Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Triennale in Informatica

Proxy SIP/RTP/RTCP a supporto della mobilità: Prestazioni.

Tesi di Laurea in Reti di Calcolatori

Relatore:
Chiar.mo Prof.
Vittorio Ghini

Presentata da:
Luca Montanari

Sessione II
Anno Accademico 2012/2013

Ringraziamenti

Vorrei ringraziare innanzitutto la mia famiglia che mi ha sostenuto moralmente ed economicamente durante i mie studi, permettendomi di arrivare a tale traguardo.
Voglio ringraziare la mia ragazza il cui contributo è stato fondamentale per affrontare lo stress e la tensione degli esami più difficili.
Ringrazio anche tutti gli amici e compagni di studio, con la quale ho condiviso questa esperienza universitaria ma in particolar modo il mio relatore, il dott. Vittorio Ghini, che mi ha guidato e aiutato durante lo sviluppo della presente tesi.
Un pensiero speciale per mio nonno Gianni che mi ha sempre sostenuto e che avrebbe tanto voluto condividere questo traguardo assieme a me.

Indice

Introduzione

Negli ultimi anni abbiamo assistito ad una crescente diffusione della rete Internet a banda larga e ad 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 soprattutto grazie all'introduzione ed alla ricerca, a partire dagli anni '90, della tecnologia *Voice over IP* (acronimo VoIP) 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 della libreria PJSIP. Scopo del progetto, condotto in collaborazione con il mio collega Michele Fabbri, è la creazione di un primo prototipo di Outbound Proxy Server per un dispositivo VoIP.

Nel primo capitolo verrà presentata la tecnologia in questione e spiegato il suo funzionamento, assieme alla descrizione dei protocolli coinvolti. Verrà poi mostrato un tipico scenario di comunicazione, nel quale 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 approfondito l'obiettivo, descritto lo strumento usato per raggiungere il nostro scopo, la libreria PJSIP, e come configurare l'ambiente di lavoro.

Nel terzo e quarto capitolo si parlerà invece dell'architettura e dell'implementazione di ciò che è stato realizzato.

Infine nel quinto ed ultimo capitolo si parlerà di una funzionalità non ancora completamente perfezionata: la funzione di outbound.

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 in grado di tramutare il segnale della voce analogico in un flusso di dati digitali, rendendo così 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.

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), 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 il quale siano stati trasmessi. Infatti può capitare 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:

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

Arriviamo alla Telefonia “Voice Over Internet Protocol”, che necessita, per poter operare 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 digitali 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.

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 arriva 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), 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.
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 Client

(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.

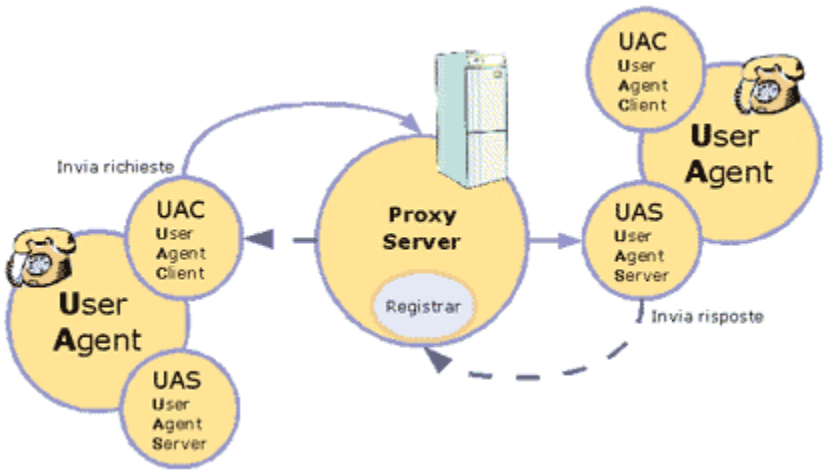


Figura 1: Architettura VoIP

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:

- 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 StateLess, ossia senza conservare traccia dei messaggi già inoltrati, oppure StateFull, 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. 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:

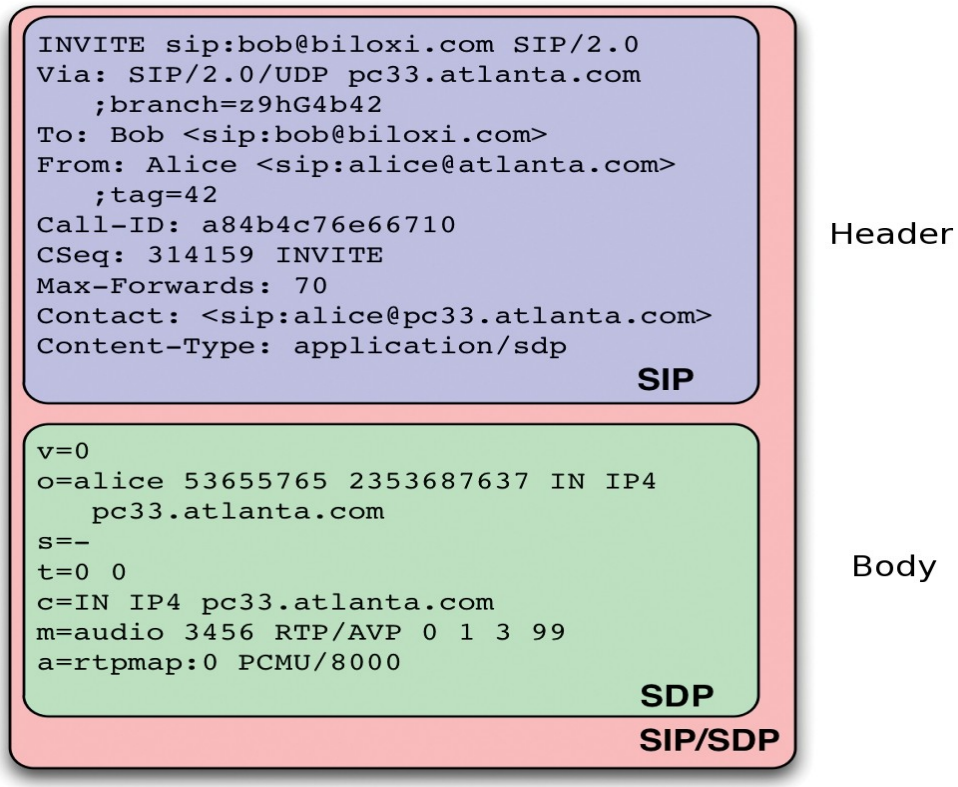


Figura 2: Pacchetto SIP/SDP

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.

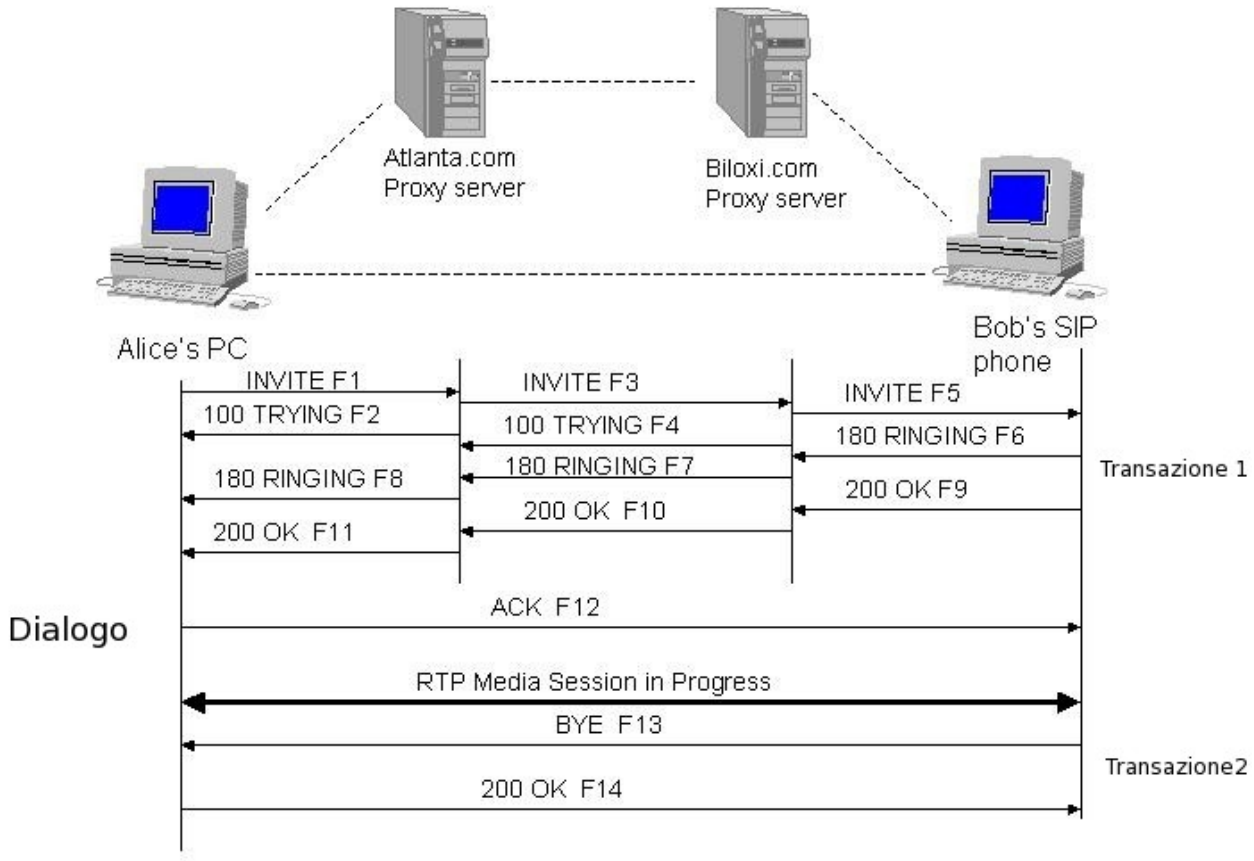


Figura 3: Trapezoide SIP

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:

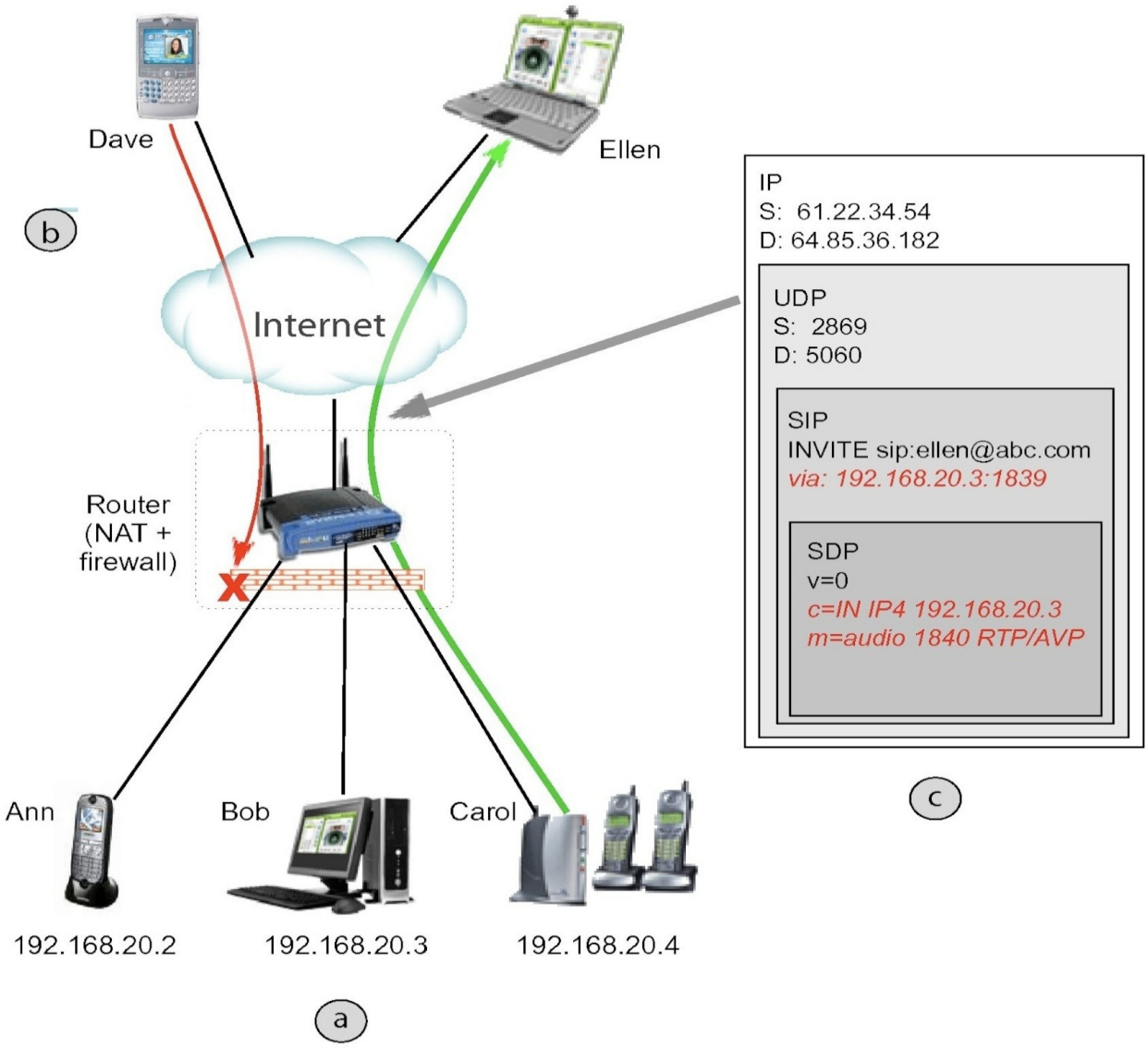


Figura 4: Scenario VoIP

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 sottorete;

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 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.
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.
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 ha 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.

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.

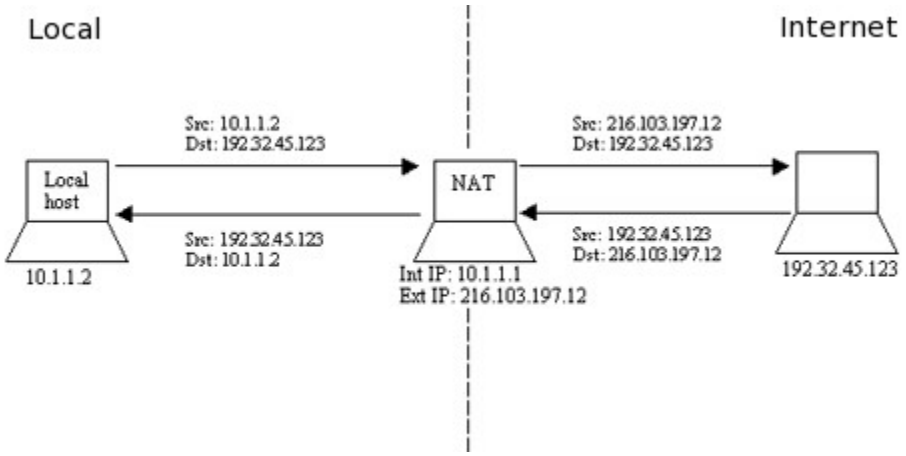


Figura 5: Meccanismo NAT

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 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'uso di un proxy esterno che funga da Relay.

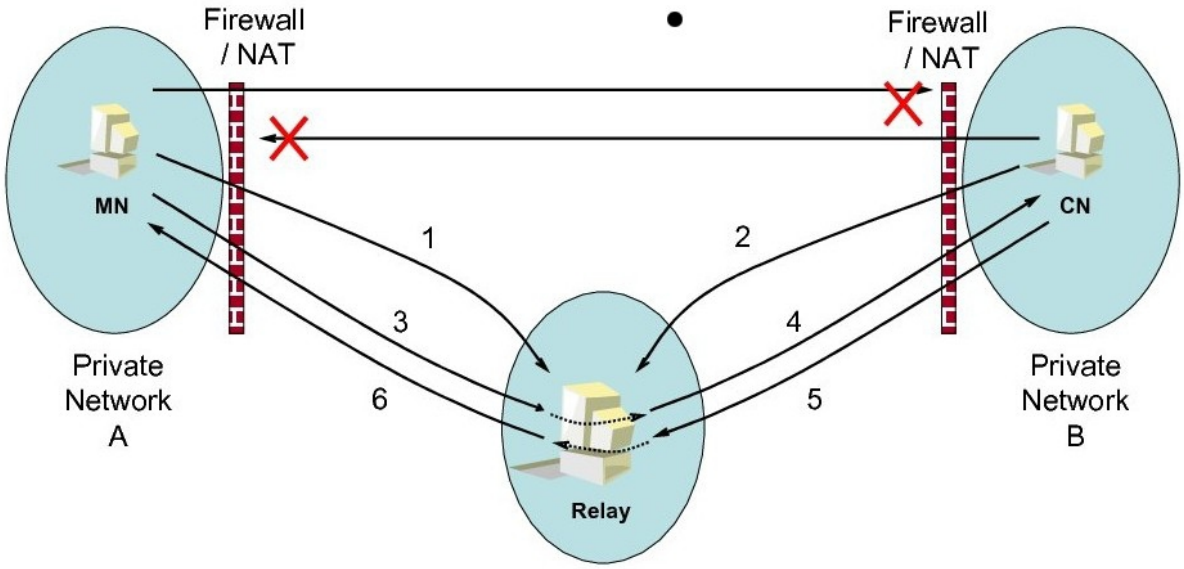


Figura 6: Relay

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 già 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

Obiettivi e Strumenti

2.1 Obiettivo

Immaginiamo uno scenario in cui un dispositivo mobile, con multiple interfacce di rete, con installata un'applicazione VoIP voglia comunicare con un altro dispositivo. Se almeno una di queste interfacce di rete è connessa allora il client sarà in grado di effettuare la chiamata. Ma cosa accade se l'interfaccia con cui è connesso non diventa più disponibile?

Non importa che siano o meno disponibili altre interfacce di rete, la comunicazione in ogni caso verrà interrotta. Impedire che ciò avvenga è l'obiettivo ultimo del nostro progetto. Si vuole infatti trovare un modo per cambiare interfaccia di rete senza che cada la comunicazione.

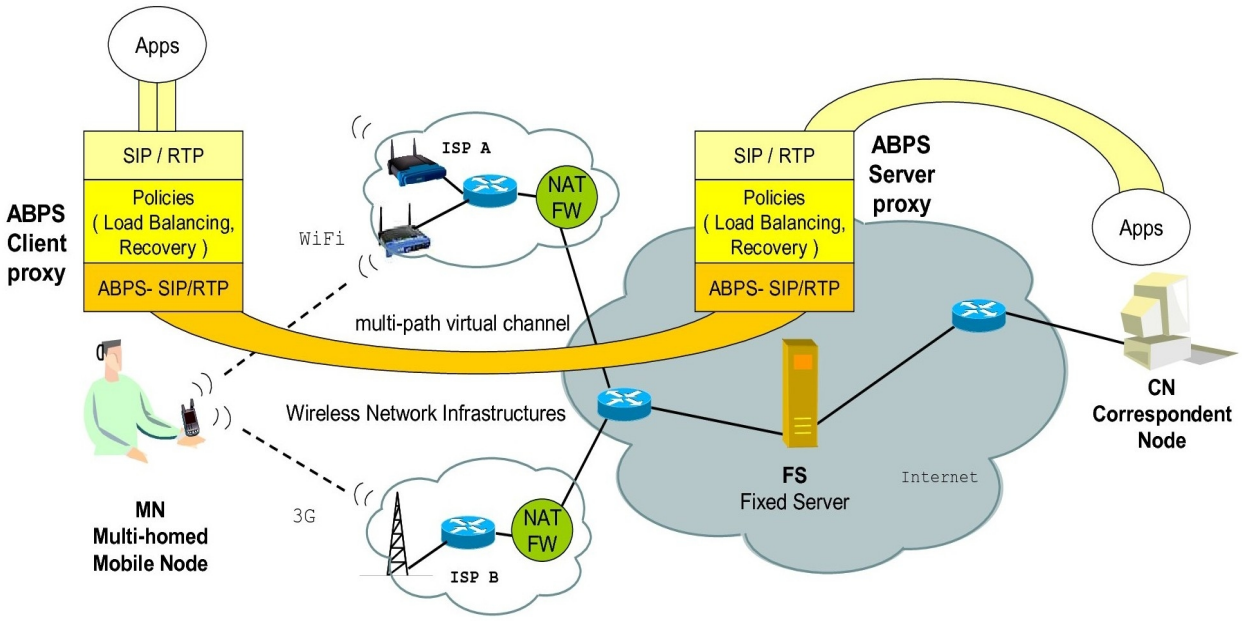


Figura 7: Architettura ABPS

Il motivo per cui accade è che se anche il client VoIP fosse in grado di gestire il cambio dell'interfaccia di rete i pacchetti verrebbero visti dal destinatario come appartenenti ad

un'entità distinta: cambiando interfaccia di rete cambierebbe l'indirizzo IP del mittente e quindi il destinatario scarterebbe tali pacchetti, sebbene in realtà il mittente sia sempre lo stesso.

L'approccio che si è deciso di adottare a tal fine è quello di creare due Outbound Proxy Server in modo che l'indirizzo del mittente venga mascherato, in questo modo se il mittente cambia indirizzo IP il destinatario non se ne accorgerà e la comunicazione potrà continuare finché almeno una interfaccia di rete sia disponibile.

Servono quindi due Proxy:

1. *Locale*, installato direttamente sul dispositivo mobile (facendo riferimento all'immagine nel multi-homed mobile node). Scopo del Proxy locale sarà di creare un tunnel multipath diretto al Proxy Esterno, attraverso tutte le interfacce disponibili (come ad esempio 3G o WiFi).
2. *Esterno,* installato su di un Server esterno e che interagirà con uno o più Proxy locali. Scopo del Proxy esterno sarà di comportarsi come Back-To-Back User Agent tra un Proxy Locale e il resto della rete. Il Proxy esterno crea quindi un canale logico con il Proxy locale in modo da riconoscere il mittente dei messaggi ricevuti, indipendentemente dall'interfaccia di rete utilizzata. Mascherando poi l'indirizzo del Proxy locale, il Correspondent Node non avrà problemi a interpretare i messaggi; non sarà infatti in grado di percepire ciò che avviene alle spalle del Proxy esterno.

Quello di cui ci si è occupati in questo progetto è di implementare un primo prototipo di Proxy locale, chiamato pj_relay.

Nei successivi paragrafi andremo a vedere la tecnologia usata per realizzare ed implementare il nostro progetto e come configurare l'ambiente di lavoro.

2.2 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 32 e 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 riferimento 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).

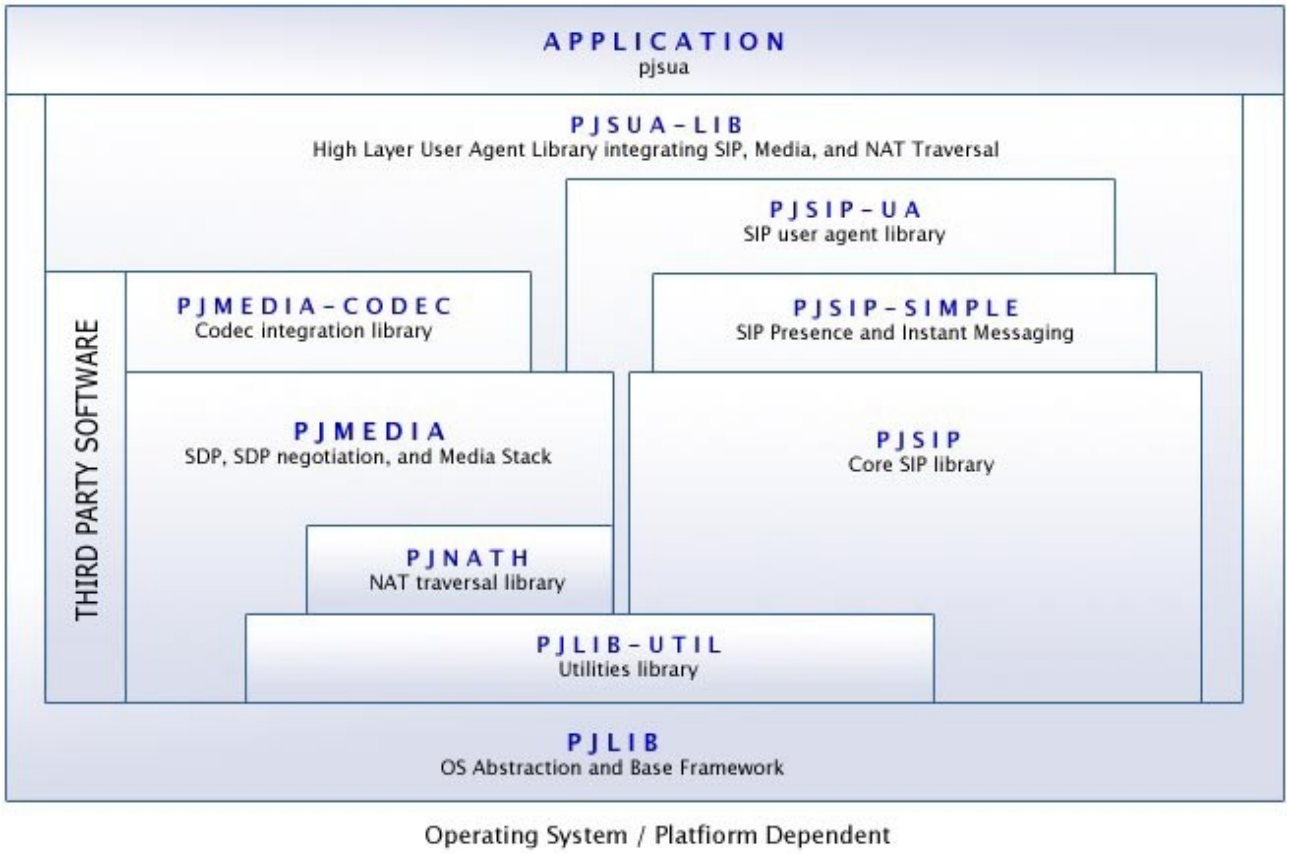


Figura 8: Stack delle librerie PJSIP

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:

- gestisce una *Pool Factory* , allocando le pool per tutti i 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;

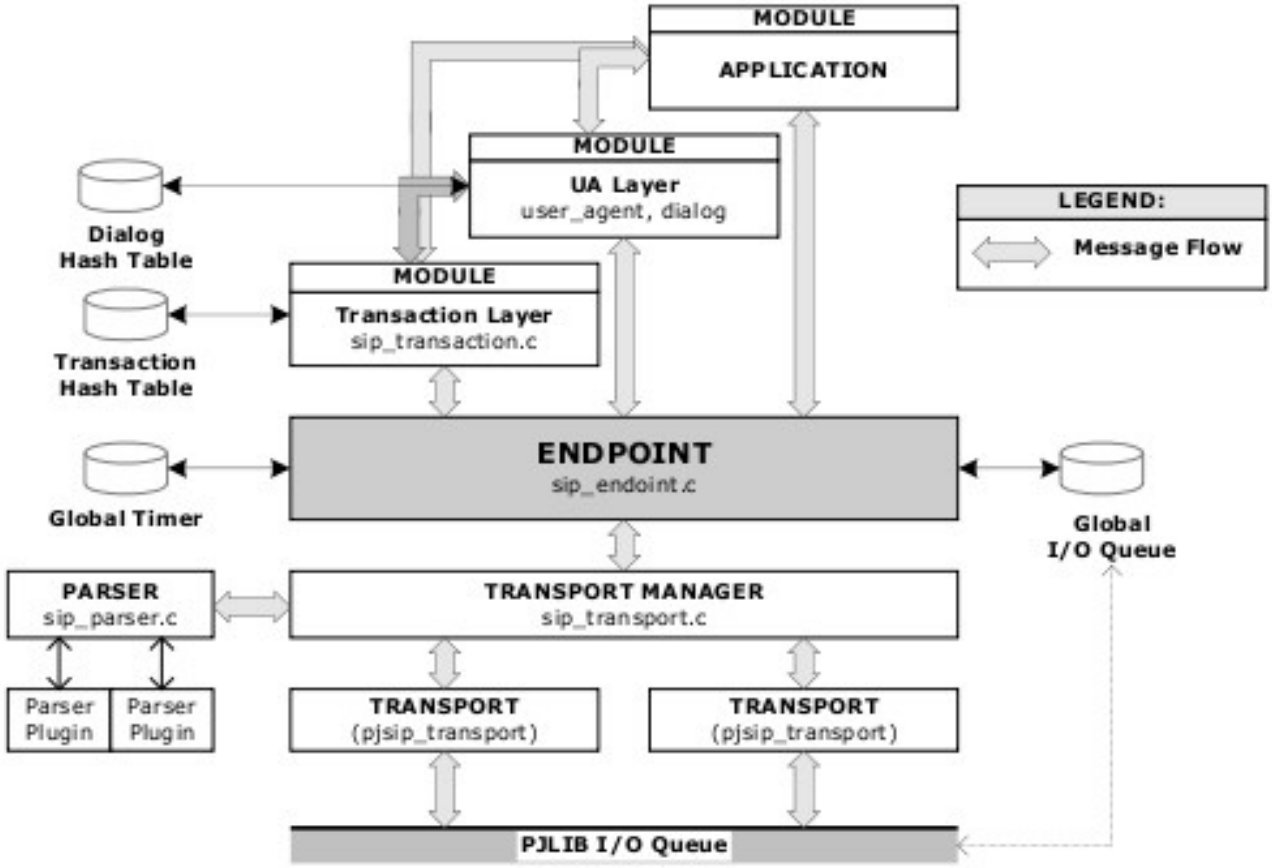


Figura 9: Diagramma di collaborazione

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 10: 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.

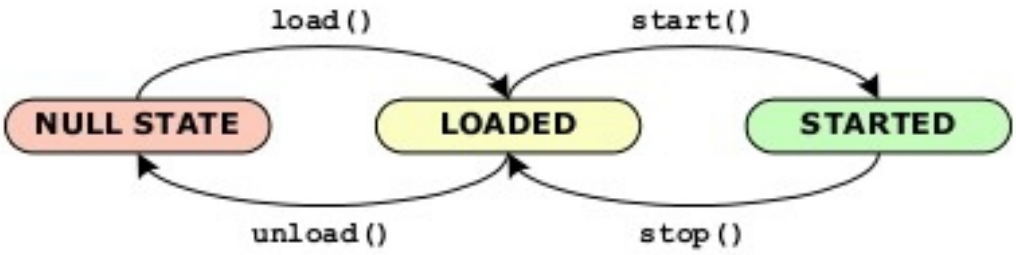


Figura 11: Ciclo di vita di un modulo

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 una *callback* torna non zero (ossia *true*), significa che il modulo ha provveduto alla gestione del messaggio e l'endpoint ne terminerà la distribuzione 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 dà 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 il 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 12: 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 il messaggio anche ad altri moduli.
Il diagramma seguente ci mostra un esempio di come i moduli possono chiamarne altri "a cascata".

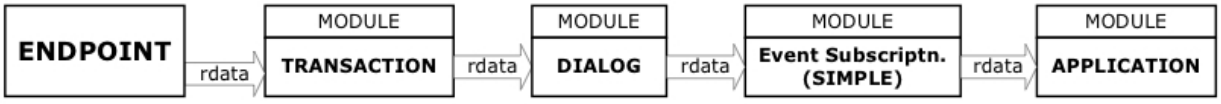


Figura 13: Cascade Callback

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*).

2.3 Configurazione dell'ambiente di lavoro

Configurare l'ambiente di lavoro per implementare il pj_relay è un'operazione semplice e veloce, a patto di sapere cosa fare; per questo motivo si è deciso di scrivere questo capitolo.
Per prima cosa è necessario registrare due account presso ekiga.net (o anche un altro SIP registrar), questi account saranno infatti utilizzati per i due client da utilizzare nell'ambiente di lavoro.
Successivamente bisogna creare le cartelle di lavoro; in quella voluta bisogna creare quindi le sotto-directory *pj_relay*, *pjproject*, *configurazione* ed una *report* (queste ultime due sono opzionali ma mi è sembrato un modo funzionale di lavorare).
La cartella pjproject sarà quindi la cartella dove andremo a scompattare i file di pjsip, che sono scaricabili all'indirizzo http://www.pjsip.org/download.htm. Ora è necessario configurare, compilare e installare pjsip:

1. *./configure --prefix=<percorso assoluto della cartella di lavoro>/pj_relay*
   Il comando prefix permette di impostare la cartella nella quale in fase di installazione verranno creati i file della libreria che vengono usati dal pj_project. Con questo comando i file vengono allocati in due cartelle direttamente dentro la cartella di lavoro del pj_relay. Il makefile del pj_relay è configurato per ricercare il file libpjproject.pc, che contiene le informazioni per risolvere gli include presenti nel nostro progetto, secondo questa configurazione. Modificare il percorso del parametro prefix richiede la modifica del makefile del pj_relay.
2. *make dep*
3. *make*
   Se dovessero comparire errori in compilazione sarà necessario ripartire dal punto 1 e abilitare dei flag durante l'esecuzione del configure. Per esempio, per poter compilare pjsip nelle macchine di laboratorio, è stato necessario usare il flag : *--disable-ffmpeg.*
4. *make install*

Una volta compilato pjsip è possibile copiare i file del pj_relay nella rispettiva cartella e compilarlo.
Il client SIP di pjsip, denominato pjsua, sarà presente nella cartella pjproject/pjsip-apps/bin. Prima di lanciarlo, per comodità, conviene creare due file nella cartella configurazione

denominati *Client1.pjsua.conf* e *Client2.pjsua.conf*. Questi file contengono parametri di configurazione per pjsua, sono di facile utilizzo (in fondo al capitolo ho inserito due esempi) e permettono di risparmiare molto tempo per configurare il client. Per utilizzare un file di configurazione basta usare la chiamata:
./pjsua-<versione della compilazione> --config-file ../../../configurazione/client1.pjsip.conf

2.3.1 Siproxd

Se si intende lavorare dietro un firewall o un NAT, a causa delle problematiche descritte nel capitolo 1.3, occorre operare in due modi: lavorare in ssh nei computer di laboratorio ovvero sfruttare le potenzialità di 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 e quella di installazione non sono 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*.

2.3.2 File di configurazione per Client interno

--log-file ../../../report/Client1.log.txt

--log-level 6

--app-log-level 1

SIP parameters

--id sip:<user da registrare>@ekiga.net

--realm ekiga.net

--registrar sip:ekiga.net # DNS SRV, or FQDN

--username <user da registrare>

--password <password>

#BUDDY

--add-buddy sip:<user da chiamare>@ekiga.net

SIP

--local-port 44444

--no-tcp

--outbound sip:127.0.0.1:5060;lr

#RTP

--rtp-port 60010

default of 55 will be rejected as being too short by sipX

--reg-timeout 100

mix WAV file into the audio stream

--null-audio

#questo parametro permette di riprodurre un file audio per testare problemi con i pacchetti

--play-file ../../../Configurazione/1.wav

--auto-play

--no-vad

--auto-answer=200

2.3.3 File di configurazione per Client esterno

--log-file ../../../report/Client2.log.txt

--log-level 6

--app-log-level 1

SIP parameters

--id sip:<user da registrare>@ekiga.net

--realm ekiga.net

--registrar sip:ekiga.net # DNS SRV, or FQDN

--username <user da registrare>

--password <password>

SIP

--local-port 44445

--no-tcp

#BUDDY

--add-buddy sip:<user da chiamare>@ekiga.net

#RTP

--rtp-port 60010

default of 55 will be rejected as being too short by sipX

--reg-timeout 100

mix WAV file into the audio stream

--null-audio

--auto-play

--no-vad

--auto-answer=200

#questo comando permette di registrare la conversazione nel file specificato

--rec-file ../../../report/Client2.rec.wav

--auto-rec

Capitolo 3

Scelte Implementative

Il protocollo SIP prevede la possibilità di utilizzare uno o più proxy per gestire la comunicazione tra due host. Tuttavia, dal momento che viene instaurata la comunicazione, i pacchetti vengono scambiati direttamente tra i due host, utilizzando i protocolli RTP/RTCP, escludendo quindi il proxy.

La comunicazione diretta però, può causare dei problemi se i due host si trovano dietro a un firewall simmetrico o se il dispositivo cambia il proprio indirizzo IP: la comunicazione infatti verrebbe instaurata senza problemi attraverso il proxy, ma i vari pacchetti scambiati potrebbero essere scartati o non arrivare a destinazione.

La soluzione proposta consiste nell'utilizzare un proxy come se fosse un relay: il suo compito sarà quindi quello di intercettare i messaggi di INVITE (utilizzati per scambiarsi i dati necessari, al fine di iniziare lo scambio vero e proprio di pacchetti) e sostituire gli opportuni campi con i propri dati. In questo modo il destinatario crederà che sia il proxy il mittente e l'eventuale risposta sarà quindi spedita a lui e non all'host mittente. Successivamente il proxy si occuperà di inoltrarla al mittente sempre cambiando i relativi campi, facendo sì che quando inizierà lo scambio di pacchetti questi passeranno tutti attraverso il proxy, al quale spetterà il compito di instradarli correttamente al mittente/destinatario.

L'applicazione è stata realizzata partendo dal proxy di esempio della libreria PJSIP. Poiché c'era la necessità di mantenere le informazioni sulle comunicazioni, è stato scelto di partire dallo statefull proxy.

La scelta, purtroppo, si è rivelata subito errata: l'applicazione dalla quale siamo partiti implementa un tipo di statefullness incompatibile con i nostri scopi. Tutto infatti è bastato su un modulo di libreria che si occupa di evitare i messaggi duplicati: il *transaction layer*. Altri compiti di questo modulo sono quelli di mantenere informazioni sui client, lanciare messaggi di CANCEL in caso di caduta della comunicazione, inviare messaggi di informazione ed altre funzionalità non utilizzate dal proxy.

Lo statefull proxy fa un uso abbondante del transaction layer. Questo strato processa i messaggi prima delle applicazioni a livello utente (come il proxy), poiché la priorità a cui viene registrato tale modulo è più alta di default di quella riservata agli altri moduli.
Ci si trova quindi davanti a una serie di problemi di coerenza: nel caso in cui nel corpo della funzione *on_rx_request()* il messaggio sip venga modificato, il nuovo messaggio conterrà l'indirizzo del proxy al posto dell'indirizzo dell'host mittente. Il transaction layer agisce prima di questa funzione, aspettandosi una risposta diretta all'host originale, ma il sistema non riesce a riconoscerla e quindi, non risultando coerente, la risposta sarà scartata.
Questo procedimento è stato verificato controllando il parametro *branch* nell'header FROM; si tratta di un checksum creato sulla base del messaggio originale. Infatti, in caso di manomissione del messaggio, questo diventerà inverificabile.
È stata a questo punto valutata l'ipotesi di agire prima del transaction layer, istanziando un modulo a priorità maggiore che modifichi in primo luogo il messaggio: al sopraggiungere di una richiesta di INVITE il messaggio viene modificato, e passa al modulo principale del proxy dove viene inviato al destinatario. Il protocollo SIP prevede però che venga inviato un messaggio informativo di 100TRYING al mittente, generato a partire dalle informazioni contenute in un messaggio però già modificato: il proxy risulterà dunque il mittente. Il risultato sarà quindi inviato al proxy stesso e l'applicazione o non sarà in grado di risolvere l'indirizzo (scartando dunque la comunicazione), oppure, caso peggiore, viene dato inizio ad un ciclo di forward dello stesso messaggio dal proxy a se stesso.
Per questi motivi si è dunque scelto di cambiare strada, indirizzandosi quindi verso lo *stateless_proxy:* la *statefullness* necessaria viene implementata dalle strutture e funzioni realizzate e si applica benissimo ad un proxy che non tiene traccia d'altro.
Il codice che svolge le operazioni necessarie al relay è realizzato nelle funzioni *relay_pre_request()* e *relay_pre_response()*: esse risalgono al tipo di messaggio (REGISTER, INVITE, ACK, BYE, CANCEL, etc) invocando poi la funzione relativa *handle_[tipo]_request|response()*.
Sono necessari due moduli perchè le modifiche risultino coerenti, in quanto queste devono essere svolte "da un solo lato" del proxy ovvero, considerando l'ambito di una transazione SIP, la posizione che assumerebbe una quarta applicazione che si trovi tra il proxy e uno dei

client: questa ipotetica applicazione A riceverebbe i messaggi di richiesta inviati dall'host H prima del proxy P, e viceversa riceverebbe i messaggi di risposta per A dopo P.
Se venisse creata un'istanza con priorità maggiore del modulo proxy, le funzioni che vi apparterrebbero agirebbero sempre prima del proxy e quindi le richieste verrebbero modificate prima che il proxy le possa vedere (dunque non sarebbe a conoscenza della richiesta originale), ma le risposte verrebbero ripristinate (dunque rese conformi alla richiesta originaria) prima di raggiungere il proxy durante il ritorno. Impostando un modulo a priorità inferiore a quella del proxy, si avrebbe ovviamente una situazione analoga.
La soluzione più semplice è istanziare due moduli differenti per richiesta e risposta in modo tale da farli agire "da un lato solo" del proxy.

3.1 Thread e mutua esclusione

La libreria PJSIP implementa le primitive di mutua esclusione dandone una propria versione portabile. Uno dei vincoli che sono imposti, richiede che il thread che rilascia la mutua esclusione sia lo stesso ad averla precedentemente ottenuta. Per questo motivo la funzione utilizzata per inizializzare il modulo, *node_info_init*, deve essere invocata dallo stesso thread che gestirà poi i pacchetti.
In conclusione, l'inizializzazione di tutti i moduli tramite la funzione *init_pj_relay()* deve avvenire nel *worker_thread* e non nel main.

Capitolo 4

Progettazione

4.1 Proxy

Il compito del proxy è quello di inserirsi tra due host A e B, intercettando quindi i loro messaggi di INVITE/200OK. Vengono quindi creati due nodi (*node_info*), utilizzati per rappresentare i due estremi della comunicazione, in cui saranno salvati i socket da utilizzare per lo stream dei dati. L'associazione tra i nodi avviene tramite il campo *linked_node*.
Il modulo del proxy, inoltre, gestisce opportunamente i messaggi attraverso gli altri moduli che abbiamo implementato.

4.1.1 Moduli

I moduli utilizzati dal proxy sono:

- *forward*
- *tsx_via*
- *src_info*
- *call_id*
- *URI*
- *Garbage Collector*

La registrazione di un modulo per l'endpoint (tramite la funzione *pjsip_endpt_register_module*) è fondamentale: l'endpoint chiamerà quindi le funzioni di *load* e *start* necessarie per inizializzare correttamente il modulo e per assegnargli un ID univoco.
I moduli sono memorizzati all'interno di uno stack secondo un criterio di priorità: ogni modulo infatti possiede un campo che ne indica la priorità e quindi l'indirizzo di memoria all'interno dello stack.
A seconda dei diversi eventi che possono verificarsi, questi moduli fanno uso di determinate

callback (maggiori informazioni sulle callback sono presenti nel capitolo 2.2): *load, start, stop, unload, rx_request, rx_response, tx_request, tx_response, tsx_state.*

4.1.2 Messaggi

Sono state create delle funzioni di gestione dei messaggi per i principali metodi SIP: REGISTER, INVITE, BYE. I messaggi con metodo diverso da quelli appena citati vengono gestiti con handler generico.
Quando viene quindi ricevuta una richiesta il proxy analizza la tipologia di messaggio e invoca il gestore appropriato. In caso di modifica del messaggio, sarà necessario ricostruire l'intero buffer.

4.2 Forward

La parte di fordward ha lo scopo principale di comportarsi da relay: modificare correttamente il messaggio SDP per potersi inserire tra i due host, ricevere i pacchetti RTP/RTCP da un endpoint e inoltrarli all'altro.

Il proxy quindi, deve intercettare i messaggi di INVITE e di BYE, per poter aprire e chiudere un flusso di forward tra gli host. Si rende quindi necessario tener traccia delle informazioni relative agli host coinvolti: è stata dunque progettata la struttura dati *node_info* che ha appunto questo scopo.

A seguito del ricevimento di un INVITE, è prevista anche una gestione dei messaggi di errore come il *408 Request Timeout* e *486 Busy Here.*

4.2.1 Processo di forwarding – *forwarding process*

L'inoltro al destinatario dei pacchetti RTP/RTCP è affidato al *forwarding process*, un thread il cui scopo è quello di rimanere in attesa di pacchetti RTP/RTCP, individuarne il mittente e inoltrarli quindi al destinatario.

Al ricevimento di un messaggio di INVITE, vengono memorizzate le porte RTP/RTCP da utilizzare per l'inoltro dei pacchetti audio. Si sono però verificate anomalie: in certi casi il client non specificava la porta RTCP da usare; la soluzione che abbiamo scelto di adottare è quella di utilizzare come porta RTCP il numero della porta RTP aumentato di 1.

Il proxy deve essere in grado di gestire più flussi di comunicazione, poiché potrebbe esserci anche un flusso di dati video. Per questo motivo è stata utilizzata la primitiva *select* con lo scopo di fare I/O multiplexing sincrono (*pj_sock_select*). PJSIP reimplementa questa funzionalità nella *pj_sock_select* che lavora su un insieme di socket ed un timeout

Il processo di forward viene implementato con un ciclo infinito che esegue questi tre passaggi:

- *Riempimento dell'insieme dei socket*: l'insieme viene riempito con i socket corrispondenti ai nodi attualmente presenti nella lista
- *Calcolo del massimo*: La select richiede tra i parametri il socket con valore massimo incrementato di 1

- *Select:* L’esecuzione viene bloccata fino a quando almeno un socket è pronto per ricevere. Al termine, nell’insieme rimangono solo i socket pronti. Di seguito, la lista dei nodi viene scansionata ottenendo i nodi corrispondenti ai socket pronti.
- *Touch*: Viene aggiornato il timestamp, per indicare che la comunicazione è attiva.
- *Inoltro*: Il pacchetto viene ricevuto, tramite il campo *linked_node* viene ottenuto il nodo corrispondente all’altro lato della comunicazione, e il pacchetto gli viene quindi spedito.

Sono stati previsti tre tipi di socket, ciascuno con scopi diversi: lettura, scrittura e gestione delle eccezioni: il *forwarding process* considererà soltano il primo socket.

4.2.2 Struct *node_info*

La struttura dati *node_info* è usata per rappresentare un lato della comunicazione rtp/rtcp, che collega il proxy con uno dei due host.
I campi che compongono questa struttura sono i seguenti:

- *local:* sockaddr del proxy.
- *remote*: sockaddr dell'host.
- *sock:* il socket utilizzato per scambiare i paccheti.
- *linked_node:* puntatore ad un node_info, cui è associata la struttura corrispondente all'altro lato della comunicazione.
- *pool*: necessario per l'allocazione e deallocazione dinamica della memoria, dal momento che PJSIP non utilizza *malloc* e *free* ma le reimplementa.
- *timestamp*: stabilisce quando una struttura è da considerarsi scaduta, chiamando così un Garbage Collector che si occuperà di rimuovere tutte le strutture non più necessarie.
- *next*: puntatore utilizzato per realizzare una lista di nodi.

4.2.3 Lista *node_info*

I nodi che descrivono le comunicazioni nelle quali è coinvolto il proxy si trovano all'interno di una lista, scorribile partendo dalla testa puntata dalla variabile globale *node_info_top*.
Due host A e B che cercheranno di comunicare attraverso il proxy quindi, genereranno quattro flussi di comunicazione (quindi quattro *node_info* nella lista): due flussi RTP, uno verso A e uno verso B, e due RTCP sempre uno verso A e uno verso B. Saranno quindi usate le seguenti funzioni:

- *nodeinfo_append*
- *node_info_remove*
- *node_info_max_socket*: individua il nodo con il socket più alto
- *node_info_socket_count:* conta il numero totale dei nodi presenti in lista
- *node_info_is_empty:* controlla se una lista è vuota o meno

Se la lista dei nodi è vuota, il *forwarding_process* deve interrompere la sua esecuzione. É stato dunque implementato un meccanismo di mutua esclusione, che si occupa di bloccare e sbloccare questo processo, a seconda che la lista sia vuota o meno.

4.2.4 Stream Management

Questa è la parte del modulo che si occupa di aprire e chiudere i flussi di comunicazione dal proxy verso un host attraverso le due funzioni *node_info_open_strea*m e *node_info_close_stream.*
L'apertura di un flusso avviene a seguito di un messaggio di INVITE e del relativo 200OK. Sarà quindi modificata la parte SDP del messaggio, sostituendo indirizzo e porta del mittente con indirizzo e porta del proxy, per far si che il destinatario risponda poi al proxy e non al mittente reale.
Successivamente verrà creato il nodo contenente le relative informazioni dell'host mittente: il nodo appena creato non viene aggiunto immediatamente alla lista, perché da quel momento il *forwarding_process* si aspetterebbe pacchetti provenienti da quell'host. Se questo avviene prima della 200OK, il *forwarding_process* va a cercare un *linked_node* che

non esiste, dal momento che verrà creato quando sarà gestita la risposta 200OK. Per questo motivo l'inserimento del nodo in lista avverrà solo dopo il ricevimento della risposta.

La chiusura del flusso di dati, consiste semplicemente nella chiusura del socket, nella rimozione del nodo dalla lista e nella deallocazione della struttura.

4.3 TSX_VIA

Questa struttura ha lo scopo di rappresentare una singola transazione SIP di richiesta-risposta in ambiente *Back-to-Back (B2B) User Agent:* si occupa infatti di salvare le informazioni del messaggio di richiesta che saranno poi necessarie per concludere la risposta.

Un Back-To-Back User Agent è un proxy che si frappone tra due host mascherandone reciprocamente la posizione. Prendiamo come esempio due macchine, A e B, che (come nel nostro caso) comunicano messaggi SIP tra loro sfruttando un proxy P. In genere P si comporterà come un comune proxy e si limiterà ad inserire il proprio indirizzo nella catena di header VIA, così che in ogni transazione SIP i due estremi siano A e B.

Al contrario, un B2B User Agent "impersona" l'host A nelle conversazioni con B e viceversa; con questo sistema, posizionando P in una rete pubblica, si può garantire l'apertura di una comunicazione tra due host entrambi nascosti da firewall simmetrici.

Le conseguenze dell'impiego di questo metodo sono il raddoppio delle transazioni SIP e la necessità di mantenere in memoria informazioni sui messaggi SIP originali e sugli host.

La struttura tsx_via si presenta in questo modo:

```
struct tsx_via_struct{
        DECL_LIST_MEMBER(struct tsx_via_struct);
        pjsip_cseq_hdr *cseq;          /* cseq header identifies the transaction */
        pjsip_via_hdr *via_chain;       /* pointer to the head of a via header list */
        pj_bool_t unregistering;       /* true if request had Expires=0 */
        pj_pool_t *pool;
}
```

4.3.1 Header VIA

Per stabilire quale sia il destinatario, sono utilizzati due metodi a seconda del tipo di messaggio. Entrambi i metodi fanno riferimento agli header di esso:

- *METHOD*: le richieste si basano sulla risoluzione dell'indirizzo specificato in questo header.

- *VIA*: le risposte percorrono al contrario la catena di header VIA, aggiunti ogni qualvolta il messaggio attraversa un hop.

É necessario modificare una parte del messaggio SIP, in modo che gli host siano convinti che il proxy sia il destinatario reale. Per far questo nel caso di messaggi di richiesta viene "asportato" il blocco dei VIA precedenti al proxy e modificato l'header CONTACT sostituendo l'indirizzo del proxy con quello del mittente.
Mentre nei messaggi di risposta gli header VIA, rimossi in precedenza, sono reinseriti per consentire al messaggio di raggiungere il destinatario corretto. Anche in questo caso è necessario modificare l'header CONTACT.
Il compito del salvataggio e ripristino dei VIA è affidato al modulo *tsx_via.*
La creazione di una struttura *tsx_via* avviene mediante la funzione *tsx_via_create()*, la quale si occupa di instanziare tale struttura e salvare al suo interno gli header VIA estratti dal messaggio SIP.
Gli header vengono reinseriti nel messaggio all'arrivo di una risposta appartenente ad una transazione registrata in precedenza (tramite il CSEQ del messaggio che serve come identificatore della trasmissione) tramite due funzioni:

- *tsx_via_proceed_transaction()*: utilizzata per le risposte di informazione 1XX (come il 100 TRYING, che segue l'INVITE), questa funzione si limita a reinserire i campi VIA.
- *tsx_via_complete_transaction()*: reinserisce gli header e completa la transazione eliminando la struttura dalla memoria.

Il parametro unregistering della struttura tsx_via è necessario per le funzioni di garbage_collecting: infatti quando un utente si disconnette, viene inviato un messaggio di REGISTER al suo registrat con header EXPIRES con valore 0. Se, al sopraggiungere della risposta 200OK, viene intercettata una richiesta di questo tipo, allora le informazioni sull'host salvate in memoria devono essere eliminate: viene controllato il flag unregistering (tramite la funzione *tsx_via_complete_transaction()*) e se necessario viene invocata la rimozione del relativo *src_info,* che spiegheremo in dettaglio qui di seguito.

4.4 SRC_INFO

```
struct src_info_struct {
        DECL_LIST_MEMBER(struct src_info_struct);   /* pointer to the next/preceding */
        pj_str_t username;
        pj_sockaddr src_addr;
        int active_calls;
        int src_type;                   /* source type: PERM_HOST or TEMP_HOST */
         pj_timestamp touch;
        pj_pool_t* pool;
}
```

La struttura *src_info* si occupa di salvare e, quando necessario, fornire l'identità e la posizione dei due utenti tra i quali il sistema si frappone. È fondamentale in quanto la statefullness è un requisito necessario per implementare un relay sip, e per farlo bisogna mantenere in memoria un altro set di informazioni: non è sufficiente dunque tenere traccia delle transazioni al fine di evitare messaggi ridondanti.

Questa struttura basa il riconoscimento dei client sullo username con cui gli host si registrano presso un registrar SIP, che viene salvato nell'omonimo campo: in caso di assenza di questa informazione, il client viene identificato tramite il suo indirizzo IP. Questo indirizzo viene memorizzato a prescindere nel campo src_addr sotto forma di pj_sockaddr per poter essere compatibile con IPV6 (nonostante l'attuale implementazione non preveda questa compatibilità).

Sono inoltre presenti tre variabili utilizzate dal modulo di garbage collecting:

- *src_type:* identifica il tempo di client, temporaneo o permanente.
- *active_calls*: numero delle chiamate attive nel client. Quando non ci sono chiamate attive, nel caso in cui il client sia temporaneo può essere deallocato.
- *touch*: timestamp dell'ultima attività del client. Il src_info può essere rimosso quando il touch è troppo datato.

4.4.1 Host temporanei e Host permanenti

Gli host vengono distinti in due “categorie”:

- Permanenti: registrati presso un registrar con il relay come proxy obbligatorio.
- Temporanei: dialogano con gli host permanenti ma non usano il relay come proxy.

La ricezione di una richiesta di REGISTER da parte di un utente verso il suo server di registrar, determina la “categoria” alla quale appartengono gli host.

Questa distinzione è stata implementata dal momento che, senza di essa, le informazioni di qualsiasi client che mandi messaggi attraverso il relay verrebbero salvate in memoria con il risultato che tutti i messaggi verrebbero intercettati, modificati e reinviati al sistema.

4.4.2 Metodi

Il modulo src_info mette a disposizione diversi metodi per la gestione delle strutture e delle relative liste:

- *src_info_create()*: istanzia la struttura dati, in questo caso richiede che venga specificato il tipo di host di cui si stanno salvando le informazioni.
- *src_info_append()*: aggiunge la struttura ad una lista dello stesso tipo.
- *src_info_find()*: cerca la struttura in una lista di strutture dello stesso tipo e ritorna un puntatore alla struttura cercata.
- *src_info_remove()*: rimuove la struttura da una lista di strutture dello stesso tipo e ritorna un puntatore alla struttura rimossa.
- *src_info_release()*: dealloca la memoria riservata alla struttura cancellando le informazioni contenute.
- *src_info_cmp()*: ritorna un booleano che indica se due strutture si riferiscono allo stesso client (il controllo viene effettuato sugli username se presenti, altrimenti sugli indirizzi).
- *src_info_touch()*: aggiorna il timestamp della struttura e la porta in fondo alla lista di appartenenza (se presente), in modo da lasciare in cima le strutture più datate.
- *src info update()*: aggiorna le informazioni sull’indirizzo attuale del client in caso di mobilità (attualmente è uno stub).

4.5 CALL_ID

```
struct call_id_struct {
        DECL_LIST_MEMBER(struct call_id_struct);  /* pointer to the next/preceding */
        pj_str_t id;                              /* call id header for search */
        struct src_info_struct *hostA;            /* hosts involved in the comunication */
        struct src_info_struct *hostB;            /* A: From, B: To */
        pj_bool_t in_call;
        pj_timestamp touch;
        struct node_info_struct *rtp_node;
        struct node_info_struct *rtcp_node;
        struct node_info_struct *streams [MAX_STREAMS];
        dl_list transactions;    /* transactions from this call */
        pj_pool_t* pool;
}
```

L'header CALL_ID nasce dalla necessità di mantenere in memoria informazioni sulle comunicazioni (ovvero le chiamate SIP). Infatti per garantire una comunicazione anche in presenza di firewall simmetrici, bisogna instradare anche il traffico RTO/RTCP attraverso il proxy: non sarebbe sufficiente instradare solo la comunicazione SIP attraverso di esso (esistono già protocolli che se ne occupano, ma non garantiscono la comunicazione appunto in presenza di firewall simmetrici).
La struttura dati call_id contiene tutte le informazioni necessarie per la gestione di una chiamata:

- *id*: la chiamata in questione
- *struct src_info_struct *hostX*: src_info dei client coinvolti
- *streams*: lista di node_info utilizzati per gli stream multimediali
- *transactions*: lista delle tsx_via che identificano le transazioni appartenenti alla chiamata

- *in_call*: flag che indica se ci sono flussi audio/video attivi per quella chiamata

I restanti campi hanno la funzione di allocare la memoria, gestire le liste e stabilire tempo di vita delle strutture allocate.

4.5.1 Funzioni

La struttura call id offre alcune semplici funzioni di controllo:

• *call_id_create()*: costruttore della struttura: richiede che almeno il mittente della comunicazione sia noto, il destinatario può essere aggiunto in un secondo momento (i.e. quando il proxy riceve una richiesta di INVITE normalmente è a conoscenza delle informazioni del mittente ma non del destinatario, informazioni che apprenderà solo al sopraggiungere della risposta).

• *call_id_add_hostB()*: imposta il secondo capo della comunicazione se precedentemente non era stato specificato.

• *call_id_dead_host()*: chiude le chiamate in cui l'host era coinvolto. Questa funzione viene chiamata ogni qualvolta un src_info viene rimosso dalla memoria

• *call_id_start_call()/call_id_end:call()*: impostano il flag che indica lo stato della chiamata.

• *call_id_add_stream()*: imposta nell'array streams il *node_info* relativo ad un particolare stream multimediale. Normalmente, per ogni comunicazione vengono settati due stream (rtp e rtcp), ma potrebbe esserci la necessità di utilizzare uno stream video.

Le funzioni *call_id_append()*, *call_id_remove()* e *call_id_find()* sono funzionali alla gestione delle liste, la funzione *call_id_touch()* aggiorna il tempo di vita delle strutture mentre la *call_id_release()* è utilizzata per la gestione della memoria.

4.6 URI – Unified Resource Identifier

Modulo per fornire un supporto all'utilizzo degli URI di PJSIP. In PJSIP infatti, un URI è rappresentato tramite struttura pjsip_uri che però non consente l'accesso diretto ai propri campi, quali *indirizzo ip*, *porta*, *etc*. Vengono quindi messe a disposizione due funzioni di libreria che consentono di ovviare a questo problema:

- *pjsip_uri_print*: stampa l'URI in un buffer
- *pjsip_parse_uri*: costruisce un nuovo URI leggendo le informazioni da un buffer

Dal momento che il proxy realizzato deve funzionare come relay, esso deve modificare singole parti dell'URI: si è quindi deciso di realizzare un modulo a parte per gestire meglio le funzionalità di *parsing*, *split* e *management*

4.6.1 URI Parsing

La sintassi di un URI è la seguente:

‹sip : [username@]domain[: port][; param]›

dove *username, port* e *param* sono parametri opzionali.

Questo modulo si occupa quindi di suddividere un URI in cinque diversi token:

- *before:* tutto quello che precede l'indirizzo (i.e. "*‹sip:*")
- *username*
- *domain*
- *port*
- *after:* tutto quello posposto all'indirizzo (i.e. "*[; param]›*")

La suddivisione in token viene fatta facendo uso delle funzioni *uri_parser_uri()*, che si occupano di isolare l'indirizzo dal resto dell'URI ottenendo così un'espressione della forma *[username@]domain[: port]*. Questa espressione sarà poi passata alla seconda funzione utile a questo scopo*: uri_parser_address(),* il cui compito è quello di separare le varie componenti (username, porta, ip).

4.6.2 URI Split

Lo split dell'URI viene fatto tramite una famiglia di funzioni il cui compito è quello di generare le stringhe relative ai diversi token: come argomenti gli vengono passati dei doppi puntatori a strutture *pjsip_str_t* a cui saranno assegnate le relative parti dell'URI.
Le due principali tipologie di funzioni sono:

- *uri_split_uri:* prende in input una struttura pjsip_uri e ne estrae il relativo buffer.
- *uri_split_buffer*: chiamata dalla *uri_split_uri*, svolge il lavoro vero e proprio di divisione della stringa.

I doppi puntatori alle stringhe, possono essere passati separatamente oppure all'interno di una array (pj_str_t ***token) oppure separatamente, specificando cinque o tre puntatori: quest'ultima versione richiede quindi solo i puntatori ai token *username*, *indirizzo ip* e *porta*.

4.6.3 URI Management

Questa famiglia di funzioni ha il compito di modificare i token voluti di un URI già esistente, lasciando gli altri immutati.
Al ricevimento di un messaggio, il proxy deve spesso modificare l'URI dell'host mittente specificando il proprio indirizzo IP e la porta: il resto dell'URI però, non deve cambiare.
Esistono diverse versioni di queste funzioni, differenziandosi per il tipo di dati ricevuti:

- *uri_management_buffer:* scompongono e sostituiscono le parti richieste di una stringa
- *uri_management_uri:* estraggono il buffer da una struttura pjsip_uri, e ottengono una nuova stringa passando questo buffer alla prima famiglia di funzioni. Da questa stringa sarà poi ottenuto il nuovo URI

4.7 Garbage Collector

È stato implementato un thread dedicato al garbage collecting, al fine di migliorare l'efficienza e di evitare malfunzionamenti a causa di informazioni troppo datate. Questo thread si attiva ogni 5000 millisecondi (5 sec) e controlla le liste delle strutture dati che sono presenti in memoria.

Sono state definite alcune costanti nel file pj_relay.h che valutano se le strutture siano scadute o meno:

- MAX SRC LIVING TIME: impostata a 650 secondi, è il tempo massimo di vita di una src info inattiva;
- MAX CID LIVING TIME: impostata a 300 secondi, è il tempo massimo di vita di una call id inattiva;
- MAX NODE LIVING TIME: impostata a 30 secondi, è il tempo massimo di vita di una node info inattiva;
- MAX NODE DELAY TIME: impostata a 10 secondi, è il tempo massimo di vita di una coppia di node info (linked node) di cui una attiva e l'altra no.

Le due liste generali di strutture (*call_id_top* e *src_info_top*) vengono mantenute ordinate dalle funzioni di *touch*, che spostano sistematicamente in coda alla lista gli elementi più recenti. È sufficiente dunque controllare l'età degli elementi in testa alla lista.

Per le call_id il discorso però è differente: una chiamata potrebbe non essere defunta, ma potrebbe non avvenire alcuno scambio di messaggi SIP per un tempo superiore a 300 secondi. Occorre quindi controllare innanzitutto se sono attivi i *node_info* relativi alle chiamate, e in caso negativo interrompere la chiamata. A questo punto, occorre controllare il FLAG che indica lo stato della chiamata: la call_id sarà sicuramente attiva se questo è settato a TRUE, in caso contrario l'eliminazione della struttura sarà decisa in base alla costante *MAX_CID_LIVING_TIME*.

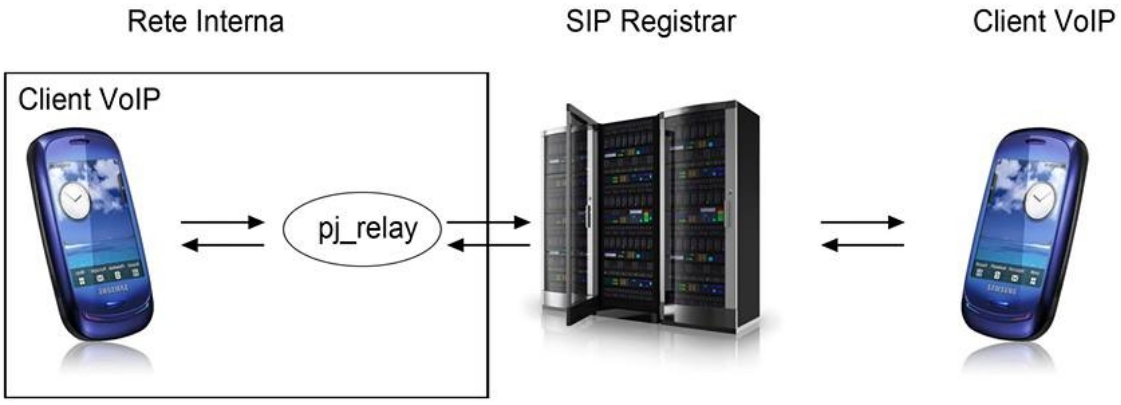
È inoltre necessario controllare tutte le call_id presenti e non limitarsi alla verifica degli elementi in testa alla lista call_id_top, poiché il garbage collector relativo alle call_id deve controllare anche lo stato delle *node_info.*

Il garbage collecting per le *src_info* invece, si limita ad eliminare iterativamente il primo elemento della lista globale fintanto che questi risultino scaduti.
Nel caso in cui vengano introdotte nuove strutture dati, le funzioni atte al controllo e all'eliminazione dei nuovi elementi si trovano nel corpo del ciclo *while(!global.quit_flag),* all'interno della funzione *garbage_collector()*.

Capitolo 5

Verso l'Outbounding

Quando si guardano le opzioni di configurazione sulla maggior parte dei telefoni IP, viene visualizzato un campo chiamato "Outbound Proxy" o "Outbound Proxy Server". In questo campo è possibile immettere un indirizzo IP, un nome host.domain o semplicemente un nome di dominio (fintanto che tale nome può essere risolto in un indirizzo IP dal DNS). Se l'inserimento del valore nel campo "Outbound Proxy Server" è valido allora ogni tipo di richiesta SIP da inviare dal telefono verrà inviato al Proxy in prima istanza. Ciò significa che anche una richiesta di REGISTRAZIONE che altrimenti sarebbe andata direttamente al server di registrazione, dovrà passare attraverso il Proxy in uscita. L'Outbound Proxy Server è quindi in grado, in base alle proprie regole interne, di bloccare, modificare e ritrasmette la richiesta al server di registrazione opportuno. In questo ruolo, il server proxy in uscita agisce come una barriera di sicurezza gestita a livello centrale che, in combinazione con adeguate regole del firewall, è simile a quello di un web proxy di una rete aziendale.



Come precedentemente asserito questo è il ruolo del nostro Proxy ma, per realizzare l'obiettivo ultimo del progetto, questa funzionalità deve essere prevista anche nel pj_relay. Mentre il pj_relay sarà il Proxy personale del client locale per come è stato teorizzato il

progetto dovrà venire implementato un Proxy Server centrale che si comporti come Outbound Proxy Server per tutti i pj_relay.

Per implementare questa funzionalità è però necessario introdurre due argomenti prima: la necessità di determinare il verso della comunicazione e il masking dei campi VIA dei messaggi SIP.

Per "determinare il verso della comunicazione" si intende analizzare il messaggio in transito attraverso il nostro Proxy e riconoscerne il destinatario. I messaggi infatti che necessitano di essere reindirizzati verso un secondo Outbound Proxy Server dal pj_relay, sempre che venga abilitato l'outbounding anche in pj_relay tramite l'opzione *--outbound sip:<indirizzo_ip>:<porta>*, sono solo quelli che vengono inviati dal nostro client interno, quello per cui stiamo svolgendo il servizio di Outbounding. Per esempio se il nostro Client 'A' volesse chiamare un certo Client 'B' allora pj_relay dovrà modificare l'instradamento del messaggio di INVITE ma non il messaggio di accettazione 200 OK, nel caso in cui 'B' abbia accettato la chiamata. Il controllo del mittente è una procedura che è importante eseguire su ogni messaggio, questo perchè anche rimanendo nel caso dell'esempio precedente in cui A chiama B ciò non significa che B non possa mandare messaggi ad A, per esempio potrebbe essere B a voler chiudere la comunicazione inviando BYE.

Il masking dei campi VIA è una funzionalità che abbiamo voluto implementare nel pj_relay in modo da fornire un servizio da Back-to-Back User Agent. Questa politica può sembrare inizialmente poco utile, in quanto se ogni hop si comportasse seguendo lo standard, il masking non avrebbe alcuna utilità; tuttavia non sono rari i server che hanno comportamenti anomali. Potrebbe accadere ad esempio che il destinatario dei nostri messaggi voglia comunicare, per qualche ragione, direttamente con il nostro Client escludendo così il pj_relay e impedendogli di svolgere quindi il suo lavoro di outbounding. La procedura di default prevede che, nell'inoltro di una risposta, il messaggio venga inviato all'host identificato dal più recente campo VIA, il quale rimuoverà tale campo e reinvierà all'indirizzo fornito da quello che è diventato il campo VIA in cima alla pila. Tuttavia un host potrebbe avere un comportamento anomalo, ad esempio inviando il messaggio di risposta direttamente all'indirizzo nel campo VIA in coda, bypassando così tutti gli hop nel mezzo, tra cui l'Outbound Proxy Server. Grazie al masking, operato dal pj_relay, questo non

può accadere in quanto i messaggi che arriveranno a destinazione avranno come ultimo campo VIA inserito quello creato dal nostro pj_relay.

Il punto in cui mi è sembrato più opportuno effettuare quindi l'analisi della direzionalità del messaggio è appunto la funzione strip_via_stack. In questo modo ogni volta che pj_relay riceve un messaggio di richiesta può effettuare due controlli:

- Se il messaggio è un messaggio di REGISTER allora può memorizzare, in una apposita struttura dati, l'indirizzo del campo VIA in coda al messaggio (presumibilmente dovrebbe essercene solo uno di campi in quanto pj_relay dovrebbe essere installato in locale sul dispositivo mobile) e riconoscere quindi il client che effettua la REGISTER come il nostro client interno, settando quindi la direzionalità del messaggio verso l'esterno.
- Se il messaggio non è di REGISTER allora quando i campi VIA vengono estratti è possibile fare dei controlli sugli indirizzi memorizzati nei campi (soprattutto, nel caso siano presenti, tra i parametri *received* e *rport*) per vedere se è presente l'indirizzo del nostro client locale. Se lo è allora la conversazione è una conversazione verso l'esterno, altrimenti la conversazione è verso l'interno in quanto il client locale è, in questo caso, il destinatario del messaggio.

Capire questo ci permette di discriminare le casistiche in cui fare outbounding una volta che verrà invocata una delle due funzioni deputate all'inoltro dei messaggi:

- on_rx_request: è addetta alla configurazione e all'inoltro delle richieste. L'outbounding in questo caso è da effettuare unicamente se si è stabilito l'esterno come verso della comunicazione. Per poter modificare il destinatario del messaggio è sufficiente aggiungere un campo ROUTE con l'indirizzo dell'Outbound Proxy. Il campo viene aggiunto con l'invocazione della funzione *proxy_process_routing* a cui deve essere passato l'intero che identifica la direzione della comunicazione. È importante che il campo venga inserito nella posizione giusta nel messaggio, in un primo momento infatti non ci si era prestato attenzione e quindi sembrava che questa soluzione non funzionasse; in realtà il corretto posizionamento del campo è fondamentale. Questa soluzione è attuabile dato che ci si è resi conto che pjsip con l'invocazione di *pjsip_endpt_send_request_stateless* analizza i campi del messaggio

per determinare il prossimo hop. Inizialmente pensavo che per cambiare l'indirizzo del hop intermedio a cui inviare il messaggio servisse configurare il terzo parametro della funzione *pjsip_endpt_create_request_fwd* il *pjsip_uri** con i dati dell'Outbound Proxy Server di pj_sip. Tuttavia così facendo non si modifica solo l'indirizzo del prossimo Hop ma anche il campo contact, compromettendo quindi il senso e l'integrità del messaggio.

- on_rx_response: è addetta alla ritrasmissione dei messaggi di risposta, l'outbounding in questo caso è da effettuare unicamente se si è stabilito l'interno come verso della comunicazione. Per cambiare il destinatario del messaggio è sufficiente modificare la variabile res_addr inserendo nei campi dst_host.addr.host e dst_host.addr.port l'indirizzo e la porta del destinatario voluto.

Con l'attuale implementazione, quando il programma viene invocato con il flag --outbound sip:indirizzo:porta, questi dati sono memorizzati, tramite la funzione init_options in proxy.h, nella variabile globale global.proxy_uri di tipo pjsip_uri*. È importante sottolineare che per poter fare il parsing dei dati memorizzati in una variabile pjsip_uri, per accedere ad esempio ai campi host e port, è necessario prima fare un cast di tipo in pjsip_sip_uri. Questo è dovuto al fatto che la struttura pjsip_uri è un tipo adatto a memorizzare un generico URI e tutte le tipologie specifiche di URI, implementate nella libreria (come sip:, sips:, tel: e name-addr), hanno come primo campo una 'virtual' function table. Questa tabella provvede ad instaurare un comportamento polimorfico tra gli URI.

In conclusione, questa implementazione dell'outbounding sembra funzionale; le funzioni di stampa delle destinazioni dei messaggi, invocate dalla libreria pjsip al momento dell'invio del messaggio, mostravano che il destinatario era l'host da noi voluto. Sono stati effettuati alcuni test utilizzando un secondo pj_relay come Outbounding Proxy Server per il primo.

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 una parte dei problemi legati al VoIP.

Siamo riusciti a sviluppare quindi un Relay in grado di svolgere un lavoro da Back-to-Back User Agent, con in aggiunta la possibilità di impostare un Outbound Proxy Server. Ci sono ancora molte modifiche da apportare al progetto affinchè si possa raggiungere l'obiettivo descritto nel capitolo 2:

- Bisogna introdurre il supporto a IPv6 che è già supportato da pjsip.
- Bisogna valutare quali modifiche apportare in modo che il relay possa essere usato sia come proxy client che come proxy remoto. Una funzionalità da realizzare che ci siamo resi conto essere necessaria è di implementare una politica analoga a quella compiuta con i messaggi VIA anche con i campi *ROUTE* e *RECORD-ROUTE*. Se infatti non si mascherano questi campi nel messaggio verrà lasciata una traccia del percorso effettuato e questo può portare a problemi analoghi a quelli che derivano dal mancato mascheramento dei campi VIA, descritti nel precedente capitolo.
- Bisogna ancora implementare la gestione delle interfacce di rete.

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 dovranno affrontare questo progetto. Spero infatti di semplificare loro, almeno in parte, la prima fase di studio e analisi del problema da risolvere.

Bibliografia

1. Wikipedia, enciclopedia libera – http://wikipedia.org
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. RTP, Real Time Protocol, RFC 3550- http://tools.ietf.org/html/rfc3550
9. SDP, Session Description Protocol, RFC 2327 - http://tools.ietf.org/html/rfc2327
10. SIP, Session Initiation Protocol, RFC 3261 - http://tools.ietf.org/html/rfc3261
11. RTSP, Real Time Streaming Protocol, RFC 2326 - http://tools.ietf.org/html/rfc2326