Alma Mater Studiorum
Università di Bologna

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

Multihoming su Symbian per VoIP

Tesi di Laurea in Architettura

Autore:
Marcello Borsari

Relatore:
Dott. Vittorio Ghini

Anno Accademico 2009-2010
Sessione II

Indice

Elenco delle figure

Introduzione

Il mercato della telefonia mobile ha conosciuto nel corso degli ultimi anni un'enorme espansione, legata soprattutto alla diffusione di telefoni *smartphone*, ovvero dispositivi con i quali è possibile telefonare, leggere la posta elettronica, navigare in Internet, registrare video, ascoltare musica e molto altro. Nel secondo quadrimestre del 2010 le vendite di telefoni cellulari hanno raggiunto a livello mondiale un totale di 325,6 milioni di unità, con un aumento del 13,8% rispetto allo stesso periodo del 2009, mentre le vendite di smartphone hanno costituito il 19% delle vendite totali, con un aumento del 50,5% rispetto al secondo quadrimestre del 2009[1]. Per quanto riguarda la diffusione dei sistemi operativi per smartphone, troviamo in testa Symbian (Nokia) con 25,4 milioni di unità vendute (41,2%), seguito da BlackBerry OS di RIM con 11,2 milioni di terminali (18,2%), tallonato a sua volta da Google Android con 10,6 milioni (17,2%). Quest'ultima piattaforma ha recentemente superato Apple iOS, con 8,7 milioni (14,2%), seguito da Windows Mobile di Microsoft con 3,1 milioni di terminali (5%), da Linux con 1,5 milioni di unità (2,4%) e dagli altri sistemi operativi con circa 1 milione di terminali (1,8%) (figura 1).

Una così ampia diffusione degli smartphone è certamente legata alla possibilità di installarvi ulteriori programmi applicativi che aggiungono nuove funzionalità, e alla capacità di connettersi ad Internet mediante una o più interfacce di rete (GPRS, EDGE, HSDPA o Wi-Fi). Le tariffe di navigazione, inoltre, si stanno sempre più abbassando, adeguandosi al largo consumo. Questi fattori hanno reso possibile il grande sviluppo del VoIP (Voice over IP)

[1]Fonte: Gartner - http://www.gartner.com/it/page.jsp?id=1421013

in campo mobile, il cui obbiettivo principale è quello di abbattere nettamente i costi di telefonia tradizionale.

Worldwide Smartphone Sales to End Users by Operating System in 2Q10 (Thousands of Units)

Company	2Q10 Units	2Q10 Market Share (%)	2Q09 Units	2Q09 Market Share (%)
Symbian	25,386.8	41.2	20,880.8	51.0
Research In Motion	11,228.8	18.2	7,782.2	19.0
Android	10,606.1	17.2	755.9	1.8
iOS	8,743.0	14.2	5,325.0	13.0
Microsoft Windows Mobile	3,096.4	5.0	3,829.7	9.3
Linux	1,503.1	2.4	1,901.1	4.6
Other OSs	1,084.8	1.8	497.1	1.2
Total	**61,649.1**	**100.0**	**40,971.8**	**100.0**

Source: Gartner (August 2010)

Figura 1: *Diffusione dei sistemi operativi per smartphone - agosto 2010*

Attualmente uno smartphone dotato di diverse interfacce di comunicazione non può trarre vantaggio dalle potenzialità offerte dall'utilizzo contemporaneo di più connessioni, in quanto le applicazioni VoIP sfruttano solamente una sola interfaccia per la trasmissione dei dati. In generale questa caratteristica non rappresenta un problema, perché le applicazioni più utilizzate, come browser o instant messaging, non richiedono particolari requisiti per quanto riguarda le prestazioni di connessione. Nel campo della telefonia VoIP è invece decisamente importante che la qualità della chiamata sia ottimale, perché eccessive latenze o perdite di pacchetti renderebbero la conversazione incomprensibile.

Per tale motivo all'interno del Dipartimento di Scienze dell'Informazione dell'Università degli Studi di Bologna è stata sviluppata una tecnologia che consente di ottimizzare il traffico VoIP sfruttando contemporaneamente tutte le interfacce a disposizione dei dispositivi mobili. Tale tecnologia, denominata *Always Best Packet Switching for SIP-based mobile multimedia*

services[1], è stata inizialmente realizzata per dispositivi equipaggiati con il sistema operativo Linux.

Scopo di questa tesi è quindi estendere l'utilizzo di questa tecnologia anche agli smartphone con sistema operativo Symbian, realizzando un software in grado gestire in maniera adeguata il bilanciamento del traffico.

Nel capitolo 1 saranno illustrati nel dettaglio i protocolli VoIP e SIP.

Nel capitolo 2 sarà descritta l'architettura ABPS e lo scenario a cui fa riferimento, valutando lo stato dell'arte e i requisiti del multihoming.

Nel capitolo 3 sarà presentato il sistema operativo Symbian OS, illustrando in dettaglio l'architettura delle comunicazioni di cui dispone. Saranno inoltre fornite le istruzioni per l'installazione e la configurazione dell'ambiente di sviluppo e sarà illustrata l'architettura di PJSIP.

Nel capitolo 4 saranno presentate le soluzioni adottate per l'implementazione del multihoming su sistema operativo Symbian.

Parole chiave: *VoIP, multihoming, connettività, SIP, Symbian, ABPS, Always Best Packet Switching, bilanciamento traffico.*

Capitolo 1

VoIP

Voice over IP (Voce over Internet Protocol), acronimo VoIP, è una tecnologia che rende possibile effettuare una conversazione telefonica sfruttando una connessione Internet o un'altra rete dedicata che utilizza il protocollo IP. Più specificamente con VoIP si intende l'insieme delle tecnologie che descrivono la trasmissione digitale della voce nelle reti a commutazione di pacchetto.

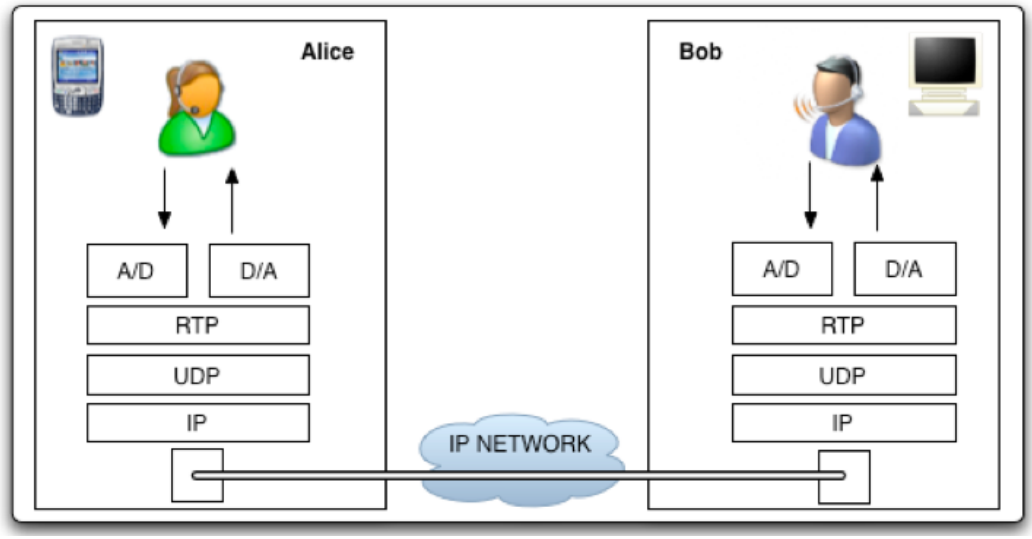


Figura 1.1: *Conversazione VoIP*

Uno dei vantaggi di questa tecnologia è che permette di fare leva su risorse di rete preesistenti, consentendo una notevole riduzione dei costi in ambito

sia privato che aziendale, specialmente per quanto riguarda le spese di comunicazione interaziendali e tra sedi diverse. Una rete aziendale, infatti, può essere sfruttata anche per le comunicazioni vocali.

La tecnologia VoIP richiede l'utilizzo di due tipologie di protocolli di comunicazione, una per il trasporto dei dati (pacchetti voce su IP), ed una per i messaggi di segnalazione (ad esempio localizzazione, registrazione e segnalazione di chiamata). Per il trasporto dei dati, nella grande maggioranza delle implementazioni VoIP viene adottato il protocollo RTP (Real-time Transport Protocol). Per quanto riguarda la seconda tipologia di protocolli, la gestione delle chiamate voce su rete IP è, al momento, indirizzata verso due differenti proposte, elaborate in ambito ITU e IETF, che sono rispettivamente H.323 e SIP (Session Initiation Protocol).

H.323 è un protocollo creato nel 1996 che propone uno standard per le sessioni di comunicazione audio, video e dati, attraverso reti orientate alla connessione nelle quali non è garantita la qualità di servizio (QoS). Si tratta di un protocollo indubbiamente stabile e collaudato, ma realizzare una rete che sfrutti tale standard risulta molto costoso e complesso.

SIP nasce come alternativa ad H.323, ed è stato creato come protocollo per la comunicazione voce in tempo reale su IP. Possiede un'architettura modulare e scalabile, ovvero capace di crescere con il numero degli utilizzatori del servizio. Inoltre, a differenza di H.323, è caratterizzato da una minore complessità e dalla somiglianza ai diffusi protocolli HTTP e SMTP. Queste caratteristiche hanno permesso a SIP di essere attualmente il protocollo VoIP più diffuso nel mercato residenziale e business.

1.1 Il protocollo SIP

Session Initiation Protocol (SIP) è un protocollo di segnalazione definito dalla IETF (Internet Engineering Task Force) nel Marzo 1999, poi aggiornato dalla RFC 3261 del Giugno 2002. Si tratta di un protocollo testuale operante a livello applicazione all'interno dello stack ISO/OSI, ed è basato su un modello di richiesta-risposta il cui scopo è quello di permettere a due o più partecipanti di stabilire una sessione composta da flussi multimediali

multipli, come audio, video o altre tipologie di comunicazione (eg. giochi distribuiti, applicazioni condivise etc).

La struttura di SIP è fortemente ispirata al protocollo HTTP e definisce la tipologia di messaggi, metodi ed entità presenti in un sistema SIP sfruttando protocolli preesistenti per compiti specifici. Le entità logiche definite sono: User-Agent (UA), Redirect Server, Proxy Server e Registrar Server. La comunicazione tra le varie entità è basata principalmente sul modello client-server ed alcune di queste entità possono svolgere entrambi i ruoli in base al tipo di operazione in corso.

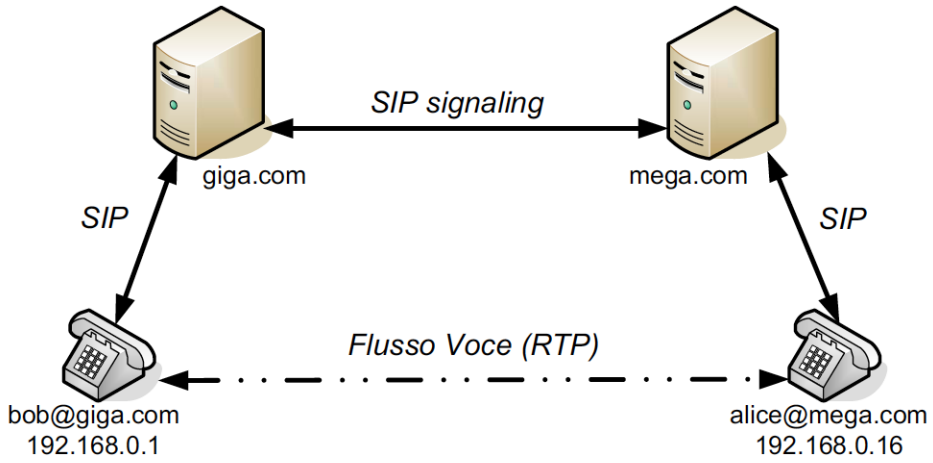


Figura 1.2: *Trapezio SIP*

L'architettura di sistema punta a concentrare la logica di protocollo presso gli User-Agent, con lo scopo di mantere l'infrastruttura intermedia (Proxy Server, Registrar Server, ecc.) il più semplice e minimale possibile. In questo modo si ottiene un sistema molto flessibile e scalabile. Gli UA rappresentano, per l'utenza, il punto di accesso ad un sistema SIP e si dividono in UAC (User Agent Client) ed UAS (User Agent Server). Ogni UA può svolgere entrambi i ruoli, in base al tipo di operazione in corso. Esempi tipici di User-Agent sono softphone, hardphone, webphone, tramite i quali l'utente può instaurare sessioni di varia natura, come audio, video e messaggistica istantanea.

La gestione specifica della sessione è affidata ad un ulteriore protocollo denominato SDP (Session Description Protocol), che ha il compito di descrivere e gestire le caratteristiche di ogni tipologia di sessione. Più precisamente

tale protocollo usato da SIP durante la fase di creazione o modifica di una sessione per negoziarne le caratteristiche specifiche.

Il protocollo più usato nel trasporto dei messaggi di segnalazione è UDP (User Datagram Protocol), di tipo connectionless, che non gestisce il riordinamento dei pacchetti e la ritrasmissione di quelli persi. È quindi generalmente considerato di minore affidabilità, ma in compenso molto rapido ed efficiente e quindi ideale per le applicazioni multimediali che richiedono di spedire una grande quantità di pacchetti. Recenti revisioni di SIP ne permettono comunque l'utilizzo attraverso TCP (Transmission Control Protocol) e TLS (Transport Layer Security). In particolare, quest'ultima capacità fornisce la possibilità di cifrare i dati tramessi sulla rete.

La gestione vera e propria dei vari flussi multimediali viene assegnata ad un ulteriore protocollo, denominato RTP (Real-time Transport Protocol). In sintesi ogni protocollo usato ha un compito molto specifico, e questa caratteristica fornisce un elevato grado di flessibilità, favorendo inoltre la possibilità di future espansioni dei servizi forniti. Questa estrema flessibilità è alla base del successo di SIP e ne ha promosso l'utilizzo negli ambiti più diversi.

1.1.1 Indirizzi

Un SIP-URI (SIP Uniform Resource Identifier) rappresenta lo schema di indirizzamento utilizzato per chiamare un altro soggetto attraverso il protocollo SIP. In altre parole, un SIP-URI è il recapito telefonico SIP di un utente.

Il formato standard per definire un indirizzo SIP è simile a quello per identificare gli indirizzi di posta elettronica:

```
sip:<username>@<hostname>
```

username: può essere un nome utente o un numero telefonico. Esistono infatti molti provider che propongono un servizio a pagamento di instradamento delle chiamate SIP su rete telefonica tradizionale;

hostname: rappresenta un dominio o un indirizzo IP.

La porta UDP di default è la 5060 ma può essere cambiata specificandola all'interno dell'indirizzo. Possono essere inseriti anche altri parametri che descrivono ulteriormente le caratteristiche del contatto, ad esempio:

```
sip:giorgio@mioDominio.it;transport=TCP
sip:carlo@194.150.20.21:5068
sip:+39-81-12345@mioGateway.com;user=phone

sips:luca@mioDominio.it
```

Come avviene per gli indirizzi e-mail, anche per SIP è possibile inserire un collegamento ipertestuale all'interno in una pagina web.

1.1.2 Entità

I componenti del network SIP sono molteplici e a ciascuno è assegnato un particolare compito.

SIP User Agent

Le entità fondamentali di SIP sono gli User Agent (UA), ovvero device in grado di accettare oppure richiedere una comunicazione. Gli UA sono composti da due entità logiche: un client UAC (User Agent Client) ed un server UAS (User Agent Server). Questi elementi si scambiano richieste e risposte consentendo a due o più terminali di comunicare tra loro.

Lo User Agent rappresenta l'unica entità con la quale l'utente interagisce direttamente, ed è quindi il punto di accesso ad un sistema SIP. Ad ogni utente possono essere associati molteplici UA: un utente può, ad esempio, essere rintracciabile contemporaneamente presso il proprio palmare, smartphone e computer.

Il compito di un UA è quello di fornire un'interfaccia opportuna per la gestione dei servizi offerti dal sistema. Nel caso di VoIP tali servizi sono tipicamente conversazioni audio o video, messaggistica istantanea o conferenze.

Di conseguenza, l'interfaccia deve contenere gli strumenti utili a gestire tali servizi, il cui numero e tipologia varia in base al terminale utilizzato.

SIP Registrar Server

Un Registrar Server è un server che accetta richieste di tipo REGISTER e memorizza le informazioni ricevute per la gestione del servizio di localizzazione. Il suo compito è quindi quello di accettare o rifiutare l'ingresso di un UA all'interno del network SIP, solitamente mediante un processo di autenticazione (figura 1.3).

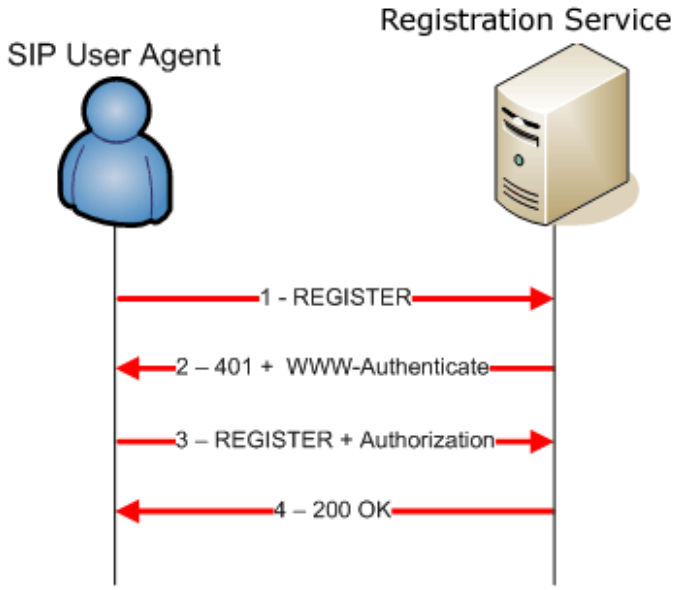


Figura 1.3: *Autenticazione*

Nel messaggio di REGISTER è specificata una lista di indirizzi ai quali l'utente può essere contattato. Questo permette all'utente di essere rintracciato su più dispositivi contemporaneamente (eg: palmare, notebook, smartphone), ovvero indipendentemente dal punto di accesso alla rete e dal dispositivo utilizzato.

All'interno del messaggio è inoltre specificato il periodo di validità della registrazione, scaduto il quale il Registrar Server considera l'utente disconnesso. Di conseguenza, un UA può inviare multipli messaggi di REGISTER, sia per rinnovare una registrazione prossima alla scadenza, sia per aggiorna-

re la lista degli indirizzi. Quest'ultima operazione avviene quando l'utente disattiva un UA precedentemente registrato oppure quando cambia il punto di accesso (ovvero l'indirizzo IP).

Di conseguenza la prima e più importante operazione eseguita da un UA, nella sua fase di avvio, è la registrazione del suo attuale indirizzo IP presso il Registrar Server.

SIP Proxy Server

Un Proxy Server è una entità intermedia che agisce sia da server che da client. Fondamentalmente inoltra i messaggi tra gli UA coinvolti nella conversazione, replicandone i comportamenti. Per questo motivo, i messaggi generati sono percepiti dagli UA di destinazione come trasmessi dal proxy stesso. Richieste e risposte possono attraversare più Proxy Server prima di raggiungere la destinazione finale.

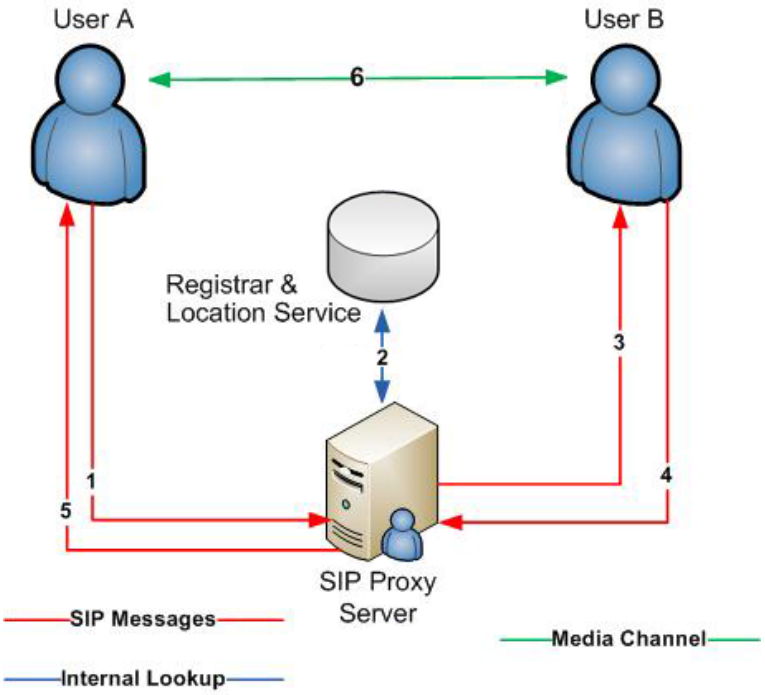


Figura 1.4: *Proxy Server*

Il Proxy Server non rappresenta direttamente una entità ma una categoria di entità: i compiti che svolge all'interno di un sistema SIP variano dal

semplice instradamento dei messaggi fino a funzionalità più avanzate. In generale i Proxy Server possono essere classificati in base alla quantità di informazioni che preservano ed utilizzano durante una sessione. Una gestione passiva prevede infatti che il proxy si limiti ad instradare opportunamente il messaggio; una gestione attiva può prevedere funzionalità aggiuntive e arbitrariamente complesse, che si sviluppano per tutta la durata della sessione stessa e che richiedono il salvataggio di opportune informazioni di stato. Le principali tipologie di proxy sono le seguenti:

Stateful Proxy: ciascuna richiesta SIP viene eseguita considerando le informazioni ottenute dalle richieste precedenti. Vengono mantenute informazioni sullo stato delle sessioni esistenti, permettendo l'uso di alcuni servizi aggiuntivi, come la ritrasmissione delle richieste in mancanza di risposta (liberando l'UA da questo compito e riducendo l'utilizzo delle risorse di rete), oppure l'autenticazione. Un tipo particolare di Stateful Proxy è il Transaction Stateful Proxy, che si limita a tenere traccia dei messaggi fino a quando la singola transazione non viene completata. La transizione ha inizio con la ricezione della richiesta ed ha termine con la ricezione della risposta definitiva. Rientra in questa categoria anche il Call Stateful Proxy, che partecipa alla gestione di una sessione per tutta la sua durata. È quindi in grado di collezionare ed utilizzare informazioni sullo stato di un'intera sessione.

Stateless Proxy: tale tipologia di proxy non conserva alcuna informazione di stato. Per ogni richiesta o risposta pervenuta si limita a inoltrare il messaggio all'entità SIP successiva, sfruttando unicamente informazioni di instradamento fornite nel messaggio stesso.

SIP Redirect Server

Il compito di un Redirect Server è quello di semplificare la localizzazione di un utente, fornendo informazioni opportune su dove si possa trovare il destinatario ricercato. Si appoggia ad un Database oppure ad un Location Service per l'operazione di localizzazione dell'utente. SIP offre infatti un servizio di ricerca utenti: elementi di rete SIP come Proxy Server e Redirect

Server sono responsabili, dopo la ricezione di una richiesta, di determinare dove essa vada inviata. Le entità consultano il Location Service, il quale fornisce il legame utente-indirizzo IP o il riferimento ad altri URI, in modo analogo al funzionamento dei sistemi DNS di Internet.

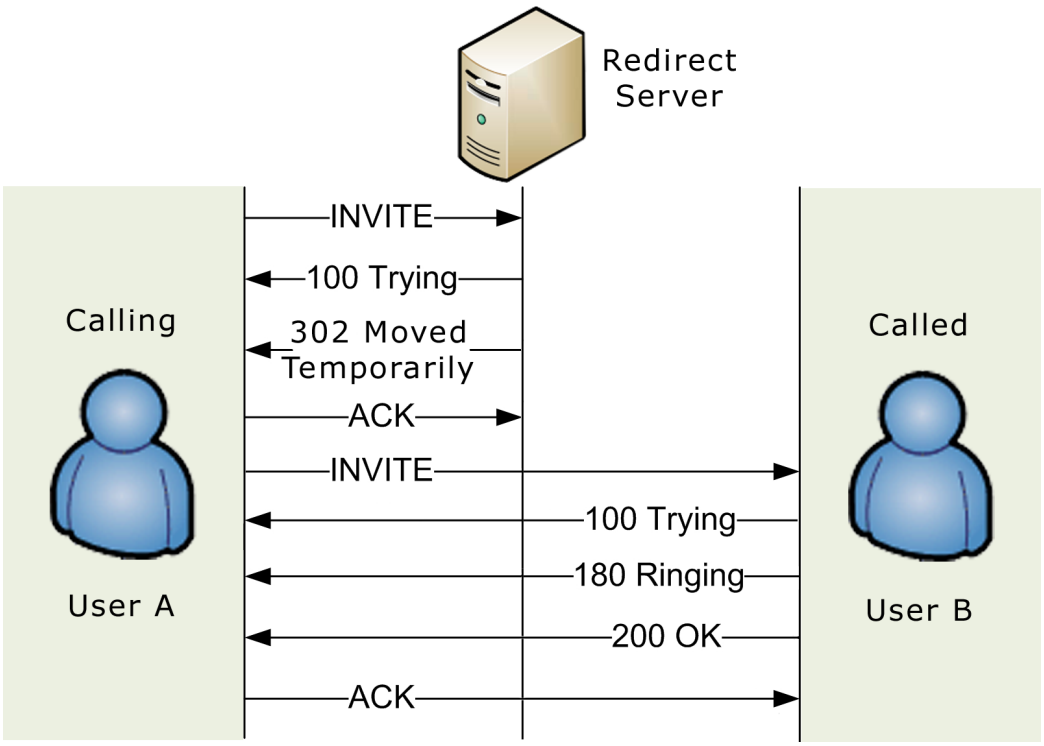


Figura 1.5: *Redirect Server*

Un Redirect Server non si limita a fornire l'intera lista dei contatti attivi (ovvero degli UA) appartenenti ad un utente, piuttosto suggerisce dove è più probabile che si trovi al momento della richiesta. Ogni utente può infatti specificare dove può essere rintracciato ad una determinata ora della giornata, o in un determinato giorno settimanale. Nel caso in cui il Redirect Server non possieda il contatto diretto dell'utente ricercato, può restituire l'indirizzo di un ulteriore Redirect Server in possesso di informazioni più dettagliate. La possibilità di concatenare i vari server permette una gestione molto articolata e complessa della localizzazione di un UA. Tale proprietà è valida non solo per i Redirect Server ma anche per i Proxy Server.

È interessante notare che l'utilizzo di un Redirect Server al posto di un Proxy Server ha spesso lo scopo di non sovraccaricare il dominio di pertinenza dell'utente cercato. Un Proxy Server si occupa direttamente della ricerca e del contatto dell'utente, impegnando quindi determinate risorse di rete. Al contrario, un Redirect Server si limita a fornire opportune informazioni, lasciando allo UA il compito di contattare l'utente in base alle informazioni ricevute.

1.1.3 Messaggi

Il modello usato per la sintassi del protocollo SIP è text-based, che trae ispirazione dal protocollo HTTP; di conseguenza ogni messaggio è anch'esso testuale e composto da un insieme di header più un eventuale body. Per instaurare una sessione, avviene un three-way handshaking (concettualmente simile a quello che avviene con il protocollo TCP). La similitudine con HTTP risiede anche nella tipica logica di richiesta-risposta.

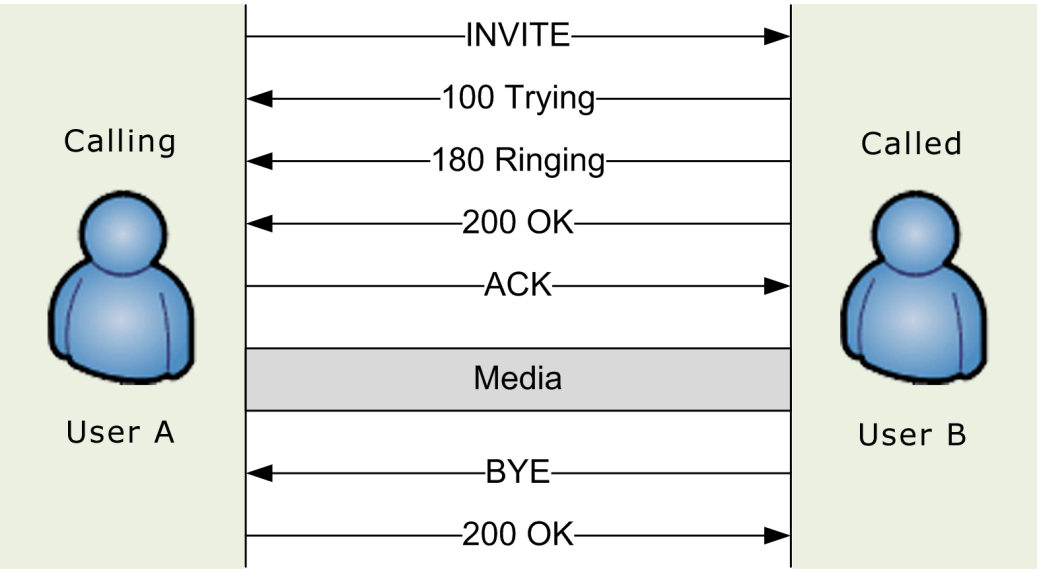


Figura 1.6: *Esempio di sessione SIP*

Messaggi di richiesta

Una richiesta è composta da una linea iniziale che specifica la release del protocollo SIP utilizzato, l'intestazione del messaggio, una linea vuota e il corpo opzionale del messaggio. Il formato di intestazione di una richiesta è composto da:

- **Method:** indica il tipo di operazione richiesta;
- **Request-URI:** indica il destinatario, cioè l'entità che dovrebbe processare l'operazione;
- **SIP Version:** indica la versione del protocollo SIP.

Dopo l'intestazione si trovano i dati specifici del messaggio che è stato inviato. I metodi definiscono le azioni che possono essere richieste.

INVITE: richiede l'apertura di una sessione. All'interno della richiesta sono presenti informazioni relative alla sessione che si vuole stabilire, definite tramite il protocollo SDP (Session Description Protocol). L'identificativo di sessione viene generato dall'UA che intende richiedere l'apertura della sessione, e sarà utilizzato in tutti i messaggi successivi. È possibile generare un ulteriore messaggio INVITE, denominato re-INVITE, con lo scopo di rinegoziare oppure modificare i parametri di sessione. Un re-INVITE non può essere inviato durante l'instaurazione dei una sessione. In questo caso è necessario utilizzare il metodo UPDATE.

ACK: conferma la ricezione di una risposta definitiva relativa ad una precedente richiesta di INVITE. Nel momento in cui il destinatario riceve un INVITE, viene generata immediatamente una risposta provvisoria (tipicamente 180, il telefono sta squillando), tuttavia l'eventuale risposta definitiva (200 OK) viene generata dal destinatario solo quando l'utente ha effettivamente accettato la chiamata. Idealmente possono passare svariati secondi, di conseguenza, per accertarsi che lo UA chiamante sia ancora in attesa di instaurare la sessione, quest'ultimo ha il dovere di generare un ACK per confermare la sua presenza dopo aver ricevuto il 200 OK.

OPTIONS: richiede le capacità di un UA. Un client si può servire di questo metodo per ottenere informazioni relative allo stato di un utente e alle funzionalità da esso supportate.

BYE: richiede la terminazione di una sessione attiva. Nel caso in cui la sessione coinvolga due partecipanti, l'abbandono da parte di uno di questi comporta la terminazione della sessione. Quando invece sono coinvolti più partecipanti non si hanno conseguenze sulla sessione.

CANCEL: richiede la cancellazione di una richiesta precedentemente tramsessa. Tipicamente viene utilizzato quando un UA decide di interrompere un tentativo di chiamata. Una richiesta CANCEL non ha effetto nel caso in cui la sessione sia già stata instaurata.

REGISTER: richiede la registrazione di un UA, ovvero informa il relativo Registrar Server che l'utente identificato da un determinato SIP-URI è contattabile sull'indirizzo fornito. La richiesta specifica inoltre un periodo di validità della registrazione (tipicamente di un'ora), rinnovabile tramite un nuovo messaggio REGISTER.

INFO: questo metodo viene usato per scambiare informazioni senza modificare lo stato della sessione. Il percorso seguito dal messaggio è quello (diretto o mediato da Server Proxy) stabilito in fase di call setup. Le informazioni di interesse possono essere inserite in un apposito campo di intestazione INFO o semplicemente nel corpo del messaggio.

PRACK: consente ad un client di riscontrare la ricezione delle risposte provvisorie, che possono essere così trasmesse in maniera affidabile, al pari di quelle definitive.

SUBSCRIBE: richiede la sottoscrizione ad un evento al fine di ricevere notifiche attraverso richieste di tipo NOTIFY. La sottoscrizione implica la creazione di una sessione. Il messaggio contiene anche un campo che specifica la validità temporale della sottoscrizione, che può essere rinnovata all'interno della stessa sessione.

NOTIFY: questa primitiva rende possibile la notifica di un evento verificatosi su un nodo appartenente al network. La richiesta viene trasmessa all'interno di una sessione precedentemente creata mediante una richiesta SUBSCRIBE.

UPDATE: consente ad un client di aggiornare i parametri di una sessione (quali tipologie di flussi multimediali e relative codifiche) senza modificare lo stato della sessione. Il metodo è usato prima che la sessione sia stata completamente stabilita tramite il messaggio INVITE.

MESSAGE: consente ad un utente di inviare messaggi immediati nell'ambito di una sessione già stabilita. Il contenuto dei messaggi viene inviato all'interno del corpo della richiesta utilizzando il formato MIME (Multipurpose Internet Mail Extensions), tipico della posta elettronica e adatto al trattamento di dati multimediali.

REFER: con questo messaggio un client può invitare il destinatario a contattare un altro indirizzo, indicato in un apposito campo di intestazione denominato Refer-To. Questo meccanismo consente a SIP di supportare una serie di funzionalità, come il trasferimento di chiamata.

Messaggi di risposta

Il formato di un messaggio di risposta è il seguente:

- versione SIP;
- codice di stato;
- indicazione della ragione.

Il codice di stato è un numero di tre cifre che indica quale è stato l'esito della richiesta effettuata. La prima cifra identifica la classe a cui appartiene tale risposta, le restanti identificano una specifica risposta all'interno di tale classe. Le risposte appartenenti alla classe 1XX sono provvisorie ed informative, mentre le altre sono invece definitive. Lo standard prevede che per ogni richiesta invocata si possano ottenere una o più risposte provvisorie, ma al più una risposta definitiva.

Le classi di risposta, corredate da alcuni esempi di messaggio, sono le seguenti:

1XX In ricerca, squillo, accodato: indica una risposta informativa e provvisoria. Le risposte appartenenti a questa classe sono:

- **100 Trying:** il server SIP sta cercando l'utente destinatario della richiesta;
- **180 Ringing:** l'utente è stato localizzato e lo User-Agent in esecuzione sul suo terminale sta cercando di avvertirlo della chiamata. Si è in attesa di una risposta;
- **181 Call Is Being Forwarded:** si sta inoltrando la chiamata verso una diversa destinazione;
- **182 Queued:** l'utente non è al momento disponibile. La chiamata è stata messa in coda;
- **183 Session Progress:** questo messaggio può essere usato per comunicare al chiamante informazioni relative allo stato di una chiamata.

2XX Successo: indica che la richiesta è stata eseguita con successo. La risposta definitiva inviata dal server è 200 OK.

3XX Forwarding: indica che la richiesta è stata inoltrata. Questa classe di risposte fornisce informazioni riguardanti la nuova posizione dell'utente o i servizi alternativi che potrebbero portare a buon fine la chiamata.

- **300 Multiple Choices:** l'indirizzo fornito nella richiesta ha condotto all'individuazione di più terminali. L'utente dovrà quindi effettuare una scelta;
- **301 Moved Permanently:** l'utente cercato non è più raggiungibile all'indirizzo specificato nella richiesta. Il chiamante deve quindi inviare una nuova richiesta all'indirizzo fornito nel campo Contact dell'intestazione;

- **302 Moved Temporarily:** l'utente cercato non è temporaneamente raggiungibile all'indirizzo specificato nella richiesta. Il chiamante deve inviare una nuova richiesta all'indirizzo fornito nel campo Contact dell'intestazione. Può essere indicata la validità temporale di questo nuovo indirizzo in un apposito campo;
- **305 Use Proxy:** la risorsa a cui è destinata la richiesta deve essere acceduta attraverso un Proxy Server, il cui indirizzo è fornito nel campo Contact;
- **380 Alternative Service:** la chiamata non ha avuto successo, ma sono disponibili servizi alternativi descritti nel corpo del messaggio.

4XX Errori lato client: indica il fallimento, imputabile al client, della richiesta.

- **400 Bad Request:** la richiesta non è stata compresa a causa di una sintassi errata. La descrizione associata al codice di stato può fornire ulteriori informazioni sull'errore;
- **401 Unauthorized:** la richiesta necessita di un'autorizzazione da parte di un UAS o di un Registrar Server;
- **403 Forbidden:** la richiesta è stata rifiutata;
- **404 Not Found:** l'utente cercato non può essere trovato;
- **405 Method Not Allowed:** il metodo tentato non è consentito sull'indirizzo specificato. La risposta deve contenere un campo di intestazione con l'elenco dei metodi ammessi;
- **407 Proxy Authentication Required:** la richiesta necessita di un'autenticazione presso un Proxy Server;
- **415 Unsupported Media Type:** il server rifiuta di soddisfare la richiesta in quanto il corpo del messaggio si presenta in un formato non supportato;
- **420 Bad Extension:** il server non supporta o non comprende l'estensione del protocollo richiesta dal metodo e specificata nel campo Require o Proxy-Require dell'intestazione;

- **485 Ambiguous:** l'indirizzo indicato nella richiesta è ambiguo e non può quindi essere associato ad un utente in maniera univoca. Una serie di indirizzi simili non ambigui viene fornita dal campo Contact della risposta.
- **486 Busy Here:** il destinatario è stato contattato con successo, ma al momento è occupato.

5XX Errori lato server: indica un errore da parte del server.

- **500 Server Internal Error:** il server ha incontrato degli ostacoli imprevisti nel tentativo di soddisfare la richiesta. Se tale condizione è momentanea, il server può usare il campo di intestazione Retry-After per indicare al client quando ripetere la richiesta;
- **501 Not Implemented:** il server non supporta la funzionalità necessaria a soddisfare la richiesta. Questa risposta viene generata quando il metodo usato nella richiesta non è stato riconosciuto, distinguendosi pertanto dal codice 405 (Methot Not Allowed);
- **503 Service Unavailable:** il server al momento non è in grado di elaborare la richiesta a causa di un sovraccarico o di operazioni di manutenzione in corso. Il client può rivolgere la richiesta ad un altro server, ma non dovrebbe tentare di ricontattare il primo server prima del termine stabilito dal campo Retry-After della risposta;
- **504 Server Time-out:** il server non ha ricevuto una risposta in tempo utile da un server esterno.

6XX Occupato, rifutato, non disponibile: indica un errore globale.

- **600 Busy Everywhere:** il destinatario è stato contattato su ogni indirizzo disponibile, ma non può rispondere perché occupato. Il campo Retry-After dell'intestazione può suggerire al client un momento migliore per ritentare. Un server genera questo codice solo se è certo che l'utente non sia disponibile su nessun altro terminale. In caso contrario il codice restituito sarà 486 (Busy Here).

- **603 Decline:** questa risposta viene inviata se il destinatario, contattato con successo, non vuole o non può accettare la richiesta di comunicazione.
- **604 Does Not Exist Anywhere:** il server sa con certezza che l'utente indicato nel Request-URI non esiste all'interno della rete SIP.
- **606 Not Acceptable:** lo User-Agent cercato è stato contattato con successo, ma alcuni aspetti della descrizione della sessione presenti nella richiesta (quali mezzi necessari per la comunicazione, ampiezza di banda o stile di indirizzamento) non sono stati accettati. La risposta può contenere un elenco delle ragioni per cui la richiesta non può essere accolta.

1.1.4 Attraversamento NAT

Nel campo delle reti telematiche, il Network Address Translation (NAT), ovvero traduzione degli indirizzi di rete, è una tecnica che consiste nel modificare gli indirizzi IP dei pacchetti in transito all'interno di una rete. Spesso implementato dai router e dai firewall, si divide in source NAT (SNAT) e destination NAT (DNAT), a seconda che venga modificato l'indirizzo sorgente o l'indirizzo destinazione del pacchetto che inizia una nuova connessione. I pacchetti che viaggiano in senso opposto verranno modificati in modo corrispondente, in modo da dare, ad almeno uno dei due computer che stanno comunicando, l'illusione di essere connessi con un indirizzo IP diverso da quello effettivamente utilizzato dalla controparte.

Utilizzando il NAT diventa possibile connettere multipli device ad Internet, utilizzando un unico indirizzo IP pubblico (in caso contrario gli indirizzi disponibili previsti dal protocollo IP versione 4 sarebbero da tempo esauriti). Un altro vantaggio evidente è che protegge gli utenti da attacchi dolosi provenienti dall'esterno, impedendo ai nodi Internet esterni di iniziare connessioni nei confronti di quelli mascherati dal NAT.

Questa caratteristica rappresenta uno dei maggiori ostacoli per le comunicazioni VoIP basate su SIP (e su protocolli associati, come RTP). Senza NAT o firewall tutti i device sarebbero in grado di comunicare tra loro senza

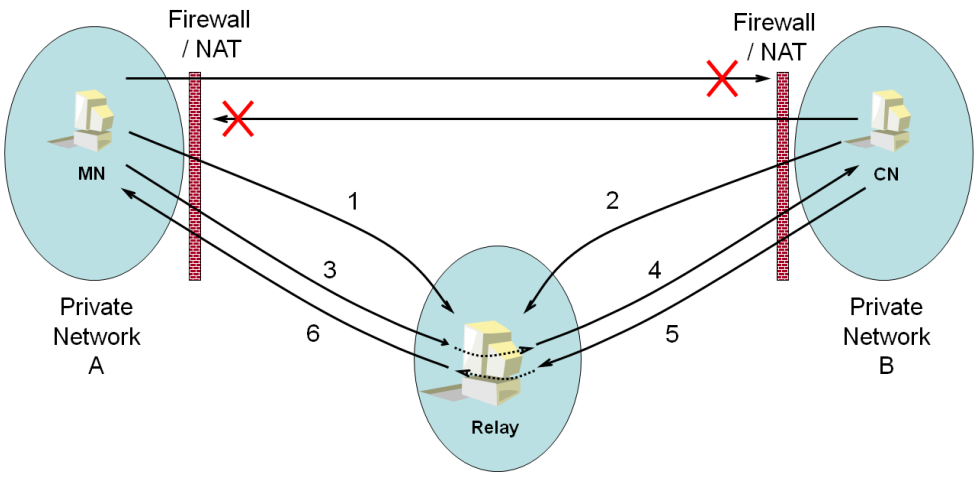


Figura 1.7: *NAT e Firewall*

alcun tipo di intermediario ad eccezione dei routers, in cui a ciascun device sarebbe assegnato un indirizzo IP pubblico verso cui instradare l'opportuno traffico. In realtà la maggior parte dei device appartenenti ad una stessa LAN e connessi a Internet fanno uso della funzione di NAT presente nel router perimetrale.

Il problema nasce quando la connessione è originata dall'esterno. In questo caso, l'utente che vuole avviare una comunicazione può solo indicare come destinatario l'indirizzo IP pubblico del router ma quest'ultimo, non avendo alcuna connessione attiva di riferimento, non sa dove recapitare i pacchetti.

Il problema viene ulteriormente complicato a causa della varietà di NAT disponibili attualmente sul mercato, ed anche a causa dell'ampio numero di possibili scenari operativi ed architetturali.

Full Cone: tutte le richieste provenienti da un IP address interno e una certa porta vengono mappate sulla stessa porta dello stesso IP address esterno. Di conseguenza, qualsiasi host esterno può inviare un pacchetto all'host interno, inviandolo all'indirizzo esterno così mappato. Un NAT Full Cone mappa quindi le sessioni attive su di una coppia indirizzo interno/porta su una specifica coppia indirizzo esterno/porta. Tali sessioni possono essere iniziate sia da host interni che esterni.

Restricted Cone: tutte le richieste provenienti da un IP address interno e una certa porta vengono mappate sulla stessa porta dello stesso IP address esterno. Differentemente dal NAT di tipo Full Cone, un host esterno (con un dato IP address) può inviare un pacchetto all'host interno solamente nel caso l'host interno abbia precedentemente inviato un pacchetto a quello stesso IP address. Di conseguenza, solamente host interni possono iniziare una sessione attraverso NAT di tipo Restricted Cone.

Port Restricted Cone: simile al Restricted Cone NAT. La restrizione è estesa ai numeri di porta, i pacchetti provenienti da host esterni devono contenere come indirizzo IP sorgente/porta una coppia precedentemente utilizzata da un host interno.

Symmetric: tutte le richieste provenienti da un IP address interno e una certa porta vengono mappate sulla stessa porta dello stesso IP address esterno. La differenza risiede nella destinazione e nella dinamicità del mapping: ogni volta che indirizzo IP o porta di destinazione cambiano, sarà utilizzato un mapping diverso sull'interfaccia esterna del router, ovvero una differente combinazione pubblica di indirizzo e porta.

Per risolvere il problema dell'attraversamento di NAT è stato creato STUN, ovvero Simple Traversal of User datagram protocol through Network address translators. Si tratta di un protocollo che permette a particolari applicazioni di scoprire la presenza ed i tipi di NAT e firewall presenti tra loro e la rete pubblica. STUN permette a questi programmi di conoscere gli indirizzi con cui il dispositivo NAT li rende visibili alla rete pubblica. STUN opera con molti NAT presistenti e non richiede particolari configurazioni. Come risultato, STUN assicura ad una grande varietà di applicazioni IP (ad esempio, i telefoni VoIP) di lavorare attraverso le varie strutture NAT presistenti.

STUN è un protocollo client-server. Un telefono o un software VoIP può includere un client STUN, che invierà una richiesta ad un server STUN. Il server riporterà al client l'indirizzo IP pubblico e la porta UDP che il dispositivo NAT (es. router) sta associando al client per il traffico entrante

nella rete. Le risposte permettono anche al client STUN di determinare che tipo di NAT sia in uso. Ci sono tre tipi di NAT che è possibile attraversare tramite STUN: Full Cone, Restricted Cone e Port Restricted Cone. STUN non lavora con il quarto tipo di NAT, detto simmetrico o bidirezionale, questo a causa del fatto che i dati trovati dal server STUN non saranno validi per terze parti, in quanto il NAT bidirezionale non permette a terzi di riusare IP e porte abilitate, differenziando le associazioni a seconda dell'host contattato.

Client e server STUN sono utilizzati con protocolli come SIP tramite UDP per il trasferimento di traffico voce/video/testo su Internet.

1.2 Terminal Mobility

Il VoIP viene tipicamente classificato come servizio P2P (peer-to-peer): una conversazione che coinvolga due terminali prevede tipicamente l'instaurazione di un canale di comunicazione diretto tra di essi. Affinchè la comunicazione possa aver luogo è necessario che entrambi i terminali conoscano l'indirizzo IP dell'altro interlocutore. Nel caso in cui uno di essi sia un terminale mobile, un eventuale cambio di indirizzo IP causa la distruzione del canale di comunicazione. Il terminale fisso dovrà quindi recuperare il nuovo indirizzo IP del terminale mobile prima di poter ripristinare la comunicazione. La velocità con cui tale aggiornamento avviene influenza completamente la possibilità di continuare la comunicazione per i due interlocutori.

Risulta quindi necessario utilizzare un supporto che fornisca ad un dispositivo la capacità di muoversi da una rete ad un'altra, continuando, durante lo spostamento, ad essere raggiungibile da nuove richieste e mantenendo le informazioni di sessione. Uno smartphone moderno dotato di più di una interfacca di comunicazione può utilizzare contemporaneamente più connessioni. Questa capacità è denominata *multihoming*, e permette ad un terminale di continuare la trasmissione dei dati anche nel caso in cui una delle interfacce smetta di funzionare. Il multihoming è inoltre utile per distribuire il carico delle trasmissioni dei dati tra le diverse interfacce, oltre ad evitare episodi di inattività causati da disconnessioni non immediatamente rilevate, causate per esempio da una eccessiva diminuzione del segnale di ricezione o da interferenze temporanee.

Capitolo 2

L'architettura ABPS

2.1 Stato dell'arte

Le architetture per l'integrazione di reti eterogenee, note con il nome di *Seamless Host Mobility Architectures*, sono responsabili di identificare univocamente un nodo multihomed (MN), permettendogli di poter essere raggiunto dagli altri nodi con cui ha già effettuato una connessione e selezionando la rete migliore in modo da proseguire la conversazione. Queste architetture non hanno una posizione ben definita all'interno del classico stack ISO/OSI, possono essere invece implementate ad ogni differente livello.

Nelle comunicazioni VoIP, l'indirizzo IP ha il compito di identificare il nodo, in quanto rappresenta l'univoca destinazione per i pacchetti provenienti dagli altri nodi con cui comunica. Trattandosi di un nodo mobile, esso riceverà un diverso indirizzo IP ad ogni ricollegamento, perdendo ad ogni selezione l'identità precedente. In questo modo, i nodi corrispondenti (CN) non riusciranno nuovamente a contattarlo prima di essere a conoscenza del suo nuovo indirizzo IP. Ne risulterà quindi una discontinuità all'interno della comunicazione. La soluzione comune a tutte le architetture di *Mobile Management*, anche se implementate in strati ISO/OSI differenti, si basa su due semplici principi:

1. definire un identificatore univoco per il nodo, indipendente alla provenienza dell'host;

2. fornire un servizio di localizzazione, sempre raggiungibile dai nodo corrispondente, per mantenere un'associazione tra l'identificatore univoco del nodo e il suo reale indirizzo di provenienza.

Il servizio di localizzazione è composto da un Location Registry (LR), attivo su un server con indirizzo IP fisso e pubblico e raggiungibile da qualsiasi host. Se il CN è a conoscenza dell'identificatore univoco del MN, sarà sufficiente mettersi in contatto con il Location Registry per recuperarne l'indirizzo IP, per poi inizializzare o ripristinare una comunicazione diretta. Il Location Registry è rappresentato da una funzione di mapping simile al DNS che opera come servizio esterno alla rete di provenienza dei nodi. Ogni MN utilizza il protocollo SIP per inviare un messaggio REGISTER al server, aggiornando la propria localizzazione. Quest'ultima soluzione non risulta efficiente perché, quando avviene una riconfigurazione dell'indirizzo IP, il tempo impiegato dal nodo mobile per comunicare al server l'aggiornamento introduce un ritardo inaccettabile, durante il quale viene interrotta la comunicazione.

2.2 Requisiti del multihoming

Viste le limitazioni delle soluzioni attualmente esistenti, ci si rende conto della necessità di una soluzione specifica. I requisiti essenziali per un completo supporto alla seamless mobility nello scenario appena descritto sono i seguenti:

Trasparenza a livello utente: il roaming deve essere concluso il più velocemente possibile. L'utente non deve notare interruzioni nella comunicazione e quando questo non risulti possibile, è necessario ridurre al minimo tali interruzioni.

Trasparenza a livello rete: non deve essere richiesto esplicito supporto nelle varie reti di accesso. Queste devono solo garantire connettività su protocollo IP.

Compatibilità: la soluzione deve essere completamente compatibile con lo scenario preesistente, ovvero con relative entità e protocolli. In una comunicazione tra un terminale mobile ed uno fisso non deve essere

richiesto supporto specifico da parte del terminale fisso. Questo quindi deve rimanere ignaro della mobilità del terminale corrispettivo.

QoS: la mobilità del terminale MN deve essere gestita rispettando adeguatamente i requisiti di QoS.

Full-Mobile: deve essere supportata la possibilità che entrambi i terminali in comunicazione siano mobili. Essi devono quindi essere in grado di proseguire una comunicazione indipendentemente dai rispettivi spostamenti.

NAT-Friendly: la soluzione deve essere compatibile con la presenza di politiche di NAT sulle reti di accesso, in modo da non risultare di ostacolo alle preesistenti tecniche di NAT-Traversal.

Si nota che, in base al requisito *trasparenza a livello rete*, le reti di accesso sulle quali si sposta il terminale devono solo fornire connettività su protocollo IP. Non si vuole infatti porre alcun limite alle tipologie di roaming gestibili, in modo da fornire una mobilità completa. L'idea è quella di sfruttare il multihoming per fornire la continuità della comunicazione in piena mobilità, gestendo opportunamente eventuali riconfigurazioni dell'indirizzo IP utilizzato.

2.3 Scenario

Lo scenario a cui fa riferimento il modello *Always Best Packet Switching* (ABPS)[1] ha come oggetto principale la comunicazione VoIP su un dispositivo mobile, che può essere un laptop o, con maggiore probabilità, un dispositivo di telefonia mobile, equipaggiato con più di un'interfaccia di rete (NIC) wireless come WiMax, WiFi (IEEE802.11/a/b/g/n), GPRS, EDGE, 1xRTT, ZigBee o altre.

La figura 2.1 illustra lo scenario composto da un terminale VoIP multihomed equipaggiato con due o più interfacce wireless, una Wi-Fi 802.11b/g/n e una 3G, posizionato in una tipica area metropolitana che offre sia copertura 3G che WiFi.

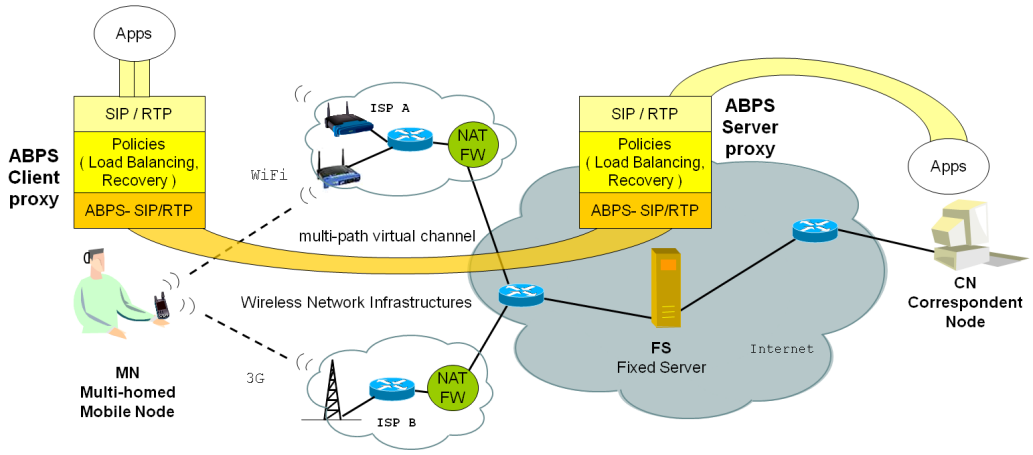


Figura 2.1: *Il sistema ABPS*

Il vantaggio di avere a disposizione interfacce di rete eterogenee sullo stesso dispositivo mobile è evidente soprattutto nel caso di interfacce wireless, perché rende possibile la connettività nel caso venga meno una delle connessioni, estendendo la copertura delle aree Wi-Fi con la copertura GPRS/UMTS. Una situazione del genere può verificarsi spesso in ambienti urbani, dove un utente si sposta in diversi punti della città cambiando access point e interfacce di rete, e viene definita dalle specifiche 3GPP come *Voice Call Continuity*[2] (VCC). Un altro vantaggio è la selezione, nel caso sia disponibile più di un'interfaccia, del NIC preferito per accedere ad Internet. Quest'ultimo approccio è utilizzato dalla funzione di mobility management delle reti wireless, seguendo il modello di *Always Best Connected*[3] (ABC), che suggerisce appunto di selezionare la migliore interfaccia NIC da usare come singolo punto di accesso a Internet. Nel caso in cui le prestazioni di un'interfaccia degradino eccessivamente, il dispositivo mobile rileverà una nuova interfaccia NIC preferita e la sostituirà alla prima.

L'architettura ABPS si basa su questo modello, ponendosi come scopo quello di offrire all'utente mobile il massimo della qualità di comunicazione, sfruttando al meglio le capacità aggregate di tutte le interfacce di rete disponibili. A causa di limitazioni tecniche ed economiche, l'utilizzo di servizi multimediali come le conferenze audio/video su Internet mediante i dispositivi precedentemente descritti presentano ancora problematiche irrisolte,

nonostante le continue innovazioni in campo. Le applicazioni multimediali VoIP e Video on Demand (VoD) risentono maggiormente di queste limitazioni tecniche perché richiedono esigenze di Quality of Service (QoS) restrittive per poter offrire all'utente un buon livello di Quality of Experience (QoE). Ad esempio, le raccomandazioni ITU-T contenute nel documento G.1010[4] stabiliscono che l'utente VoIP necessiti, per ottenere un servizio soddisfacente, di una latenza end-to-end minore di 150ms e una percentuale di pacchetti persi inferiore al 3%.

Il sistema ABPS è una soluzione completa ed efficace di QoS e Terminal Mobility per il VoIP wireless basato su SIP, ed è composta da due entità:

SIP Mobility: un'entità realizzata a livello applicazione che gestisce le conseguenze di un handover layer-3 (ovvero la riconfigurazione dell'indirizzo IP), che rispetti i requisiti precedentemente elencati.

Vertical Mobility: un'entità realizzata a livello data-link e network per monitorare lo stato di ogni interfaccia di rete presente sul dispositivo, gestendo in base ad opportune metriche e politiche la selezione dell'interfaccia di trasmissione.

La prima entità è realizzata mediante un server posto sulla rete pubblica, mentre la seconda è una applicazione eseguita direttamente sul terminale mobile. Il server rappresenta il punto d'ancora del terminale mobile ed ogni comunicazione VoIP del terminale passa attraverso di esso: qualunque entità VoIP-SIP che debba comunicare con il terminale contatta il server stesso, credendo di dialogare effettivamente con il terminale. In questo modo la mobilità del terminale viene gestita direttamente ed esclusivamente dal server.

In conseguenza di una scelta progettuale dovuta soprattutto a questioni pratiche, la versione corrente del sistema ABPS prevede che l'applicazione includa diversi livelli con diverse funzioni. L'idea originale era infatti quella di utilizzare un proxy server ABPS in locale, sul dispositivo Mobile Node (MN), per aggiungere ai pacchetti SIP ed RTP gli header ABPS, interfacciandosi con il Fixed Server (FS) in modo del tutto trasparente rispetto all'applicazione client. In questa nuova implementazione invece viene integrato tutto in un'unica applicazione client specifica per ABPS, divisa in più livelli, ad

ognuno dei quali è assegnato un compito specifico dell'architettura. In particolare l'applicazione completa è composta da un livello più basso di selezione dell'interfaccia di rete per effettuare handover e load balancing, un secondo livello formato da un insieme di funzioni e strutture per effettuare autenticazione mutuale e firma dei pacchetti, e di un terzo e un quarto livello che riguardano il client SIP vero e proprio, ovvero l'utilizzo dei livelli sottostanti per trasmettere al Proxy Server FS e lo sviluppo della GUI di un softphone SIP.

Le due entità descritte, il client ABPS su MN e il proxy su FS creano un tunnel logico tra esse permettendo la continuità della comunicazione in modo completamente trasparente, sia al terminale che alle altre entità in comunicazione con quest'ultimo.

Capitolo 3

Symbian OS

Symbian OS è un sistema operativo per dispositivi mobili e smartphone. È l'erede del sistema operativo EPOC[1] creato dalla Psion alla fine degli anni novanta per la propria linea di palmari. Nato nel giugno del 1998 con la creazione della compagnia indipendente *Symbian Limited*, Symbian dispone di funzionalità di multithreading, multitasking e protezione della memoria. Il funzionamento di Symbian è limitato esclusivamente alla categoria dei processori ARM, ed è basato sulla gestione di eventi. La CPU è automaticamente disabilitata se non vi sono eventi attivi e l'utilizzo di questa tecnica assicura una maggiore durata della batteria.

La *Serie 60* di Nokia è molto diffusa in tutto il mondo ed i modelli di base che la utilizzano hanno costi relativamente moderati. Per questo motivo ha raggiunto nel corso degli anni una vasta diffusione, incentivata ulteriormente dalla possibilità di caricare svariati programmi (gratuiti e a pagamento) che permettono di ampliare le funzioni del cellulare.

Nel giugno del 2008 Nokia ha fondato l'organizzazione non-profit *Symbian Foundation*. Symbian, a partire dal 04 febbraio 2010, è quindi diventato un sistema operativo libero. Il sito dell'organizzazione[2] offre la possibilità di scaricare il software necessario allo sviluppo, oltre che a fornire la documentazione relativa alle API, esempi e supporto tecnico mediante forum e mailing-list.

[1] http://it.wikipedia.org/wiki/EPOC
[2] http://www.symbian.org/

3.1 Symbian C++

Il sistema operativo Symbian supporta lo sviluppo di applicazioni in Java e in liguaggio C++, a cui sono state aggiunte particolari estensioni proprie di Symbian.

3.1.1 Classi

In Symbian si definiscono diversi tipi di classi, ognuna delle quali possiede differenti caratteristiche. Questa suddivisione consente di descrivere le principali proprietà ed il comportamento degli oggetti di ogni classe, come la possibilità di allocarli nello stack o nello heap (o in entrambi) e di deallocarli, in modo da effettuare una efficiente pulizia della memoria. Per consentire una facile distinzione dei tipi di classi, in Symbian si utilizza una semplice convenzione che prevede di inserire dei prefissi, come T, C, R o M, all'interno del nome della classe.

T Classes: le *T classes* si comportano come i tipi predefiniti di C++, non prevedono l'implementazione del distruttore e di conseguenza non devono contenere membri che a loro volta lo implementino. Le T classes contengono variabili membro di tipo predefinito, oggetti di altre T classes e riferimenti o puntatori relativi ad associazioni ("uses a") piuttosto che ad aggregazioni ("has a"), che sono allocati di solito all'interno dello stack. Il prefisso T è anche usato per le enumerazioni: Symbian mette a disposizione dei *basic types* che appartengono alla categoria delle T classes.

C Classes: le *C classes* derivano dalla classe CBase (definita in *e32base.h*) ed ereditano da essa due importanti caratteristiche. Per prima cosa la classe CBase ha un distruttore virtuale: in questo modo un oggetto di una classe derivata può essere distrutto appropriatamente tramite un puntatore alla classe CBase. Qualora si verifichi una *leave*, l'oggetto sarà cancellato attraverso il puntatore. Il distruttore virtuale di CBase garantisce che vengano chiamati i distruttori delle classi derivate nell'ordine corretto (partendo dalla classe derivata più in basso nella gerarchia).

R Classes: il prefisso R delle *R classes* indica una risorsa, di solito un *handle*. Una classe di tipo R possiede un costruttore che setta a zero l'handle, indicando che l'oggetto della classe non è collegato ad alcuna risorsa all'atto della sua creazione. Si deve evitare di inizializzare l'handle all'interno del costruttore, poiché potrebbe causare una *leave*. Per collegare un oggetto di una classe R ad una risorsa, tipicamente si implementano le funzioni membro *Open()*, *Create()* ed *Initialize()*, che allocano la risorsa, inizializzano l'handle correttamente e gestiscono gli errori in queste fasi. Le R classes sono classi di piccole dimensioni, raramente contengono altre variabili membro al di fuori dell'handle e, rispetto alle T e C classes, prevedono meno vincoli di implementazione.

M Classes: le *M classes* sono classi di tipo astratto. In Symbian solitamente le classi derivate sfruttano l'ereditarietà multipla, con una derivazione principale da un classe base (come CBase) ed ereditando proprietà aggiuntive con una derivazione da una o più M classes. L'ereditarietà da più classi astratte è l'unico tipo di ereditarietà multipla che si dovrebbe utilizzare in Symbian, evitando in questo modo di aumentare la complessità della classi progettate.

3.1.2 Eccezioni

Symbian era stato progettato quando ancora la gestione delle eccezioni non faceva parte dello standard C++. Dopo la sua introduzione, si è verificato che essa determinava un notevole overhead della RAM a run-time, anche nel caso in cui le eccezioni non venissero effettivamente utilizzate. Si rese quindi necessario creare il meccanismo delle *leaves*, adeguato allo stile compatto del sistema operativo ed al codice sviluppato per quest'ultimo, ma meno elegante rispetto allo standard C++. Questo meccanismo è comunque caratterizzato da semplicità d'uso, efficienza e leggerezza, in termini sia di dimensioni del codice che di overhead a tempo di esecuzione. Una *leave* può verificarsi a seguito di una chiamata ad una *leaving function* o ad una funzione di sistema che la genera. Una leave è usata per generare un'eccezione e consentire la propagazione di un valore d'errore lungo la pila delle chiamate a funzione, per poi essere intercettato da una *trap* e gestito appropriatamente.

3.1.3 Active Objects

Symbian è un sistema operativo che fa largo impiego di codice *event-driven*, ovvero basato su eventi, sia per l'interazione con il livello utente, che a livello di sistema per comunicazioni asincrone con periferiche di input e output. In questo ambito è normale fare ricorso ad una programmazione multithreaded e in Symbian i thread sono schedulati in modalità preempitve, in cui il context switch tra essi implica comunque un certo overhead, sebbene minore rispetto a quello tra processi. Gli Active Objects consentono invece di utilizzare un meccanismo di multi-tasking non preemptive nel contesto di un singolo thread, semplificando la programmazione per la gestione asincrona di eventi e evitando quindi il ricorso al multi-threading.

La comunicazione tra Active Object (AO) è più veloce di quella che avviene tra i thread (gli AO sono eseguiti nel contesto di un unico thread e condividono le stesse risorse), grazie allo scheduling non preemptive che non richiede salvataggio dello stato. In questo modo è possibile evitare il ricorso a strutture come i semafori per regolare l'accesso concorrente alla memoria condivisa, che aumentano la complessità del codice e incidono sulla velocità d'esecuzione.

Una tipica applicazione sviluppata in Symbian è costituita da un unico thread per la gestione di eventi, il quale esegue uno scheduler (denominato *Active Scheduler*), che ha il compito di coordinare i vari Active Object. Ognuno di essi effettua una richiesta asincrona ad un server e gestisce in seguito l'esito della richiesta, consentendo la cancellazione di richieste inoltrate e fornendo la gestione di errori.

3.2 Architettura delle comunicazioni

3.2.1 Socket

Il Sockets Client API (ESOCK) è una libreria che fornisce un'interfaccia generica da utilizzare per accedere ai protocolli di comunicazione mediante i socket. Questi ultimi forniscono a loro volta un'interfaccia ai protocolli di rete, utilizzata per le comuni operazioni che riguardano le comunicazioni

via rete, come inviare dati, ricevere dati, stabilire connessioni e configurare protocolli di rete.

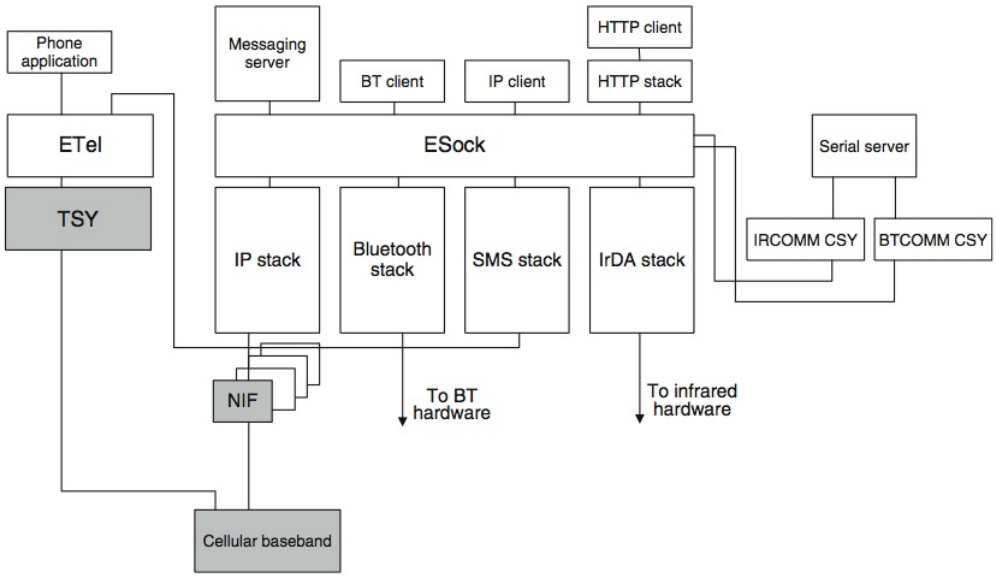


Figura 3.1: *Architettura delle comunicazioni su Symbian*

Le Socket API sono state introdotte inizialmente allo scopo di semplificare l'uso di connessioni TCP/IP nell'ambiente BSD UNIX, e sono poi diventate uno standard per una vasta gamma di piattaforme, come ad esempio UNIX e Windows. Anche per Symbian OS è disponibile una implementazione di queste API, con alcune differenze rispetto alle versioni standard. I concetti principali legati a queste API restano comunque invariati.

In Symbian OS i componenti chiave per la gestione dei socket sono i seguenti:

Socket Server: ha lo scopo di fornire un accesso generico e condiviso ai protocolli.

API client-side: fornisce i metodi di controllo per l'accesso al server.

Plug-ins: chiamati anche *protocol modules* o *PRTs*.

Attraverso l'utilizzo della libreria ESOCK è quindi possibile accedere agli stack relativi al Bluetooth, all'IrDA e al protocollo TCP/IP, come si può notare in figura 3.2.

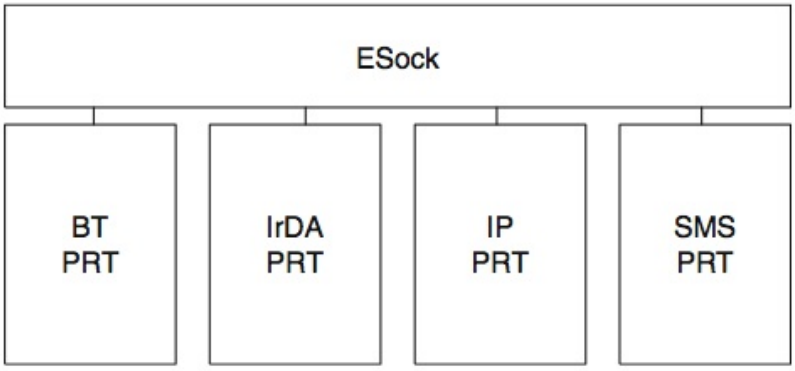


Figura 3.2: *Libreria ESOCK*

Per poter utilizzare queste API è necessario innanzitutto connettersi al Socket Server e stabilire una sessione client/server. La connessione è gestita attraverso la classe *RSocketServer*, i cui metodi principali sono:

```
TInt Connect(TUint aMessageSlots=KESockDefaultMessageSlots);
TInt Connect(const TSessionPref& aPref,
TUint aMessageSlots=KESockDefaultMessageSlots);
TVersion Version() const;
TInt NumProtocols(TUint& aCount);
TInt GetProtocolInfo(TUint anIndex,TProtocolDesc& aProtocol);
TInt FindProtocol(const TProtocolName& aName,
        TProtocolDesc& aProtocol);
void StartProtocol(TUint anAddrFamily,TUint aSockType,
TUint aProtocol,TRequestStatus& aStatus);
void StopProtocol(TUint anAddrFamily,TUint aSockType,
TUint aProtocol,TRequestStatus& aStatus);
```

3.2.2 Connessioni

Un programma può stabilire una connessione, che viene definita una *active connection*, oppure può ricevere una connessione da un peer, in questo caso si parla di *passive connection*.

Active connection: Una connessione di questo tipo viene solitamente stabilita utilizzando il metodo RSocket::Connect():

```
void Connect(TSockAddr& anAddr,TRequestStatus& aStatus);
void Connect(TSockAddr& anAddr,const TDesC8& aConnectDataOut,
        TDes8& aConnectDataIn,TRequestStatus& aStatus);
```

L'indirizzo del device a cui è indirizzata la connessione viene specificato nel parametro *TSockAddr*. Le informazioni necessarie per l'indirizzamento possono variare da protocollo a protocollo, per questo motivo la maggior parte di essi fornisce una classe derivata da *TSockAddr* che contiene le informazioni specifiche richieste dal protocollo stesso. Il metodo *Connect()* può essere usato sia per protocolli *connession-oriented* che per protocolli *connectionless*, e in quest'ultimo caso la chiamata al metodo si limita a impostare l'indirizzo di default verso il quale i pacchetti saranno inviati.

Passive connection: la gestione delle connessioni passive è più complessa, in quanto coinvolge tre diverse chiamate alle API (*Bind*, *Listen* e *Accept*) ed è necessario utilizzare due socket:

```
TInt Bind(TSockAddr& anAddr);
TInt SetLocalPort(TInt aPort);
void Accept(RSocket& aBlankSocket,TRequestStatus& aStatus);
void Accept(RSocket& aBlankSocket,TDes8& aConnectData,
        TRequestStatus& aStatus);
```

```
TInt Listen(TUint qSize);
TInt Listen(TUint qSize,const TDesC8& aConnectData);
```

Il primo socket, definito *listening socket*, è legato (Bind) all'indirizzo in cui si attendono nuove connessioni (Listen). Il metodo *Accept()* richiede invece l'utilizzo di un nuovo socket, sul quale si effettuerà la connessione vera e propria.

3.2.3 Trasmissione

Per effettuare la trasmissione dei dati è possibile utilizzare le seguenti funzioni:

```
void Send(const TDesC8& aDesc,TUint someFlags, TRequestStatus& aStatus);
void Send(const TDesC8& aDesc,TUint someFlags, TRequestStatus& aStatus,
        TSockXfrLength& aLen);
void SendTo(const TDesC8& aDesc, TSockAddr& anAddr, TUint flags,
        TRequestStatus& aStatus);
void SendTo(const TDesC8& aDesc, TSockAddr& anAddr, TUint flags,
        TRequestStatus& aStatus, TSockXfrLength& aLen);
void CancelSend();
```

Il metodo *Send()* può essere usato solo per un socket connesso, sul quale quindi è stata effettuata una *Connect()* senza errori. Il metodo *SendTo()* è invece utilizzato su protocolli connectionless e necessita dell'indirizzo remoto per poter inviare i dati.

3.2.4 Ricezione

La ricezione dei dati coinvolge i seguenti metodi:

```
void Recv(TDes8& aDesc,TUint flags,TRequestStatus& aStatus);
void Recv(TDes8& aDesc,TUint flags,TRequestStatus& aStatus,
        TSockXfrLength& aLen);
void RecvOneOrMore(TDes8& aDesc,TUint flags,
        TRequestStatus& aStatus,TSockXfrLength& aLen);
void Read(TDes8& aDesc,TRequestStatus& aStatus);
void RecvFrom(TDes8& aDesc,TSockAddr& anAddr,TUint flags,
        TRequestStatus& aStatus);
void RecvFrom(TDes8& aDesc,TSockAddr& anAddr,TUint flags,
        TRequestStatus& aStatus,TSockXfrLength& aLen);
void CancelRecv();
void CancelRead();
```

La funzione *Recv()* fornisce la possibilità di ricevere dati da un peer, la cui dimensione massima è specificata del descriptor. *RecvFrom()* differisce da *Recv()* perché è utilizzata su socket connectionless.

3.2.5 Disconnessione

È possibile disconnettere un socket attraverso il metodo *Shutdown()*, che chiude il socket e disconnette qualunque connessione sia attiva su di esso. La medesima operazione viene compiuta anche dal metodo *Close()*, ma in questo caso l'operazione è asincrona. I metodi per effettuare una disconnessione sono i seguenti:

```
void Shutdown(TShutdown aHow,TRequestStatus& aStatus);
void Shutdown(TShutdown aHow,const TDesC8& aDisconnectDataOut,
        TDes8& aDisconnectDataIn,TRequestStatus& aStatus);
```

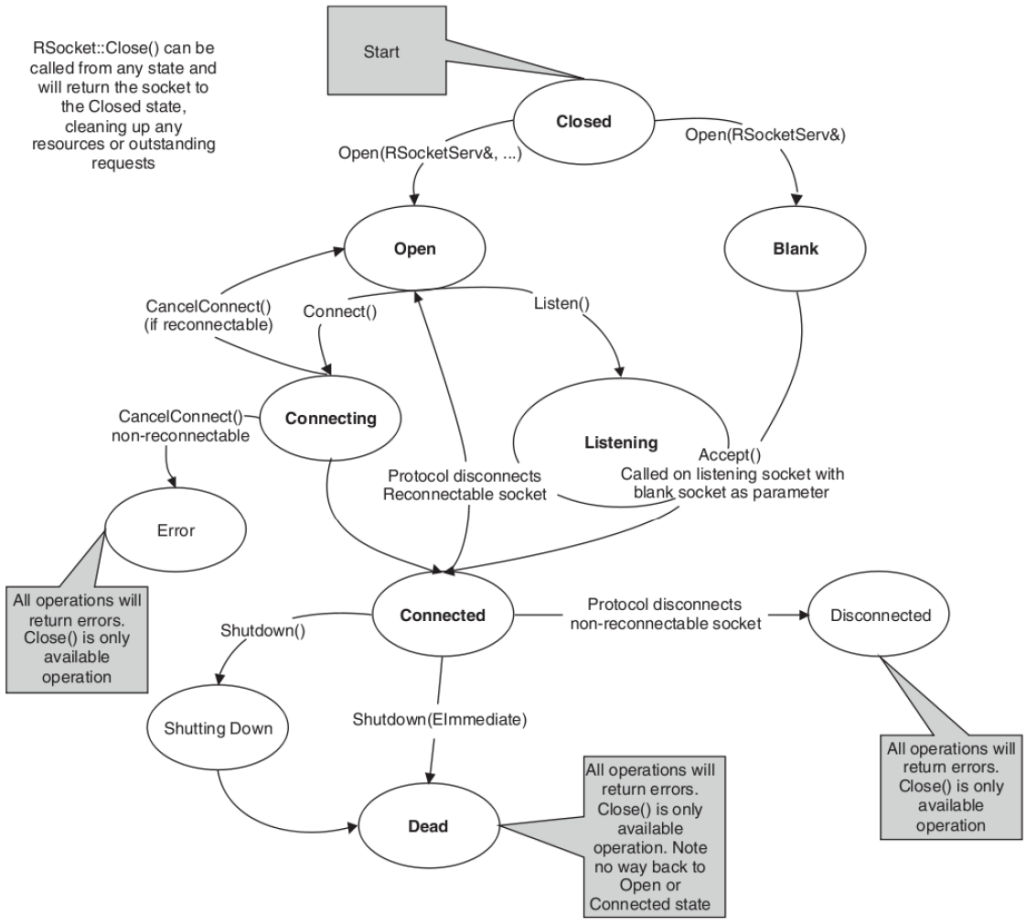


Figura 3.3: *Ciclo di vita di un socket*

3.2.6 Gestione delle connessioni

Le API forniscono inoltre meccanismi per enumerare, iniziare, chiudere e monitorare una connessione:

```
TInt Open(RSocketServ& aSocketServer,
        TUint aConnectionType = KConnectionTypeDefault);
TInt Open(RSocketServ& aSocketServer, TName& aName);
void Close();
void Start(TRequestStatus& aStatus);
void Start(TConnPref& aPref, TRequestStatus& aStatus);
```

```
TInt Start();
TInt Start(TConnPref& aPref);
TInt Stop();
TInt Stop(TConnStopType aStopType);
TInt Stop(TSubConnectionUniqueId aSubConnectionUniqueId);
TInt Stop(TSubConnectionUniqueId aSubConnectionUniqueId,
        TConnStopType aStopType);
```

Una volta stabilito il canale client/server con ESOCK, l'oggetto *RConnection* può essere associato con un *bearer* specifico. Più oggetti RConnection possono essere associati ad un singola connessione e questo consente a più processi di controllare e monitorare la medesima connessione. Per avviare un bearer occorre utilizzare il metodo *Start()* (sincrono) o *Start(TRequestStatus)* (asincrono). La connessione ad un bearer può impiegare molto tempo per essere stabilita. In questo caso è possibile ottenere delle notifiche sui progressi effettuati o monitorare la connessione per osservarne i cambiamenti di stato.

I metodi usati per raccogliere informazioni sulla connessione sono i seguenti:

```
void ProgressNotification(TNifProgressBuf& aProgress,
        TRequestStatus& aStatus,
        TUint aSelectedProgress = KConnProgressDefault);
void ProgressNotification(TSubConnectionUniqueId aSubConnectionUniqueId,
        TNifProgressBuf& aProgress, TRequestStatus& aStatus,
        TUint aSelectedProgress = KConnProgressDefault);
void CancelProgressNotification();
void CancelProgressNotification(TSubConnectionUniqueId
        aSubConnectionUniqueId);
TInt Progress(TNifProgress& aProgress);
TInt Progress(TSubConnectionUniqueId aSubConnectionUniqueId,
        TNifProgress& aProgress);
TInt LastProgressError(TNifProgress& aProgress);
```

I metodi rimanenti della classe RConnection possono essere usati per controllare lo stato di un bearer. Quest'operazione viene eseguita attraverso i metodi:

```
TInt EnumerateConnections(TUint& aCount);
TInt GetConnectionInfo(TUint aIndex, TDes8& aConnectionInfo);
TInt Attach(const TDesC8& aConnectionInfo, TConnAttachType aAttachType);
```

Il metodo *EnumerateConnections()* restituisce il numero delle connessioni attive e successivamente è possibile utilizzare il metodo *GetConnectionInfo()* per ottenere informazioni su una specifica connessione. Non è necessario preoccuparsi del fatto che il numero delle connessioni attive cambi tra la chiamata *EnumerateConnections()* e la chiamata a *GetConnectionInfo()* perché il server ESOCK mantiene in memoria le informazioni relative al momento in cui il metodo è stato chiamato.

Il metodo *Attach()* può essere chiamato per legare un oggetto della classe RConnection ad una connessione attiva. Indipendentemente dal fatto che l'associazione sia avvenuta per mezzo del metodo *Start()* o per mezzo del metodo *Attach()*, è possibile ottenere informazioni sulla connessione associata utilizzando i seguenti metodi:

```
TInt GetIntSetting(const TDesC& aSettingName, TUint32& aValue);
TInt GetBoolSetting(const TDesC& aSettingName, TBool& aValue);
TInt GetDesSetting(const TDesC& aSettingName, TDes8& aValue);
TInt GetDesSetting(const TDesC& aSettingName, TDes16& aValue);
TInt GetLongDesSetting(const TDesC& aSettingName, TDes& aValue);
TInt Name(TName& aName);
```

3.3 Ambiente di sviluppo

3.3.1 Installazione

Per poter sviluppare applicazioni Symbian è necessario disporre di un computer con Microsoft Windows. Nei prossimi paragrafi saranno illustrati i passaggi necessari per la corretta installazione dell'ambiente di sviluppo. Ulteriori informazioni possono essere trovate a questo indirizzo:

http://wiki.forum.nokia.com/index.php/SDK_and_Carbide.c%2B%2B_installation_guide

Perl

ActivePerl è un interprete per Windows di script programmati in linguaggio Perl[3]. In pratica rende disponibile al programmatore il necessario per creare, testare e fare il debug dei programmi scritti con questo linguaggio. È possibile scaricare ActivePerl al seguente indirizzo:

http://www.activestate.com/activeperl/downloads

SDK

Il pacchetto Software Development Kit (SDK), fornito da Nokia, include tutte le risorse necessarie per lo sviluppo di applicazioni su Symbian, come compilatore, librerie di sistema (API), documentazione, esempi e un emulatore. È consigliabile installare il pacchetto all'interno della cartella *C:\Symbian*, per evitare problemi in seguito. Il Software Development Kit dà accesso alle API pubbliche, e lavorando esclusivamente su queste ultime si ha la sicurezza che le applicazioni sviluppate funzioneranno sull'ampio ventaglio di device disponibili attualmente e in futuro. Ad oggi Symbian Foundation non ha ancora rilasciato la versione open source del kit, ed è quindi necessario scaricare il software presso il sito di Nokia, al seguente indirizzo:

http://www.forum.nokia.com/info/sw.nokia.com/id/ec866fab-4b76-49f6-b5a5-af0631419e9c/S60_All_in_One_SDKs.html

[3] http://it.wikipedia.org/wiki/Perl

ADT

L'Application Development Toolkit (ADT) contiene gli strumenti necessari allo sviluppo delle applicazioni. All'interno del toolkit è presente un Integrated Development Environment (IDE) denominato Carbide.c++, basato su Eclipse[4], che costituisce il principale strumento per lo sviluppo C++ su questa piattaforma. I plugin di Carbide includono funzionalità di building, debugging, analisi statiche, analisi dinamiche e una varietà di strumenti ed utility specializzate. Carbide.c++ permette quindi allo sviluppatore di editare, costruire ed effettuare debugging di un'applicazione su emulatore o direttamente su smartphone. È possibile scaricare ADT al seguente indirizzo:

https://developer-secure.symbian.org/main/tools_and_kits/downloads/view.php?id=2

Nokia PC Suite

Nokia PC Suite consente di acquisire il controllo completo del dispositivo Nokia attraverso il PC. Si tratta di una raccolta di programmi tramite i quali è possibile collegare il proprio PC con il telefono cellulare. Utilizzando Nokia PC Suite è anche possibile installare applicazioni direttamente sul cellulare. È possibile scaricare il software al seguente indirizzo:

http://www.nokia.it/supporto/software/nokia-pc-suite

3.3.2 Configurazione

Per lo sviluppo di un'applicazione VoIP su protocollo SIP si è deciso di basarsi sullo stack open source PJSIP. Si tratta di un set di librerie complete, che forniscono:

ottima portabilità: le applicazioni scritte possono essere eseguite su altre piattaforme, come Microsoft Windows, Microsoft Windows Mobile, Linux, Unix, Mac OS X, Symbian OS, iPhone, ecc.

[4] http://it.wikipedia.org/wiki/Eclipse_(informatica)

basso consumo di memoria: meno di 150KB occupati per fornire il set completo di funzionalità SIP.

performance: maggiori ottimizzazioni offrono un minore consumo di tempo di calcolo e batteria.

ampia documentazione: per ogni struttura e funzione dello stack PJSIP è disponibile una chiara descrizione nella documentazione online[5].

PJSIP

I sorgenti di PJSIP possono essere scaricati al seguente indirizzo:

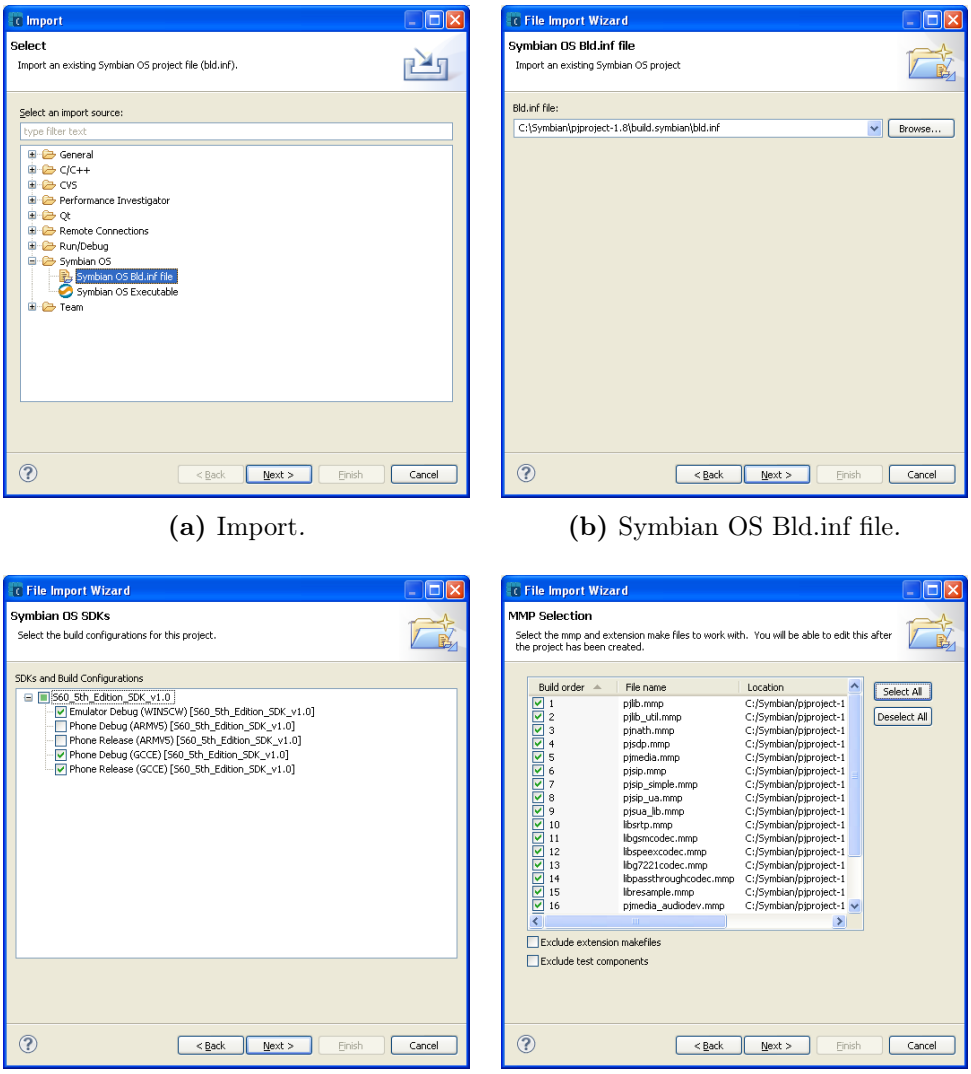
http://www.pjsip.org/download.htm

Una volta decompresso il pacchetto, è possibile importarlo in Carbide.c++, attraverso i seguenti passaggi:

1. una volta eseguito Carbide.c++, selezionare *File* e poi *Import*;
2. all'interno della finestra di dialogo, selezionare dalla lista *Symbian OS*, e successivamente *Symbian OS Bld.inf file*; infine cliccare sul bottone *Next*;
3. cliccare quindi su *Browse* e scegliere il file *bld.inf* contenuto all'interno della cartella *build.symbian*;
4. a questo punto selezionare le configurazioni WINSCW (emulatore) e GCCE (telefono) e cliccare su *Next*;
5. all'interno della finestra di dialogo, cliccare su *Select All* ed infine su *Next*;
6. il progetto è quindi pronto per essere importato, premerendo il bottone *Finish*. A questo punto Carbide exporterà tutti i file MMP all'interno del nuovo workspace. Attendere quindi che il processo sia terminato;

[5] http://www.forum.nokia.com/Develop/Other_Technologies/Symbian_C++/Documentation/

7. il progetto PJSIP è quindi disponibile all'interno della finestra *Project Explorer*.



(a) Import. **(b)** Symbian OS Bld.inf file.

(c) Symbian OS SDKs. **(d)** MMP Selection.

Figura 3.4: *Import del progetto PJSIP all'interno di Carbide.c++*

Per poter effettuare il building del progetto è necessario selezionare all'interno del menu principale *Project*, *Build Configuration*, *Set Active* e quindi *Phone Debug (GCCE) [S60_5th_Edition_SDK_v1.0]* (figura 3.5). È quindi possibile eseguire il building, facendo click con il tasto destro sul progetto, e

poi click su *Build Project*. Il processo può richiedere diversi minuti, al termine dei quali occorre configurare la compilazione del file SIS, che sarà poi installato successivamente sul telefono.

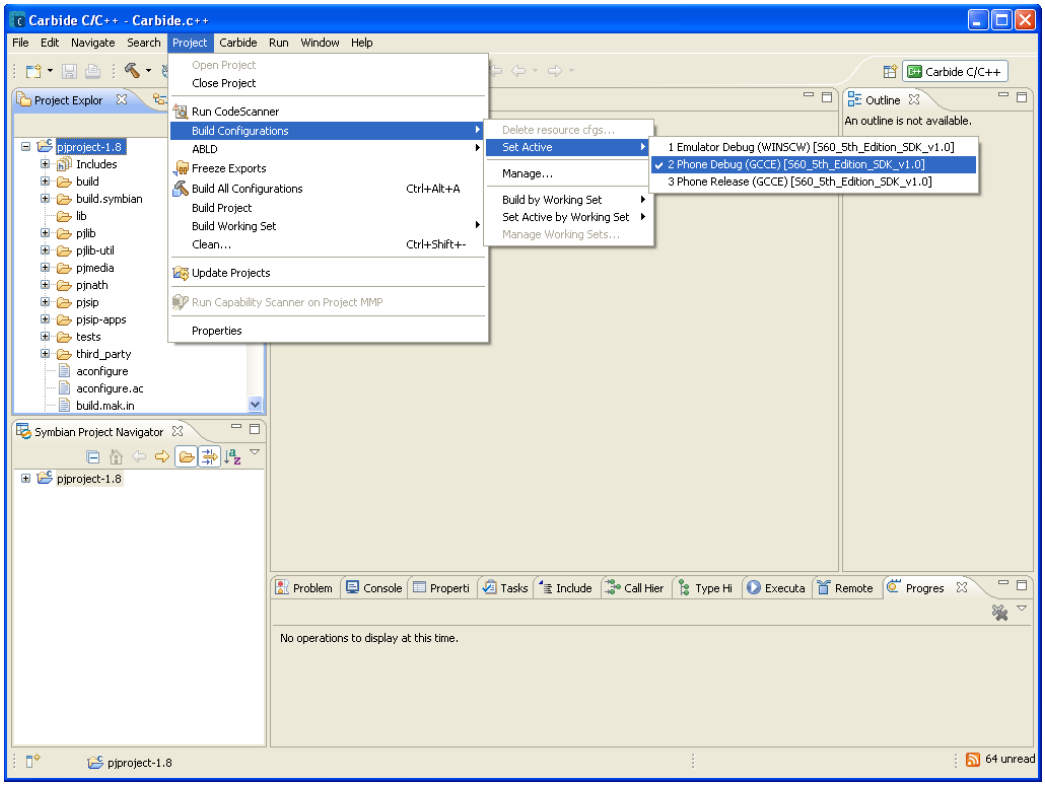


Figura 3.5: *Build Configuration su Carbide.c++*

Il formato SIS è una particolare tipologia di file compresso usato sui telefoni cellulari Symbian per consentire l'installazione di applicazioni. Un file SIS può essere trasferito sul dispositivo mobile via OTA (over the air), BlueTooth ed infine tramite USB usando l'applicazione Nokia PC Suite. Alla fine della procedura di installazione, nella griglia delle applicazioni apparirà una nuova icona associata al software appena installato. Con l'uscita della terza edizione del sistema operativo Symbian è stata introdotta la certificazione dei programmi. I file SIS devono quindi essere certificati prima di poter essere installati sul telefono. Un programma certificato possiede estensione SISX. Attraverso Carbide è possibile compilare e certificare un programma:

1. cliccare con il tasto destro del mouse sul progetto, ed infine su *Properties*;
2. selezionare *Carbide.c++*, ed poi cliccare su *Build Configurations*;
3. controllare che sia selezionato come configurazione attiva *Phone Debug (GCCE) [S60_5th_Edition_SDK_v1.0]*;
4. cliccare su *Add* e selezionare il file *symbian_ua.pkg* contenuto nella cartella *build_symbian*;
5. all'interno della finestra *SIS Properties* cliccare su OK per utilizzare le configurazioni di default;
6. a questo punto cliccare con il tasto destro del mouse sul progetto, e poi su *Refresh*;
7. attendere il completamento del processo;
8. cliccare con il tasto destro sul progetto, ed infine su *Build Project* per compilare.

Il processo di compilazione partirà, ed è possibile controllarne lo stato all'interno del tab *Console*. Una volta terminata la compilazione, sarà possibile osservare all'interno della *Console* una scritta di questo tipo:

```
Signing

***SIS Creation Complete

Total Time: 12 sec
```

Eventuali errori di compilazione possono essere visualizzati all'interno del tab *Problems*. A questo punto è possibile installare il programma sul telefono:

1. localizzare il file *symbian_ua.sisx*, contenuto all'interno della cartella *build_symbian*;

2. aprire il file con un doppio click del mouse;

3. seguire le istruzioni di installazione proposte dal programma Nokia PC Suite.

Ulteriori informazioni sulla configurazione di PJSIP all'interno di Carbide.c++ sono presenti all'indirizzo:

http://trac.pjsip.org/repos/wiki/Getting-Started/Symbian

TRK

Carbide permette anche di effettuare il debug direttamente sul device. Questa tecnica, denominata *On-Device Debugging*, risulta fondamentale nello sviluppo, poiché l'utilizzo dell'emulatore pone grossi limiti per quanto riguarda la gestione della rete e dell'audio, come specificato nel paragrafo 3.5.1.

Seguendo le istruzioni riportate nel paragrafo precedente, al termine di una compilazione si ottiene un programma auto-certificato. Di norma Symbian non accetta l'installazione di programmi di questo tipo, per cui risulta necessario effettuare una particolare configurazione sul telefono[6]:

1. sul telefono, aprire il menu;

2. entrare nel *Pannello di Controllo*, ed infine su *Gestione Applicazioni*;

3. selezionare *Opzioni*, e poi *Impostazioni*;

4. all'interno di *Installazione software*, selezionare *Completa*.

Dopo aver completato la configurazione, è possibile installare il software *TRK* sul telefono, attraverso le seguenti istruzioni:

1. collegare il telefono al computer utilizzando un cavetto USB;

2. all'interno di Carbide, selezionare *Help* e successivamente *On-Device Connections*;

[6]Le istruzioni si riferiscono ad un Nokia E52.

3. nella finestra di dialogo, selezionare USB come *Connection Type* e cliccare quindi su *Next*;

4. selezionare il tab *Install remote agents* e quindi selezionare l'installer appropriato secondo il seguente schema[7]:

Versione Symbian	**Installer**
S60 3rd Plain/MR	3.0.0
S60 3rd FP1	3.1.0
S60 3rd FP2	3.2.0
S60 5th	5.0.0

Tabella 3.1: *TRK Application Installer*

5. cliccare su *Install* e seguire le istruzioni di installazione fornite da Nokia PC Suite.

Al termine dell'installazione, è possibile eseguire l'applicazione TRK[8]:

1. sul telefono, aprire il menu e selezionare *Applicazioni*;

2. all'interno della lista visualizzata, selezionare *TRK*;

3. apparirà una finestra di dialogo, all'interno della quale viene chiesto di collegarsi via Bluetooth. Selezionare *NO*, in quanto verrà utilizzato un collegamento USB;

4. selezionare *Opzioni* ed infine *Settings*;

5. cambiare il tipo di connessione da Bluetooth a USB e premere *Indietro*;

6. selezionare *Opzioni*, ed infine *Connect*;

7. a questo punto dovrebbe essere visibile lo stato *Connected* all'interno del programma.

[7] Per il telefono Nokia E52 selezionare *Application TRK 3.1.0*.
[8] Le istruzioni si riferiscono ad un Nokia E52.

Ritornare quindi a Carbide e cliccare su *Set Connection Settings*. Selezionare la porta seriale collegata al device e la versione dell'installer utilizzata in precedenza. Per testare la connessione, selezionare il servizio TRK e cliccare su *Initiate service testing*. Se il servizio è configurato in modo corretto, cliccare su *Finish*. A questo punto all'interno del tab *Remote connections* è possibile verificare il corretto funzionamento del servizio TRK. È quindi possibile effettuare il debug *On-Device*.

RDebug

RDebug è un set di funzioni che permettono allo sviluppatore di generare messaggi di output all'interno della *Console* di Carbide. Per utilizzare RDebug è necessario includere all'interno dei file l'header *e32debug.h*. All'interno del codice è quindi possibile inserire:

```
#include <e32debug.h>

// code before log
// Print a HBufC
RDebug::Print( _L("Test string: %S"), hbuf );
// code after log
```

Ulteriori informazioni e configurazioni sono reperibili all'indirizzo:

http://wiki.forum.nokia.com/index.php/How_to_use_RDebug

3.4 PJSIP

Le librerie PJSIP sono costituite da molteplici livelli di API, attraverso i quali viene ridotta la complessità implementativa del sistema.

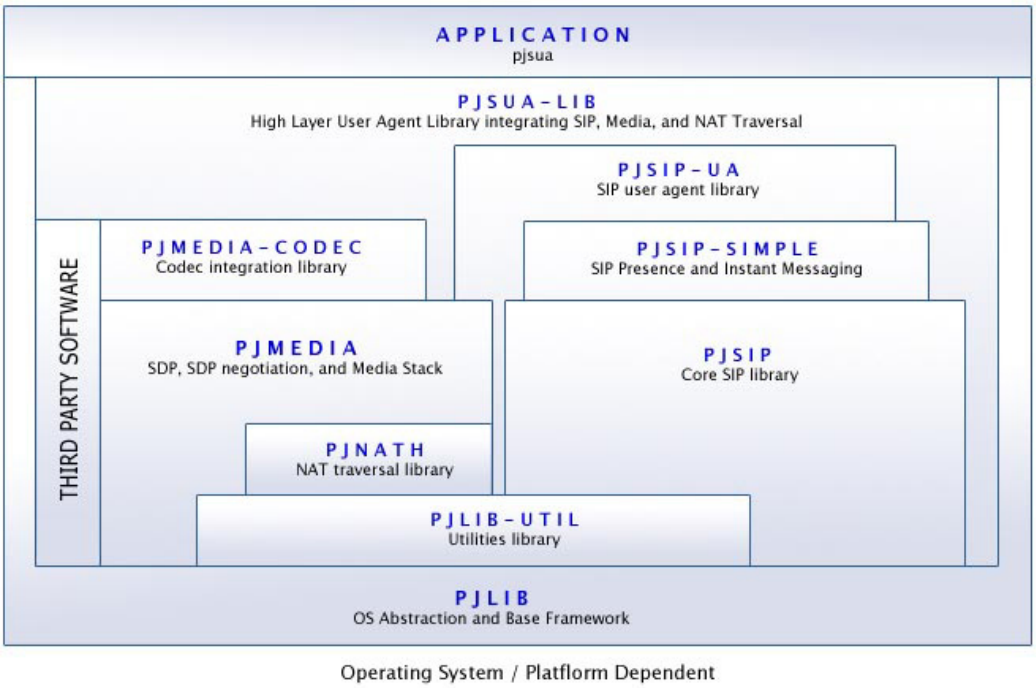


Figura 3.6: *Architettura PJSIP*

Come si può osservare dalla figura 3.6, lo strato di livello più alto è costituito dalla libreria *PJSUA-LIB*, che integra al suo interno le librerie *PJSIP*, *PJMEDIA* e *PJNATH*, formando una serie di API di alto livello. La libreria *PJSIP* è anch'essa costituita da altre librerie, come *PJSIP base library* (pjsip core, transaction layer e dialog management), *SIP user agent library* (pjsip-ua, che include invite session management, client registration, call features, ecc.), e *SIP presence and messaging library* (pjsip-simple).

Il media stack include *PJMEDIA* e *PJMEDIA-CODEC*, i quali contengono i componenti SDP (Session Description Protocol) come message representation, parsing, e SDP offer answer negotiation. Nel caso di Symbian le componenti SDP sono spostate in una libreria a parte, denominata *PJSDP library*, e quindi non sono presenti all'interno di *PJMEDIA*.

Lo strato inferiore di PJSIP è costituito dalle librerie più generali che forniscono strutture e funzioni di comune utilità: *PJLIB* e *PJLIB-UTIL*. La prima, di livello più basso, contiene strutture semplici come stringhe, liste linkate, alberi bilanciati, semafori, funzioni per l'allocazione di memoria e per l'utilizzo delle strutture precedentemente elencate. La seconda contiene invece strutture e funzioni più avanzate, utili per la crittografia (algoritmi SHA1, MD5, HMAC e CRC32), per il parsing e la manipolazione di testo e stringhe semplici e di testi in formato XML.

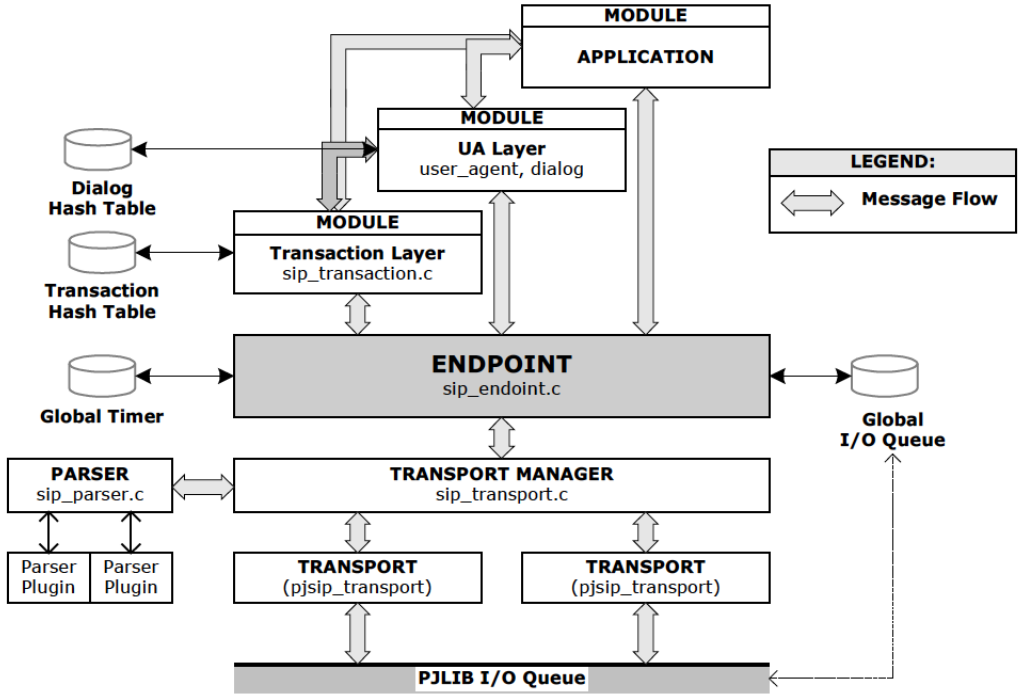


Figura 3.7: *Diagramma delle collaborazioni di PJSIP*

3.4.1 Gestione delle riconnessioni

Per un device mobile la gestione di un'eventuale disconnessione, e conseguente riconnessione, costituisce un problema di primaria importanza. Nel caso in cui il terminale si sia collegato ad un nuovo Access Point, è molto probabile che abbia ricevuto un nuovo indirizzo IP.

In PJSIP il monitoraggio della caduta di una connessione è affidato all'applicazione. In particolare, nella versione Symbian è utilizzato un *connection progress monitor*. Nel caso in cui l'applicazione rilevi un cambio di indirizzo IP, PJSIP propone due soluzioni al problema: riavviare l'intera applicazione o aggiornare singolarmente le strutture fondamentali.

Riavviare PJSIP

La soluzione più semplice è quella di riavviare l'intera applicazione. All'interno di *pjsua* sarà quindi chiamata la funzione *pjsua_ destroy()*. Questo drastico metodo, facile da implementare, è attualmente utilizzato in PJSIP nel caso venga rilevato un cambio di indirizzo IP.

Aggiornare le strutture

Questo approccio richiede alcuni specifici accorgimenti e configurazioni. Quando viene rilevato un cambio di indirizzo IP sarà necessario:

aggiornare il proprio indirizzo SIP: la libreria *PJSUA-LIB* possiede la capacità di rilevare un cambio di indirizzo IP controllando la risposta ricevuta dopo l'invio di un messaggio di REGISTER. In questo modo è possibile aggiornare automaticamente la registrazione con l'indirizzo IP rilevato dal server. Questa funzionalità è abilitata di default in PJSIP attraverso la variabile *pjsua_ acc_ config.allow_ contact_ rewrite*. La soluzione può quindi essere quella di inoltrare una nuova registrazione chiamando la funzione *pjsua_ acc_ set_ registration()* una volta instaurata una nuova connessione;

aggiornare gli indirizzi RTP/RTCP: se STUN è abilitato, allora gli indirizzi dei media transport saranno automaticamente abilitati sull'indirizzo pubblico mappato da STUN. In caso contrario, si potrebbero ricreare i media transport e fornirli quindi alla libreria *PJSUA-LIB* tramite la funzione *pjsua_ media_ transports_ attach()*;

informare immediatamente il nodo corrispondente: nel caso sia in atto una comunicazione, non esistono attualmente soluzioni al problema.

In PJSIP è presente una funzione *pjsip_inv_reinvite()* che permetterebbe di cambiare il *Contact URI*, ma questa funzione non è utilizzabile all'interno di *PJSUA-LIB* (*pjsua_call_reinvite()* non permete all'applicazione di cambiare il *Contact URI*).

3.4.2 Soluzioni su Symbian

In Symbian le connessioni sono gestite all'interno di oggetti denominati *RConnection*. Ogni socket è invece creato all'interno di una RConnection tramite l'utilizzo di un oggetto *RSocket*.

Il metodo *RConnection.ProgressNotification()* può essere usato per registrare un *Active Object*, che sarà eseguito quando cambierà lo stato della connessione, così da poter gestire in modo appropriato l'evento.

Quando una connessione di tipo RConnection cade, il socket creato verrà slegato dalla connessione e non sarà più possibile riutilizzarlo, anche nel caso in cui venga ripristinata la connessione. Se l'applicazione tenta di utilizzare il socket, tramite ad esempio la funzione *SendTo()*, sul telefono sarà visualizzata una finestra di dialogo, all'interno della quale occorrerà selezionare un nuova connessione alla rete. Nel caso in cui l'utente selezioni una differente connessione, in caso di una nuova *SendTo()*, lo stato definito dalla variabile *TRequestStatus* associato all'operazione bloccherà l'invio per un lungo periodo di tempo (circa un minuto e mezzo), e in caso di un ulteriore invio, *TRequestStatus* potrebbe bloccare definitivamente l'operazione.

Per questo motivo in PJSIP è stata implementata una funzione, denominata *pj_symbianos_set_connection_status()*, che impedisce all'applicazione di utilizzare un qualsiasi socket se la connessione è stata impostata su *down*.

Il seguente codice sorgente, presente in PJSIP all'interno del file *symbian_ua\ua.cpp*, implementa le funzionalità appena descritte:

```
class CConnMon : public CActive {
public:
    static CConnMon* NewL(RConnection &conn, RSocketServ &sserver) {
        CConnMon *self = new (ELeave) CConnMon(conn, sserver);
        CleanupStack::PushL(self);
```

```
            self->ConstructL();
            CleanupStack::Pop(self);
            return self;
        }

        void Start() {
            conn_.ProgressNotification(nif_progress_, iStatus);
            SetActive();
        }

        void Stop() {
            Cancel();
        }

        ~CConnMon() { Stop(); }

    private:
        CConnMon(RConnection &conn, RSocketServ &sserver) :
            CActive(EPriorityHigh),
            conn_(conn),
            sserver_(sserver)
        {
            CActiveScheduler::Add(this);
        }

        void ConstructL() {}

        void DoCancel() {
            conn_.CancelProgressNotification();
        }

        void RunL() {
            if (nif_progress_().iStage == KLinkLayerClosed) {
                pj_status_t status;
                TInt err;
```

```
// Tell pjlib the connection has been down.
pj_symbianos_set_connection_status(PJ_FALSE);

PJ_LOG(3, (THIS_FILE, "RConnection closed, restarting PJSUA.."));

// Destroy pjsua
pjsua_destroy();
PJ_LOG(3, (THIS_FILE, "PJSUA destroyed."));

// Reopen the connection
err = conn_.Open(sserver_);
if (err == KErrNone)
    err = conn_.Start();
if (err != KErrNone) {
    CActiveScheduler::Stop();
    return;
}

// Reinit Symbian OS param before pj_init()
pj_symbianos_params sym_params;
pj_bzero(&sym_params, sizeof(sym_params));
sym_params.rsocketserv = &sserver_;
sym_params.rconnection = &conn_;
pj_symbianos_set_params(&sym_params);

// Reinit pjsua
status = app_startup();
if (status != PJ_SUCCESS) {
    pjsua_perror(THIS_FILE, "app_startup() error", status);
    CActiveScheduler::Stop();
    return;
}


PJ_LOG(3, (THIS_FILE, "PJSUA restarted."));
PrintMenu();
```

```
        }

        Start();
    }

private:
    RConnection& conn_;
    RSocketServ& sserver_;
    TNifProgressBuf nif_progress_;
};
```

La funzione *RunL()* viene richiamata quando cade la connessione. Il suo compito è quello di chiudere *PJSUA*, aprire una nuova connessione, reinizializzare tutte le variabili di sistema e riavviare quindi l'intera applicazione.

Ulteriori informazioni sono reperibili all'indirizzo:

http://trac.pjsip.org/repos/wiki/Symbian_AP_Reconnection

3.5 Problemi riscontrati

3.5.1 Emulatore

Utilizzare l'emulatore del device mobile è utile per lo sviluppo e il debugging iniziale delle applicazioni per Symbian. È quindi possibile iniziare a sviluppare software prima di avere a disposizione l'hardware reale. Effettuando debugging è possibile aggiungere *breakpoint* e *step through*, oppure osservare il contenuto delle variabili, come accade normalmente durante lo sviluppo di un programma.

Nonostante sia uno strumento utile e molto funzionale per lo sviluppo, l'utilizzo dell'emulatore fa emergere comunque grossi limiti e differenze con il dispositivo reale. Ad esempio, l'emulatore non mette a disposizione il quadro generale delle periferiche di cui il cellulare è provvisto, come fotocamera, vibrazione e soprattutto la gestione delle reti. Quest'ultima, molto limitata, rende disponibile una sola interfaccia di rete, vista come una LAN, impedendo

quindi ogni tipo di sviluppo in ambito multihoming. In generale, comunque, un emulatore non rappresenta un device reale: alcune caratteristiche come la temporizzazione, le performance reali delle applicazioni e la gestione della memoria non sono rappresentate alla perfezione, e questo può creare diversi problemi durante lo sviluppo.

3.5.2 Fase di debugging

TRKProtocolPlugin: Failed to launch the application

Attraverso Carbide è possibile compilare ed installare automaticamente l'applicazione in via di sviluppo. Questa funzionalità è fornita mediante l'utilizzo di TRK, come specificato nel paragrafo 3.3.2. Durante lo sviluppo può capitare che non vada a buon fine l'avvio dell'applicazione per effettuare il debug. In questi casi il problema è stato risolto effettuando un *clean* ed un *rebuild* dell'intero progetto, e successivamente installando manualmente il programma mediante l'utilizzo di Nokia PC Suite. Seguendo questa procedura, le successive installazioni non hanno generato ulteriori errori.

TRKProtocolPlugin: Unable to install the application

Nelle prime fasi di configurazione dell'ambiente di sviluppo è capitato di incontrare un problema relativo alla impossibilità di installare l'applicazione sul telefono. Questa difficoltà è causata da un problema relativo ai certificati e all'impostazione dell'orologio nel computer e nel telefono. Se infatti i due orologi non sono perfettamente impostati, quando si tenta di installare un'applicazione con un certificato generato (apparentemente) qualche minuto dopo, si incorrerà in un errore. Occorre quindi che il telefono sia perfettamente sincronizzato o impostato (per sicurezza) qualche minuto più avanti rispetto al computer.

Mancata installazione

Non è stato a volte possibile installare l'applicazione sul telefono a causa di un errore sconosciuto. In questi casi il problema si è risolto riavviando semplicemente il telefono.

3.5.3 Stack size

Durante lo svilppo ed il debugging può capitare di ottenere un *crash* dell'applicazione dovuto ad un errore di tipo *KERN-EXEC 3*. Questo significa che la dimensione dello stack impostata di default è insufficiente per gestire l'esecuzione dell'applicazione.

Con l'introduzione di Symbian OS v9, la dimensione di default dello stack è stata ridotta da 20KB a 8KB[9], in modo da ottimizzare il consumo di memoria causato dal sempre più elevato numero di processi eseguiti dal sistema. In realtà uno *stack size* di 8KB è spesso insufficiente a gestire applicazioni complesse, come nel caso di PJSIP. Generalmente l'applicazione è in grado di funzionare correttamente nelle prime fasi di avvio, ma il rischio di ottenere un crash di sistema è molto alto. Per questo motivo, aumentare la dimensione dello Stack a 20KB è raccomandato per tutte le applicazioni scritte per la piattaforma *S60 3rd Edition*. Per effettuare questa modifica è necessario modificare il parametro *EPOCSTACKSIZE* contenuto all'interno del file *symbian_ua.mmp*.

3.5.4 Thread

CleanupStack

Il sistema operativo Symbian gestisce le eccezioni in modo molto differente da quanto accade con lo standard C++. Symbian offre infatti alcuni meccanismi di *cleanup*, come l'utilizzo delle funzioni di *TRAP* e *TRAPD* assieme ad un *CleanupStack*, oltre ad alcune convenzioni di programmazione come *leaving methods*. In particolare, fino alla versione Symbian 9.1, il linguaggio non supportava l'utilizzo dei classici meccanismi *try and catch* proposti dal linguaggio C++. A partire dalla versione 9.1, invece, è stato aggiunto il supporto a queste funzioni, anche se, comunque, viene raccomandato da Symbian l'utilizzo delle proprie funzioni.

[9] http://wiki.forum.nokia.com/index.php/KIS000387_-_Small_default_stack_size_in_S60_3rd_Edition

In Symbian ogni thread creato deve possedere il proprio CleanupStack, oltre alla definizione delle funzioni TRAP/TRAPD che gestiscono le eccezioni e il cleanup.

```
void DoTheThingsForMeL(CConsoleBase* aConsole)
{
        //Do something here.
}
void DoExampleL()
{
        CConsoleBase* console;
        // Make the console and push it on the cleanup stack.
        console = Console::NewL(_L("Console"),
                TSize( KConsFullScreen, KConsFullScreen));
        CleanupStack::PushL(console);
        DoTheThingsForMeL(console);
        CleanupStack::PopAndDestroy(console);
}

TInt E32Main()
{
        __UHEAP_MARK;
        //Create a cleanup stack
        CTrapCleanup* cleanup = CTrapCleanup::New();
        //Call some Leaving methods inside TRAP
        TRAPD(error, DoExampleL());
        __ASSERT_ALWAYS(!error, User::Panic(KAPConsoleTest, error));
        //Destroy cleanup stack
        delete cleanup;
        __UHEAP_MARKEND;
        return 0;
}
```

TLS

In Symbian, ogni thread creato possiede un proprio stack size di 8KB di dimensione. È possibile specificare un valore diverso di stack durante la creazione del thread, e questo valore non può essere incrementato una volta avviato il thread. Nel caso in cui la memoria dello stack venga occupata interamente, il sistema andrà in *PANIC*. Un thread possiede un'ulteriore memoria denominata *heap*, che può essere propria o condivisa con quella del thread generatore. Normalmente un thread può avere da un minimo di 4KB ad un massimo di 1MB di memoria heap. Anche in questo caso, è possibile configurare l'apposito file MMP attraverso l'uso della keyword *epocheapsize*, oppure specificarne manualmente il valore durante la creazione del thread. A differenza della memoria stack, la dimensione di un heap può incrementare se necessario. Nel caso in cui il sistema non possieda abbastanza memoria libera, sarà generato un errore di tipo *-memory*.

Le variabili globali e i dati statici sono normalmente caricati all'interno della memoria heap una volta creato un nuovo thread. Per questo motivo è necessario utilizzare una funzionalità denominata Thread Local Storage (TLS)[10], che permette di accedere e modificare in modo corretto le variabili globali. Trattandosi di una allocazione in memoria di risorse, è necessario specificare anche una funzione che si occuperà di liberare la memoria all'uscita dal thread.

```
void InitLibrary()
{
    //allocate the structure containing global data on heap
    GlobalData* p = new GlobalData();
    if ( p ) {
        Dll::SetTls( p ); //Set this Dll's Tls with this structure pointer
    }
    else
        //Panic the thread
}
```

[10] http://wiki.forum.nokia.com/index.php/How_to_use_Thread_Local_Storage

```
void CleanupLibrary()
{
    //Get the Dll's Tls
    GlobalData* p = (GlobalData*) Dll::Tls();
    if ( p )
    {
        //Do some global variable specific cleanup activity if required
        Cleanup( p );
        delete p;  //Deallocate the memory now
     }
}
```

Capitolo 4

Multihoming

4.1 Obiettivi

La realizzazione di questo lavoro di tesi prevede di modificare un client VoIP basato su PJSIP, introducendo la gestione del multihoming all'interno di tale client, concentrandosi principalmente nello sviluppo della componente delle comunicazioni e realizzando la gestione delle connessioni e l'invio di dati attraverso le varie interfacce a disposizione dello smartphone.

Il client VoIP è stato strutturato in due modalità alternative per verificare quale di queste fosse la più adeguata in termini di performance, riusabilità del software ed efficienza energetica. La prima modalità prevede di inserire la gestione delle connessioni all'interno della libreria PJSIP, mentre la seconda modalità prevede di gestire le connessioni in un proxy UDP esterno al client VoIP, riuscendo così a minimizzare gli interventi sulla libreria PJSIP. Gli strumenti per il monitoraggio e la gestione delle connessioni implementate in questa tesi sono stati utilizzati in entrambe le modalità.

In questo elaborato non è stata inclusa la parte di sistema che si occupa della gestione della sicurezza, in quanto sarà realizzata all'interno di un lavoro di tesi parallelo. L'assenza della gestione della sicurezza genera però diversi problemi. Oltre alle problematiche classiche relative alla crittografia, la presenza del contesto di sicurezza all'interno delle comunicazioni risolve un problema tutt'altro che secondario: l'identificazione del mittente di un messaggio.

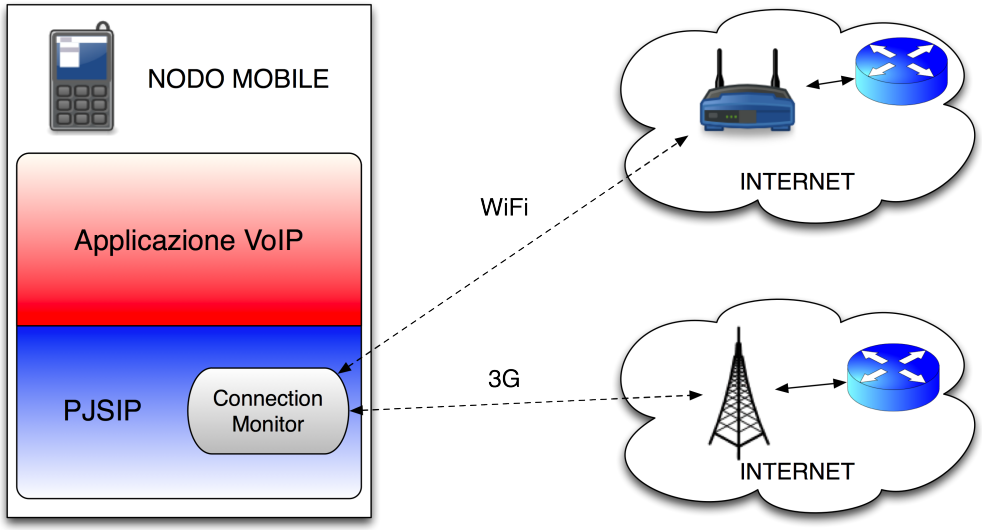


Figura 4.1: *Architettura modalità integrata*

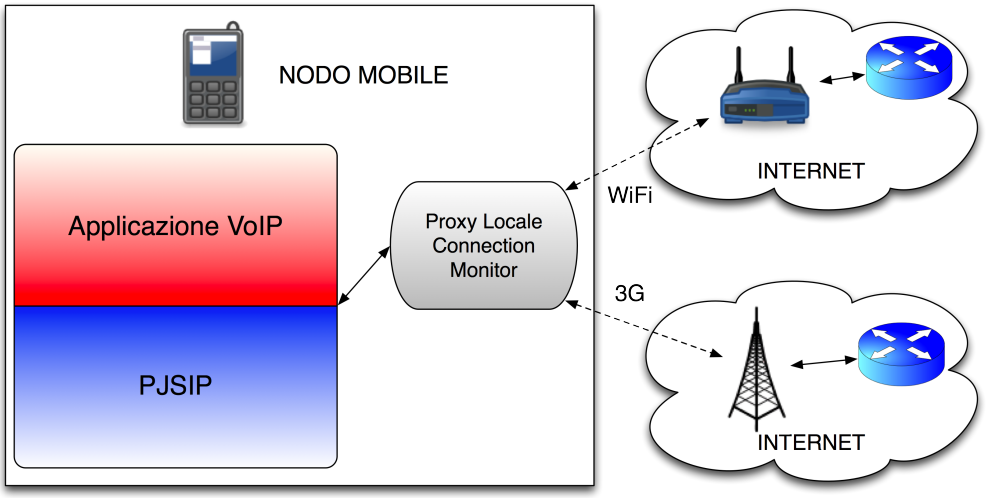


Figura 4.2: *Architettura modalità proxy locale*

Il Fixed Server presente in ABPS necessiterà quindi di operare senza le informazioni relative alla sicurezza, comportandosi come un Proxy Server SIP standard. Un proxy di questo tipo riconosce i client connessi in base all'indirizzo IP di provenienza dei messaggi.

Il ruolo del Proxy Server è di fondamentale importanza perché l'indirizzo del client mobile potrà variare da un momento all'altro e senza preavviso. Il server proxy dovrà quindi incaricarsi di tenere costantemente traccia dell'indirizzo del nodo mobile, in modo da inoltrare nella corretta direzione ogni pacchetto che riceve. Questa funzione è del tutto trasparente alle operazioni del nodo corrispondente, che continuerà in ogni caso a spedire i propri pacchetti al proxy server, ignorando quindi ogni eventuale cambio di indirizzo IP da parte del client mobile.

4.2 Implementazione

4.2.1 Routing

Un problema comune che si riscontra operando con connessioni multiple è l'ambiguità nel routing dei pacchetti. In altre parole, non è immediato stabilire l'interfaccia corretta sulla quale inviare un determinato pacchetto, o quale interfaccia IP possa essere utilizzata come indirizzo di base per le nuove connessioni. Una semplice soluzione consiste nello stabilire un'interfaccia di default, ma è possibile che alcuni servizi non siano ovunque accessibili. Ad esempio, è possibile inviare o ricevere un MMS solamente utilizzando la rete dell'operatore telefonico.

Symbian OS risolve i problemi legati al multihoming consentendo alle applicazioni di dichiarare quale interfaccia intendono usare. Attraverso le API legate alle connessioni di tipo RConnection è possibile associare i socket ad una determinata connessione. Per questo motivo, un'applicazione può inviare un pacchetto su un determinato socket senza sapere quale interfaccia stia realmente utilizzando. In realtà tecnologie differenti forniscono caratteristiche diverse nell'ambito delle performance, mentre un'applicazione che funziona perfettamente su rete WLAN potrebbe avere dei problemi nel caso in cui venga utilizzata attraverso una rete GPRS.

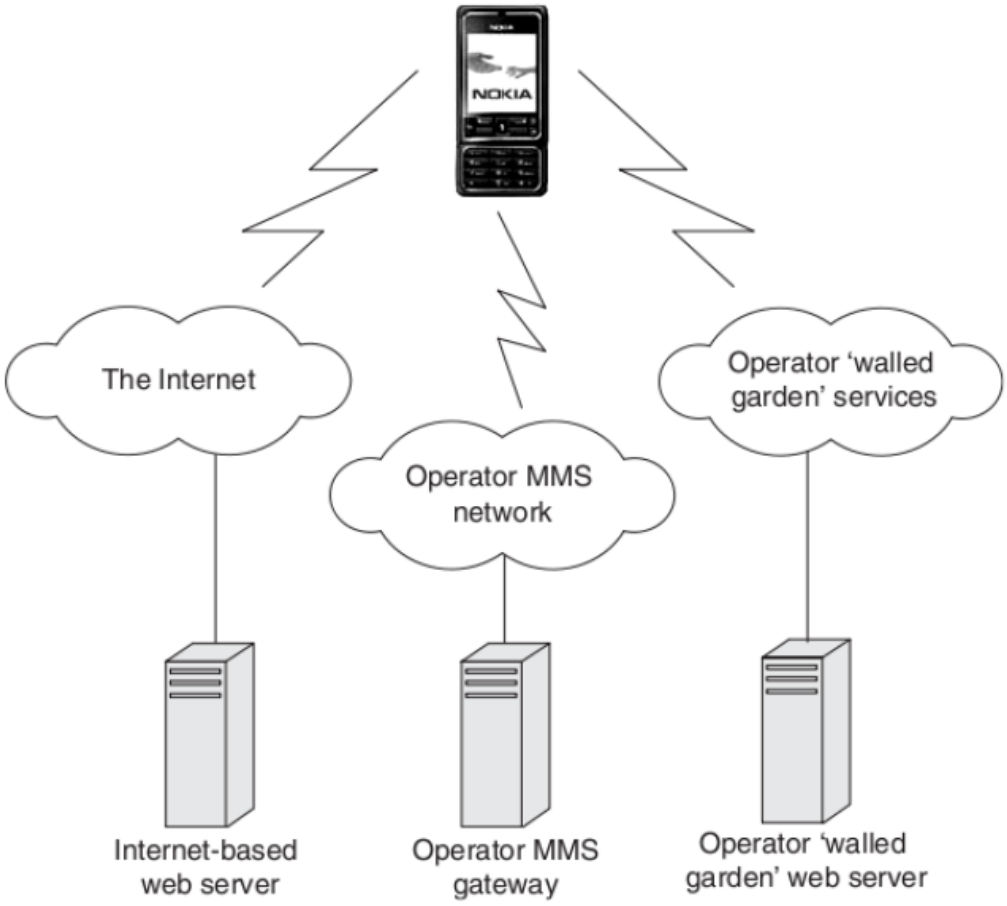


Figura 4.3: *Multihoming*

Symbian OS supporta le seguenti tecnologie IP, sebbene alcune di esse non siano disponibili per tutte queste piattaforme:

Packet-switched data connections over GPRS/W-CDMA: fornisce i servizi dell'operatore telefonico, come ad esempio accesso a Internet, MMS, streaming di musica o video.

Circuit-switched data calls: fornisce accesso ad Internet attraverso dial-up ISPs.

Bluetooth PAN: fornisce una rete Ethernet virtuale attraverso il dispositivo Bluetooth. Può essere utilizzata per giochi multiplayer o per creare delle reti ad-hoc.

Ethernet-based technologies: fornisce supporto a connessioni di tipo Ethernet, come ad esempio le wireless LAN.

Dal punto di vista della programmazione esistono tre modi per stabilire una nuova connessione. Ognuna di queste modalità può avere comportamenti diversi e generare richieste, come un prompt per la scelta della connessione:

Socket: le applicazioni possono utilizzare direttamente i socket, senza aver inizializzato esplicitamente nessuna connessione. In questo modo la connessione sarà avviata automaticamente nel momento in cui si tenta di inviare un pacchetto dati IP. L'utente può essere avvisato attraverso una finestra di dialogo, all'interno della quale può selezionare la rete più idonea. Questo tipo di connessione è utilizzato nella maggior parte delle applicazioni incluse nei terminali della serie S60. È possibile comunque cambiare l'impostazione di default selezionando, all'interno delle configurazioni del telefono, il punto di accesso desiderato. Utilizzando questo tipo di connessione non si ha modo di monitorare lo stato della connessione, quindi diventa difficile scoprire quali sono le cause di un malfunzionamento sulla connessione. Gli errori generati dalle chiamate ai socket possono fornire un'indicazione dei problemi di rete, ma le informazioni acquisibili sono limitate e vi è uno stacco temporale tra il momento in cui il problema si è generato e il momento in cui l'applicazione ne viene a conoscenza.

RConnection::Start(): questo metodo fornisce all'applicazione più controllo e visibilità nei confronti della connessione che si è appena instaurata. L'applicazione può utilizzare lo stesso oggetto RConnection per tracciare i progressi della connessione. È compito dell'utente selezionare l'IAP che verrà usato, esattamente come nel caso precedente. Nel caso in cui la connessione alla rete non vada a buon fine, l'applicazione sarà avvisata in maniera asincrona attraverso l'oggetto *TRequestStatus*. Anche in questo caso è consigliato utilizzare la versione asincrona di questa funzione, dato che le fasi di creazione di una connessione di rete posso richiedere parecchio tempo.

RConnection::Start(IAP): per connettersi ad un preciso punto di accesso, l'applicazione può effettuare una chiamata al metodo RConnection::Start() specificando un IAP ID come argomento. Quest'ultimo può essere scelto direttamente dall'utente in fase di configurazione del programma, oppure dall'applicazione mediante l'utilizzo di specifiche API, denominate *CommsDat*.

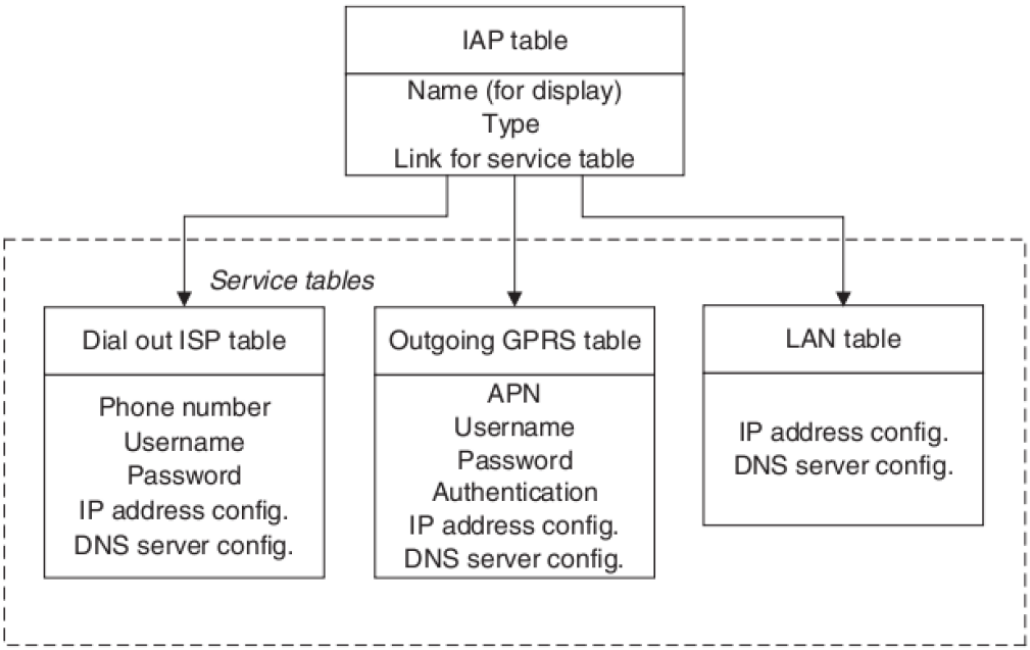


Figura 4.4: *CommsDat API*

La figura 4.4 mostra una vista semplificata della struttura CommsDat. La tabella di maggiore interesse è la *IAP table*, che contiene in particolare il nome dell'IAP e il link all'appropriato tipo di servizio che fornisce. La tabella dei servizi contiene quindi informazioni che possono variare in base al tipo di IAP: contiene ad esempio l'APN per una connessione GPRS o il SSID per una connessione WLAN, username e password per accedere al servizio, il tipo di configurazione dell'indirizzo IP (statica o negoziata per mezzo del servizio DHCP), eccetera.

4.2.2 Configurazione interfacce

Per instaurare una connessione sul terminale è stata implementata una funzione, denominata *addConnection*, che ha il compito di collegarsi automaticamente una punto di accesso, popolando correttamente le strutture dati.

Questa operazione è svolta utilizzando alcune sottofunzioni:

searchIAPs: scorre e salva l'elenco delle connessioni di tipo *type* presenti all'interno del telefono. In caso di connessione UMTS, la funzione setta i parametri per l'unica connessione disponibile. Quando si instaura invece una connessione Wireless, dopo aver recuperato la lista degli Access Point presenti nell'area in cui ci si trova (tramite la funzione *listAvailAPs*), si effettua una ricerca incrociata tra gli Access Point disponibili e quelli configurati all'interno del telefono. Il sottoinsieme generato è quindi ordinato per potenza di segnale. Al termine dell'operazione saranno quindi configurati, all'interno di apposite strutture dati, tutti gli Access Point rilevati.

listAvailAPs: effettua una scansione degli Access Point disponibili, salvando i risultati all'interno di una lista che contiene il nome dell'Access Point e la potenza di segnale. Al termine dell'operazione restituisce il numero degli Access Point rilevati.

sort_by_SignalStrength: ordina gli Access Point in base alla potenza di segnale rilevata dallo smartphone.

Se l'instaurazione della connessione ha successo, è possibile creare i socket che saranno successivamente utilizzati per i protocolli SIP, RTP e RTCP, ed infine sarà attivato un *Connection Monitor*. Da questo momento la connessione sarà quindi monitorata e gli eventi che la riguardano potranno essere gestiti in modo appropriato dall'applicazione.

4.2.3 Invio e ricezione pacchetti

Il client VoIP deve poter trasmettere i propri dati indipendentemente dall'interfaccia e dall'indirizzo IP usati. In generale, il terminale non utilizzerà sempre lo stesso indirizzo IP per trasmettere i suoi dati, ma potenzialmente potrà cambiarlo frequentemente in base alle decisioni del livello di Vertical Mobility. L'obiettivo è quello di nascondere questi cambi di indirizzo IP sia all'UA eseguito sul terminale che ad ogni entità in comunicazione con questo, assicurando che un'eventuale sessione in corso non venga interrotta e possa quindi continuare in modo del tutto trasparente.

Per questo motivo all'interno del sistema si è realizzato un sistema di invio/ricezione dati che coinvolge più interfacce contemporaneamente. In pratica quando un messaggio deve essere inviato si controlla quali siano in quel momento le interfacce disponibili, selezionando la prima disponibile. Si tenta quindi di inviare il messaggio e, nel caso in cui si verifichino problemi (rilevati dal Connection Monitor o a causa di timeout) si tenta di utilizzare un'altra interfaccia disponibile. In qualunque caso non sono inviati messaggi duplicati e, se si verifica un errore il Connection Monitor, si dovrà occupare di configurare, se possibile, una nuova connessione.

Il flusso dati SIP ha caratteristiche molto differenti da quello RTP/RTCP: la dimensione media di un pacchetto SIP è di 900 Byte, la frequenza di trasmissione è molto variabile (tipicamente mai più di qualche pacchetto al secondo; potenzialmente un pacchetto viene spedito a distanza di vari minuti dall'altro), infine viene gestita la ritrasmissione di un pacchetto se questo viene considerato perso. Al contrario la dimensione media di un pacchetto RTP è di 170 Byte (a seconda del codec usato) inclusi anche gli header dei vari protocolli (RTP, UDP, IP, MAC), la frequenza di trasmissione è dell'ordine di un pacchetto ogni 40 millisecondi. In sintesi un pacchetto SIP è quindi meno sensibile ad overhead sulla dimensione o a ritardi sulla trasmissione rispetto ad un pacchetto RTP. Quest'ultimo tipicamente non può essere ritrasmesso ed una sua perdita ha effetti molto negativi sulla qualità della conversazione.

In base a queste osservazioni risulta fondamentale minimizzare l'overhead necessario alla gestione della terminal mobility, in particolare bisogna evitare di introdurre ritardi nella trasmissione dei pacchetti o di aumentare ecces-

sivamente la dimensione degli stessi. Per rispettare queste necessità è stato deciso di non affidarsi ad alcun protocollo ad-hoc per la trasmissione dei dati tra client e proxy server (ad esempio per informare riguardo all'indirizzo IP utilizzato) e non devono scambiarsi informazioni particolari per la gestione della Terminal Mobility.

4.2.4 Aggiunta e rimozione connessioni

La funzione *addConnection*, specificata nel paragrafo 4.2.2, è lanciata durante l'inizializzazione delle connessioni. Nelle fasi iniziali, infatti, sono definiti alcuni *Connection Monitor* (uno per connessione) il cui compito è quello di monitorare lo stato delle connessioni. Le notifiche di cambio stato riguardanti le connessioni possono essere abilitate mediante l'utilizzo della funzione *ProgressNotification*. Nel momento in cui lo stato della connessione cambia, l'Active Scheduler risveglierà il Connection Monitor e sarà quindi possibile gestire l'evento.

Durante l'inizializzazione della connessione è aperto un socket che consente al programma di recuperare alcuni importanti dati, come indirizzo IP ricevuto, subnet mask, gateway e DNS. Nel caso in cui tutte le operazioni descritte non generino problemi, la connessione raggiungerà lo stato di *KLayerOpen* e sarà quindi completamente instaurata. A questo punto sarà possibile aprire i socket SIP, RTP e RTCP e da questo momento sarà possibile utilizzare la connessione per inviare e ricevere pacchetti.

In caso di problemi sulla connessione, il Connection Monitor si accorgerà dell'evento e inizierà l'esecuzione della funzione *RunL*, all'interno della quale sono definite le operazioni di chiusura della connessione mediante la disattivazione di tutti i socket che la riguardano e della RConnection. Da questo momento non sarà più possibile utilizzare la connessione interrotta e il Connection Monitor tenterà quindi di aprire una nuova RConnection utilizzando la funzione *addConnection*, cercando una nuova rete a cui collegarsi in modo asincrono per non bloccare l'intera applicazione. Qualora questa operazione non vada a buon fine, sarà necessario chiudere la RConnection ed attendere un certo periodo di tempo prima di ritentare un nuovo collegamento. Nel caso in cui tutte le connessioni siano interrotte, l'applicazione dovrà

essere bloccata in attesa di un ripristino di connessione su almeno una delle interfacce disponibili.

4.2.5 Proxy Server

All'interno dell'architettura di ABPS risulta di fondamentale importanza l'utilizzo di un Fixed Server. Si tratta sostanzialmente di un Call Stateful Proxy server che si inserisce all'interno della comunicazione tra i due endpoint. Questi ultimi dovranno quindi inviare ogni messaggio direttamente ad esso, a prescindere dalla reale destinazione. Quando ad esempio lo UA invierà il messaggio di REGISTER, il proxy server ne modificherà il contenuto prima di inviarlo al Registrar Server. La modifica riguarderà l'indirizzo IP specificato dallo UA, al posto del quale sarà inserito l'indirizzo del proxy. Mediante l'utilizzo delle entità descritte il sistema è quindi in grado di fornire Terminal Mobility e QoS in modo del tutto trasparente: ogni volta che un UA subirà una riconfigurazione dell'indirizzo IP, non sarà necessario propagare questa modifica fino al Registrar Server. Gli unici interessati a tale evento saranno UA e Proxy Server, che si preoccuperanno di portare avanti le comunicazioni in corso, evitandone eventuali interruzioni.

Il terminale mobile può usare per comunicare qualunque interfaccia in suo possesso in qualunque momento. Può quindi capitare che il proxy server riceva i dati di un client da indirizzi IP diversi, e intuitivamente questo pone il problema di come il proxy possa riconoscere ed accettare correttamente questi dati come provenienti da quel client specifico. Tali requisiti definiscono una nuova soluzione che non sia più strettamente ip-centrica, ovvero il cui funzionamento non sia fortemente dipendente dall'indirizzo IP utilizzato.

Il Proxy Server dovrà quindi accettare comunicazioni indipendentemente dall'indirizzo IP da cui provengono, riconoscendo il traffico appartenente ad uno specifico UA. Un proxy può supportare contemporaneamente un numero elevato di client ed è quindi necessario tenere traccia degli spostamenti per poter inoltrare correttamente ogni dato ricevuto. Per effettuare questa operazione, il proxy mantiene una lista di indirizzi IP da cui riceve le comunicazioni, che sarà nel tempo popolata analizzando ogni pacchetto ricevuto.

Attraverso l'utilizzo di questa lista, il proxy potrà decidere a quale indirizzo IP spedire i dati destinati ad un particolare client.

La costruzione dello Proxy Server nel sistema ABPS è avvenuta modificando un progetto opensource, ovvero Siproxd[1].

4.3 Modalità integrata

La modalità integrata prevede l'implementazione del multihoming e del monitoraggio delle connessioni direttamente all'interno della libreria PJSIP. Questo tipo di architettura richiede una profonda modifica delle strutture e delle funzioni di cui dispone PJSIP.

PJSIP inizia la propria esecuzione all'interno della funzione *ua_main()*, contenuta nel file *ua.cpp* presente nella cartella *pjsip-apps\src\symbian_ua*. All'interno della stessa cartella è stato aggiunto un nuovo file, denominato *multihoming.cpp*, che contiene le principali modifiche apportate a PJSIP per l'aggiunta del multihoming, come le funzioni che instaurano, monitorano e chiudono una connessione.

All'interno delle librerie PJSIP e PJMEDIA sono stati definiti nuovi tipi di transport basandosi sui quelli già esistenti per il protocollo UDP, come specificato nel paragrafo 4.2.3. Queste modifiche sono presenti all'interno dei file *sip_transport_ggproxy.c* e *transport_ggproxy.c*, che contengono rispettivamente le funzioni e le strutture necessarie per l'invio e la ricezione dei messaggi SIP e RTP/RTCP utilizzando più interfacce.

La modifica principale riguarda la registrazione sulla IOQueue dei socket coinvolti e la duplicazione delle strutture dati necessarie alla ricezione dei messaggi. Ad una prima analisi questa operazione non sembrava necessaria in quanto, per la natura dei messaggi SIP, essi vengono ricevuti contemporaneamente. In realtà PJSIP, alla ricezione di un messaggio, rilascia la memoria del *receive data buffer*, per poi riallocarla successivamente per il messaggio successivo. In questo modo le operazioni di lettura rimaste in sospeso utilizzavano un indirizzo errato del buffer in cui memorizzare i dati.

[1] http://siproxd.sourceforge.net/

4.4 Modalità proxy locale

Una versione alternativa dell'applicazione potrebbe invece utilizzare un server proxy UDP locale, eliminando in questo modo le numerose modifiche da apportare a PJSIP. In questa modalità, denominata *proxy locale*, i pacchetti non sono inviati dall'applicazione verso Internet, ma ad un indirizzo *localhost* sul quale è attivo il proxy. Su quest'ultimo è quindi aperta una porta per ogni tipo di flusso dati, ovvero SIP, RTP e RTCP. I pacchetti inviati dall'applicazione sono quindi ricevuti sulle porte aperte sul proxy locale, il cui compito è quindi quello di monitorare costantemente le connessioni ed inoltrare correttamente i pacchetti sulla rete Internet.

Le modifiche apportate a PJSIP riguardano il datagram spedito dall'applicazione, che deve essere modificato in modo tale da aggiungere in testa 2 byte che indicano la porta di destinazione sul proxy server. In questo modo PJSIP penserà di inviare direttamente al proxy server, rendendo trasparente l'operazione.

All'interno del proxy locale è mantenuta una tabella che salva le informazioni riguardanti le porte utilizzate dal proxy server e dall'applicazione. Quando viene ricevuto un pacchetto dall'esterno sarà quindi possibile controllarne l'intestazione, scoprendo da quale porta proviene e a quale destinatario è indirizzato. Queste informazioni sono quindi salvate nella tabella precedentemente descritta, in modo tale da poter inoltrare correttamente il messaggio all'arrivo di un nuovo datagram.

Capitolo 5

Valutazione

La modalità integrata e la modalità proxy locale presentate nel capitolo precedente sono state analizzate al fine di valutare gli aspetti positivi e negativi che presentano. In particolare l'analisi è stata orientata ad evidenziare le differenze di riusabilità del software, di performance e di efficienza energetica.

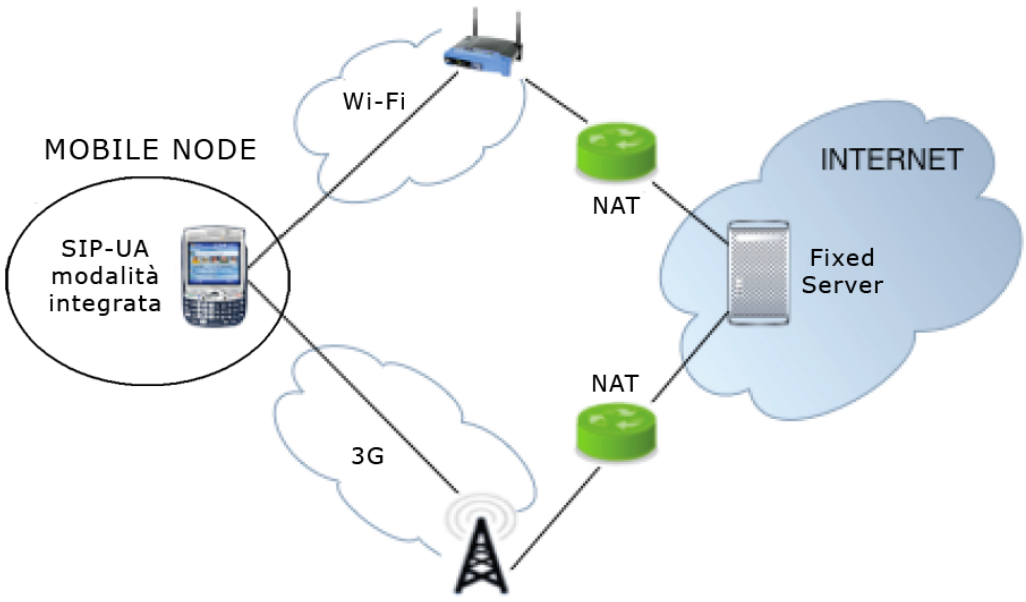


Figura 5.1: *Modalità integrata*

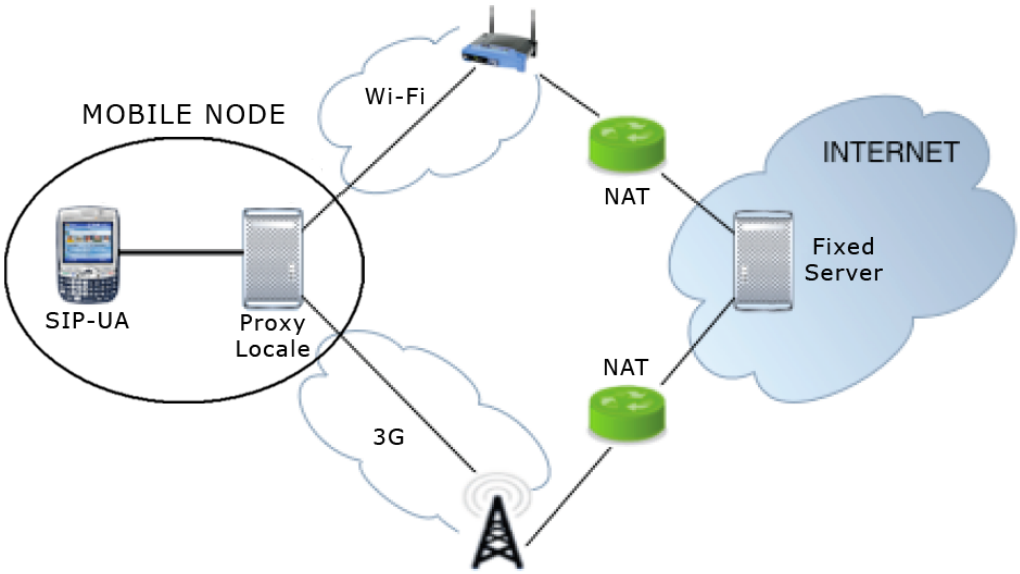


Figura 5.2: *Modalità proxy locale*

5.1 Riusabilità

La modalità proxy locale nasce dall'idea di eliminare le numerose modifiche necessarie all'implementazione del multihoming all'interno dell'applicazione. Tale approccio ha imposto l'uso di un'entità aggiuntiva, il proxy locale, contattabile dall'applicazione e raggiungibile in qualsiasi momento, in modo da poter operare in maniera totalmente indipendente dagli eventi che riguardano il livello network. Per tale motivo il proxy locale risulta essere implementato in un'applicazione separata che viene eseguita contemporaneamente allo User Agent sul nodo mobile.

L'introduzione del proxy locale permette quindi di minimizzare gli interventi sull'applicazione VoIP. Questa importante caratteristica consente quindi di poter riutilizzare il proxy server anche su sistemi operativi diversi, su altre applicazioni o, nel caso di aggiornamenti, su versioni successive di PJSIP. Gli sviluppatori di PJSIP hanno infatti rilasciato negli ultimi mesi diversi aggiornamenti, portando il progetto alla versione 1.8[1]. Quest'ultima versione introduce diversi bugfix e supporto ad iPhone e iPad.

[1]PJSIP versione 1.8 è stato rilasciato il 9 settembre 2010

5.2 Performance

In un contesto specifico come quello degli smartphone in generale, e del sistema Symbian in particolare, l'approccio descritto dalla modalità proxy locale risulta essere poco performante. Le linee guida proposte da Symbian consigliano di utilizzare una programmazione basata sugli eventi anziché sui thread, in modo da minimizzare i context switch all'interno di un singolo processo. Nella modalità proxy locale il sistema necessita invece di eseguire contemporanemente due processi indipendenti, in cui il proxy locale dovrà inoltre occuparsi di ricevere i messaggi, modificarli ed inviarli successivamente allo User Agent o, a seconda della direzione della comunicazione, al Fixed Server. Il consumo di risorse risulterà quindi notevole in un contesto come quello degli smartphone, ovvero in sistemi dotati di un quantitativo esiguo di risorse.

La modalità integrata propone invece di modificare la struttura originale di PJSIP al fine di integrare le funzionalità del proxy locale all'interno dello User Agent. Il risultato, schematizzato in figura 5.1, fornisce un'architettura più leggera e performante, che prevede l'uso di un componente in meno rispetto alla versione con proxy locale.

5.3 Efficienza energetica

Una delle peculiarità di un sistema operativo per dispositivi mobili è la gestione della CPU volta alla minimizzazione dei consumi energetici. Per questo motivo nella costruzione di un'applicazione occorre utilizzare codice event-driven (basato su eventi) sia per l'interazione con il livello utente, che a livello di sistema per comunicazioni asincrone con periferiche di input e output. Evitare quindi di impegnare eccessiamente il dispositivo, anche quando sufficientemente performante da gestire contemporaneamente una moltitudine di processi e di thread, comporta un minore impiego delle risorse, che si traduce in una maggiore durata delle batterie. Per questo motivo la modalità proxy locale, a causa della propria natura implementativa, impegnerà maggiormente il dispositivo, come specificato anche nel paragrafo 5.2.

Capitolo 6

Conclusioni e sviluppi futuri

In questo elaborato si è illustrato come è avvenuto lo studio e lo sviluppo sperimentale di un'applicazione VoIP che implementasse la tecnologia di multihoming. Lo sviluppo di una soluzione personalizzata per il sistema Symbian costituisce un obiettivo necessario se si considera la grande diffusione di telefonini dotati di questo sistema all'interno del mercato degli smartphone.

Le modalità presentate nei capitoli precedenti, integrata e proxy locale, possiedono entrambe vantaggi e svantaggi da valutare accuratamente. Nel caso in cui si decida di dare maggiore importanza alla portabilità e alla semplicità d'implementazione è consigliato utilizzare la modalità di proxy locale, mentre se si preferisce ottenere performance migliori e minore consumo energetico è consigliato orientarsi verso la soluzione integrata.

Il sistema multihoming necessita inoltre di ulteriori modifiche per poter essere utilizzato all'interno del sistema ABPS. Dal punto di vista del funzionamento è necessario inserire la parte relativa alla sicurezza, in modo da firmare i pacchetti in uscita e fornire un meccanismo di identificazione del mittente. Un client mobile che trasmette da un determinato indirizzo IP può infatti cambiare tale indirizzo in un qualunque momento della sessione. Di conseguenza risulta necessario distinguere il traffico proveniente da un client specifico senza affidarsi all'indirizzo IP. Un contesto di sicurezza tra client e proxy può quindi fornire proprietà crittografiche alle comunicazioni, dato che per ogni pacchetto trasmesso vengono garantite autenticità ed integrità. Questa caratteristica semplificherebbe l'implementazione del proxy server,

migliorando inoltre il routing dei pacchetti attraverso la rete.

Dal punto di vista dell'usabilità è necessario realizzare un'interfaccia grafica che consenta all'utente un accesso agevole all'applicazione. Durante lo sviluppo è stata infatti utilizzata una versione testuale del client per Symbian, che, al di fuori della fase di debug, risulta poco usabile da un utente non esperto. In particolare, per realizzare tale interfaccia è consigliato anche seguire le apposite linee guida fornite dalla Symbian Foundation, tenendo in grande considerazione ciò che sta avvenendo con la recente introduzione della versione 3.0 di Symbian OS.

Ringraziamenti

Desidero ringraziare Vittorio, relatore di questa tesi, per la sua infinita disponibilità e per l'aiuto che mi ha offerto in questi mesi. Non riesco a contare le ore trascorse insieme su Skype a effettuare test e prove *giusto per non sapere né leggere, né scrivere*, oltre alle giornate passate a Bologna durante le vacanze estive. Credo che nessun altro professore possa essere di così grande aiuto e supporto.

Un sentito ringraziamento ai miei genitori Gabriele e Monica che, con il loro enorme sostegno morale ed economico, mi hanno permesso di raggiungere questo traguardo.

Desidero infine ringraziare la mia ragazza Giorgia che, anche questa volta, mi ha aiutato a revisionare l'intera tesi per renderla leggibile. È stata un'estate piuttosto corta se contiamo i giorni di vacanza trascorsi assieme, ma avremo modo di rifarci nei prossimi mesi. Promesso.

Bibliografia

[1] **V. Ghini, L.E. Tomaselli, G. Lodi, F. Panzieri, A. Messina**, *"Always Best Packet Switching" for SIP-based mobile multimedia services*, 2009

[2] **3GPP**, *Voice Call Continuity (VCC) between Circuit Switched (CS) and IP Multimedia Subsystem (IMS)*, 2007
http://www.3gpp.org/ftp/Specs/html-info/23206.htm

[3] **E. Gustafsson et al.**, *Always Best Connected*, IEEE Comm. Mag., vol. 10, no. 1, Feb. 2003, pp. 49–55.

[4] **ITU-T**, *G.1010 : End-user multimedia QoS categories*, 2002
http://www.itu.int/rec/T-REC-G.1010-200111-I

[5] **Michele Dallachiesa**, *Il VoIP: SIP, H.323, SCCP ed RTP/RTCP*, 2007
http://dallachiesa.com/docs/voip/

[6] **Iain Campbell et al**, *Symbian OS Communications Programming, 2nd Edition*, 2007
http://developer.symbian.org/wiki/index.php/Symbian_OS_Communications_Programming

[7] **PJSIP**, *Developer's Guide*, 2006
http://www.pjsip.org/release/0.5.4/PJSIP-Dev-Guide.pdf