Alma Mater Studiorum · Università di Bologna

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

Approccio ABPS per VoIP su dispositivi mobile con Android.

Tesi di Laurea in Reti di Calcolatori

Relatore:
Chiar.mo Prof.
Vittorio Ghini

Presentata da:
Francesco Eugenio Susi

Sessione III
Anno Accademico 2010/2011

Ringraziamenti

Vorrei ringraziare innanzitutto la mia famiglia che mi ha sostenuto moralmente ed economicamente durante i mie studi, permettendomi di arrivare a tale traguardo.

Ringrazio anche tutti gli amici e compagni di studio, con la quale ho condiviso questa esperienza universitaria e tutti i professori della Facoltà di Informatica, in particolar modo il mio relatore, il dott. Vittorio Ghini, che mi ha guidato durante lo sviluppo della presente tesi.

Indice

Introduzione

Negli ultimi anni abbiamo assistito ad una crescente diffusione della rete Internet a banda larga e un abbassamento dei costi delle connessioni per i dispositivi mobili, che hanno portato allo sviluppo di applicazioni multimediali capaci di fornire servizi audio, video e dati in modalità *real-time*.

Questo complesso servizio multimediale si è sviluppato sopratutto grazie all'introduzione ed alla ricerca, a partire dagli anni '90, della tecnologia *Voice over IP* (acronimoVoIP) ed oggi è sulla buona strada per diventare il principale mezzo di comunicazione multimediale dei prossimi anni. Tuttavia questa tecnologia è ancora in fase di sviluppo e presenta diverse problematiche, in particolar modo per quanto riguarda i dispositivi mobili.

Con la presente tesi si vuole proporre un metodo alternativo per affrontare alcune difficoltà che ruotano attorno al VoIP, grazie all'ausilio dell'architettura *ABPS*.

Nel primo capitolo verrà presentata la tecnologia in questione e spiegato il suo funzionamento, assieme alla descrizione dei protocolli coinvolti. Verrà poi mostrato uno tipico scenario di comunicazione, in cui si evidenziano i principali meccanismi che ostacolano il corretto funzionamento del VoIP, quali *Firewall* e *NAT*. Infine verranno suggerite delle possibili soluzioni, già oggi adottate, per affrontare problemi di questo tipo.

Nel secondo capitolo verrà descritto l'obbiettivo e gli strumenti usati per raggiungere tale obbiettivo: partendo dalle librerie PJSIP, fino ai tools di sviluppo.

Infine, nel terzo capitolo, si parlerà della fase di sviluppo, sottolineando le problematiche riscontrate lungo questo percorso, e dei potenziali sviluppi futuri.

Capitolo 1

Scenario

1.1 La Telefonia VoIP

Con il termine "*Voice over IP"* (che tradotto significa "*Voce tramite Protocollo Internet"*), si intende una tecnologia che sia in grado di tramutare il segnale della voce analogico in un flusso di dati digitali, rendendo cosi possibile effettuare conversazioni telefoniche usando una connessione ad Internet, ma non solo, grazie a numerosi provider VoIP è possibile effettuare telefonate anche verso la tradizionale rete telefonica (PSTN).

Più specificamente con VoIP si intende l'insieme dei protocolli che rendono possibile la comunicazione a livello applicativo, attraverso una rete dedicata a commutazione di pacchetto, che utilizzi il protocollo IP, senza connessione per il trasporto dati [1].

È possibile usufruire di tali servizi mediante:

- una applicazione eseguita su di un computer (*Softphone*);
- un telefono tradizionale con adattatore oppure attraverso un dispositivo del tutto simile ad un telefono fisso (*telefono VoIP*), ma connesso ad Internet anziché alla PSTN;
- una applicazione eseguita su di un *telefono mobile* capace di connessione dati (WiFi, GPRS o UMTS);

Grazie alle sue caratteristiche e numerose funzionalità, ma grazie sopratutto alla costante diffusione della banda larga e alla smisurata produzione di dispositivi mobile (basti pensare che tra il '04 e il '05 la produzione degli smartphone è raddoppiata e attualmente nel mondo rappresentano il 30% del mercato [1]), il VoIP sta prendendo sempre più piede, non solo in ambiente lavorativo ma anche in quello privato.

I vantaggi di questa tecnologia sono numerosi: innanzitutto può utilizzare una sola rete integrata sia per voce che per i dati, garantendo una riduzione dei costi delle infrastrutture e uno scambio agevolato delle informazioni, offrendo così dei servizi che vanno oltre la classica telefonata. Molti arrivano a considerare la telefonia tradizionale ormai "obsoleta", mentre definiscono il VoIP come una delle cosiddette "tecnologie del futuro", in quanto permette di fare cose che prima erano impensabili.

Questa tecnologia non solo permette una gestione congiunta della telefonia e dei sistemi di messaggistica, come mail, fax e segreteria telefonica, ma può essere utilizzato per telefonate più complesse, come video-conferenze in multi-point, oppure usare applicazioni aggiuntive come l'*Application Sharing* (applicativi su desktop condivisi) ed il *White Boarding* (applicazioni che consentono di vedere e interagire con una sorta di lavagna condivisa). Inoltre è importante sottolineare che l'implementazione di future funzionalità e servizi, non richiederà la sostituzione o modifica dell'hardware.

Uno dei principali vantaggi, che ha permesso lo sviluppo del VoIP, è stato il notevole abbattimento dei costi. Se nella telefonia tradizionale tempo e distanza sono i parametri da applicare al costo di una telefonata, nella telefonia VoIP questi parametri assumono pesi diversi: in generale un servizio VoIP costa meno di un servizio equivalente tradizionale, sia per chiamate nazionali che internazionali o intercontinentali (il costo della chiamata praticamente non varia sia che si parli nel proprio Paese che dalla parte opposta del mondo). C'è da dire anche che i costi di una telefonata per gli utenti VoIP, sono basati sui reali consumi, in quanto viene considerata la quantità effettiva di informazioni trasferite, quindi i dati, e non la durata complessiva della telefonata.

Per quanto riguarda gli svantaggi di tale tecnologia, sicuramente la scarsa qualità della telefonata e' tra le più frequenti: purtroppo le reti IP non dispongono di alcun meccanismo in grado di garantire che i pacchetti dati vengano ricevuti nello stesso ordine con la quale siano stati trasmessi. Infatti può capitare, a volte, che il network si "congestioni" ed i pacchetti arrivino in ritardo o peggio ancora, vadano persi. Nel primo caso, il ritardo genera eco, fastidiosi disturbi quali fruscii e sovrapposizione delle conversazioni. Nel secondo caso, se la perdita dei pacchetti è superiore al 10%, sarà molto difficile svolgere una conversazione chiara e lineare.

Oggi però sono state introdotte nuove tecniche, che correggono e rendono trascurabili questi inconvenienti, permettendo quasi sempre conversazioni fluide e chiare. Sono stati persino introdotti meccanismi per la gestione dei silenzi, delle pause o dei respiri prolungati, eliminando cosi circa il 50% dei "tempi morti" ed evitando un inutile sovraccarico sulla rete.

Oltre a queste problematiche relative alla qualità del servizio, QoS (*Quality of Service*), il VoIP deve tener conto anche di quei meccanismi di difesa che popolano la rete, come NAT e Firewall.

Ma non anticipiamo le cose: prima di vederli dettagliatamente, andiamo a descrivere quali protocolli vengono coinvolti nella tecnologia VoIP.

1.2 Protocolli coinvolti

Per molti, le telefonate Internet sono automaticamente associate all'uso di *Skype*, che gode di un particolare successo di diffusione e di immagine, ma i cui protocolli operativi sono proprietari e quasi sconosciuti. D'altra parte, la possibilità di usare le reti per dati anche a scopo di trasmissione vocale è stata a lungo inseguita dagli operatori telefonici tradizionali, prima offrendo un accesso numerico con ISDN (*Integrated Services Digital Network*), poi integrando la trasmissione di voce e dati su

di una stessa rete con ATM (protocollo di livello data link), fino al punto che in sede ITU (*International Telecommunications Union*) fu standardizzata la famiglia di protocolli aperti *H.323*, orientata ad offrire servizi multimediali su reti a pacchetto, facilitando allo stesso tempo l'interoperabilità con le reti a circuito preesistenti.
Nel frattempo, l' IETF (*Internet Engineering Task Force*) stava procedendo ad introdurre nuove modalità di realizzazione di servizi multimediali (come videoconferenze e streaming) su rete Internet, attraverso la definizione dei seguenti protocolli [3],

- RTSP, *Real Time Streaming Protocol* [14], per il controllo di una sorgente multimediale;
- SDP, *Session Description Protocol* [12], per la sintassi di descrizione delle *modalità di codifica e trasmissione* per dati multimediali;
- RTP, *Real Time Protocol* [11], per il formato di pacchettizzazione e la trasmissione dei dati multimediali;

Arriviamo alla Telefonia "Voice Over Internet Protocol", che necessita, per poter funzionare correttamente, di due tipi di protocollo di comunicazione, i quali devono funzionare in parallelo: uno serve ad inviare e ricevere la voce digitalizzata sotto forma di dati e l'altro per ricodificare i dati digitale in segnale voce analogico (ricostruzione del frame audio, sincronizzazione, identificazione del chiamante, etc…). Per il trasporto dei dati, nella grande maggioranza delle implementazioni VoIP, viene adottato il protocollo RTP, mentre per la seconda tipologia di protocolli, il processo di standardizzazione non si è ancora concluso[1].

Al momento, la gestione delle chiamate VoIP è indirizzata verso due differenti proposte: una l'abbiamo già citata, ovvero il protocollo *H.323,* elaborato in ambito ITU, mentre l'altra proposta, presentata dal *IETF,* riguarda il protocollo *SIP (Session Initiation Protocol).*

Nonostante la suite di protocolli H.323 sia stata, inizialmente, l'unica soluzione

standard adottata dai produttori di dispositivi per telefonia su IP e in generale per applicazioni multimediali, la proposta SIP, sta incontrando sempre più maggiore approvazione, grazie sopratutto alla sua ottima integrazione con gli altri protocolli della suite TCP/IP (mentre H.323 è pensato per una generica rete a pacchetto), a tal punto che si arrivi a vedere nel protocollo SIP lo stesso impatto che hanno avuto nella comunicazione su internet, i protocolli SMTP per le mail e HTTP per il web.

1.2.1 Protocollo SIP

Il protocollo SIP (acronimo *Session Initiation Protocol*) è un protocollo di livello applicativo basato su IP (definito da RFC 3261)[13], che fornisce meccanismi per instaurare, modificare e terminare una *sessione* di comunicazione, tra due o più utenti.
È importante sottolineare che questo protocollo non fornisce servizi, ma meccanismi per l'implementazione e a differenza di altri protocolli che svolgono funzioni analoghe, SIP è progettato e generalizzato per la scalabilità e l'interconnettività globale, utilizzando servizi Internet già disponibili, come ad esempio il DNS.[3]
Tale protocollo non si occupa di negoziare il tipo di formato multimediale da usare nella comunicazione, né di trasportare il segnale digitalizzato: questi due compiti sono invece svolti rispettivamente dai protocolli SDP e RTP; il protocollo SIP si occupa di mettere in contatto le parti da coinvolgere nella comunicazione, definendo così una *Sessione*, attraverso le seguenti operazioni:

- individuare l'utente
- invitarlo a partecipare (se disponibile)
- instaurare una connessione
- cancellare la sessione

Le entità essenziali di una rete SIP sono gli *User Agent* (abbreviato UA), ovvero

endpoint che possono fungere sia da client che da server. Quando funge da client, *User Agent Clien*t (UAC), da inizio alla transazione originando richieste, mentre quando funge da server, *User Agent Server* (UAS), riceve le richieste e se possibile le soddisfa. Questi due ruoli sono dinamici, nel senso che durante il corso di una sessione, un client può fungere da server che viceversa.

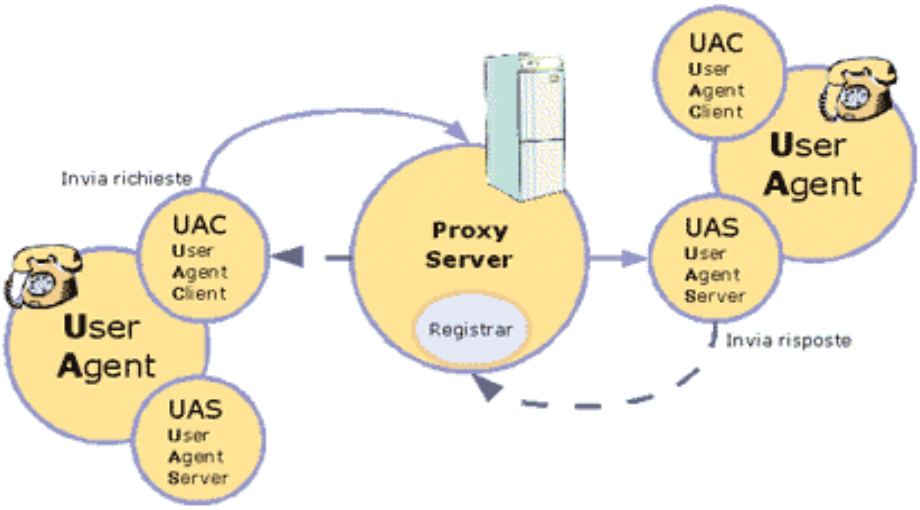

Figura 1: Architettura VoIP

Alla base della comunicazione c'è un server che funge da punto di incontro tra gli UA e servono per mantenere informazioni sulle sessioni degli utenti. Nello specifico viene usato un *Proxy Server,* cioè un server intermedio che analizza i parametri di instradamento dei messaggi e può decidere se rispondere direttamente alle richieste oppure inoltrarle ad un client, ad un server o ad un ulteriore proxy. Tra le sue prerogative ricordiamo [3]:

- la gestione di diversi tipi di politiche come:
  - autenticazione e autorizzazione
  - tariffazione
  - instradamento
  - controllo della qualità del servizio
- la possibilità di inoltrare la chiamata, anziché al Registrar di destinazione, verso

un ulteriore *Proxy di Transito*, o verso un *Gateway*;

- la capacità di deviare la chiamata, comportandosi come un *Redirect Proxy*, rispondendo con un messaggio SIP in cui viene specificato un diverso proxy a cui inoltrarla;
- il funzionamento in modalità S*tateLess*, ossia senza conservare traccia dei messaggi già inoltrati, oppure S*tateFull*, che mantiene in memoria lo stato delle transazioni.

Esiste anche il *Registrar Server,* cioè un server dedicato o collocato in un proxy: quando un utente è iscritto ad un dominio VoIP (es Ekiga), invia un messaggio di registrazione del suo attuale punto di ancoraggio alla rete ad un Registrar Server, in modo tale da poter essere raggiunto [1]. Infine quando un UA invia richieste sistematicamente ad un proxy di default, parliamo allora di *Outbound Proxy.*

Sebbene l'architettura VoIP definita sia basata su di un modello *peer to peer*, il protocollo SIP opera sulla base di messaggi *text-based,* che utilizzano un modello di comunicazione di tipo client-server, simile a quello HTTP, anch'essi suddivisi in un header e in un body. Ecco come si presenta (incapsulato) un tipico messaggio di richiesta SIP:

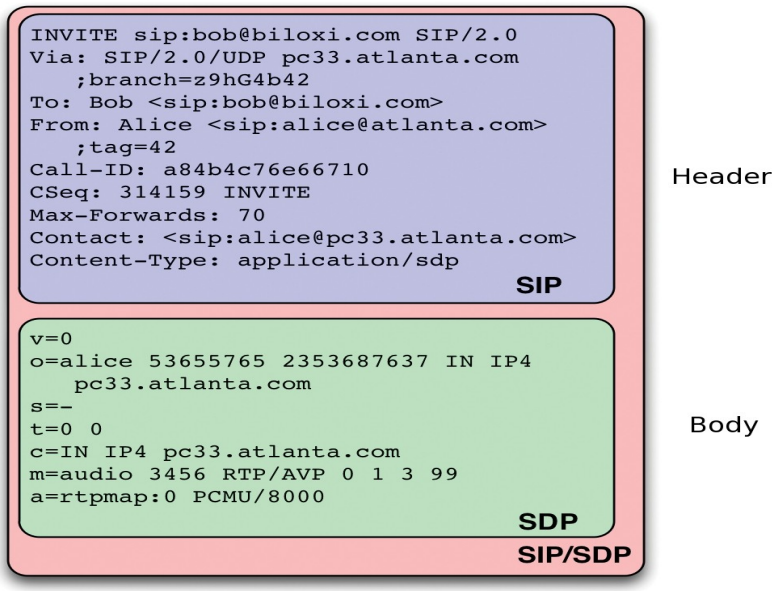

Figura 2: Pacchetto SIP/SDP

Come possiamo vedere in *Figura 2,* la “start line” indica che il messaggio di richiesta è un INVITE, usato per avviare una sessione di comunicazione, mentre le altre informazioni in essa contenute riguardano il *Request-URI*, cioè l’indirizzo SIP dell’utente da contattare (bob@domain.com) e la versione del protocollo usata dal terminale che ha generato il messaggio (SIP/2.0). Il campo *Via*, che in questo caso è relativo al terminale chiamante, può essere più di uno e serve a tenere traccia del percorso compiuto dalla richiesta. Infine, come è facile intuire, i campi *From* e *To,* identificano il chiamante e il chiamato.

Solitamente quando si cerca di instaurare una connessione è comune che la richiesta di invito, prima di raggiungere lo UAS chiamato, attraversi almeno due proxy: l'*Outbound Proxy* del chiamante e il *Proxy Registrar* del chiamato. Questa forma di instradamento prende il nome di *SIP Trapezoid*, in base al particolare modo di raffigurare l'instradamento, come riportato qui sotto.

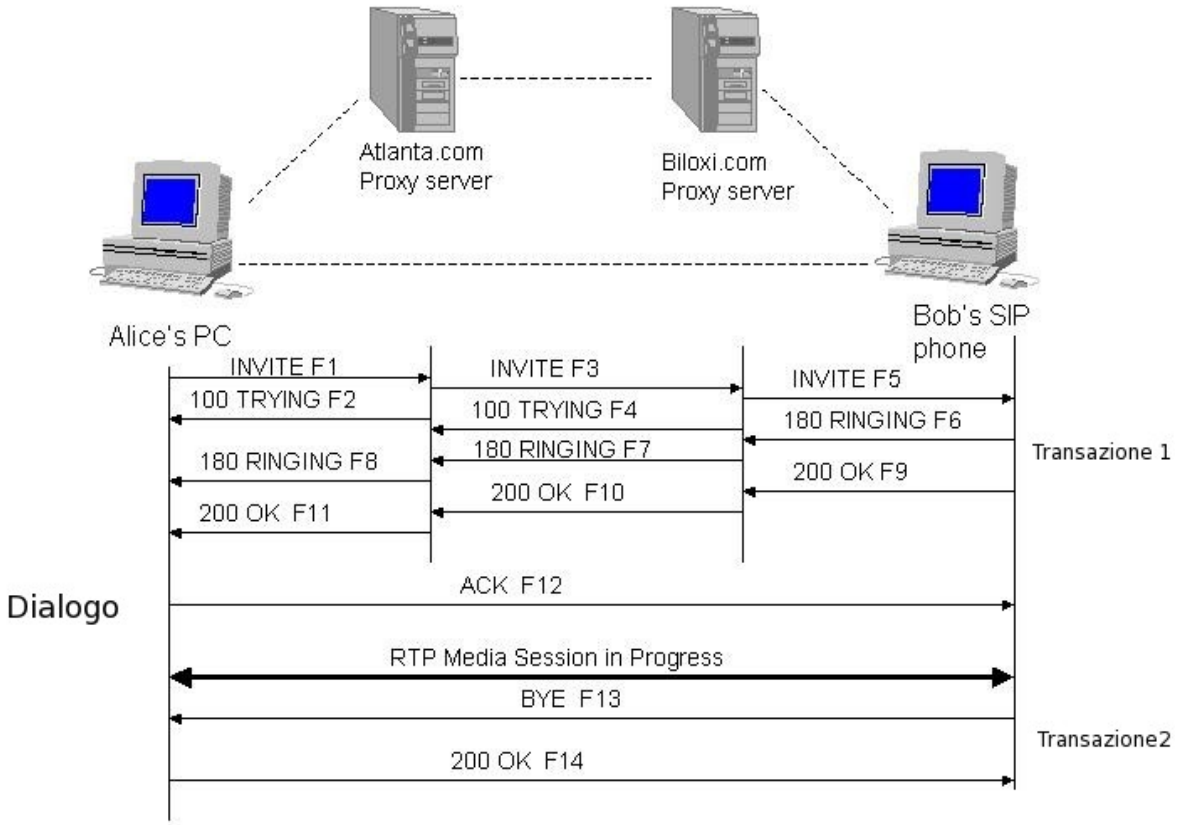

Figura 3: Trapezoide SIP

Tutti i messaggi scambiati tra due UA dall'inizio di una comunicazione fino alla sua conclusione, vanno a costituire un *Dialogo* che viene individuato presso ogni UA in base al valore delle intestazioni contenute nell'header (es: Call-ID, tag, From, To).

A sua volta, all'interno di un dialogo distinguiamo due precise sequenze di messaggi tra gli UA e/o i proxy, che prendono il nome di *Transazioni*. La prima transazione determina l'inizio di un dialogo, durante il quale vengono negoziati i parametri da usare (in SDP) e viene esteso il campo *Via* per tener traccia del percorso che collega i due UA, mentre la seconda transazione determina la fine del dialogo.

1.3 Scenario Network

Abbiamo accennato in precedenza alle problematiche che ruotano attorno al VoIP, citando anche firewall e NAT. Ecco cosa succede in loro presenza:

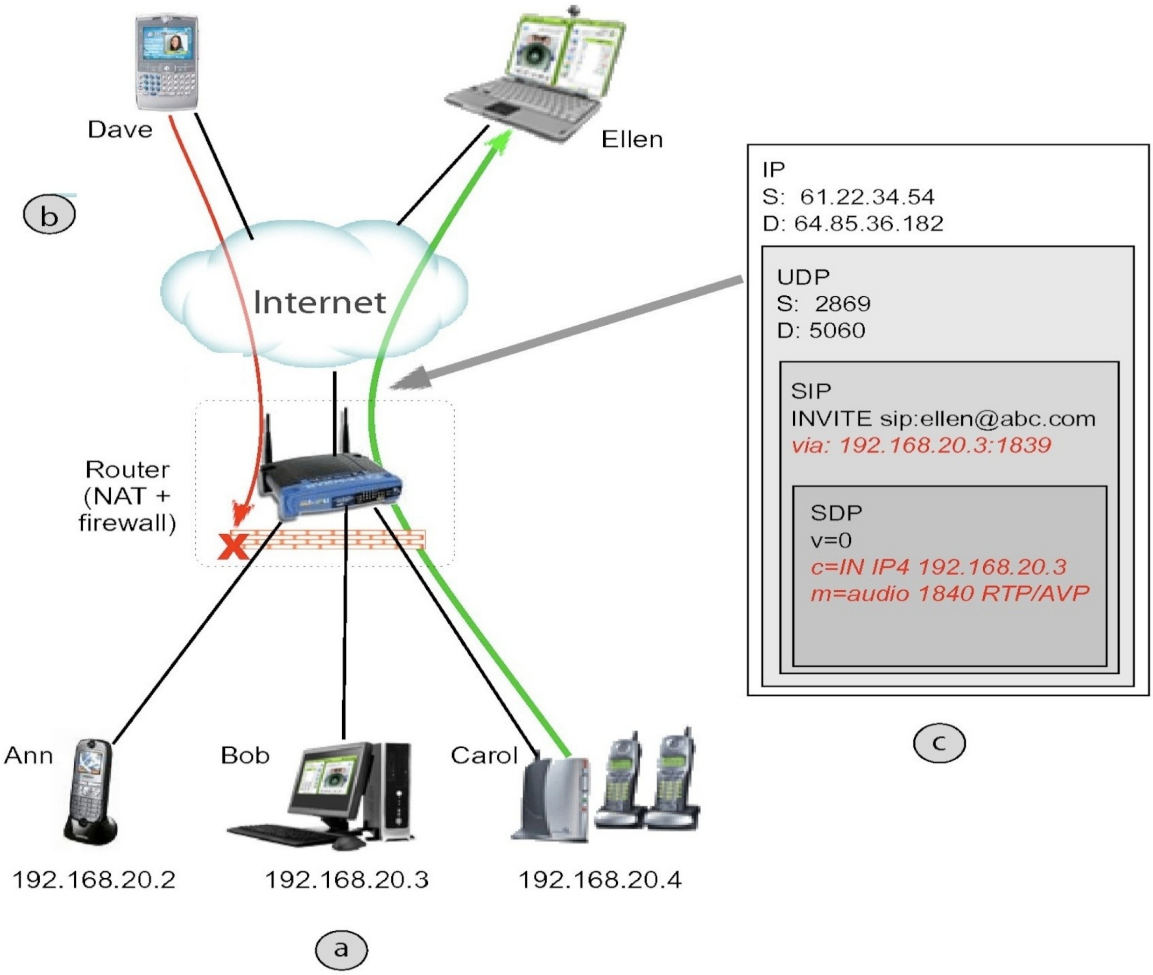

Figura 4: Scenario VoIP

A. il NAT permette l'esistenza di una sotto-rete con indirizzi IP privati;

B. il Firewall blocca l'ingresso nella sotto-rete ad ogni utente non autorizzato;

C. e' possibile una connessione VoIP solo se viene aperta da un utente della sotto-rete;

La figura precedente ci mostra tre utenti sotto la stessa rete, ovvero Ann, Bob e Carol, i quali tra loro possono effettuare comunicazioni VoIP senza alcun problema, mentre gli utenti Dave ed Ellen, che sono esterni alla sotto-rete, troveranno alcune difficoltà ad instaurare una connessione verso l'interno: nel caso in cui Dave volesse contattare Ann, il Firewall glielo impedirebbe (*Figura 4-B*).

Uno scenario di questo tipo però, presenta una condizione particolare, ovvero, permette ad un utente della sotto-rete di contattare una persona all'esterno, ma non il contrario: se prendiamo in considerazione, ad esempio, Carol, vediamo come, nonostante usi un indirizzo IP privato, riesca ad instaurare una connessione con Ellen senza alcun problema, proprio grazie al NAT. Come possiamo vedere nella *Figura 4-C*, sebbene la richiesta SIP sia generata con un indirizzo privato (vedi il campo *Via*), l'indirizzo IP che verrà usato nella comunicazione invece sarà quello quello pubblico.

Nei paragrafi successivi andremo a vedere con maggior dettaglio il funzionamento questi meccanismi di difesa, per capire come mai possono arrivare a compromettere il corretto funzionamento del VoIP.

1.3.1 Firewall

In informatica, nell'ambito delle reti di computer, con il termine *Firewall* si intende un componente passivo di difesa perimetrale che può anche svolgere funzioni di collegamento tra due o più tronconi di rete [1].

Usualmente la rete viene divisa in due sotto-reti: una, detta esterna, comprende l'intera rete Internet mentre l'altra interna, detta LAN (*Local Area Network*), comprende una sezione più o meno grande di un insieme di computer locali [1].

Lo scopo principale dei firewall è quello di proteggere una rete interna dagli accessi di utenti non autorizzati. Normalmente il traffico in entrata da host esterni viene consentito solo se la sessione è stata avviata dalla rete interna, pertanto le chiamate in

entrata, provenienti da una rete esterna (WLAN), verranno filtrate e l'applicazione non riuscirà a stabilire una connessione con gli utenti finali.

Il firewall agisce sui pacchetti in transito da e per la rete interna, grazie alla sua capacità di "aprire" il pacchetto IP per leggere le informazioni presenti nel suo header, e in alcuni casi anche di effettuare verifiche sul contenuto del pacchetto stesso, potendo eseguire su di essi operazioni di:

- controllo
- modifica
- monitoraggio

Una tipica soluzione al problema e' configurare il firewall in modo tale da usare solo specifiche porte per le connessioni VoIP, ma questo approccio presenta un forte rischio per la sicurezza, poiché un malintenzionato che è a conoscenza delle porte o che può arrivare a conoscerle, ne potrebbe approfittare. In più questa e' una soluzione che richiede una configurazione da parte di tecnici o amministratori, visto che l'utente medio non hanno le conoscenze necessarie per farlo, oppure, peggio ancora, possono essere i provider stessi a non permettere questa configurazione.

1.3.2 NAT Traversal

Nel campo delle reti telematiche il *NAT* (Network Address Translation), conosciuto anche come *Network Masquerading*, è una tecnica che consiste nel modificare gli indirizzi IP dei pacchetti in transito su un sistema che agisce da router [1].

L'idea di fondo del NAT è quello di consentire a dispositivi diversi, che sono sotto la stessa rete locale , di condividere un unico indirizzo IP pubblico per accedere alla rete internet: ogni dispositivo sarà identificato da un indirizzo IP privato e per ogni

“transazione” il NAT interverrà con una riscrittura appropriata di tali indirizzi.

Non solo questa tecnica facilita la condivisione di indirizzi IP pubblici tra molti ospiti di una stessa rete, ma può anche essere applicata “a cascata”, ovvero un router connesso a internet usando IP pubblici, fornisce indirizzi IP privati di una seconda serie di router. Ogni nuova serie di router, può a sua volta fornire indirizzi ad uno o più host.

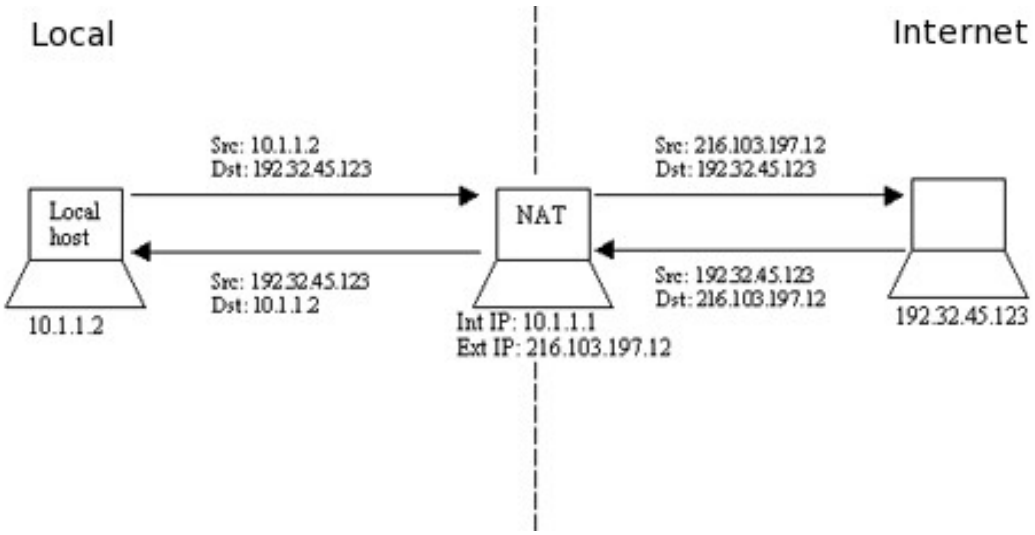

Figura 5: Meccanismo NAT

Il NAT non è ben visto dai puristi delle reti, in quanto mina profondamente la semplicità di IP, e in particolare viola il principio della comunicazione "da qualsiasi host a qualsiasi host" (*any to any*), che si ripercuote in conseguenze pratiche:

- L'instradamento dei pacchetti viene a dipendere non solo dall'indirizzo IP destinazione, ma anche dalle caratteristiche del livello di trasporto.
- Le configurazioni NAT possono diventare molto complesse e di difficile comprensione.
- L'apparato che effettua il NAT ha bisogno di mantenere in memoria lo stato

delle connessioni attive in ciascun momento. Questo a sua volta viola un principio insito nella progettazione di IP, per cui i router non devono mantenere uno stato relativo al traffico che li attraversa.

Critiche “filosofiche” a parte, i Firewall ed i meccanismi NAT giocano un ruolo importante nel garantire sicurezza ed usabilità di una rete interna, ma impongono forti vincoli per la diffusione dei VOIP.

1.3.3 Soluzioni Possibili

Per affrontare questo problema, non ci sono soluzioni semplici. Possiamo seguire i “big” dell'industria che hanno tentato qualche soluzione introducendo ALG (*Application Level Gateway*) e SBC (*Session Border Controller*) oppure utilizzare dei server esterni che implementano meccanismi IETF, il quale ha messo a disposizione una suite di protocolli per affrontare i limiti delle attuali soluzioni disponibili per il problema coi NAT:

- *STUN* (Session Traversal Utilities for NAT):
  e' un protocollo client-server che permette alle applicazioni di scoprire l'indirizzo IP pubblico e le porte con i cui il NAT li sta rendendo visibili sulla rete pubblica.

- *TURN* (Traversal Using Relays around NAT):
  assegna (o segnala) un indirizzo IP pubblico ed una porta su un server raggiungibile a livello globale, il quale fungerà da relay tra le parti comunicanti.

- *ICE* (Interactive Connectivity Establishment):
  e' un framework di più protocolli che definisce le modalità di instradamento,

scegliendo quali tra i protocolli STUN e TURN sia meglio usare per la connessione.

Tuttavia queste soluzioni non soddisfano tutti gli scenari e contesti possibili, ma ci suggeriscono l'approccio da adottare per risolvere il problema, che consiste nel "girarci attorno", attraverso l'suo di un proxy esterno che funga da *Relay*.

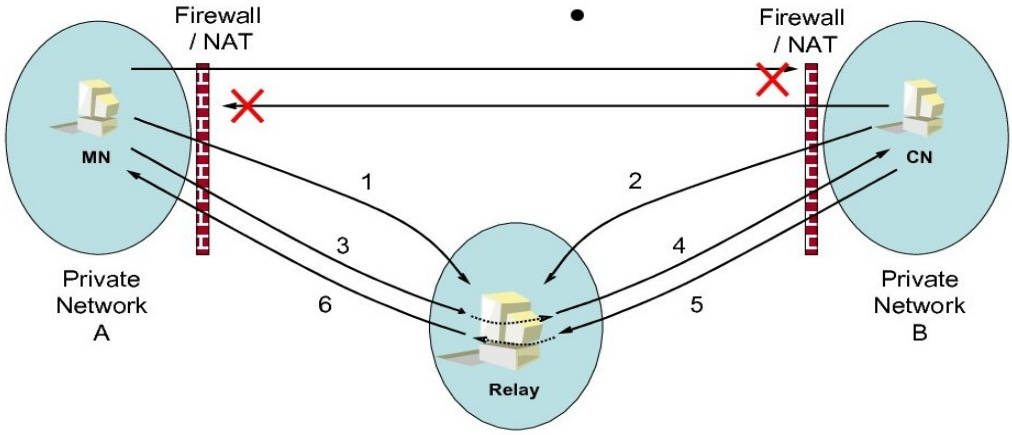

Figura 6: Relay

Infine, prendendo in considerazione che il VoIP sia ancora una tecnologia in fase di "sviluppo" e che poggia su dei protocolli non ancora standardizzati, possiamo riassumere le caratteristiche fondamentali che ci si aspetta da una soluzione, nel seguente modo:

- **Sicurezza**: le soluzioni non devono compromettere le impostazioni di protezione NAT e/o Firewall;
- **Completezza**: La soluzione deve garantire il completamento di una chiamata tra gli utenti, indipendentemente dai tipi di NAT e/o Firewall utilizzati. Inoltre, ha bisogno di massimizzare il peer-to-peer tra le chiamate, al fine di ridurre il

carico sui server di inoltro (relay);

- **Integrazione**: la soluzione deve integrarsi con prodotti o servizi gia esistenti;
- **Conformità ed Interoperabilità:** la soluzione deve interagire con apparecchi diversi e deve basarsi su alcuni standard per garantire la corretta comunicazione tra diversi dispositivi.

Capitolo 2

Obbiettivi e Strumenti

2.1 Obbiettivo

L'obbiettivo della seguente tesi era quello di estendere una applicazione VoIP su dispositivi mobile Android, introducendo un *SIP Proxy Client* con il fine di reindirizzare le connessioni VoIP verso una architettura *ABPS*.

Tale obbiettivo è stato definito prendendo in considerazione l'esistenza di strumenti per il VoIP già presenti e funzionanti, come appunto l'architettura ABPS, ma anche attraverso scelte progettuali, come l'adozione delle librerie PjSip per l'implementazione delle sessioni di comunicazione.

È importante sottolineare come ABPS ricopra un ruolo determinate per gli scopi di questa tesi: l'intenzione di fondo era quella di arrivare ad interagire con tale architettura, implementando uno *Stateful Proxy* che fosse in grado di intervenire sui "pacchetti VoIP" (cioè sui protocolli RTP, SDP, SIP), modificando opportunamente gli indirizzi IP e le porte da usare nella sessione di comunicazione, per poi inoltrarli verso un proxy server ABPS, attraverso le interfacce di rete disponibili.

Eccone uno schema:

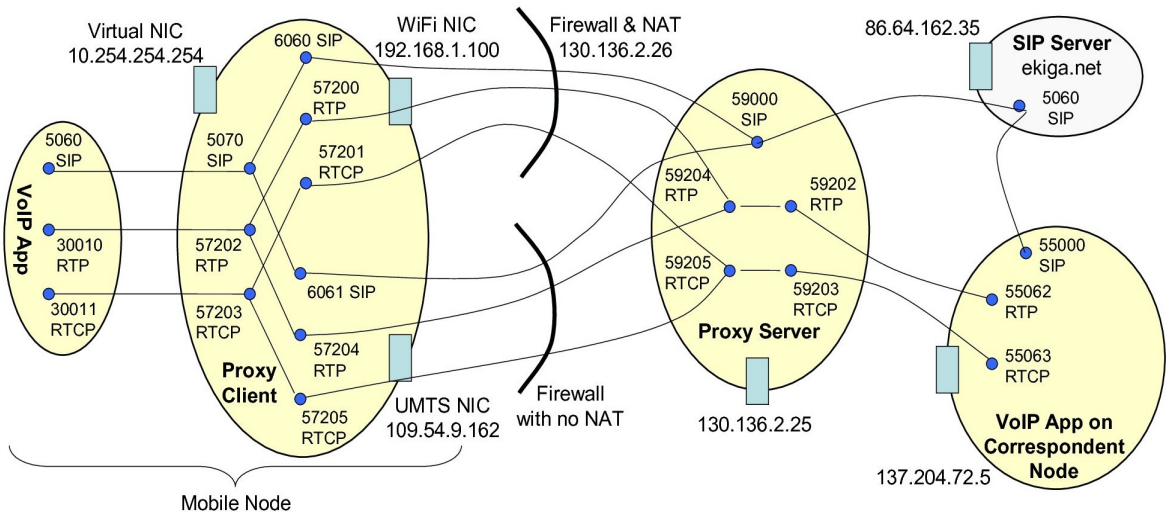

Figura 7: Possibile scenario di utilizzo

È possibile introdurre sul lato mobile una interfaccia di rete virtuale (o *Virtual NIC*), in modo tale da far disinteressare l'applicazione dalle interfacce fisiche che può usare per la connessione: tutti i pacchetti verranno inoltrati verso l'indirizzo di rete virtuale, soltanto il proxy sarà a conoscenza di quelli reali. La Virtual NIC funge da ponte di comunicazione tra l'applicazione e il Proxy Client, sarà poi quest'ultimo a preoccuparsi di applicare le opportune modifiche per raggiungere il Proxy Server.

L'applicazione da usare non subirà sostanziali modifiche, ma verrà semplicemente estesa con una funzionalità in più, permettendo all'utente di avere la possibilità di scegliere un “percorso alternativo” per instradare le sue chiamate e quindi decidere se usufruire o meno dei servizi e benefici che l'architettura ABPS offre.

Ma andiamo a vedere come e' possibile realizzare tutto ciò e con quali strumenti: nei paragrafi successivi verranno approfondite l'architettura ABPS e le librerie che sono state coinvolte per raggiungere i nostri scopi.

2.2 ABPS

Attualmente un dispositivo mobile è dotato di più interfacce di rete wireless eterogenee, denominate *NIC* (Network Interface Cards), che permettono la connessione a diverse reti wireless, come 3G e Wi-Fi. Solitamente però, le politiche di *Mobility Management* (ovvero "Gestione della Mobilita") non permettono ad un nodo mobile (MN) di usare tutti i NIC disponibili, né tantomeno di poter scegliere quello più appropriato, in base al suo contesto reale. Con il termine "più appropriato" si intendono tutti quei parametri, molto diversi tra loro, che possono condizionare la scelta del NIC, come ad esempio, il costo economico, la copertura di rete, la velocità di trasmissione, il QoS, la sicurezza e le preferenze dell'utente. [4]

In risposta a questa problematica, viene proposto il modello ABPS (*Alway Best Packet Switching*), che consente alle applicazioni di utilizzare contemporaneamente tutte le schede di rete disponibili e di inoltrare ogni datagram IP verso la schede di rete che in quel momento è più adatta.

I componenti principali di questa architettura sono:

- un **Proxy Server** fisso, che agisce come relè per il nodo di telefonia mobile e consente comunicazioni da/verso questo nodo, con lo scopo di superare possibili sistemi NAT e firewall
- un **Proxy Client** in esecuzione sul nodo mobile, che mantiene un tunnel multi-path con il Proxy Server descritto prima, attraverso tutte le schede NIC disponibili

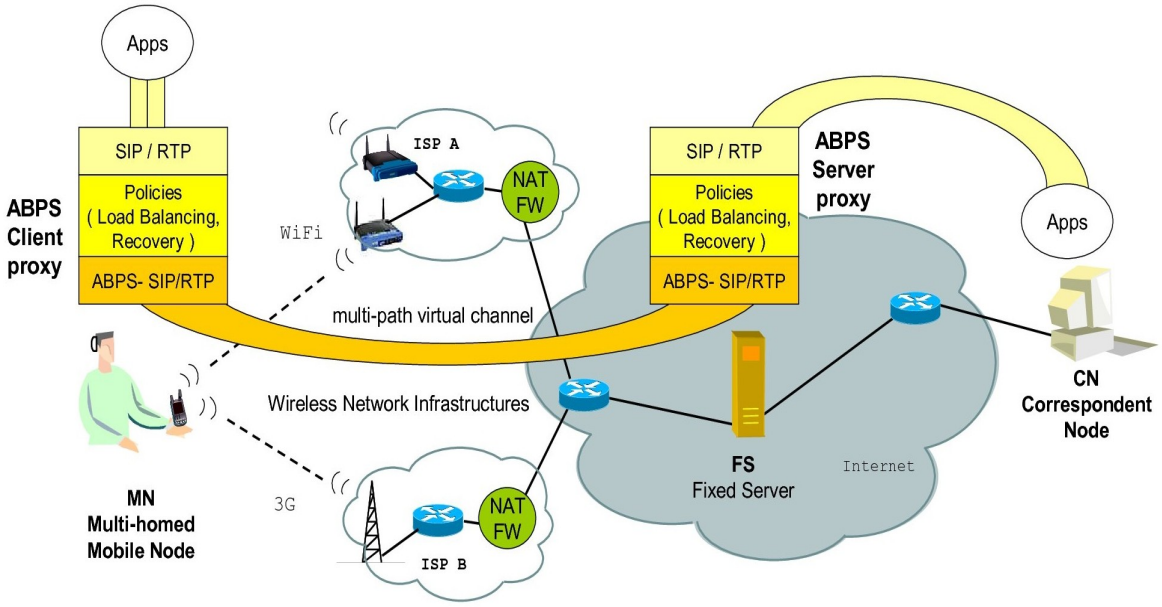

Figura 8: Architettura ABPS

Questo modello permette di creare un “canale logico” per la comunicazione, tra il nodo mobile ed il proxy server, in cui ogni pacchetto può essere identificato anche quando cambia l’indirizzo IP di provenienza, consentendo così alle applicazioni mobile di creare politiche per il *load balancing* ed il *recovery* della connessione.

ABPS adotta un particolare meccanismo *crosslayer,* che prende il nome di *Robust Wireless Multi-Path Channel*, o più semplicemente RWMPC, in grado di fornire una effettiva interoperabilità a bassa percentuale di pacchetti persi. Sfortunatamente lo studio approfondito di questo meccanismo esula dagli scopi della presente tesi, ma sottolineiamo la capacità di RWMPC, di garantire un delay pari a 150ms e circa il 3% dei pacchetti persi. Per avere informazioni maggiori e più dettagliate, rimandiamo alla lettura della documentazione ufficiale [4].

In conclusione, possiamo riassumere che ABPS e' una soluzione completa ed efficace di *terminal mobility* (per il VoIP), progettata in modo da supportare direttamente le applicazioni basate su protocolli SIP e RTP, senza che queste subiscano alcuna modifica, rispettando a pieno i requisiti QoS fondamentali per il Voice Over IP.

2.3 PjSip

PjSip è una suite di librerie Open Source (con licenza GPL 2.0 e scritte in C) strettamente connesse tra loro, ideata per lo sviluppo di applicazioni VoIP embedded e non. Nello specifico PjSip implementa un *SIP Stack* e un *Media Stack* per le comunicazioni multimediali basate su protocollo SIP/RTP, offrendo i seguenti vantaggi:

- **Portabilità:** è supportato da numerosi sistemi operativi, come Windows, Windows Mobile, Unix-likes, Symbian OS, Android OS, Mac OS, su architetture a 32e 64 bit, sia *little endian* che *big endian;*
- **FootPrint limitato:** la capacità di occupare poco spazio, siamo sull'ordine di pochi KB, lo rende adatto ai dispositivi come gli smartphone;
- **Alte prestazioni:** e' in grado di gestire più transazioni SIP contemporaneamente mantenendo un consumo minimo delle risorse, come la CPU;
- **Ampie funzionalità:** dispone di molte funzionalità ed estensioni SIP, come le connessioni multiple all'interno di uno stesso dialogo, i messaggi istantanei, il trasferimento di chiamata, etc....

L'architettura delle librerie PJSIP è altamente modulare e strutturata a livelli multipli, stratificati uno sopra l'altro. Riportiamo di seguito la descrizione solo di quelle più importanti:

- *PJSIP:* rappresenta uno stack SIP che supporta un insieme di features ed estensioni del protocollo SIP.
- *PJMEDIA:* è una piccola libreria estremamente portabile, che si occupa della gestione e del trasferimento dei dati multimediali.

- *PJLIB:* a questa libreria fanno rifermento tutte le altre, in quanto fornisce una astrazione ampia e completa delle sue funzionalità indipendentemente dal sistema operativo sulla quale poggia; Può essere considerata una revisione della libreria *libc*, con l'aggiunta di alcune features come la gestione dei socket, funzionalità di logging, gestione thread, mutua esclusione, semafori, critical section, gestione eccezioni e definizione di strutture dati di base (liste, stringhe, tabelle, ecc...).
- *PJSUA:* rappresenta il livello più alto della suite e funge da *wrapper* verso le altre librerie. Inoltre agevola notevolmente la scrittura di applicazioni, in quanto implementa direttamente UA (client e server)

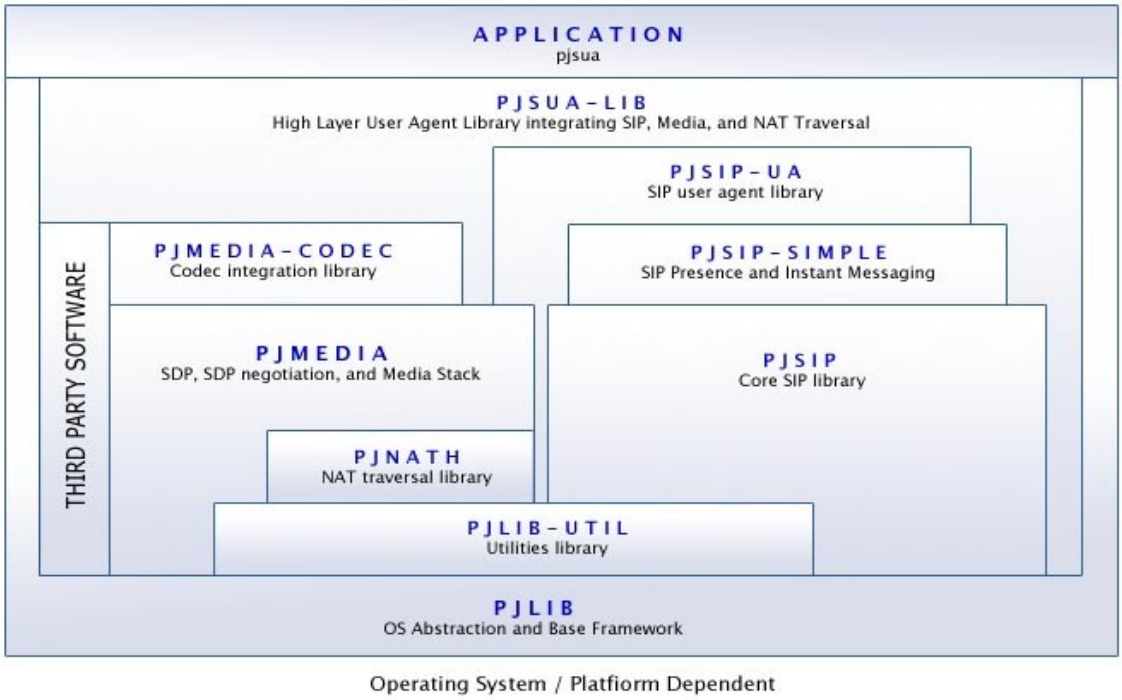

Figura 9: Stack delle librerie PJSIP

Tutti i componenti software in PJSIP, compreso il livello per le transazioni e quello per il dialogo, sono implementati come moduli. Senza questi il *Core Stack* non saprebbe come gestire i messaggi SIP. Il "cuore" di questa architettura è rappresentato dal *SIP_ENDPOINT,* che si occupa dei seguenti compiti: [5]

- gestisce una *Pool Factory* , allocando le pool per tutti moduli SIP;
- si occupa della temporizzazione (*Timer Heap*) e schedula gli eventi da notificare a tutti i moduli SIP;
- gestisce le varie istanze dei moduli di trasporto e controlla il parsing dei messaggi;
- gestisce i moduli PJSIP che sono la struttura primaria per poter estendere la libreria e fornire nuove funzionalità di parsing e trasporto;
- riceve messaggi SIP dal livello trasporto e li ridistribuisce ai moduli interessati;

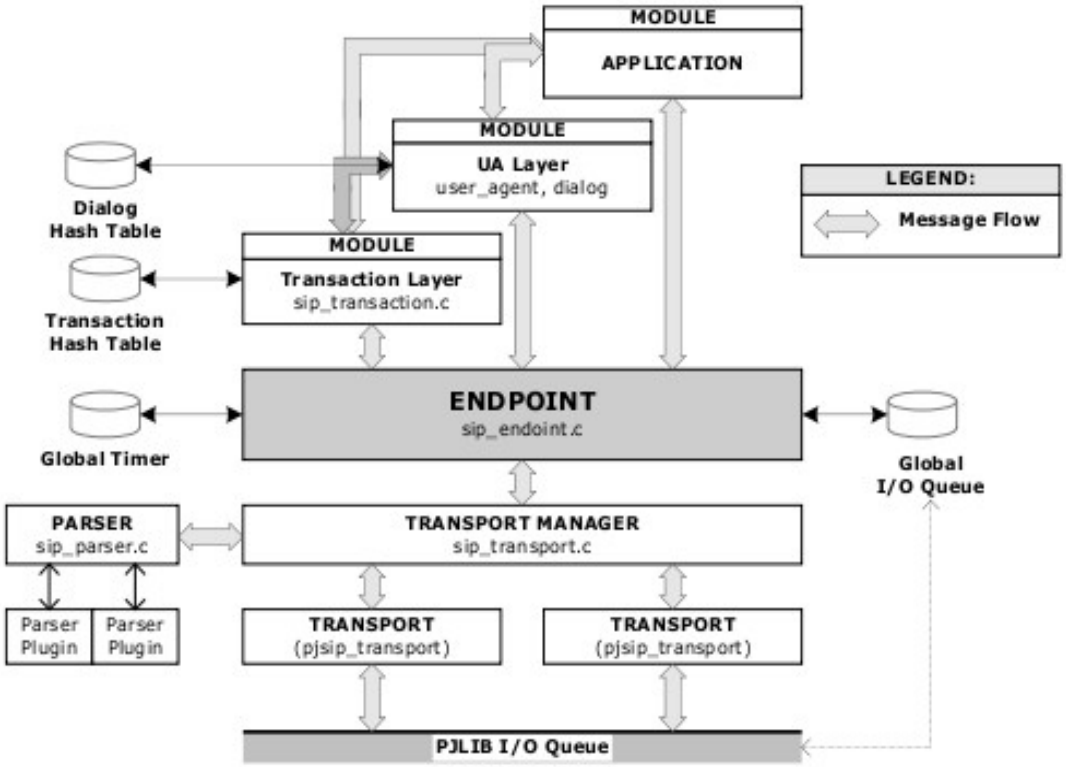

Figura 10: Diagramma di collaborazione

Il framework in questione consente la creazione di nuovi moduli per l'implementazione di nuove funzionalità, i quali occuperanno una posizione all'interno della struttura a livelli, in base al valore di priorità assegnatoli. Ecco come si presenta la struttura di un modulo:

```
struct pjsip_module
{
   PJ_DECL_LIST_MEMBER(struct pjsip_module);              // For internal list mgmt.
   pj_str_t             name;                             // Module name.
   int                  id;                               // Module ID, set by endpt
   int                  priority;                         // Priority

   pj_status_t (*load)          (pjsip_endpoint *endpt);  // Called to load the mod.
   pj_status_t (*start)         (void);                   // Called to start.
   pj_status_t (*stop)          (void);                   // Called top stop.
   pj_status_t (*unload)        (void);                   // Called before unload
   pj_bool_t   (*on_rx_request) (pjsip_rx_data *rdata);   // Called on rx request
   pj_bool_t   (*on_rx_response)(pjsip_rx_data *rdata);   // Called on rx response
   pj_status_t (*on_tx_request) (pjsip_tx_data *tdata);   // Called on tx request
   pj_status_t (*on_tx_response)(pjsip_tx_data *tdata);   // Called on tx request
   void        (*on_tsx_state)  (pjsip_transaction *tsx,  // Called on transaction
                                 pjsip_event *event);     // state changed
};
```

Figura 11: Modulo PJSIP

Le quattro funzioni, *load*, *start*, *stop* e *unload*, sono chiamati dall'endpoint per controllare lo stato dei moduli. Attraverso queste funzioni è possibile determinare il *ciclo di vita* di un modulo.

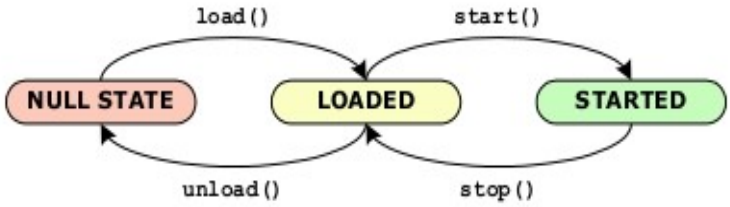

Figura 12: Ciclo di vita di un modulo

All'interno della struttura del modulo, troviamo anche dei puntatori a funzione che sono tutti opzionali: se non vengono specificati, il valore di ritorno sarà

SUCCESSFUL, altrimenti andranno a svolgere i loro rispettivi compiti, descritti qui di seguito:

- *ON_RX_REQUEST()* e *ON_RX_RESPONSE()* sono i mezzi principali per ricevere i messaggi dall'endpoint SIP o da altri moduli, dove il valore di ritorno è estremamente importante. Se un *callback* torna non zero (cioe *true*), significa che il modulo ha provveduto al messaggio e l'endpoint terminerà la distribuzione del messaggio ad altri moduli.

- *ON_TX_REQUEST()* e *ON_TX_RESPONSE()* sono chiamati dal gestore dei trasporti prima che un messaggio venga trasmesso: in questo modo si da la possibilità ad alcuni moduli (ad esempio quello dedito alla firma dei pacchetti) di fare un ultima modifica al messaggio. Tutti i moduli devo poi restituire come valore di ritorno PJ_SUCCESS, altrimenti la connessione verrà annulla.

- *ON_TSX_STATE()* viene utilizzato per ricevere una notifica ogni volta che lo stato della trasmissione viene cambiato: per esempio a causa del ricevimento di un messaggio, oppure dalla scadenza di alcuni timer o da errori dovuti al trasporto.

PJSIP si basa su di un semplice, ma molto potente, “concetto astratto di gerarchia”, in cui ogni modulo è identificato da un valore che ne indica la priorità. Questo valore specifica l'ordine con la quale i moduli vengono chiamati: più sarà basso il valore, maggiore priorità avrà il modulo. Lo schema seguente mostra i valori standard usati per impostare le priorità dei moduli.

```
enum pjsip_module_priority
{
   PJSIP_MOD_PRIORITY_TRANSPORT_LAYER   = 8,  // Transport
   PJSIP_MOD_PRIORITY_TSX_LAYER         = 16, // Transaction layer.
   PJSIP_MOD_PRIORITY_UA_PROXY_LAYER    = 32, // UA or proxy layer
   PJSIP_MOD_PRIORITY_DIALOG_USAGE      = 48, // Invite usage, event subscr. framework.
   PJSIP_MOD_PRIORITY_APPLICATION       = 64, // Application has lowest priority.
};
```

Figura 13: Priorità dei moduli

Una volta chiamato un modulo, questo andrà prima a richiamare rispettivamente le funzioni *on_rx_request()* e *on_rx_response()* per gestire i messaggi in entrata e soltanto dopo le funzioni *on_tx_request()* e *on_tx_response()* per i messaggi in uscita. Se un modulo invece desidera accedere alla struttura del messaggio, prima del livello trasporto, dovrà essere impostato una priorità maggiore.

Quando arriva un messaggio, questo viene rappresentato all'interno della struttura *pjsip_rd_data*. Il gestore dei trasporti analizza il messaggio, lo bufferizza in questa struttura e lo passa all'endpoint. Quest'ultimo distribuisce poi il messaggio a ciascun modulo registrato, chiamando *on_rx_request()* oppure *on_rx_response()* a partire dal modulo con priorità maggiore, cioè col valore di priorità più basso, finchè uno di loro non restituisce un valore diverso da zero (cioè *true*). In tal caso, l'endpoint interrompe la distribuzione del messaggio verso altri moduli, in quanto presuppone che il modulo sia occupato nel trattamento del messaggio. Il modulo che restituisce *true* a sua volta può inoltrare ulteriormente il messaggio ad altri moduli.

Il diagramma seguente ci mostra un esempio di come i moduli possono chiamare altri moduli “a cascata”.

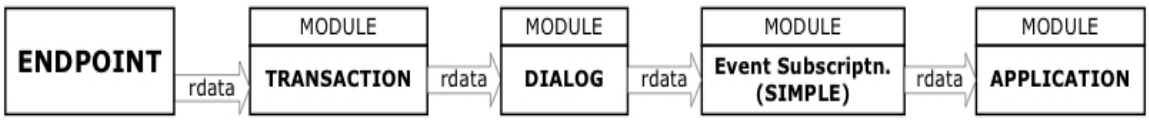

Figura 14: Cascade Callback

Per quanto riguarda i messaggi in uscita o di risposta, anche questi vengono rappresentati all'interno di una struttura, che prende il nome di *pjsip_tx_data*, la quale contiene anche un buffer contiguo, un pool per la memoria e informazioni di trasporto. Quando viene eseguita la funzione *pjsip_transport_send()* per inoltrare un messaggio, il gestore dei trasporti, chiama le funzioni *on_tx_request()* oppure *on_tx_response()*, per tutti i moduli, a partire ovviamente da quello con priorità maggiore.

Riassumendo, i messaggi in arrivo, vengono distribuiti a tutti i moduli registrati presso il SIP endpoint, partendo da quello con priorità maggiore, mentre i messaggi in uscita, prima di essere inoltrarli al destinatario, vengono distribuiti ai quei moduli che lo desiderano, per consentire loro di applicare le ultime modifiche ai messaggi stessi.

Per avere maggiori dettagli su come sono strutturate le librerie PJSIP, rimandiamo al sito web ufficiale (*http://www.pjsip.org*) e alla guida per gli sviluppatori (*PJSIP: Developer's Guide*)[5][6].

2.4 Architettura Android

Android è un sistema operativo per dispositivi mobili costituito da un vero e proprio stack di strumenti software che include un middleware per le comunicazioni, un insieme di librerie native, una Virtual Machine e alcune applicazioni di base.

Come ogni altra tecnologia, anche Android e' nata da una esigenza: quella di fornire una piattaforma aperta e per quanto possibile standardizzata, per la realizzazione di dispositivi mobili. Tra i suoi numerosi obbiettivi, evidenziamo la capacità di fornire tutto ciò di cui un operatore o un venditore o uno sviluppatore ha bisogno per raggiungere i propri interessi.

La fondamentale caratteristica di questo sistema operativo è quella di essere “open”, in quanto utilizza tecnologie Open Source*,* prime fra tutte il kernel Linux nella versione 2.6. Anche l'intero codice è sotto licenza *OpenSourceApacheLicense 2.0,* chiunque può consultarlo, che sia per semplice documentazione oppure per contribuire a migliorarlo.

È importante sottolineare che il linguaggio di programmazione adottato, per lo sviluppo di Android è *Java:* nel caso in cui Google avesse introdotto un nuovo linguaggio, avrebbe dovuto anche realizzare delle specifiche, un compilatore, un debugger, degli IDE opportuni oltre alle librerie con apposita documentazione. Ma non solo, la scelta di *Java* implicherebbe anche l'uso di una JVM (*Java Virtual Machine*) per l'esecuzione del proprio codice, andando così in contrasto con quella che è la “filosofia open” di cui parlavamo prima, perché significherebbe realizzare un sistema operativo che può essere usato liberamente, ma soltanto attraverso il pagamento di una *royality* alla società Sun Microsystem (ora Oracle), che detiene i diritti della piattaforma Java.

Come risposta a questo problema, Google ha adottato una propria VM che prende il nome di *Dalvik* (nome di una località islandese). Si tratta di una VM ottimizzata per l'esecuzione di applicazioni in dispositivi a risorse limitate, che esegue codice contenuto all'interno di file con estensione *.dex,* ottenuti a loro volta, in fase di

building, a partire da file *.class* di bytecode Java. Alla fine il file che verrà installato sul dispositivo fisico, si presenterà con estensione *.apk* (Android Package).

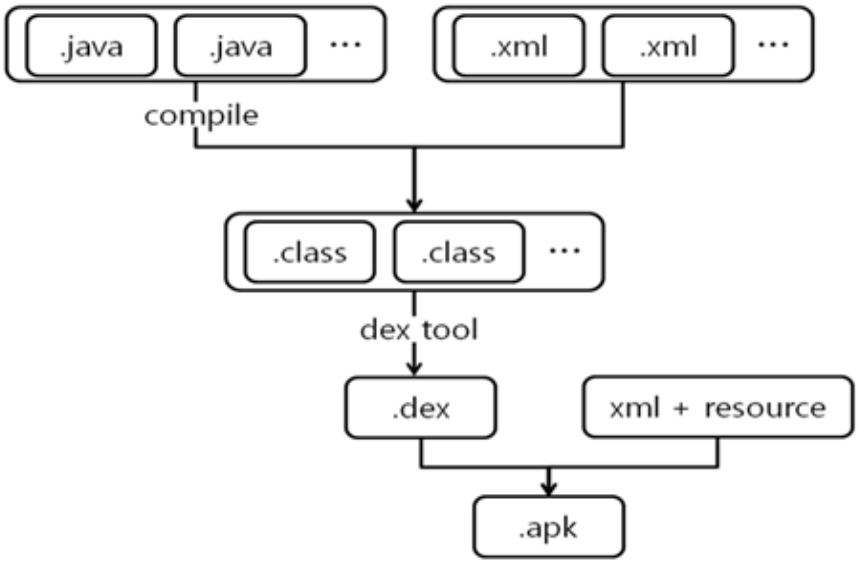

Figura 15: Da *java* ad *apk*

La Dalvik Virtual Machine dispone di alcune caratteristiche specifiche, che la distinguono dalle altre Virtual Machine:

- Un file *.dex* occupa circa il 50% in meno di spazio rispetto al corrispettivo file *.jar*
- solo dalla versione 2.2 diAndroid è stato introdotto un compilatore *Just In Timer* (JIT)
- L'insieme delle costanti è stato modificato per utilizzare un indirizzamento di soli 32-bit per semplificare l'interprete
- Usa il suo stesso *bytecode* e non quello di Java
- E' stata progettata per essere eseguita con più istanze contemporaneamente,

ogni applicazione Android viene avviata nella propria istanza della Dalvik VM.

- Non rispetta i profili delle librerie Java SE/ME, ma usa una sua libreria creata a partire dall'implementazione Java: Apache Harmony, un'implementazione Java free sotto licenza Apache 2.0.

Ora vediamo quali sono le caratteristiche di un'applicazione Android, dal punto di vista delle relazioni con il sistema operativo:

- Ogni esecuzione dell'applicazione genera un singolo processo Linux il quale viene terminato da Android non appena l'applicazione viene chiusa.
- Ogni processo possiede una propria Virtual Machine (necessaria per l'esecuzione di un file java) il quale ha il compito di gestire il processo. Ciò implica che ogni processo viene eseguito da Android in maniera autonoma, isolata dal resto delle altre operazioni e dalle routine di sistema.
- Ad ogni applicazione viene associata un User ID univoco. Questa caratteristica permette ai file delle applicazioni di essere visibili a tutte le altre applicazioni che hanno lo stesso User ID e all'applicazione stessa.

Soffermiamoci un attimo su quest'ultimo punto: le applicazioni in Android possono “interagire” con altre applicazioni, condividendo elementi in comune. Per esempio, se in un'applicazione avessimo realizzato una slideshow di immagini, prese tra le foto scattate con la fotocamera dall'utente, e in un'altra applicazione avessimo bisogno della medesima slideshow, non ci sarebbe bisogno di inserire all'interno della seconda applicazione (e quindi duplicare) il codice sorgente della slideshow: sarà sufficiente richiedere al sistema l'esecuzione della parte di codice relativo alla slideshow.

Solitamente quando si sviluppa una applicazione Android, vengono coinvolte quattro componenti differenti: le classi che verranno implementate andranno ad estendere

almeno uno delle seguenti classi esistenti:

1. SERVICES

   La peculiarità di questo componenete è quella di non avere nessuna interfaccia grafica (che invece possiede un'Activity); solitamente vengono eseguiti in background. Un esempio di *service* potrebbe essere la riproduzione musicale eseguita in background mentre l'utente compie altre azioni.

2. BROADCAST RECEIVERS

   Un componenete di questo tipo non deve svolgere un'operazione prestabilita, ma bensì rimane in ascolto e reagisce di conseguenza quando cattura un "avviso". Chi trasmette questi avvisi solitamente è il sistema operativo che notifica alle applicazioni un messaggio, come per esempio che è stato terminato il download di un file. Ovviamente nella nostra applicazione possiamo configurare i nostri ricevitori a reagire solamente ad alcuni tipi di annunci.

3. CONTENT PROVIDERS

   Il compito di questo componenete è quello di rendere disponibili un insieme di dati di una certa applicazione alle altre applicazioni. Solitamente i dati che vogliamo rendere, per così dire, "pubblici" vengono salvati o nel file system oppure in un database SQLite.

4. ACTIVITY

   A differenza delle altre questa dispone di una interfaccia grafica ed è lo strumento principale attraverso il quale l'utente interagisce con l'applicazione. Un'esempio di un' Activity è una lista di bottoni cliccabili in un menù dell'applicazione che permette ad un utente di compiere azioni diverse a

seconda del bottone cliccato. Siccome in un'applicazione possono coesistere più Activity, che sono delegate ad uno specifico compito, risulta evidente che ognuna è indipendente dalle altre. E' necessario però identificare una sorta di gerarchia tra loro.

A questo punto risulta ovvia un'altra proprietà delle applicazioni Android che è quella di non possedere un unico punto di accesso per l'esecuzione, come potrebbe essere per altri progetti, dove l'esecuzione è delegata esclusivamente alla funzione *main()*.

Dal punto di vista applicativo, è importante sapere e capire che le *Activity* seguono una logica "a stack": per tutte le applicazioni, viene tracciata la sequenza di visualizzazione delle Activity in uno stack. Ogni volta che una applicazione chiede di visualizzare una Activity questa viene posta in cima allo stack e quella che era precedentemente visualizzata diventa la penultima. Quindi in una applicazione, l'utente non fa altro che navigare da una Activity all'altra (sempre della stessa applicazione), visualizzando di volta in volta quella che serve in base alle esigenze.

Il "ciclo di vita" di una Activity e' organizzato in tre stati:

- **Attivo**: in questo stato l'Activity viene mostrata in primo piano sullo schermo e l'utente può interagire con l'interfaccia grafica mostrata.
- **In pausa**: l'Activity viene comunque mostrata a schermo, ma l'utente non può interagire con essa perchè un'altra Activity è stata sovrapposta alla primaria. Un classico esempio è quando viene ricevuta una chiamata dove l'Activity relativa alla chiamata viene sovrapposta a quella dell'applicazione che l'utente stava utilizzando.
- **Ferma**: si ricade in questo stato quando l'Activity viene completamente sovrapposta da un'altra e dunque non è più visibile all'utente. Di fatto l'Activity iniziale è nascosta all'utente e nel caso in cui la memoria stia per esaurirsi Android chiuderà l'Activity nascosta.

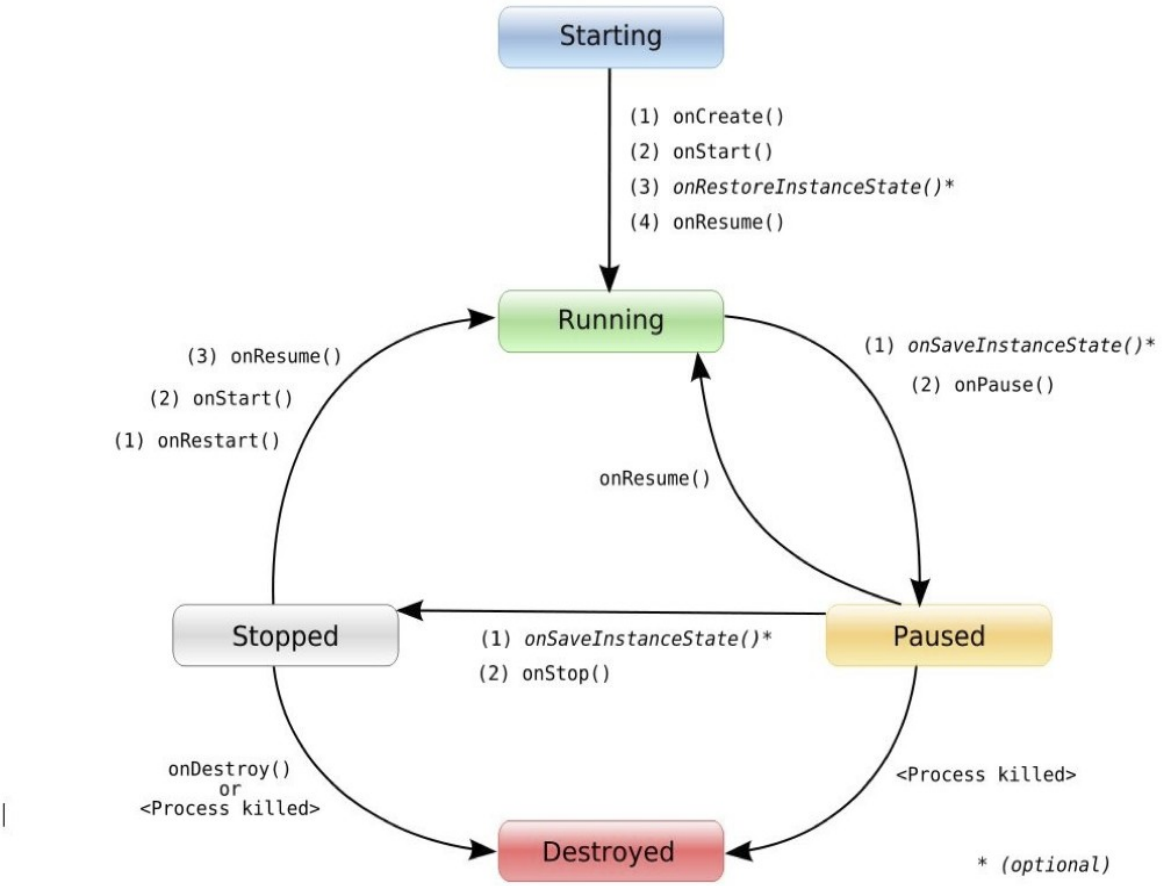

Figura 16: Ciclo di vita di una Activity

2.4.1 ADT

Google mette a disposizione diverse componenti per poter sviluppare una applicazione Android, più specificatamente, fornisce numerosi tools per poter estendere il proprio *ambiente di sviluppo integrato,*(acronimo inglese IDE). Nel mio caso la scelta dell'IDE è ricaduta su Eclipse, principalmente per due motivi: primo perché è *Open Source,* secondo perché Google ha sviluppato una serie di plug-in specifici per Eclipse, tutti riunificati in una suite che prende il nome di *Android Development Tool* (ADT).

ADT integra in Eclipse numerose *features,* che velocizzano lo sviluppo delle applicazioni, in particolar modo per quanto riguarda l'interfaccia grafica. Molte di queste sono “attività di lavoro” e processi che vengono eseguiti, spesso in background, per consentire il corretto sviluppo dell'applicazione. Inoltre vengono messe a disposizione anche due editor, uno *Java*, per la verifica della sintassi e per una corretta compilazione, l'altro *XML* che consente la creazione e modifica di layout grafici per l'utente, attraverso una interfaccia “drag and drop”.

Ogni applicazione Android, per poter essere sviluppata deve usufruire di un “kit di sviluppo”, fornito sempre da Google, che prende il nome di SDK (*Software Development Kit*), il quale fornisce tutti gli strumenti sufficienti per poter lavorare.

Ecco cosa contine:

- Il file *Android.jar* che contiene tutte le classi di Android, necessarie per la costruzione dell'applicazione.
- La documentazione viene fornita sia in locale che sul web, sottoforma di javadoc; comprende anche collegamenti con la comunità Android, per far riferimento a problematiche di sviluppo.
- Viene fornito tutto il codice sorgente, che comprende anche applicazioni demo.
- Tutti gli strumenti da riga di comando per creare applicazioni Android, tra i quali ricordiamo il compilatore ANT e Android Debug Brige (o ADB).
- Una directory contenente i driver necessari per fare girare l'applicazione su architetture differenti
- un *emulatore* sulla quale e' possibile testare direttamente le applicazioni sviluppate (ma anche quelle già esistenti) sul proprio computer, senza l'uso di un dispositivo fisico, ma purtroppo con qualche inconveniente: le conversazioni telefoniche non possono essere effettuate, cosi come non è possibile usare la fotocamera, i diversi sensori e altri strumenti hardware.

Google ha messo a disposizione anche un altro “kit per lo sviluppo”, ovvero il *Native Development Kit*, acronimo NDK. Come suggerisce il nome, questo kit permette agli sviluppatori di scrivere porzioni di codice utilizzando i linguaggi nativi di Android, come il C ed il C++, ma anche le librerie *libc*, *libm* (librerie matematiche), *libz* (per la compressione) e *liblog* (utilizata per inviare messaggi *logcat* al kernel).

Il principale vantaggio nell'utilizzare codice nativo è dato dalla possibilità di bypassare tutti i controlli e le mediazioni della virtual machine di Android, sfruttando più a fondo la risorse. NDK serve soprattutto quando si ha necessità di scrivere applicazioni che facciano un uso intensivo della CPU moderando, ad esempio, l'uso della RAM, oppure se si vuole riutilizzare codice C/C++ già esistente.

È importante sottolineare come NDK lavori in stretto contatto con JNI, ovvero Java Native Interface, un framework di programmazione con lo scopo di permettere ad applicazioni Java di interfacciarsi con funzioni scritte in altri linguaggi di programmazione, come C/C++ o Assembly.

JNI non solo consente il *wrapping* dal codice nativo a java, ma permette anche l'operazione inversa: è possibile definire classi in Java, che delegano poi al codice nativo l'implementazione dei loro metodi, cosi come l'invocazione di codice Java da parte di programmi scritti in linguaggio nativo.

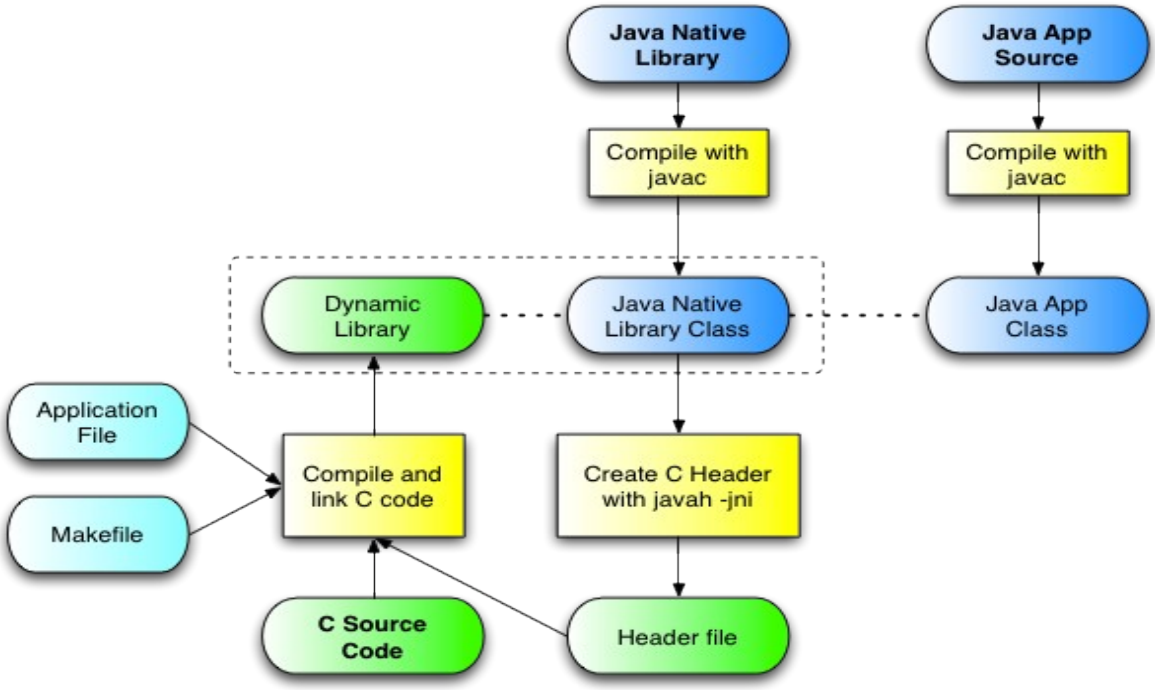

Figura 17: Schema NDK

Capitolo 3

Progettazione

3.1 Un Nuovo Scenario

La progettazione di questa tesi è andata progredendo per “fasi” (o step), dove le librerie PJSIP sono state il primo e vero punto di riferimento, nonché il mezzo con la quale raggiungere il nostro fine, ovvero l'architettura ABPS.

Per prima cosa sono stati effettuati dei “Test & Check” per valutare quel fosse l'applicazione migliore per Android. Più specificatamente, è stato cercato un *Software Sip Open Source* implementato con librerie PJSIP. Sebbene le applicazioni che ricadono in questo contesto sono molto poche e, più o meno, presentano le medesime caratteristiche, è stata preferita tra tutte l'applicazione *CSipSimple,* grazie alla sua maggiore configurabilità e sopratutto efficienza.

Una volta trovata l'applicazione sono stati pensati tre ipotetici scenari, tutti potenzialmente validi per raggiungere il nostro scopo, ma come è doveroso sottolineare, sono tutti scenari studiati da un punto di vista strettamente teorico. Come vedremo più avanti, le problematiche di sviluppo saranno numerose e di diversa natura, e porteranno ad una soluzione alternativa.

- *Scenario Uno*

  L'applicazione, implementata con librerie PJSIP, viene integrata ed estesa con delle funzionalità di proxy, così che possa comunicare soltanto attraverso una interfaccia virtuale: in altre parole, non viene implementato un proxy, ma un “meccanismo di proxy”, molto simile ad uno switch, il quale si preoccupa dell'instradamento dei messaggi VoIP tra l'interfaccia virtuale e le reali interfacce fisiche.

- *Scenario Due*

  In questo caso l'applicazione con PJSIP viene affiancata da un Proxy Client, già esistente e funzionante, la cui particolarità sarà quella di essere configurabile nel dettaglio, così da poter inoltrare i messaggi VoIP verso il Proxy Server esterno (nel nostro caso ABPS).

- *Scenario Tre*

  Anche in questo caso, l'applicazione viene affiancata da un Proxy Client, ma a differenza degli altri scenari, sarà possibile usare una applicazione qualsiasi, a patto che sia configurabile per l'uso di uno proxy esterno. Invece il proxy che viene introdotto, dovrà essere implementato con librerie PJSIP, oppure, nella peggiore delle ipotesi, bisognerà adattare un proxy gia esistente a tal contesto.

Come possiamo vedere, il primo scenario descrive più uno stratagemma, per “ingannare” l'applicazione, mentre gli altri due, fanno riferimento ad un vero e proprio Proxy Client. Sfortunatamente gli unici Proxy a disposizione, sono quelli HTTP e FTP: ancora non esistono, purtroppo, SIP Proxy Client open source per dispositivi mobile.

Difronte a questo problema, gli scenari descritti sopra, vengono fortemente

compromessi, in particolar modo gli ultimi due, lasciando come unica alternativa lo sviluppo del primo. Tuttavia la soluzione che viene proposta è quella di unificare il secondo scenario con il terzo, ottenendone uno nuovo, in cui l'applicazione di partenza dovrà essere implementata con librerie PJSIP ed il nodo mobile dovrà includere uno *Stateful Proxy Client SIP/RTP*, scritto anch'esso, preferibilmente, con PJSIP.

Lo scenario di comunicazione che si vuole ottenere sarà diverso dal *SIP Trapezoid* descritto nel primo capitolo, in cui i parametri da usare nella sessione SIP venivano negoziati con l'aiuto dei proxy e la comunicazione RTP, avveniva in maniera *peer-to-peer*, tra i partecipanti coinvolti.

Il proxy in questione avrà la peculiarità di essere un proxy di tipo SIP/RTP, cioè in grado di intervenire sia sui pacchetti SIP che su quelli RTP, partecipando attivamente nella comunicazione VoIP: una volta aperte le porte da usare nella sessione di comunicazione, il proxy andrà a modificare tutti i pacchetti in transito, sostituendo con i propri valori, l'indirizzo IP e le porte che identificano il nodo precedente (il quale può essere un endopoint o un altro proxy). In questo modo gli User Agent non effettueranno una comunicazione diretta, come nel caso del trapezoide, perché i pacchetti RTP seguiranno lo stesso tragitto percorso dai pacchetti SIP, passando per ogni nodo coinvolto,

Ecco un semplice schema descrittivo:

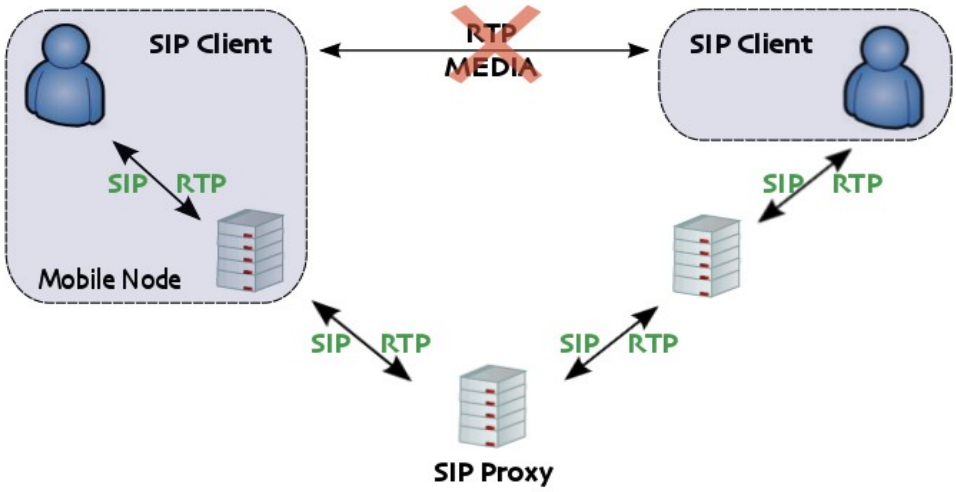

Figura 18: Nuovo Scenario

Per poter implementare un proxy di questo tipo, sono stati considerati due approcci:

- uno, che sfrutti le potenzialità del sistema operativo Android, per incorporare e adattare un Proxy SIP/RTP già esistente, come *Siproxd*;
- l'altro, attraverso le librerie PJSIP, che mettono a disposizione dei moduli specifici per la creazione e gestione di proxy;.

3.2 Stateful Proxy

Come definito dal RFC 3261 [13], un SIP Proxy è l'unico elemento standard SIP a non essere un endpoint, il cui compito è quello di inoltrare una richiesta verso una o più destinazioni, raccogliere le risposte da ciascun destinatario e notificarle al mettente.

Sempre l' RFC 3261 definisce e descrive due modalità di funzionamento dei proxy, facendo riferimento a quelle che sono le transazioni in una sessione SIP, distinguendole in transazioni stateless e transazioni stateful.

Si parla quindi di *Statefull Proxy* quando si mantiene lo stato delle transazioni (non della chiamata): il proxy mantiene in memoria le richieste, in modo da poter associare le risposte del destinatario col mittente e viceversa. Invece si parla di *Stateless Proxy,* quando il proxy non memorizza lo stato delle transazioni e solitamente, a differenza dell'altro, non ha bisogno di usare il protocollo TCP e fare operazioni di multicast e forking.

3.2.1 Siproxd

Siproxd è un *Deamon Proxy Masquerading* , scritto in C, per il protocollo SIP, cioè un proxy in grado di gestire la registrazione di utenti SIP, su di una rete IP privata, consentendo ai rispettivi client SIP di instaurare connessioni VoIP anche dietro il mascheramento di un firewall o di un router NAT, attraverso la riscrittura dei messaggi SIP stessi.

Questo tipo di proxy viene posto in uscita tra il client locale SIP ed il client remoto (o un altro eventuale proxy); Siproxd non solo consente la riscrittura "al volo" dei messaggi SIP, ma implementa anche funzionalità di proxy RTP, per il traffico dei dati multimediali, sia in entrata che in uscita. L'intervallo delle porte che viene usato per la ricezione dei dati RTP è configurabile in modo tale da consentire al Firewall il traffico in entrata solo per un piccolo range di porte.

Siproxd non si propone come una soluzione ai problemi di NAT traversal o Firewall, ma come una alternativa, uno strumento in grado di adattarsi ad un determinato contesto o scenario.

La fase di compilazione ed installazione non sono per nulla complesse; ad ogni demone, e quindi proxy che verrà eseguito, sarà associato un file di configurazione che ne descrive i comportamenti, le policy ed il ruolo che deve assumere. L'unico prerequisito è quello di lavorare su sistemi operativi Unix-like in grado di usare le librerie *libosip2*.

Purtroppo, proprio per questo motivo, l'approccio con Siproxd è stato scartato, poiché le librerie in questione non sono supportate dal sistema operativo Android.

3.2.2 Proxy PJSIP

Come abbiamo visto nel capitolo precedente, grazie alla stratificazione di PJSIP, è

possibile lavorare a diverse profondità, attraverso i moduli. Le librerie PJSIP, oltre ai moduli principali forniti, mettono a disposizione per gli sviluppatori, anche degli esempi di applicazione, per testare le proprie funzionalità o per implementarne di nuove, tra cui anche un "prototipo" di S*tateful Proxy Forwarding.*

Il modulo che viene proposto da PJSIP implementa già funzionalità di Stateful Proxy, seppur incomplete: il codice seguente mostra uno "scheletro" delle funzioni *on_rx_request* e *on_rx_response* che il modulo andrà ad usare.

```
// This is our proxy module.
extern pjsip_module proxy_module;

static pj_bool_t on_rx_request( pjsip_rx_data *rdata )
{
   pjsip_account *acc;
   pjsip_uri *dest;
   pjsip_transaction *uas_tsx, *uac_tsx;
   pjsip_tx_data *tdata;
   pj_status_t status;

   // Find the account specified in the request.
   acc = ...

   // Respond statelessly with 404/Not Found if account can not be found.
   if (!acc) {
      ...
      return PJ_TRUE;
   }
   // Set destination URI from account's contact list that has highest priority.
   dest = ...

   // Create UAS transaction
   status = pjsip_endpt_create_uas_tsx( endpt, &proxy_module, rdata, &uas_tsx);

   // Copy request to new tdata with new target URI.
   status = pjsip_endpt_create_request_fwd( endpt, rdata, dest, NULL, 0, &tdata);

   // Create new UAC transaction.
   status = pjsip_endpt_create_uac_tsx( endpt, &proxy_module, tdata, &uac_tsx );

   // "Associate" UAS and UAC transaction
   uac_tsx->mod_data[ proxy_module.id] = (void*)uas_tsx;
   uas_tsx->mod_data[ proxy_module.id] = (void*)uac_tsx;

   // Forward message to UAC side
   status = pjsip_tsx_send_msg( uac_tsx, tdata );
   return PJ_TRUE;
}
```

Figura 19-A: Richiesta [5]

```
static pj_bool_t on_rx_response( pjsip_rx_data *rdata )
{
   pjsip_transaction *tsx;
   pjsip_tx_data *tdata;
   pj_status_t status;

   // Get transaction object in rdata.
   tsx = pjsip_rdata_get_tsx( rdata );

   // Check that this transaction was created by the proxy
   if (tsx->tsx_user == &proxy_module) {
      // Get the peer UAC transaction.
      pjsip_transaction *uas_tsx;
      uas_tsx = (pjsip_transaction*) tsx->mod_data[proxy_module.id];

      // Check top-most Via is ours
      ...
      // Strip top-most Via
      // Note that after this code, rdata->msg_info.via is invalid.
      pj_list_erase(rdata->msg_info.via);
      // Code above is equal to:
      //  pjsip_hdr *via = pjsip_msg_find_hdr(rdata->msg, PJSIP_H_VIA);
      //  pj_list_erase(via);

      // Copy the response msg.
      status = pjsip_endpt_create_response_fwd( endpt, rdata, 0, &tdata);

      // Forward the response upstream.
      pjsip_tsx_send_msg( uas_tsx, tdata );

      return PJ_TRUE;
   }
   ...
}
```

Figura 19-B: Risposta [5]

Il proxy in questione, così come viene fornito, si comporta come un *Proxy Trasparente*, ovvero crea un UAC e un UAS per le transazioni ed inoltra tutte le risposte dall' UAC verso l' UAS, occupandosi solo del timeout della transazione, senza intervenire sui messaggi. Ecco la sequenza delle operazioni che compie:

- *Init_Stack()*, inizializza lo stack, cioè registra i moduli, con la quale andrà ad interagire, presso l'endpoint;
- *Init_Proxy(),* setta le porte e gli indirizzi di rete da usare, con eventuali rispettivi alias;

- *Init_Stateful_Proxy(),* registra altri moduli, sempre presso l'endpoint, spefici per le transazioni di tipo stateful;
- *Thread_Create()*, viene eseguito in background il thread, che resta in attesa di eventuali transazioni.

Nonostante questo modulo, così proposto, si presenti incompleto, perché gestisce solo transazioni di tipo SIP/SDP, rimane un ottimo punto di partenza, per poter raggiungere il nostro obbiettivo, grazie alla possibilità di estenderlo con altri moduli, specifici per la gestione delle transazioni RTP.

Questi moduli li troviamo in PJMEDIA, che si preoccupa di gestire le sessioni multimediali, e sono:

- SDP Negotiation State Machine (offer/answare model, RFC 3264);
- SDP Parsing and Data Struct;
- RTP Session and Encapsulation (RFC 3550);
- RTCP Session and Encapsulation (RFC 3550);
- RTCP Extende Report (RFC 3611)

Per quanto riguarda il modulo *RTP Session and Encapsulation* le funzioni che mette a disposizione garantiscono:

- una semplice ma completa gestione delle sessioni RTP;
- la creazione di intestazioni RTP per ogni pacchetto in uscita;
- La codifica dei pacchetti RTP, nell'header RTP e in playload;
- non utilizza alcuna memoria dinamica;

Una volta inizializzata una sessione RTP, attraverso la funzione *pjmedia_rtp_session_init()*, l'applicazione per inviare un pacchetto RTP dovrà prima creare l'intestazione RTP (che non rappresenta l'intero pacchetto) e poi decidere se concatenarlo con il *playload* oppure inviare i due frammenti (intestazione e playload) utilizzando delle API per il trasporto che usino la tecnica *scatter-gather* (ad esempio sendv())[5].

Quando invece l'applicazione riceve un pacchetto RTP, deve per prima decodificarlo attraverso la funzione *pjmedia_rtp_decode_rtp()*, separando cosi l'intestazione dal playload per poi elaborarlo attraverso la funzione *pjmedia_rtp_session_update()*. La funzione di decodifica viene garantita per far puntare il playload nella corretta posizione, indipendentemente da altri eventuali opzioni presenti nel pacchetto RTP.

3.3 Considerazioni e Sviluppi Futuri

Come abbiamo potuto vedere, la tecnologia VoIP abbraccia numerosi contesti con diverse sfaccettature, che sono sempre in continuo cambiamento ed evoluzione.

Personalmente ho potuto assistere, durante un breve lasso di tempo di circa sei mesi, alla nascita e crescita di numerose applicazioni dedite al mondo della telefonia VoIP: dalle singole applicazioni SIP Client fino ad intere infrastrutture aziendali.

Ma non solo, anche le componenti VoIP qui spiegate e chiamate in causa per raggiungere l'obbiettivo, hanno subito aggiornamenti e modifiche continue, a tal punto da condizionare gli sviluppi della tesi stessa, sopratutto in fase di progettazione.

Basti pensare che l'applicazione *Siproxd*, scartata per via della mancanza di compatibilità tra le librerie che implementava ed il sistema operativo Android, è stata aggiornata di recente, con una nuova versione delle librerie *libosip2* (attualmente alla versione 3.5.0), riuscendo così ad integrarsi sia su dispositivi con Android, che su iPhone (a patto che venga usato un meccanismo di supporto TCP/TLS in background).

Anche gli altri componenti studiati, dall'architettura Android, fino alle librerie PJSIP, sono in costante evoluzione, in particolar modo l'applicazione *CSipSimple*, che rappresenta il nostro SIP Client, ha apportato, proprio negli ultimi mesi, un drastico cambiamento alla sua intera struttura: non solo sono state cambiate le parentele tra classi e alcune delle principali strutture, ma l'attuale versione è fortemente dipendente da una nuova libreria che è stata introdotta appositamente per la gestione dell'interfaccia grafica. In questo modo, l'applicazione ha guadagnato in efficienza e funzionalità, grazie all'uso separato delle librerie, perdendo però la compatibilità con le versioni precedenti.

Sebbene, l'obbiettivo di implementare uno *Stateful Proxy* non è stato raggiunto, sia per mancanza di tempo, ma sopratutto a causa dei repentini cambiamenti dello scenario VoIP studiato, possiamo suggerire agli sviluppatori futuri, che vogliono mantenere l'architettura ABPS come punto di riferimento, di usufruire appieno delle librerie PJSIP e dei mezzi che mette a disposizione.

Conclusioni

La tecnologia VoIP, i dispositivi mobile, il multihoming le sue problematiche fin qui introdotte e spiegate, sono un insieme di tecnologie in continua evoluzione, giunte ormai a disposizione di tutti.

Nonostante l'intenzione di fondo, di queste tecnologie, sia sempre stato quello di garantire le migliori prestazioni con minor problemi possibili, con la presente tesi di laurea, si voleva suggerire un metodo alternativo per risolvere gran parte dei problemi legati al VoIP. È stata presentata l'architettura ABPS, in grado di garantire ad un *device multi-homed* in movimento, la possibilità di effettuare traffico VoIP, sfruttando al meglio le potenzialità offerte dalle interfacce multiple a disposizione, che gli consentono di rientrare nei limiti richiesti per garantire la QoS necessaria ad applicazioni multimediali interattive.

Sebbene l'obbiettivo sperato non è stato raggiunto, questo lavoro mi ha permesso di approfondire le conoscenze relative alla tecnologia VoIP, toccando con mano gran parte dei componenti di tale architettura, come proxy e user agent, monitorando le loro funzionalità. Ho avuto anche la possibilità di studiare nel dettaglio l'architettura Android e acquisire familiarità con il suo funzionamento, attraverso l'implementazione di applicazioni e l'uso di librerie native, per non parlare delle librerie PJSIP, che rappresentano il vero mezzo da usare per interagire con l'architettura ABPS.

In conclusione, spero che questa tesi, oltre ad aver ampliato il mio bagaglio culturale e di esperienze, risulti utile in futuro anche per tutti coloro che vogliono avvicinarsi a tal contesto e partecipare alla risoluzione di alcuni dei problemi che ruotano attorno al VoIP.

Bibliografia

1. Wikipedia, enciclopedia libera – *http://wikipedia.org* (+ riferimenti interni!!)
2. Eyeball Networks, *"NAT Traversal for VoIP and Internet Communications using STUN, TURN and ICE"*, 2011 - *www.eyeball.com/.../anyfirewallwhitepaper.pdf*
3. Dott. Alessandro Falaschi, Università di Roma - *http://infocom.uniroma1.it/alef/labint/index.html*
4. *V. Ghini, S. Ferretti, F.Panzieri, "The Always Best Packet Switching architecture for SIP-based mobile multimedia services"*
5. PJSIP's website, "*an Open Source SIP and Media stack"* - *http://www.pjsip.org/*
6. PJSIPs website, *"PJSIP Developer's Guide"* , version 0.5.4
7. Thomas Ries, Siproxd Guide, *"Siproxd: a masquerading SIP proxy"*
8. Google Inc., *"Android SDK"* - *http://developer.android.com/sdk/index.html*
9. Google Inc., *"Android NDK"* - *http://developer.android.com/sdk/ndk/index.html*
10. Google Inc., *"Android Development Tools"* - *http://developer.android.com/guide/developing/tools/adt.html*
11. RTP, *Real Time Protocol,* RFC 3550- *http://tools.ietf.org/html/rfc3550*
12. SDP, *Session Description Protocol,* RFC 2327 - *http://tools.ietf.org/html/rfc2327*

13. SIP, *Session Initiation Protocol,* RFC 3261 - *http://tools.ietf.org/html/rfc3261*

14. RTSP, Real Time Streaming Protocol, RFC 2326 - *http://tools.ietf.org/html/rfc2326*