POLITECNICO DI MILANO

Polo Territoriale di Como

Facoltà di Ingegneria dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica

Analisi e realizzazione di un'applicazione di videocitofonia per Android

Relatore: **Prof. Luciano BARESI**

Tesi di Laurea Magistrale di:

Angelo LUPO Matricola 769939

Michele MORANDINI Matricola 766268

Anno Accademico 2012 – 2013

SOMMARIO

ELENCO DELLE FIGURE

1 INTRODUZIONE

In questo elaborato andremo ad analizzare e progettare un sistema che permetta di interfacciarsi con un impianto domotico Bticino di videocitofonia, attraverso un dispositivo mobile, come può essere uno smartphone o un tablet. Immaginiamo di possedere un classico citofono domestico, o un videocitofono che ci permette anche di osservare il nostro interlocutore. In quest'ultimo caso, il funzionamento è semplice: una persona effettua una chiamata dall' esterno della nostra abitazione verso l'interno e, da dentro l'appartamento, l'utente può controllare la situazione prima di decidere se iniziare la comunicazione oppure se non rispondere. Non sempre, però, è possibile rispondere alle chiamate magari perché, quando avvengono, non abbiamo la possibilità di accedere al dispositivo interno oppure semplicemente perché si vuole un accesso più comodo e immediato. L'idea è, quindi, quella di dotare l'impianto domotico di una funzione mobile alternativa, da poter portare sempre con sé. Un passo analogo è stato compiuto anni fa con i telefoni fissi, introducendo i telefoni senza fili, i quali si possono trasportare all'interno dell'unità abitativa. Con questo sistema, sarà possibile fare altrettanto con il proprio citofono: rispondere ad una chiamata in qualsiasi posto dell'abitazione. Oramai, uno smartphone è disponibile in quasi tutte le case ed è diventato uno strumento molto utilizzato nella vita quotidiana; viene quindi naturale pensare di dotarlo di tutte quelle funzionalità che il suo potenziale permette.

Nei prossimi capitoli verranno illustrate le problematiche indotte dal sistema e le soluzioni proposte. Saranno elencate ed esposte tutte le tecnologie utilizzate sia dall'impianto sia dall'applicazione mobile, e, dove necessario, approfondito l'argomento per far comprendere al meglio l'utilizzo e il senso della relativa funzionalità. Durante il lavoro è stato realizzato, inoltre, un prototipo del prodotto finale, funzionante, e saranno, quindi, descritte le prove e le valutazioni di quest'ultimo in una prova sul campo. Un'attenzione particolare, inoltre, verrà posta su specifici sviluppi futuri, non solo accennando a cosa potrà essere sviluppato, ma illustrando tutta una progettazione volta a predisporre le basi per l'effettiva creazione delle funzionalità aggiuntive.

1.1 TERMINOLOGIA ADOTTATA ALL'INTERNO DELLA TESI

È necessario definire a priori una terminologia, onde chiarire alcune terminologie ambigue. Definiamo:

- **Applicazioni Java:** Questo genere di applicazioni sono quelle più diffuse all'interno del mondo Android: tutto il codice è scritto in linguaggio Java.
- **Applicazioni Native:** Con questo termine facciamo riferimento unicamente alle applicazioni compilate in codice binario, ed eseguibile direttamente dal processore. In genere si effettua la compilazione dei binari tramite Android NDK.
- **Applicazioni Native con JNI:** In genere, si fa riferimento a questo tipo di applicazioni come ad "Applicazioni Native", assieme alle successive anche se, in questo caso, si accede alla Java Native Interface [1] allo scopo di interagire con il codice Java. La compilazione di queste applicazioni avviene tramite lo script *ndk-build*. In ogni caso, viene sempre prodotto un file apk, e vengono stabiliti i permessi di accesso nel Manifest. Per la compilazione è necessario disporre dell'Android NDK.
- **API**: Con Application Programming Interface si indica ogni insieme di procedure disponibili al programmatore. Spesso con tale termine si intendono le librerie software disponibili in un certo linguaggio di programmazione. Più precisamente le API di Android sono fornite da Google e cambiano di versione in versione del Sistema Operativo
- **ADB**: Android Debug Bridge è un'applicazione in grado di connettersi ad un device Android (virtuale o no) permettendo: di installare le applicazioni, di inviare e ricevere file e di eseguire comandi di *shell scripting* sul device.
- **Dalvik Virtual Machine**: Si tratta di una VM (Virtual Machine) 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).

- **Home Area Network**: è una rete informatica la cui estensione è tipicamente limitata ai dispositivi presenti in una abitazione. Una funzione importante è la condivisione dell'accesso a Internet.

Si utilizzeranno, invece, i termini generici di "***applicazione***" o "***apps***" qualora si faccia riferimento a tutte le tipologie di applicazione di cui sopra.

1.2 DISPOSITIVI ADOTTATI IN FASE DI TESTING E SVILUPPO

Elenchiamo di seguito quali sono stati gli strumenti e i device principali utilizzati all'interno della tesi:

- **Emulatore Android SDK**: L'SDK di Android include un emulatore di dispositivi mobile, un device virtuale utilizzabile su differenti sistemi operativi. Permette di "prototipare" un device reale, in modo da sviluppare e testare applicazioni Android senza utilizzare un vero dispositivo.

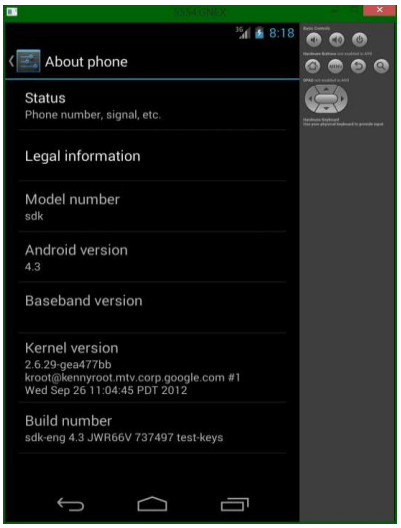


Figura 1: emulatore Android

L'emulatore Android simula le tipiche funzioni e i comportamenti di un dispositivo mobile, tranne per il fatto che non può effettuare o ricevere normali chiamate. L' emulatore è fornito di vari pulsanti e sistemi di

interazione, che possono essere premuti utilizzando il mouse o la tastiera, per generare eventi all'interno delle applicazioni testate.

Per aiutare il design e il testing delle applicazioni, l'emulatore permette alla stessa di usare i servizi della piattaforma Android per invocare altre applicazioni, accedere alla rete, eseguire audio e video, immagazzinare e recuperare dati, avvisare l'utente e interpretare transizioni grafiche e temi.

- **Nexus Device**: I Nexus sono una linea di dispositivi mobili che usano il sistema operativo Android, prodotti da Google in collaborazione con degli OEM (Original Equipment Manufacturer). I dispositivi della serie Nexus non hanno personalizzazioni, da parte di produttore o gestore telefonico, al sistema operativo Android.
  Samsung Galaxy Nexus: è uno smartphone progettato da Google e prodotto dalla Samsung. È stato recentemente aggiornato con la versione 4.3 Jelly Bean.
  Il Galaxy Nexus è uno dei pochi smartphone raccomandati dall' Android Open Source Project per lo sviluppo di software per Android [2].

- **Samsung Galaxy Tab 8.9**: è un tablet progettato e realizzato da Samsung, introdotto nel marzo 2011. Fa parte della serie Samsung Galaxy Tab e presenta uno schermo da 8.9 pollici di diagonale.
  Al rilascio, la versione di Android preinstallata era la 3.1 Honeycomb personalizzata da Samsung [3], ed è stato aggiornato ad Android 4.0.4, con una versione personalizzata chiamata TouchWiz UX [4].

Figura 2: Galaxy Nexus

Figura 3: Galaxy Tab

2 STATO DELL'ARTE E ANALISI DEL PROBLEMA

2.1 LA DOMOTICA

Il termine "**domotica**" deriva dal francese *domotique*, contrazione del latino *domus*, per indicare la dimora, il centro abitativo dell'individuo o della famiglia, e non semplicemente l'edificio. Esso indica l'**integrazione** nella vita domestica di diverse **tecnologie** quali l'**elettrotecnica**, l'**elettronica**, l'**informatica**, la **comunicazione** e più in generale l'**automazione**.

Lo scopo della domotica è, in generale, quello di **migliorare le condizioni di esistenza** dell'abitante della casa offrendo comfort, sicurezza e benessere. Il comfort si ottiene grazie a dispositivi, quali attuatori, telecomandi, sistemi di comunicazione, automatismi pre-programmati, che rendono più semplice la fruizione dei servizi (illuminazione, condizionamento), degli elettrodomestici e degli apparecchi tecnologici. La sicurezza può essere assicurata da dispositivi o servizi *attivi* quali la chiusura automatica degli accessi, il loro controllo, i dispositivi antintrusione, segnalazione e comunicazione che permettono di conoscere lo stato della casa e dei suoi abitanti e di intervenire quando necessario.

Un altro concetto è il **benessere degli abitanti**, al quale si può ricondurre, ad esempio, l'alleviamento delle condizioni di debilitazione di un malato, di un anziano o di un disabile, realizzato sia con strumenti che ne riducano la fatica fisica e aumentino la mobilità, sia facilitandone l'accesso a mezzi di comunicazione. In questa categoria si possono anche far rientrare i dispositivi che predispongono la casa ed i suoi abitanti ad un uso più consapevole ed efficiente dell'ambiente, quali il risparmio energetico.

La domotica è nata nel corso della *terza rivoluzione industriale* allo scopo di studiare, trovare strumenti e strategie per:

- Migliorare la qualità della vita;
- Migliorare la sicurezza;
- Semplificare la progettazione, l'installazione, la manutenzione e l'utilizzo della tecnologia;
- Ridurre i costi di gestione;
- Convertire i vecchi ambienti e i vecchi impianti.

La domotica svolge un ruolo davvero molto importante nel rendere intelligenti apparecchiature, impianti e sistemi. Ad esempio, un impianto elettrico intelligente può autoregolare l'accensione degli elettrodomestici per non superare la soglia che farebbe scattare il contatore.

Con il termine ***Smart Home*** o *"casa intelligente"* si indica un ambiente domestico, opportunamente progettato e tecnologicamente attrezzato, il quale mette a disposizione dell'utente impianti che vanno oltre il "tradizionale", dove apparecchiature e sistemi sono in grado di svolgere funzioni parzialmente autonome (secondo reazioni a parametri ambientali di natura fissa e prestabilita) o programmate dall'utente o, recentemente, completamente autonome (secondo reazioni a parametri ambientali dirette da programmi dinamici che cioè si creano o si migliorano in autoapprendimento).

Ad un livello superiore si parla di ***building automation*** o *"automazione degli edifici"*. L'edificio intelligente, con il supporto delle nuove tecnologie, permette la gestione coordinata, integrata e computerizzata degli impianti tecnologici (climatizzazione, distribuzione acqua, gas ed energia, impianti di sicurezza), delle reti informatiche e delle reti di comunicazione, allo scopo di migliorare la flessibilità di gestione, il comfort, la sicurezza e per migliorare la qualità dell'abitare e del lavorare all'interno degli edifici.

La casa Intelligente

La casa intelligente può essere controllata dall'utente tramite opportune interfacce utente (come per esempio pulsanti, telecomandi, device touchscreen, tastiere, riconoscimento vocale), che realizzano il contatto (invio di comandi e ricezione informazioni) con il sistema intelligente di controllo, basato su un'unità computerizzata centrale oppure su un sistema a intelligenza

distribuita. I diversi componenti del sistema sono connessi tra di loro e con il sistema di controllo tramite vari tipi di interconnessione (ad esempio rete locale, onde convogliate, onde radio, bus dedicato, ecc.).

Il sistema di controllo centralizzato, oppure l'insieme delle periferiche in un sistema ad intelligenza distribuita, provvede a svolgere i comandi impartiti dall'utente (ad esempio accensione luce cucina oppure apertura tapparella sala), a monitorare continuamente i parametri ambientali (come allagamento oppure presenza di gas), a gestire in maniera autonoma alcune regolazioni (ad esempio temperatura) e a generare eventuali segnalazioni all'utente o ai servizi di teleassistenza. I sistemi di automazione sono di solito predisposti affinché ogniqualvolta venga azionato un comando, all'utente ne giunga comunicazione attraverso un segnale visivo di avviso/conferma dell'operazione effettuata (ad esempio LED colorati negli interruttori, cambiamenti nella grafica del touch screen) oppure, nei casi di sistemi per disabili, con altri tipi di segnalazione (ad esempio sonora o tattile).

Un sistema domotico si completa, di solito, attraverso uno o più sistemi di comunicazione con il mondo esterno (ad esempio messaggi telefonici preregistrati, SMS, generazione automatica di pagine web o e-mail) per permetterne il controllo e la visualizzazione dello stato anche da remoto. Sistemi comunicativi di questo tipo, chiamati gateway o residential gateway, svolgono la funzione di router avanzati, permettono la connessione di tutta la rete domestica al mondo esterno, e, quindi, alle reti di pubblico dominio.

Caratteristiche

Le soluzioni tecnologiche che possono essere adottate per la realizzazione di un sistema domotico sono caratterizzate da peculiarità d'uso proprie degli oggetti casalinghi: Semplicità, Continuità di funzionamento, affidabilità, basso costo.
Le tecnologie per la domotica permettono inoltre di ottenere alcuni vantaggi quali ad esempio:

- **Risparmio energetico**: un sistema completamente automatizzato dovrà evitare i costi generati da sprechi energetici dovuti a dimenticanze o ad altre situazioni.

- **Automatizzazione di azioni quotidiane**: un sistema di ***home automation*** deve semplificare alcune azioni quotidiane, soprattutto quelle ripetitive, quindi non deve in alcun modo complicarle.

Tutte queste caratteristiche, se non sviluppate singolarmente ma nel loro insieme, portano alla creazione di un sistema domotico integrato che può semplificare la vita all'interno delle abitazioni. La casa diventa intelligente non perché vi sono installati sistemi intelligenti, ma perché il sistema intelligente di cui è dotata è capace di controllare e gestire in modo facile il funzionamento degli impianti presenti. Attualmente, le apparecchiature tecnologiche sono poco integrate tra loro e il controllo è ancora ampiamente manuale. Nella casa domotica gli apparati sono comandati da un unico sistema automatizzato che ne realizza un controllo intelligente.

Per quanto riguarda il sistema di automazione, fondamentalmente ne esistono di due tipi, **uno basato su un'unità di elaborazione centrale,** che permette di gestire tutte le attuazioni a partire dai risultati di rilevazione, e **uno a struttura distribuita,** dove le interazioni avvengono localmente in maniera distribuita ed eventualmente comunicate ad un'unità centrale per un controllo di coerenza generale; in genere, sistemi di questo tipo sono più affidabili dei primi.

L'interfaccia utente (interfaccia uomo-macchina) deve, in base a tutte le precedenti considerazioni, essere consistente (non deve creare conflitti fra i comandi), essere di facile impiego (si pensi ai bambini o agli anziani) ed essere gradevole (la difficoltà di interazione con il sistema non deve essere una barriera al suo utilizzo).

Automazione dell'edificio

Come detto in precedenza un settore di grande rilievo anche per le implicazioni tecnologiche e di mercato è quello della ***building automation***. Questa espressione si riferisce soprattutto alla definizione dell'automazione di edifici di dimensioni medio-grandi e di struttura complessa, per uso privato o sociale (complessi residenziali, alberghi, ospedali, centri commerciali, complessi di uffici), con obbiettivi sostanzialmente simili a quelli indicati per la domotica, ma con un maggiore accento sui problemi delle infrastrutture comuni, necessarie a supportare i vari livelli di automazione. I requisiti funzionali dell'edificio, per

produttori, progettisti e integratori di automazione dell'edificio, possono essere raggruppati in alcune aree:

- **Impianti tecnologici** (climatizzazione, continuità e risparmio dell'energia elettrica, impianti di trasporto delle persone e delle cose, distribuzione dei fluidi);
- **Sistemi di sicurezza** (rilevamento fumo e incendio, antintrusione, controllo degli accessi, protezioni perimetrali, ecc.);
- **Sistemi informatici distribuiti** (automazione dell'ufficio, trattamento dei documenti, rilevamento presenze, ecc.);
- **Sistemi telematici** (telefono, telefax, posta elettronica, videotel, banche dati, banca elettronica, autostrade informatiche, ecc.);
- **Sistemi di comunicazione a banda larga**, parzialmente sovrapposti ai precedenti che includono la distribuzione e l'utilizzo di flussi di informazione rilevanti, per l'impiego di internet a banda larga e per applicazioni con flussi multimediali (televisione via cavo o satellitare, intrattenimento).

Automazione dell'ufficio

Rispetto all'automazione domestica, quella dell'ufficio comprende funzioni caratterizzate da un più elevato contenuto informativo, legato alla creazione e alla trasmissione di testi, alla gestione di moduli e di archivi, alla copiatura e stampa di documenti, alla elaborazione e presentazione grafica di dati e documenti in genere ecc., ottenuta per mezzo di elaboratori e relativi programmi, mezzi di comunicazione, generalmente interconnessi attraverso una rete locale che, con un adeguato software, realizza di fatto l'integrazione e l'ottimizzazione delle diverse attività.

2.2 LA DOMOTICA BTICINO

BTicino S.p.A. è un'azienda metalmeccanica italiana che opera nel settore delle apparecchiature elettriche in bassa tensione destinate agli spazi abitativi, di lavoro e di produzione, distinte in soluzioni per la distribuzione dell'energia, per la comunicazione (citofonia, videocitofonia, dati) e per il controllo di luce,

audio, clima e sicurezza. Dal 1989 è stata acquisita dal gruppo industriale Legrand.

L'azienda fa parte del gruppo internazionale Legrand e opera sia sul mercato italiano che su quello mondiale, con oltre 60 sedi all'estero.

I prodotti principali nel campo del materiale elettrico sono: *Interruttori tradizionali e domotici, Placche, Salvavita, Citofoni e Videocitofoni*.

La sede principale è a Varese, dove lavorano 1500 dipendenti, suddivisi tra progettazione, produzione, qualità, test, marketing, uffici amministrativi e commerciali. Altre sedi importanti sono quella di Erba (Italia), dove vengono sviluppati e realizzati prodotti tutti i prodotti domotici e di videocitofonia, Ospedaletto Lodigiano, dove è situato il magazzino centrale, Azzano San Paolo e Torre del Greco, dove vengono sviluppati e realizzati i dispositivi industriali e salvavita.

BTicino è presente su molti mercati esteri: Europa (Belgio, Spagna, Svizzera), Sud America (Messico, Costa Rica, Venezuela, Cile), Asia (Tailandia, Cina) Africa (Egitto).

2.2.1 MyHome

My Home è la domotica di BTicino e rappresenta il nuovo modo di progettare l'impianto elettrico. My Home amplia le possibilità della progettualità e rende la vita più semplice e funzionale, grazie all'utilizzo di un'unica tecnologia impiantistica, il Bus digitale. L'impianto realizzato su Bus si applica in qualsiasi contesto abitativo e terziario con soluzioni evolute in termini di comfort, sicurezza, risparmio energetico, multimedialità e controllo locale o a distanza. La modularità "installativa" e l'integrazione funzionale dei diversi dispositivi offrono la libertà di scegliere quali applicazioni adottare fin da subito e quali integrare nel futuro, senza importanti interventi strutturali e con un'ottima gestione dei costi.

Il Sistema

Il sistema My Home è un impianto a Bus caratterizzato da dispositivi intelligenti collegati fra loro mediante una linea di segnale, dedicata sia allo scambio delle

informazioni che al trasporto della tensione di alimentazione. Il supporto fisico è costituito da un cavo a coppie ritorte e non schermato da 27 V, che, posato insieme ai cavi di energia, collega in parallelo tutti i dispositivi del sistema a BUS.

Il sistema si compone di alimentatore, attuatori e comandi. L'alimentatore fornisce energia elettrica alla linea Bus in bassa tensione (27 V d.c), gli attuatori sono preposti al controllo dei carichi e sono collegati alla linea Bus e alla linea di potenza (230 V a.c.), i comandi sono collegati unicamente al Bus e forniscono agli attuatori, secondo configurazione, l'istruzione su quale operazione svolgere. In funzione dei sistemi My Home che si desidera installare, è possibile scegliere tra due diverse tipologie di cablaggio: - con struttura libera - con struttura a stella Il cablaggio con struttura libera si applica quando si devono installare gli impianti Automazione, Antifurto, Risparmio Energia e Termoregolazione. Il cablaggio con struttura a stella è indicato per l'integrazione in un'unica conduttura degli impianti di videocitofonia, di diffusione sonora, di telefonia, TV/SAT e di trasmissione dati.

La Tecnologia

Ogni dispositivo connesso al sistema è dotato di circuito di interfaccia e di una propria intelligenza, costituita da un microprocessore programmato, per mezzo del quale il dispositivo è in grado di riconoscere l'informazione a lui destinata ed elaborarla per realizzare la funzione desiderata. Affinché ciascun dispositivo in un sistema a BUS svolga correttamente la funzione preposta, esso deve essere opportunamente programmato assegnando il rispettivo identificativo e modalità di funzionamento.

Questa procedura, denominata configurazione, può essere effettuata scegliendo tra le due modalità:

- Configurazione fisica, inserendo, in apposite sedi dei dispositivi ad innesto denominati configuratori.
- Configurazione virtuale, grazie al software denominato VIRTUAL CONFIGURATOR, installato in un PC palmare, che permette inoltre la diagnostica dei dispositivi.

Audio e Video

La moderna tecnologia di My Home mette a disposizione svariati strumenti di visualizzazione locale o a distanza, per tenere tutto sotto controllo quando si è in casa o lontano da essa.

Dispositivi videocitofonici digitali permettono di comunicare chiaramente con chi si presenta all'ingresso, anche attraverso messaggi di segreteria videocitofonica in caso di assenza del proprietario (che a distanza potrà fruire del trasferimento di chiamata piuttosto che del messaggio registrato dal visitatore).

Controllo a Distanza

Ciascuno può scegliere il tipo di attivazioni, informazioni e notifiche da dare e ricevere dalla casa, in funzione delle proprie specifiche necessità e sulla base del tipo connessione presente nella stessa (ADSL, Linea Telefonica) e dei dispositivi di controllo My Home collegati.

Controllo Locale

Consente di gestire le soluzioni My Home della propria casa dallo schermo di un PC o dallo schermo tattile del video touchscreen da 15 pollici, utilizzando un software fornito a corredo con alcuni dispositivi e costruendo un sinottico che riproduce la piantina o le foto dei locali della casa con le icone dei relativi punti luce, automatismi, allarmi eccetera.

Anche attraverso le varie schermate del touchscreen è comunque sempre possibile effettuare il comando locale delle soluzioni My Home della casa.

2.2.2 Il problema

L'impianto domotico in questione è ancora in una fase di sviluppo, in quanto si cerca sempre di aggiungere nuove tecnologie, nuove funzioni, e quindi non ci sarà mai una versione definitiva, ma solo un'ultima versione in previsione della successiva.

Come detto, tutte le chiamate vengono inoltrate verso il posto interno presente nell'abitazione, mentre non è stato ancora sviluppato un sistema di inoltro della chiamata da videocitofono ad un dispositivo mobile.

Scopo principale del nostro studio è proprio quello di permettere al sistema MY HOME di Bticino di comunicare con un device Android, non attraverso il browser, ma tramite un'applicazione nativa.

L'utilizzo dell'applicativo, ovviamente, non andrà a sostituire l'utilizzo del posto interno, ma fornirà un supporto aggiuntivo per l'utente, in modo da migliorare ancor di più l'usabilità del sistema in questione.

All'interno del nostro elaborato analizzeremo due differenti scenari, entrambi realizzati per funzionare su rete LAN:

- ***Scenario SIP***: il videocitofono esterno comunica con un device Android attraverso il protocollo SIP.
- ***Scenario OpenWebNet***: il videocitofono esterno comunica con un device Android attraverso un web server F454 (BTicino), sfruttando il protocollo OPEN.

Ulteriore attenzione verrà posta poi sulla possibilità di sfruttare le potenzialità degli applicativi anche in remoto (rete WAN) e non solo su rete LAN. Nel capitolo *sviluppi futuri* analizzeremo il problema in modo che la comunicazione tra posto esterno e dispositivo mobile avvenga attraverso le moderne reti di comunicazione cellulare.

3 ANALISI DELLE TECNOLOGIE UTILIZZATE

3.1 VOIP - VOICE OVER IP

Per VoIP (Voice Over Internet Protocol, letteralmente voce attraverso IP, il quale è il protocollo usato per Internet) si intende una tecnologia che renda possibile effettuare una conversazione telefonica tramite una connessione internet o rete dedicata a commutazione di pacchetto, e che utilizzi il protocollo IP senza connessione per il trasporto dei dati. VoIP consente una comunicazione sia audio che video, unicast o multicast su rete a pacchetto. È possibile usufruire di tali servizi mediante:

- Un' 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 (Wi-Fi, GPRS o UMTS);

In questo tipo di comunicazioni basate su Internet, e quindi sulla combinazione TCP/IP, i problemi principali che possono manifestarsi riguardano la qualità della trasmissione dei dati e la gestione dei pacchetti trasmessi. I dati vengono suddivisi in pacchetti e spediti attraverso Internet, per essere poi ricostruiti in ricezione. Per la comunicazione vocale, le difficoltà sono legate alla latenza, al jitter e all'integrità dei dati.

La tecnologia VoIP utilizza due protocolli di comunicazione che funzionano in parallelo, uno utilizzato per il trasporto dei dati (i pacchetti voci attraverso IP) e come secondo, nella maggioranza dei casi il protocollo RTP (RealTime Transport Protocol) e sue estensioni (SRTP, ZRTP). La prima tipologia di protocollo è necessaria per l'instaurazione della comunicazione e quindi per la segnalazione che assiste la conversazione. I protocolli più utilizzati sono H.323, elaborato in

ambito ITU (International Telecommunications Union), e SIP (Session Initiation Protocol) presentata dal IETF (Internet Engineering Task Force).

3.2 SIP - SESSION INITIATION PROTOCOL

Il protocollo SIP (acronimo di 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.

Il protocollo SIP 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'inter-connettività 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 (nell'ordine):

- 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 (UA), ovvero endpoint che possono fungere sia da client sia da server. Quando assume il ruolo di client, User Agent Client (UAC), dà 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 e interscambiabili, nel senso che durante il corso di una sessione, un client può fungere da server e viceversa (lo scambio dei ruoli avviene in modo automatico, a seconda delle esigenze, in quanto entrambi gli agenti hanno sia la possibilità di chiamata e

quindi di iniziare la sessione o processare richieste, sia la possibilità di essere chiamati ed effettuare risposte).

3.2.1 Caratteristiche

Il protocollo SIP supporta cinque caratteristiche di creazione e terminazione delle chiamate:

- ***User location***: determina se l'end user è disponibile per la comunicazione.
- ***User availability***: determina la disponibilità degli utenti ad instaurare la comunicazione.
- ***User capabilities***: determina i parametri media da usare.
- ***Session setup***: stabilisce i parametri di sessione che entrambi i chiamanti devono usare.
- ***Session management***: include il trasferimento e la terminazione della sessione, modifica i parametri e invoca i vari servizi.

Tipicamente SIP viene utilizzato in un'architettura con altri protocolli quali RTP (real-time transport protocol, per trasportare i dati e fornire un QoS), RTSP (real-time streaming protocol) per controllare la consegna dello streaming, MEGACO (media gateway control protocol, per controllare i gateways), SDP (session description protocol) per descrivere le sessioni multimediali e UDP (user datagram protocol). Le revisioni degli standard permettono però l'utilizzo anche di TCP e TLS oltre ad UDP, e la decisione spetta allo user agent client (il terminale mittente).

Oltre alle funzioni base del protocollo, come la localizzazione degli utente e l'invito alla partecipazione seguito da negoziazione della sessione, grazie ad alcune estensioni, il protocollo SIP può pubblicare ed aggiornare le informazioni di presenza, notificare l'evento di presenza, richiedere il trasporto di informazioni di presenza e trasportare messaggi istantanei.

SIP può essere impiegato sia in architetture client-server che in contesti peer-to-peer, e può avere server sia stateless che stateful.

3.2.2 Architetture e Protocolli

Di seguito verranno elencati i principali attori del protocollo SIP, e i protocolli usati contemporaneamente in una sessione SIP (sia durante la negoziazione come può essere per SDP, che durante un dialog per un media stream come può essere RTP).

- ***SIP user agent***: è il punto di accesso ad un sistema SIP. Può fungere sia da client che da server, i due ruoli sono dinamici durante la sessione, nel senso che in un certo momento può funzionare come se fosse il server e in un altro momento assumere le funzioni del client. Quando funge da client da origine alla transazione originando richieste, quando funge da server invece raccoglie le richieste e se possibile le soddisfa.
- **SIP *server***: è un server dedicato o condiviso. Può essere di tre tipologie: registrar server, proxy server o redirect server. Il primo viene utilizzato per la registrazione di un utente, il secondo come server intermedio per analizzare i vari parametri di instradamento, il terzo per re instradare le richieste SIP. Il proxy server può essere di tipo stateless oppure stateful.
- ***UDP***: è un protocollo di livello trasporto, che si affida al livello sottostante IP per la trasmissione e la ricezione di pacchetti (datagram). È un protocollo connectionless, ovvero orientato alla trasmissione, cioè non garantisce una trasmissione affidabile e nemmeno il loro ordinamento. UDP è compatibile e usato anche per applicazioni che effettuano trasmissioni broadcast e multicast. La perdita di pacchetti è tollerabile entro certi limiti al fine di garantire QoS.
- ***SDP***: è un protocollo utilizzato per descrivere i parametri di inizializzazione di streaming media. Gestisce l'annuncio, l'invito e altri metodi per inizializzare una sessione multimediale. Non si occupa del trasporto dei dati, ma permette di negoziare i parametri di sessione, come il tipo, formati, e codec.
- ***RTP***: è un protocollo di livello applicativo utilizzato per trasmissioni real-time di dati come audio e video. Non fornisce meccanismi per garantire la qualità del servizio, ma fa affidamento ai protocolli di livello inferiore per gestire tali problemi. Non vengono garantite né la consegna affidabile dei datagram né il loro ordine di arrivo sequenziale, tuttavia il numero di sequenza presente nell'header permette al ricevente di

ricostruirne l'ordine corretto. Insieme a RTP viene utilizzato il protocollo RTCP per monitore QoS e trasmettere informazioni ai partecipanti della sessione.

- ***SRTP***: è un profilo per RTP che garantisce la proprietà di confidenzialità, autenticazione dei messaggi e protezione da replay attack. Il protocollo mette a disposizione un framework per la cifratura e autenticazione dei messaggi. SRTP può garantire un livello elevato di throughput e una ridotta espansione dei pacchetti.

3.2.3 I Messaggi

Per instaurare una sessione avviene una three-way-handshake (tipo quella che avviene nel protocollo TCP). Gli utenti SIP sono risorse identificabili mediante URI o URL che contengono informazioni sul dominio, sull'utente, sull'host o sul numero col quale l'utente partecipa alla sessione. Gli indirizzi hanno uno stile tipo le email e possono apparire in diversi formati:

```
sip: nome@123.100.100.100
sip: nome@dominio.it
sip: 123456@dominio.it
```

Un messaggio SIP può essere una richiesta (request) o una risposta (response). Una sequenza composta da una richiesta e da una o più response è detta transazione, la quale è identificabile tramite un transaction-ID, che ne specifica la sorgente, destinazione e numero di sequenza. Entrambe le tipologie di messaggio sono costituite da una start-line, uno o più campi di intestazioni (header), e il corpo vero e proprio del messaggio che è da considerare però opzionale (body):

- **Request**: i messaggi request hanno come start-line una request-line come quella nell'esempio, contenente il metodo di richiesta, l'URI e la versione del protocollo utilizzata:

```
request-line = <<method>> <<request-URI>> <<SIP-
version>>
```

Secondo le specifiche SIP sono previsti i seguente metodi di richiesta:

- **register**: inviato da uno UA per registrare presso un registrar server il proprio punto di accesso alla rete.
- **invite**: serve ad invitare un utente a partecipare ad una sessione.
- **ack**: è un messaggio di riscontro, è inviato dallo UA chiamante, verso lo UA chiamato, per confermare la ricezione di una risposta ad un INVITE.
- **cancel**: serve a terminare un dialogo quando la sessione non ha avuto inizio.
- **bye**: utilizzato per terminare un dialogo SIP.
- **options**: è utilizzato per interrogare uno UA riguardo alle sue funzionalità, in questo modo il chiamante può decidere il tipo di comunicazione da instaurare.
- **prack**: consente ad un client di riscontrare la ricezione delle risposte provvisorie.
- **refer**: lo user agent che lo riceve trova nell'intestazione refer-to una nuova SIP URI, dopo aver chiesto conferma, viene contattato il nuovo SIP URI, ed il mittente del REFER viene notificato dell'esito con un messaggio NOTIFY
- **subscribe**: consente a chi lo invia di manifestare l'interesse a ricevere delle notifiche (appunto tramite messaggi NOTIFY) riguardanti l'evoluzione di alcuni variabili di stato, indicate mediante l'intestazione event; è utilizzato per e-presence.
- **notify**: tiene uno UA al corrente dell'evoluzione di alcune variabili di stato, può essere inviato anche senza aver prima ricevuto un messaggio SUBSCRIBE
- **message**: permette l'invio di messaggi instantanei, ospitati nel body, e descritti da un'intestazione content-type
- **update**: consente ad un client di aggiornare i parametri di una sessione senza modificare lo stato della sessione stessa. il metodo è usato dopo un messaggio di INVITE, ma prima che la sessione sia stata instaurata

  - **info**: utilizzato per inviare ad uno UA, con cui si è già instaurata una sessione, delle informazioni relative ad eventi che avvengono dall'altro lato, come ad esempio la pressione dei tasti del telefono.

- **Response**: i messaggi di response hanno come start-line una status-line costruita come nell'esempio seguente:

```
status-line = <<SIP-version>> <<status-code>> <<reason-phrase>>
```

  Essa è costituita dalla versione del protocollo usata, un numero intero di tre cifre (status code) ed una frase opzionale a commento della risposta. Lo status-code indica il tipo di risposta. le tipologie di risposta si possono distinguere in sei classi, e sono le seguenti:

  - **1xx - provisional**: risposte provvisorie necessarie ad interrompere i timer di ritrasmissione. Le principali sono: trying (100), ringing (180), call is being forwarded (181), queued (182) e session progress (183).
  - **2xx - succesfull**: indica che l'operazione è avvenuta con successo, ne fa parte il messaggio di OK (200).
  - **3xx - redirection**: richieste di redirezione della richiesta. In questo caso è il client (lo UA o il proxy che ha inoltrato la chiamata) che si deve occupare di richiamare l'indirizzo specificato. Alcuni esempi sono: multiple choises (300), moved permanently (301), moved temporarily (302), use proxy (305), alternative service (380).
  - **4xx - request failure**: la richiesta non può essere soddisfatta perchè contiene qualche errore sintattico. I principali sono: bad request (400), unauthorized (401), payment required (402), forbidden (403), method not allowed (405), not acceptable (406), proxy authentication required (407), request timeout (408), gone (410), request entity too (413), request URI too long (414), unsupported media type (415), unsupported URI scheme (416), bad extension (421), extension required (420), interval too (423), temporary unavailable (480), call does not exist (481), loop detected (482),

address incompete (484), ambiguous (485), busy here (486), request terminated (487), not acceptable here (488), request pending (491), undecipherable (493).

- **5xx - server failure**: la richiesta appare valida ma non può essere soddisfatta per un problema interno del server. Abbiamo: Server Internal error (500), not implemented (501), bad gateway (502), service unavaible (503), server timeout (504), version not supported (505), message too large (513).
- **6xx - global failure**: la richiesta non può essere accettata da parte di nessun server. I principali sono: busy everywhere (600), decline (603), does not exist anywhere (604) e not acceptable (606).

Per quanto riguarda le intestazioni, sono rappresentate nel seguente formato:

```
<<header-name>>: <<header-value>> (, <<header-value>>)
```

Dove hader-name è il nome dell'intestazione, ed header-value il valore, il terzo parametro significa che l'intestazione può avere più di un valore, inoltre per ogni valore possono essere specificati dei parametri aggiuntivi, ognuno col proprio valore e separati da punto e virgola:

```
<<header-name>>: <<value (; <<parameter-name>>=<<parameter-
value>>) >>
```

Dove <<value>> è il valore dell'intestazione, parameter-name è il nome del parametro e parameter-value è il valore corrispondente. Le intestazioni più importanti sono:

- ***to***: indica la URI del destinatario della richiesta.
- ***from***: rappresenta la URI di chi invia la richiesta.
- ***call-id***: è un identificatore semi-casuale, che resta uguale per tutti i messaggi di uno stesso dialogo, ovvero è univocamente associato ad un INVITE iniziale.

- ***cseq***: è costituita da un numero, seguito dal nome del metodo che ha dato inizio alla transazione.
- ***via***: è inserita da ogni elemento che invia una richiesta SIP, in cui indica il proprio indirizzo, porta, trasporto. Ogni elemento di trannsito che deve inoltrare la risposta rimuove l'intestazione da lui inserita, ed usa quella in cima per determinare a chi inviarla. In questo modo non occorre consultare il DNS, ed è sufficiente un proxy stateless.
- ***max-forwards***: utile per limitare il numero di volte che un messaggio è inoltrato.
- ***contact***: contiene uno o più URI presso le quali il mittente di una richiesta desidera essere ricontattato.
- ***allow***: annuncia i metodi supportati da un'entità.
- ***supported***: elenca le estensioni supportate tra quelle elencate presso IANA.
- ***record-route***: specifica la volontà di un proxy di essere mantenuto nel path dei futuri messaggi del dialogo.

3.2.4 Esempio di session flow SIP

Nella Figura 4, si può osservare un classico flow SIP tra due user agent, che comunicano tra di loro direttamente, cioè senza un intermediario, attraverso un network IP (il canale di comunicazione come specificato in precedenza non è obbligatorio che sia necessariamente una rete IP). Il primo passo lo fa lo user agent P1 chiamando tramite il suo apposito URI lo user agent P2 mandando il messaggio di INVITE per invitarlo ad unirsi alla sessione, P2 risposte con una serie di messaggi di risposta come trying, ringing per far capire all'utente che sta instaurando la connessione, e il messaggio 200 di OK, a quel punto P1 invia a P2 un messaggio di ack (acknowledge) di riconoscimento della connessione e a quel punto può partire la comunicazione stream real time (tramite RTP o altri), dopodichè uno user agent (in questo caso P1) invia il messaggio di BYE per iniziare la chiusura della sessione, e l'altro user agent invia il messaggio 200 di ack e termina la comunicazione. Vi è la possibilità ovviamente che ci siano nel mezzo della rete altri agenti/server proxy che re-dirigono la comunicazione.

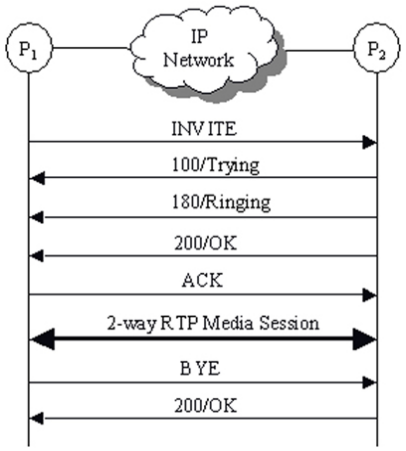


Figura 4: messaggi SIP

3.2.5 Esempio di messaggio SIP header/body

```
INVITE sip:nome@dominio.it SIP/2.0
Via: SIP/2.0/UDP sipserv.elet.polimi.it
From: sip:user@dominio.com
To: sip:sip:nome@dominio.it
Call-Id: user@dominio.com
CSeq: 1 INVITE
Subject: meeting.
Content-Type: application/sdp
Content-Length: ...
Contact: user@dominio.com
<corpo messaggio SDP>
```

I messaggi SDP sono trasportati in modo trasparente all'interno dei messaggi di SIP e possono includere:

- Nome e scopo della sessione
- Durate della sessione
- I vari media utilizzati
- L'informazione per ricevere tali media (porte, indirizzi)
- Informazioni sulla banda
- Informazioni di contatto

Di seguito alcuni parametri usati nel protocollo SDP:

```
v= (protocol version)
o= <username> <session id> <version> <network type> <address
type> <address>
s= <session name>
i= <session description>
u=<URI> (pointer to additional information about the
conference)
e=<email address> (contact information for the person
responsible for the conference)
c=<network type> <address type> <connection address>
t=<start time> <stop time>
a=<attribute>:<value> (primary means for extending SDP)
m=<media> <port> <transport> <fmt list>
```

Un esempio completo

```
INVITE sip:watson@boston.bell-tel.com SIP/2.0
Via: SIP/2.0/UDP kton.bell-tel.com
From: A. Bell <sip:a.g.bell@bell-tel.com>
To: T. Watson <sip:watson@bell-tel.com>
Call-ID: 3298420296@kton.bell-tel.com
CSeq: 1 INVITE (il numero numera le transizioni)
Subject: Mr. Watson, come here.
Content-Type: application/sdp
Content-Length: <lunghezza messaggio>
v=0 (versione)
o=bell 53655765 2353687637 IN IP4 128.3.4.5 (origine)
s=Mr. Watson, come here. (sessione)
c=IN IP4 kton.bell-tel.com (con. address)
m=audio 3456 RTP/AVP 0 3 4 5 (PCM, GSM, G.723, DVI4) (media)
```

3.2.6 Librerie/Stack SIP

Allo stato attuale del lavoro, esistono e vengono utilizzate per vari progetti varie librerie che permettono di creare uno stack SIP e di gestire così la creazione di una sessione tramite tale protocollo standard. Alcune forniscono funzioni standard e a basso livello, mentre altre si concentrano sulla creazione della comunicazione, lasciando all'oscuro l'utente (programmatore) il contenuto dei vari messaggi SIP veri e propri, fornendo le funzioni essenziali. Di

seguito verranno elencate quali librerie sono state analizzate e le loro caratteristiche.

SIP Stack di Android

Dalla versione di Android 2.3 (e quindi a partire dalla versione 9 delle API Android), sono fornite nativamente le API per il supporto del protocollo SIP.

Da quel momento Android include un vero e proprio stack SIP il quale permette di integrare servizi di gestione chiamate (in entrata e in uscita), senza preoccuparsi del livello di trasporto, della gestione della sessione e dei vari messaggi per instaurarla.

Viene fornita un'interfaccia programmazione molto ad alto livello, dove le classi principali sono il SipManager, il SipProfile e il SipAudioCall.

Il primo, SipManager, fornisce tutte quelle funzioni necessarie ad instaurare una sessione SIP, Il SipProfile serve invece ad identificare i vari agenti (crearli e gestirli) inclusa la registrazione ed autenticazione

```
SipProfile.Builder builder = new SipProfile.Builder(username,
domain);
builder.setPassword(password);
mSipProfile = builder.build();
mentre con SipAudioCall viene gestita la vera e propria
chiamata.
call = mSipManager.makeAudioCall(mSipProfile.getUriString(),
sipAddress, listener, 30);
```

Essendo funzioni ad alto livello, non è possibile intervenire sulla decisione del SIP flow, e quindi inviare autonomamente un messaggio di INVITE, oppure un messaggio di ACK, OK o quant'altro; si deve fare affidamento sulle funzioni predefinite, che si arrangiano a gestire tali messaggi, fornendo al più una stringa per descrivere la sessione

LibLinphone

Linphone è un software VoIP molto famoso, ed è disponibile per vari sistemi operativi. Questo software utilizza il protocollo SIP per creare la sessione di

comunicazione, e gli sviluppatori hanno creato una propria libreria opensource e disponibile a tutti: LibLinphone. Anche questa libreria fornisce diverse funzioni ad alto livello per gestire una sessione SIP con facilità e in maniera automatica come se fosse una semplice chiamata. Inoltre, supporta vari altri sistemi quali l'autenticazione tramite digest, il DTMF (dual-tone multi-frequency). Permette chiamate multiple simultanee, trattenuta di chiamata, trasferimento di chiamata ecc.
Mette a disposizione inoltre una propria libreria per la gestione dello stream video: Mediastreamer2, la quale supporta la maggior parte dei codec Audio e Video conosciuti. Supporta SRTP e zRTP (criptazione di audio e video), cancellazione di eco, gestione efficiente della banda e può essere "estesa" con altri plugin per esempio per aggiungere supporto ad altri codec o altre funzionalità.

PjSIP

PjSip è una libreria scritta in linguaggio C, Free e Open Source, che implementa i potocolli standard quali SIP, SDP, RTP, STUN, TURN e ICE. PjSip raccorda tutte le tre componenti principali delle applicazioni multimediali real-time che sono i segnali, i media e il NAT trasversal.

Essendo scritta in linguaggio C, è molto versatile e utilizzabile da tantissime piattaforme e sistemi operativi (Windows, Mac OS X sia PPC che Intel, Linux su qualsiasi processore, sistemi Unix, Nokia/Symbian, iOS di Apple, BlackBerry 10 e naturalmente Androido (pianificato dalla versione 2.2).

Con PjSip viene fornita sia una libreria a basso livello che un'interfaccia ad alto livello tramite PjSUA (e relative API). PjSUA è in pratica uno User Agent SIP a linea di comando scritto tramite lo stack PjSip. PjSip è costruito su PjLib e supporta i principali protocolli utilizzati in una sessione SIP; oltre ai già citati protocolli di trasporto quali UDP, TCP e TLS, supporta il routing/NAT, una gestione avanzata delle chiamate (deviazione, trattenuta, sipfrag, session timers ecc), gestione di istant messagging e altre estensioni come autenticazione AKA, INFO e ICE option tag.

Per quanto riguarda la gestione delle sessioni SIP, la libreria mette a disposizione tutte le possibili funzioni low e midlle level per gestirle, dalla parte di negoziazione fino al rilascio.

Avendo un'interfaccia anche a basso livello, per gestire i messaggi delle sessioni SIP vengono usati i metodi *Message* e *MessageBody*, in aggiunta alla gestione degli Header e dei codici degli stati (101, 200, ecc). Scambiandosi i messaggi per negoziare una sessione SIP è necessario analizzare il contenuto dove si trovano i vari parametri della sessione, per fare ciò vi sono disponibili i Parser. I Parser sono di tipo top-down e utilizzano lo scanner di PjLib, il quale è scalabile e molto veloce; usa un sistema di eccezioni try/catch che non lo semplifica, ma salva l'esecuzione da fastidiosi errori. Oltre al parser per il body del messaggio, vi sono anche i parser per l'header e per l'URI, di cui ci sono anche i manager e costruttori.

Gli indirizzi URI (Uniform Resource Indicator) sono modellati secondo uno schema ad oggetti, e vi sono vari tipi di URI come per esempio il base URI, l'URI SIP, il tel URI ecc.
Una volta creato il messaggio, è possibile inviarlo tramite le funzioni di invio che utilizzano il livello di trasporto, il quale fa uso di due buffer utilizati rispettivamente per i messaggi di request e i messaggi di response. Passando dai buffero vengono invocati gli eventi on_rx_response() e on_rx_request() che danno inizio a tutta la gestione di richieste/risposta.
Esiste poi un intero Framework per l'autenticazione dell'utente (ed eventualmente del server), per esempio quando si necessita di un'iscrizione presso un server SIP.

Più ad alto livello, troviamo invece dei metodi per gestire la Dialog Invite Session; la quale è usata dall'applicazione per gestire sessioni INVITE (incluso la gestione dei messaggi SDP), ed è costruita in maniera astratta dalle API basi del dialog. Inoltre dispone di un sistema di autenticazione automatico, di un session timout e come già anticipato di un handler per il protocollo SDP.

MJSIP

MjSip è una libreria SIP che fornisce allo stesso tempo delle API e un'implementazione di uno stack in uno stesso pacchetto. Include tutte le classi

e metodi per creare delle applicazioni basate su SIP, implementando uno stack con un'architettura su più livelli, inoltre fornisce un'interfaccia ad alto livello per il controllo delle chiamate. È formata da vari pacchetti che includono:

- Oggetti SIP standard come messaggi SIP, transazioni, dialogs…
- Alcune estensioni SIP già definite con lo standard IETF.
- API per il controllo delle chiamate.
- Un'implementazione dei sistemi SIP (sia server che User Agent).

È una libreria basata su java, molto leggera e può essere usata allo stesso tempo per implementare la parte server o la parte dei terminali; aggiunge all'interfaccia lower-leve (simile a jain-sip), un'interfaccia ad alto livello, e per la parte di implementazione server supporta sia gli stateless che stateful.

Seguendo le linee guida dello standard RFC 3261, il core di MjSip è strutturato su tre livelli base: trasporto, transaction e dialog, mentre più in alto è stato aggiunto il livello per il controllo delle chiamate.

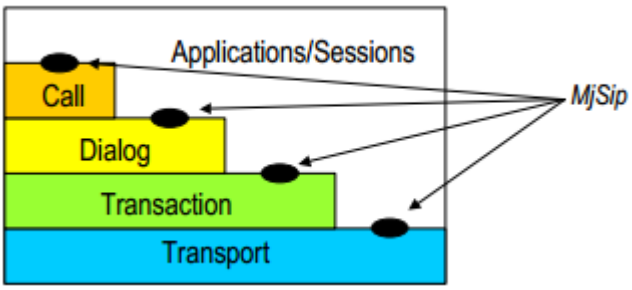


Figura 5: nome da decidere

Il livello più basso (transport) fornisce il trasporto dei messaggi SIP, attraverso l'oggetto SipProvider. È responsabile di inviare e ricevere i messaggi, ed ogni elemento SIP lo usano se vogliono accedere ai servizi di trasporto. Il livello Transaction invia e ricevere i messaggi attraverso il livello di trasporto, e serve per gestire le ritrasmissioni, i match tra risposte e richieste e i timeouts. Ci sono due tipi di transaction: two-way e three-way. Il livello Dialog unisce diverse

transazione in una stessa sessione. Facilita la sequenza di messaggi tra gli User Agents. Il livello più alto, il controllo chiamata (Call Control) implementa una completa chiamata SIP, offre una semplice interfaccia per gestire le chiamate in entrata e in uscita. Una chiamata consiste in più di un dialog.

Le API MjSip per i quattro layer sono implementate rispettivamente da:

- Classe Call (and class ExtendedCall)
- Classe InviteDialog
- Classi ClientTransaction, ServerTransaction, InviteClientTransaction, e InviteServerTransaction
- Classe SipProvider

Inoltre MjSip fornisce altri strumenti per gestire i messaggi SIP, la sintassi SDP (per la descrizione delle sessione) e la codifica:

- Classi Message, MessageFactory
- Classi Header, MultipleHeader, FromHeader, ToHeader, ViaHeader, ecc.
- Classi SessionDescription, Timer, ecc.

JAIN-SIP (JSIP)

Jain-Sip (abbreviato talvolta con JSip) è un insieme di librerie e API realizzate in Java per lo sviluppo di applicativi desktop e server basati sul protocollo SIP. Le JAIN (Java APIs for Integrated Networks, chiamate precedentemente Java APIs for Intelligent Network) sono API sviluppate in Java per la creazione di servizi telefonici.

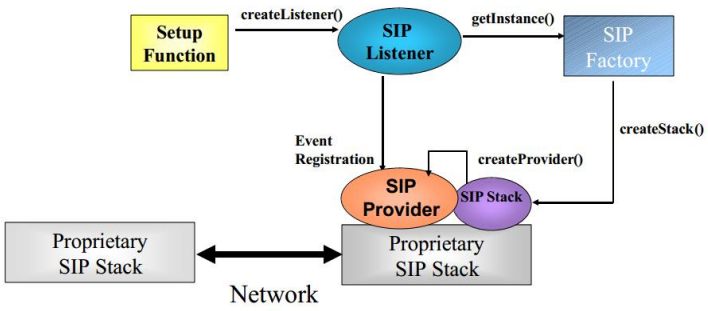


Figura 6: Architettura ad oggetti

Jain-Sip è stata creata per avere una vera interoperabilità tra diversi stack e diverse applicazioni, ed è l'interfaccia standard Java per uno stack SIP. Standardizza oltre all'interfaccia dello stack, anche l'interfaccia messaggi, la semantica degli eventi e la portabilità delle applicazioni; e può essere usata in diversiti ambiti come user agents, proxy server (sia stateful che stateless), registrar server o embedded in un contenitore di servizi.

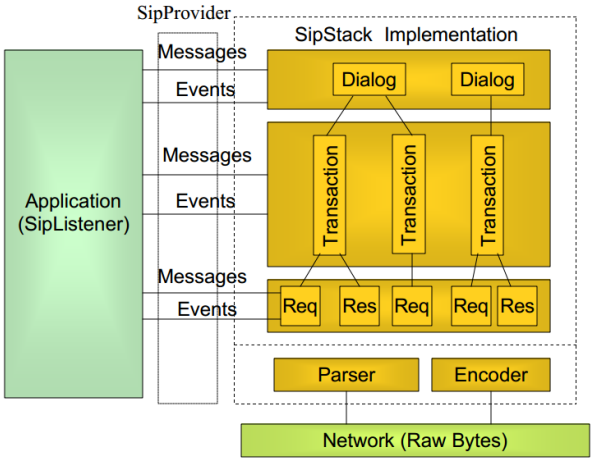


Figura 7: Implementation Structure

Jain-Sip fornisce metodi per formattare bene i messaggi SIP, e le funzioni per fare in modo che l'applicazione invii e riceva tali messaggi; fornisce inoltre un parser per i messaggi in arrivo per fare in modo che l'applicazione acceda ai campi del messaggio tramite un'interfaccia Java standardizzata. JSip invoca inoltre gli appropriati handler quando arrivano i messaggi o i time-out delle transazioni, fornisce supporto e gestione alle Transazioni e ai Dialog.
Gli attori principali di tutta la libreria sono il SIpProvider, il SIp Stack che crea il precedente, il SIplistener, il SipFactory e poi tutta una seria di funzioni.
L'interfaccia SipStack è associata ad un indirizzo IP e può avere più punti di ascolto, un'applicazione invece può avere più SipStack. L'interfaccia è istanziata dal SipFactory e definisce opzioni di ritrasmissione e informazioni di instradamento. Tutte le proprietà sono riservate e contenuto nel pacchetto javax.sip.*, il quale è stato inglobato nelle versioni più recenti di Android, e quindi occorre compilarlo e associarlo con un altro nome per non avere conflitti di librerie. Esiste un unico SIpListener per ogni SipStack, e ciò implica

che ve ne è uno per una singola architettura, gestisce le notifiche del processo delle Transazioni.

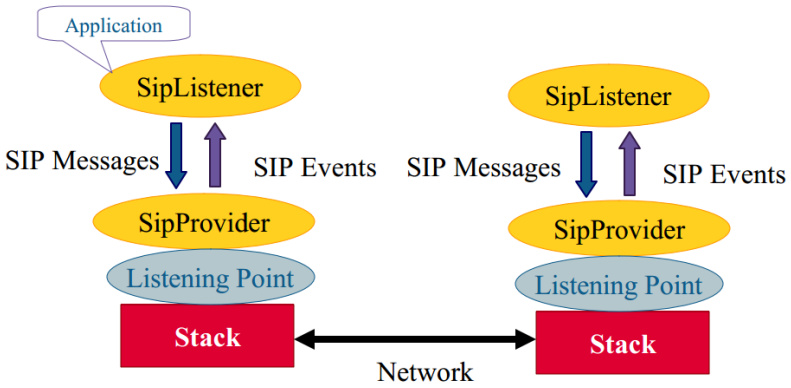


Figura 8: comunicazione jain-sip

Dato che l'architettura è sviluppata per un ambiente J2SE è basato su un modello ad eventi listener/provider, c'è uno scambio diretto tra il fornitore di eventi e il suo fruitore, il quale deve essere registrato col primo. Gli eventi "incapsulano" le richieste e le risposte, e il modello degli eventi è asincrono e usa un identificatore transazionale per correlare i messaggi. Il SipListener rappresenta il fruitore degli eventi e, resta in ascolto per gli eventi in arrivo che incapsulano i messaggi che possono dare il via ad un dialogo, il SipProvider invece è il fornitore di tali eventi, il quale riceve i messaggi dal network e li passa all'applicazione tramite appunto gli eventi.

Ci sono differenti tipi di Packages: quello generale, che definisce l'architettura dell'interfaccia, le transazioni, i dialog e gli eventi; il package degli indirizzi, che contiene il wrapper URI e definisce il SIP URI e l'interfaccia per i Tel URIs; il package per i messaggi, che definisce l'interfaccia necessaria per i messaggi Request e Response; e infine il package per gli header.

Jain-Sip definisce, inoltre, quattro tipi di "factory" differenti:

- SipFactory, definisce i metodi per creare un nuovo stack object e gli altri factory object.
- AddressFactory, definisce i metodi per creare SipURi e TelURI
- MessageFactory, definisce i metodi per creare nuovi Request e Response object

- HeaderFactory, definisce i metodi per creare nuovi Header object

Ci sono due tipologie di messaggi in SIP, quindi Jain-Sip definisce due tipi di interfacce:

- I messaggi Request sono inviati dal client al server, contengono metidi specifici e sono identificati come tipo di richiesta, una Request-URI indica l'utente o il servizio per il quale la request è indirizzata
- I messaggi Response sono inviati dal server al client in risposta ad una Request, contengono uno specifico status code che identifica il tipo di risposta, un Request-URI anche in questo caso identifica l'indirizzo, e una "reason phrase" viene aggiunta per essere compresa dagli essere umani e non dal codice dell'applicazione.

Un messaggio può contenere header multipli e header dello stesso tipo, e l'ordine degli header dentro un messaggio è significativo. Il corpo del messaggio contiene un descrittore di sessione (SDP) di cui Jain-Sip definisce il formato tramite un object che permette al body di essere una stringa o un oggetto definito dal SDP o alternativamente un array di byte.
Una transazione SIP consiste in una singola richiesta e in una qualsiasi risposta a tale richiesta. Jain-Sip standardizza l'interfaccia per un generico modello di transazione definito dal protocollo SIP (sia transazioni client che server). Una transazione è creata all'arrivo di una richiesta o può essere creata all'invio di tale richiesta. Quando una richiesta viene inviata statefully, l'applicazione deve richiedere una ClientTransaction, quando una nuova richiesta arriva, l'applicazione determina se gestirla tramite un ServerTransaction, quando una richiesta arriva in un dialog già esistente, lo stack associa automaticamente la richiesta al ServerTransaction. Quando arriva una risposta, lo stack la associa possibilmente ad un ClientTransaction create precedentemente, i messaggi sono passati al SipProvider per generare una nuova Transaction.

Un Dialog è un'associazione peer to peer tra endpoint SIp in cui si interpretano messaggi SIP, e non sono mai creati dall'applicazione ma stabiliti creando Transaction e gestiti dallo stack. I Dialog sono usati per mantenere dati necessari per messaggi futuri di ritrasmissione (Route sets, numeri di sequenza, URI ecc) e hanno una macchina a stati: early, confirmed, completed e terminated.

3.3 ANDROID

Android è un sistema operativo basato su Linux, progettato principalmente per device touchscreen, come smartphone e tablet. Il primo smartphone Android-powered viene venduto nell'Ottobre 2008. Android è open source e Google rilascia il codice tramite la Licenza Apache.

Le applicazioni sono sviluppate in linguaggio Java utilizzando l'Android Software Development Kit (SDK). Questo include un considerevole numero di strumenti di sviluppo, incluso un debugger, librerie software, un emulatore portatile basato su QEMU, documentazione, esempi e tutorials. L' Integrated Development Environment (IDE) ufficialmente supportato è Eclipse, che utilizza il plugin Android Development Tools (ADT plugin). Sono disponibili, inoltre, altri strumenti di sviluppo, come il Native Development Kit per applicazioni o estensioni in C o C++.

3.3.1 Perché Android

Analizziamo brevemente la distribuzione mondiale dei sistemi operativi mobile, in modo da giustificare la nostra scelta: sviluppare il progetto per Android.

Operating System	3Q12 Shipment Volumes	3Q12 Market Share	3Q11 Shipment Volumes	3Q11 Market Share	Year-Over-Year Change
Android	136.0	75.0%	71.0	57.5%	91.5%
iOS	26.9	14.9%	17.1	13.8%	57.3%
BlackBerry	7.7	4.3%	11.8	9.5%	-34.7%
Symbian	4.1	2.3%	18.1	14.6%	-77.3%
Windows Phone 7/ Windows Mobile	3.6	2.0%	1.5	1.2%	140.0%
Linux	2.8	1.5%	4.1	3.3%	-31.7%
Others	0.0	0.0%	0.1	0.1%	-100.0%
Totals	181.1	100.0%	123.7	100.0%	46.4%

Figura 9: distribuzione mobile OS, Q3 2013

La tabella sopra (Figura 9) mostra la distribuzione nel Q3 2013 (Luglio-Agosto-Settembre), tralasciando *Linux,* in quanto non ancora ufficialmente distribuito per smartphone e tablet e *Symbian,* non più supportato da nessun hardware

recente. Ci troviamo di fronte a quattro principali Sistemi Operativi da analizzare:

- Android, sviluppato da Google, al 75%
- iOS, sviluppato da Apple, al 14.9%
- BlackBerry, sviluppato da RIM, al 4.3%
- Windows Phone, sviluppato da Microsoft, al 2.0%

Decidiamo che, a causa della scarsa distribuzione, possiamo escludere Windows Phone e BlackBerry e concentrarci su un confronto tra Android e iOS.

Analizziamo ora un aspetto secondario della scelta di Google: il Java è un linguaggio ormai comunissimo che troviamo ovunque, da Android, alla centralina della macchina, piuttosto che al proiettore digitale del cinema o al televisore di ultima generazione; capiamo quindi come *non sia possibile avere una sola interfaccia che possa andar bene per tutti*, ed è proprio per questo motivo che Google ha deciso di rilasciare delle linee guida che tutti gli sviluppatori devono seguire quando creano le proprie applicazioni**.**

Java non è l'unico modo di sviluppare su Android. Anzi, negli ultimi tempi, ne sono nati parecchi, che in realtà altro non sono se non delle interfacce di programmazione che rendono più semplice lo sviluppo. Quella più famosa in assoluto è *Basic4Android*, che permette di scrivere applicativi utilizzando una sintassi basic-like, piuttosto che *PhoneGap,* che ci apre le porte dell'HTML5. Fortunatamente, questi framework sono ormai compatibili con tutte le piattaforme, e stanno prendendo sempre più piede proprio per la possibilità di non dover riscrivere il codice e poterlo invece solo ricompilare per la piattaforma desiderata.

Abbiamo analizzato quindi quali sono le differenze nello sviluppo dal punto di vista tecnico. Consideriamo ora, brevemente, l'aspetto economico. Se per poter mettere tutte proprie applicazioni sul Google Play Store, molte volte senza nemmeno bisogno dell'approvazione di Google, bastano 25$, ne servono, invece, circa 100$ per pubblicare le proprie applicazione sugli store di Apple o Microsoft.

3.3.2 Distribuzione Android

Grazie ai dati ufficiali distribuiti sulla dashboard da Google e pubblicati ad inizio ottobre sulla pagina ufficiale degli sviluppatori Android [5], abbiamo analizzato la situazione della distribuzione delle versioni di Android e le dimensioni degli schermi maggiormente utilizzati.

Questi dati mostrano solamente i device sui quali è installata l'ultima versione del Google Play Store che è compatibile solo con le versioni di Android superiori alla 2.2.

Version	Codename	API	Distribution
2.2	Froyo	8	1.7%
2.3.3 - 2.3.7	Gingerbread	10	26.3%
3.2	Honeycomb	13	0.1%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	19.8%
4.1.x	Jelly Bean	16	37.3%
4.2.x		17	12.5%
4.3		18	2.3%

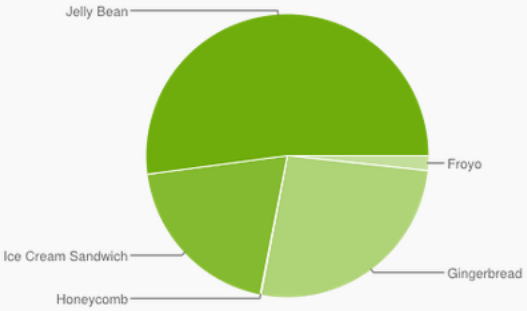


Figura 10: distribuzione Android, Novembre 2013

Possiamo vedere dalla *Figura 10* che le quattro versioni principali di Android (Honeycomb allo 0.1% è trascurabile) sono:

- Froyo 1.7% (nel mese precedente era al 2.2%).
- Gingerbread 26.3% (nel mese precedente era al 28.5%).
- Ice-Cream Sandwich 19.8% (nel mese precedente era al 20.6%).
- Jelly Bean sale dal precedente 48.6% all'attuale 52.1%.

Nel complesso, quindi, le versioni di Android superiori alla 4 sono solamente due (JB e ICS), ma detengono quasi il 70% della distribuzione totale oltre ad avere un livello di API più alto, che fornisce maggiori funzionalità per gli sviluppatori.

Per questo motivo abbiamo scelto di sviluppare il nostro progetto rendendolo compatibile solamente per dispositivi che sono equipaggiati con versioni di Android superiori alla 4.0.

3.3.3 SDK - Primi passi con Android

Come già accennato in precedenza l'IDE più utilizzato per sviluppare applicazioni Android è Eclipse. Nel Maggio 2013, Google ha presentato il suo IDE, chiamato *Android Studio*, atto a uniformare e semplificare lo sviluppo delle apps. Purtroppo, questo strumento è ancora in fase di beta test e quindi ancora non del tutto stabile, e, cosa più importante, non supporta tutte le funzionalità che il concorrente permette di utilizzare (nel nostro caso l'NDK è uno strumento fondamentale).

Per questo motivo, nello sviluppare il nostro progetto, abbiamo deciso di utilizzare Eclipse.

Tramite l'SDK (o meglio: tramite gli strumenti utilizzati mediante l'SDK), trasformiamo la nostra applicazione Android in un codice intermedio chiamato bytecode; questo è esattamente quello che accade abitualmente in Java. Questo bytecode viene eseguito dalla Dalvik Virtual Machine (DVM) descritta nel capitolo 1.1.

Ogni terminale Android ha la sua DVM, come definito nell'architettura del sistema; il suo compito è solo questo: eseguire il bytecode.

Per iniziare, è opportuno scaricare e decomprimere l'SDK e l'NDK di Android dal sito http://developer.android.com/sdk/index.html. In particolare questo pacchetto di software, chiamato ADT Bundle, contiene già tutto il necessario per poter iniziare a sviluppare senza dover installare software aggiuntivo. In particolare, l'SDK fornisce tutte le API che vanno installate tramite l'SDK Manager, uno strumento che consente allo sviluppatore di scegliere quali API di sistema installare (disponibile subito dopo aver decompresso il pacchetto scaricato), si fa riferimento all'applicazione:

```
android-sdk/SDK Manager.exe
```

D'ora in poi ci riferiremo ad android-sdk con $SDK.

Un volta ultimata questa procedura, si può passare all'installazione dell'Emulatore Android (vedi capitolo 1.2). Come prima cosa, è necessario creare un Android Virtual Device (ADV). Un ADV è una configurazione di un device per l'emulatore i Android che permette di modellare differenti device. Questa operazione può essere eseguita tramite l'interfaccia grafica attraverso il *Virtual Device Manager* nella barra degli strumenti di Eclipse, o attraverso linea di comando, tramite il seguente binario:

```
$SDK/tools/android avd -n new_machine_name -t api -sdcard file
```

Dove il parametro *api* specifica quale versione di Android emulare all'interno della macchina.

Ovviamente, è possibile eseguire l'installazione del software sviluppato anche su dispositivi reali, nel nostro caso un Samsung Galaxy Nexus e un Samsung Galaxy Tab 8.9 (vedi: paragrafo Galaxy Tab in Dispositivi adottati in fase di testing e sviluppo). L'interazione con i dispositivi Android è mediata dall'ADB (vedi il paragrafo ADB in Terminologia adottata all'interno della tesi). Questo programma, che abbiamo già visto occuparsi in precedenza della lista dei dispositivi attivi, può anche consentire il trasferimento dei files dal dispositivo dove è in esecuzione l'SDK all'emulatore o viceversa, dell'interazione con la shell del dispositivo (tramite il comando shell è possibile far, eventualmente, eseguire le istruzioni da effettuare) o la visualizzazione del file di logging (logcat).

Per prima cosa, è quindi necessario connettere il device Android, tramite cavo USB, alla macchina sulla quale è in esecuzione l'SDK. Nel nostro caso, in ambiente Windows, i nostri devices non erano correttamente riconosciuti e si è reso necessario installare driver aggiuntivi, OEM USB Drivers, forniti da Google sul sito http://developer.android.com/tools/extras/oem-usb.html. Terminata l'installazione, è necessario attivare le ***opzioni sviluppatore*** dal menù impostazioni del proprio device, e abilitare la funzionalità di ***USB debugging***, funzione già attiva di default sui Nexus device (vedi: Dispositivi adottati in fase di testing e sviluppo).

Al termine di queste operazioni, è possibile eseguire l'applicazione sul device o attraverso l'interfaccia grafica, cliccando su *Run* nella barra degli strumenti di Eclipse, o attraverso linea di comando, per mezzo dei seguenti binari:

```
$SDK/platform-tools/ adb install bin/myApp-debug.apk
```

A questo punto, sul device collegato verrà installata l'applicazione ed eseguita automaticamente.

3.3.4 NDK

Tratteremo in questa sezione di cross-compilazione, in particolare del NDK per Android, fornito dalla stessa Google: il vantaggio principale di questa collezione di tool è quello di fornire sia le librerie, sia i file di include necessari alla cross-compilazione, senza avere la necessità di cross-compilarle o di ottenerle da un altro dispositivo.

Questa tesi utilizza la versione di "r9" dell'NDK, che è comunque ottenibile al seguente indirizzo internet [6].

Lo NDK fornisce uno script, detto ndk-build, il quale può automatizzare le procedure di compilazione del codice nativo e del codice Java. Questo è presente all'interno del percorso $NDK, e deve essere eseguito all'interno della cartella dove sono presenti i sorgenti. Bisogna inoltre sottolineare come questo script venga in genere utilizzato per compilare applicazioni Native con JNI e librerie di sistema.

3.3.5 JNI - Java Native Language

La Java Native Interface o JNI è un framework del linguaggio Java che consente al codice Java di richiamare (o essere richiamato da) codice cosiddetto "nativo", ovvero specifico di un determinato sistema operativo o, più in generale, scritto in altri linguaggi di programmazione, in particolare C e C++.

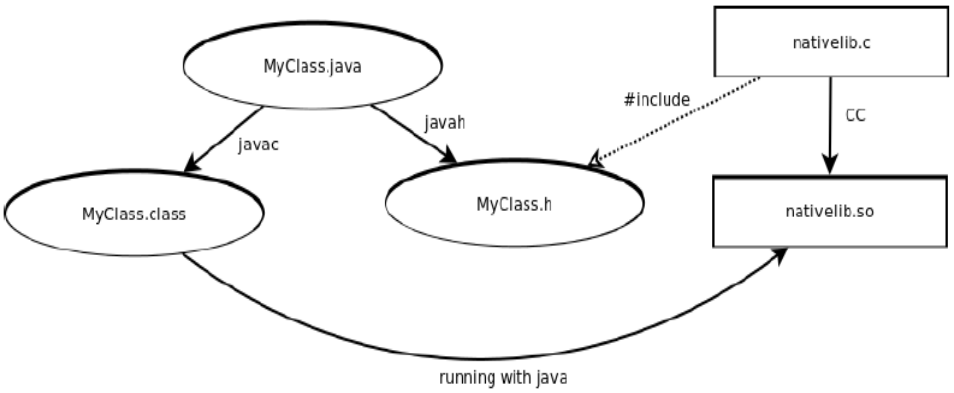


Figura 11: compilazione ed esecuzione codice Java attraverso JNI [1]

La principale applicazione della JNI è quella di richiamare all'interno di programmi Java porzioni di codice che svolgono funzionalità intrinsecamente non portabili (per esempio primitive di sistema operativo) e che, pertanto, non posso essere implementate in Java puro. L'interfacciamento è basato sulla definizione di un insieme di classi di raccordo fra i due contesti, che presentano una interfaccia Java, ma che delegano al codice nativo l'implementazione dei loro metodi.

In questa trattazione faremo riferimento alla strutturazione del codice per permettere una comunicazione tramite l'interfaccia JNI: tale procedura è comunque riassunta in Figura 11.

All'interno di una classe si può definire un numero arbitrario di metodi implementati in codice nativo. Per far questo, nel sorgente della classe il metodo deve avere la parola-chiave native e deve avere un punto e virgola al posto del corpo del metodo:

```
public class MyClass {
      public void metodo() { /* . . .  */ }
public native void metodoNativo();
}
```

Tale metodo nativo può essere definito all'interno del codice nativo, nel seguente modo:

```
#include <jni.h>

JNIEXPORT void JNICALL Java_ClassName_metodoNativo (JNIEnv
*env, jobject obj, jstring javaString){

Const char *nativeString = (*env)-> GetStringUTFChars
(env,javaString, 0);

(*env)-> ReleaseStringUTFChars (env, javaString,
nativeString);

}
```

In particolare, il primo parametro fa riferimento ad un puntatore ad un array ove sono contenuti tutti i metodi, che è possibile chiamare da codice nativo, mentre il secondo è un'istanza dell'oggetto che ha invocato tale metodo.

Il JNI viene principalmente utilizzato per consentire al programma l'accesso a funzioni definite nelle librerie del sistema operativo ospite, mediante primitive di sistema. In realtà, l'accesso avviene in modo indiretto, nel senso che le funzioni del sistema operativo vengono invocate dalle funzioni che implementano i metodi nativi della classe che li definisce e che vengono a loro volta utilizzati dal codice Java.

L'uso del JNI si rende inoltre necessario quando l'implementazione di una certa funzionalità nel programma dipende dal sistema operativo in uso a run-time e non è presente nelle librerie standard di Java. Il programma risultante non può essere definito "100%-Java", in quanto esso fa direttamente uso di codice nativo.

3.4 GSTREAMER

GStreamer è un framework multimediale basato su pipeline scritto in il linguaggio C, concesso sotto la GNU Lesser General Public License. Permette di creare una grande varietà di componenti per la gestione dei media, come ad esempio strumenti per la riproduzione audio, riproduzione audio-video, registrazione, streaming e editing. La progettazione della pipeline serve come base per creare molti tipi di applicazioni.

GStreamer è progettato per essere cross-platform, infatti è compatibile con Linux (x86, PowerPC and ARM), Solaris (Intel and SPARC) e OpenSolaris, FreeBSD, OpenBSD, NetBSD, Mac OS X, Microsoft Windows, Android, iOs e Windows Phone.

È costruito inoltre per essere compatibile, oltre al C, con i linguaggi di programmazione più usati come Python, Vala, C++, Perl, GNU Guile e Ruby.

3.4.1 La pipeline

GStreamer processa i media attraverso la connessione di un numero di *elementi* di elaborazione in una *pipeline*. Ciascun elemento è fornito da un plug-in. Gli elementi possono essere raggruppati in *bins*, che possono essere ulteriormente aggregati, formando così un grafo gerarchico (Figura 12).

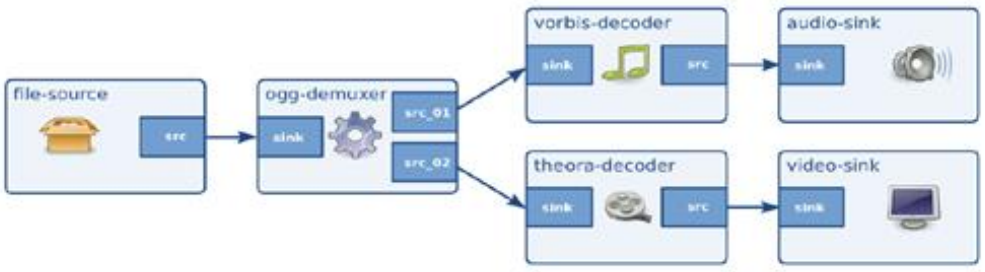


Figura 12: pipeline GStreamer di esempio

Il vero scopo di GStreamer è quello di rendere possibile la realizzazione di applicazioni multimediali, utilizzando elementi già esistenti, plugins [7], in

grado di interagire: in sostanza, definisce un protocollo/modello di comunicazione e di scambio di dati ed eventi tra elementi.

3.4.2 Gli elementi nel dettaglio

Per formare una pipeline, gli elementi vanno collegati gli uni agli altri utilizzando quello che è l'equivalente in GStreamer dei comuni connettori: i pad. Per immaginare un pad, si pensi a delle apparecchiature elettroniche che si collegano tra di loro, per esempio televisione e videoregistratore: la connessione avviene tramite i connettori SCART. In GStreamer, i pad sono caratterizzati da un particolare tipo di dati e dal tipo di connessione permessa: per continuare con l'esempio del televisore e del videoregistratore, si può dire che non è possibile collegare un connettore SCART ad un connettore per le cuffie.

Gli elementi, classificati in base al tipo e al numero di pad, possono essere di tre tipi:

- ***Source***: prelevano dati da fonti esterne al programma come file, altri programmi, connessioni di rete, etc. Hanno una uscita source pad.
- ***Sink***: mandano dati provenienti dal programma a componenti esterne come file, altri programmi, connessioni di rete, etc. Hanno un ingresso sink pad.
- ***Filter***: possono essere collegati solo ad altri elementi, quindi non sono usati per comunicare con l'esterno. Il loro scopo è quello di modificare, convertire o elaborare i dati. Hanno sempre almeno un source pad e un sink pad.

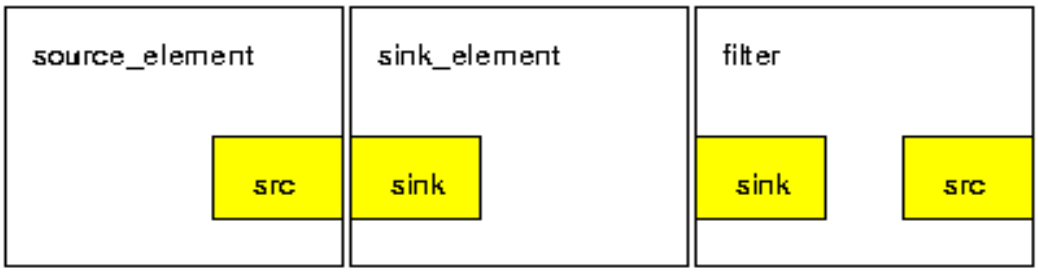


Figura 13: gli elementi della pipeline

3.4.3 Plug-in e media supportati

GStreamer utilizza una architettura plug-in [7] che rende la maggior parte delle funzionalità di GStreamer implementate come librerie condivise. Le funzionalità di base di GStreamer contengono le funzioni per la registrazione e il caricamento dei plug-in e permettono di fornire i fondamenti di tutte le classi in forma di classi di base. Queste librerie vengono caricate dinamicamente in modo da supportare una vasta gamma di codec, formati contenitore, i driver di input/output ed effetti. I plug-in possono essere installati in maniera semi-automatica, quando sono necessari.

Dalla versione 0.10, i plug-in vengono raggruppati in tre set (dal nome del film Il buono, il brutto, il cattivo), oltre a questi esiste un altro plug-in di GStreamer, chiamato *FFmpeg,* che supporta vari formati multimediali addizionali.

Grazie a questi quattro plug-in, GStreamer supporta una gran varietà di formati, protocolli e codec multimediali (tra i quali MPEG-1, MPEG-2, MPEG-4, H.261, H.263, H.264, RealVideo, MP3, WMV, FLV).

4 SCENARIO SIP

Nel primo scenario ci troviamo ad analizzare un sistema basato principalmente su due canali di comunicazione: il bus SCS utilizzato da Bticino, e il protocollo SIP.

In questo sistema vi è un classico video-citofono esterno (che da ora in poi chiameremo posto esterno) che comunica con un video-terminale (che da ora in poi chiameremo posto interno) posto all'interno dell'abitazione. In situazioni reali, di solito sul posto esterno vi è la possibilità di effettuare la chiamata a tutti i posti interni presenti, ma rimane sempre un unico, seppure a volte molto complesso, dispositivo. Il cosiddetto posto interno invece è un video-terminale, il quale può essere a seconda delle versioni, dei modelli, più o meno sofisticato; di solito consente una ripresa audio-video, e nelle versioni più avanzate anche l'utilizzo di comandi, o la gestione tramite touch di altri dispositivi della casa, tutto tramite l'unico access point device.

Il posto interno viene identificato da un indirizzo, il quale identifica piano e appartamento (nel caso ci fossero più appartamenti e più piani). Il posto esterno infatti, può gestire chiamate a centinaia di posti interni (ovviamente uno alla volta in quanto la telecamera-microfono è singolo), tutti identificati tramite il relativo indirizzo. In strutture complesse, ci si può avvalere anche di un dispositivo terzo, che sta nel mezzo: il centralino, e serve principalmente a fare da tramite tra posto interno e posto esterno.

Sia che la chiamata da posto esterno avvenga direttamente al posto interno, o tramite centralino, in entrambi i casi questa si determina tramite SIP. Dopodiché una volta instaurata la sessione, i dati viaggiano attraverso la rete con il protocollo IP.

Per concentrarsi sulla progettazione di un nuovo sistema di comunicazione, o più correttamente sullo studio di una nuova funzionalità da aggiungere a quelle già esistenti al sistema, è stato proposto uno scenario leggermente semplificato, in modo da isolare il tutto in un sottoinsieme più piccolo. Viene tralasciata la gestione dell'indirizzamento (e quindi raggiungimento) delle specifiche unità abitative (che possono essere una palazzina, un appartamento

o altro) in quanto non vi è alcuna differenza tra una versione desktop e una versione mobile (gli indirizzi vengono, infatti, gestiti allo stesso modo). La comunicazione quindi viene ridotta ad una semplice, si fa per dire, connessione punto-punto tra il posto esterno (videocitofono) e il posto interno, che in questo nuovo scenario è un dispositivo mobile e non più un semplice videocitofono interno. I due dispositivi condividono la medesima rete LAN, dove il posto interno è collegato via cavo, mentre il dispositivo mobile è di solito, ma non obbligatorio, collegato via wireless (attraverso il Wi-Fi).

4.1 COMPONENTI E SPECIFICHE TECNICHE

4.1.1 Bticino SwitchBoard (centralino)

Il centralino ricopre il ruolo solito che gli compete, e cioè riceve la chiamata dall'esterno, controlla l'interlocutore e ridirige la chiamata all'appartamento corrispondente tramite un software Bticino (switchboard). Tramite questo software si possono gestire tutte le chiamate in entrata e tutte le chiamate in uscita. Il centralino si occupa appunto di instaurare la connessione tra i due dispositivi, raccogliendo le chiamate e reindirizzandole; una volta che il canale di comunicazione è stato instaurato, la trasmissione real-time dei pacchetti media avviene direttamente tra i due partecipanti ossia posto interno e posto esterno. Come si può notare nella schermata in Figura 14, l'interfaccia è molto semplice e intuitiva.

Figura 14: software centralino di Bticino

Nella parte sinistra si trovano tutte le chiamate in entrata, disposte in un elenco, e posizionata in alto uno schermo per visualizzarne il video in ingresso dall'esterno. Sulla destra invece, si trova analogamente alle chiamate in entrata, quelle in uscita, e cioè i terminali posti nei vari appartamenti; anche qui c'è una zona dove poter visionare il video in ingresso dal posto interno. Al centro vi è il posto per visionare le telecamere di sorveglianza. Oltre ai classici pulsanti di apertura e chiusura della chiamata (cornette rosse e verdi), vi sono i comandi per collegare un posto esterno ad uno interno (e farli così comunicare direttamente tra di loro), la rubrica di cui abbiamo accennato prima e i comandi per aprire i vari cancelli.

CONFIGURAZIONE

Per poter usufruire del software centralino, occorre prima configurarlo ed avviarne i servizi. Per fare queste operazioni ci si serve del cosiddetto "switchBoard Configurator", con il quale innanzitutto si seleziona il mezzo di comunicazione e trasmissione, cioè l'interfaccia di rete che verrà usata dal centralino (è possibile scegliere tra le varie opzioni che il sistema in uso consente, e quindi LAN, wireless, etc). Dopodiché, una volta assegnato un codice univoco generale, si può passare alla vera e propria configurazione dello switchBoard (il centralino). Oltre al nome e all'indirizzo univoco (che serve per essere raggiunto dai vari posti interni/esterni e/o da altri centralini), si deve scegliere tutta quella serie di impostazioni hardware quali microfono, webcam e altoparlanti, che verranno usati durante la comunicazione. Il software propone già i dispositivi riconosciuti dal sistema. Infine vanno configurate le opzioni per il SIP, e cioè la versione e il codec utilizzato per lo stream video (disponibili per la scelta allo stato attuale MJPEG e H264).

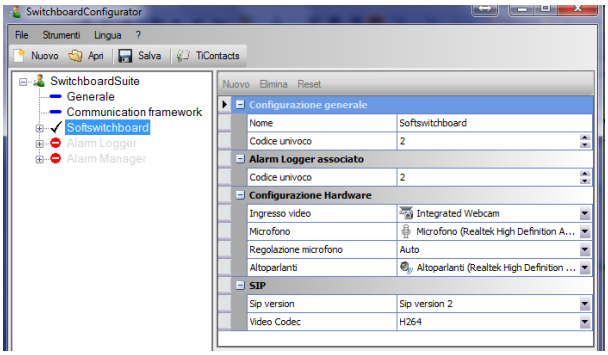


Figura 15: software di configurazione

Terminata la configurazione, si può passare all'attivazione del centralino, e all'esecuzione di quest'ultimo. Per raggiungere un altro dispositivo (che sia uno smartphone, un posto interno/esterno o un altro centralino), è necessario aggiungere i relativi indirizzi alla rubrica, memorizzata nel database del centralino; in questo modo durante l'esecuzione del software, è possibile chiamare i vari indirizzi richiamando la funzione rubrica del centralino, selezionando i destinatari senza impostare il loro indirizzo IP, ma visionandoli "nominalmente" tramite un numero identificativo.

4.1.2 VideoCitofono Esterno

Il posto esterno è composto dal medesimo videocitofono utilizzato nello scenario OpenWebNet: è composto da una telecamera a colori, dei led luminosi (che indicano l'accensione), due pulsanti per inoltrare la chiamata, un microfono e degli speaker audio utilizzati come segnali acustici e per l'audio in uscita.

Per il sistema videocitofonico di questo scenario collegato via IP e gestito nella sua connessione con il protocollo SIP, vi è un segnale audio e video in uscita dal posto esterno e in entrata allo smartphone, mentre solo l'audio viene trasmesso sul canale opposto, in quanto sul videocitofono esterno non vi è una videocamera per visualizzare l'eventuale video registratore dal mobile device. I codec video supportato è un H264, formato 800x480 a 24 frame per secondo.

4.2 ANALISI

La prima fase dell'analisi sarà osservare e descrivere il processo di negoziazione e instaurazione della comunicazione tramite protocollo SIP, in modo da integrare successivamente in un'applicazione Android, un sistema che vada ad interfacciarsi a tale protocollo e riesca a comunicare in maniera adeguata con il posto esterno, il quale utilizza una serie di codici e sequenze particolari. Come già anticipato, lo studio avverrà su uno "scenario" semplificato, punto-punto, e quindi tralasciando quelle problematiche che possono essere un'autenticazione

ad un proxy server SIP, oppure la deviazione di chiamata (che nella realtà può essere effettuato dal software centralino fornito direttamente da Bticino.

4.2.1 SIP

Qui di seguito vi è lo schema base di una semplica chiamata SIP e di come avviene lo scambio di messaggi fra il posto interno (IU - Internal Unit) e posto esterno (External Unit).

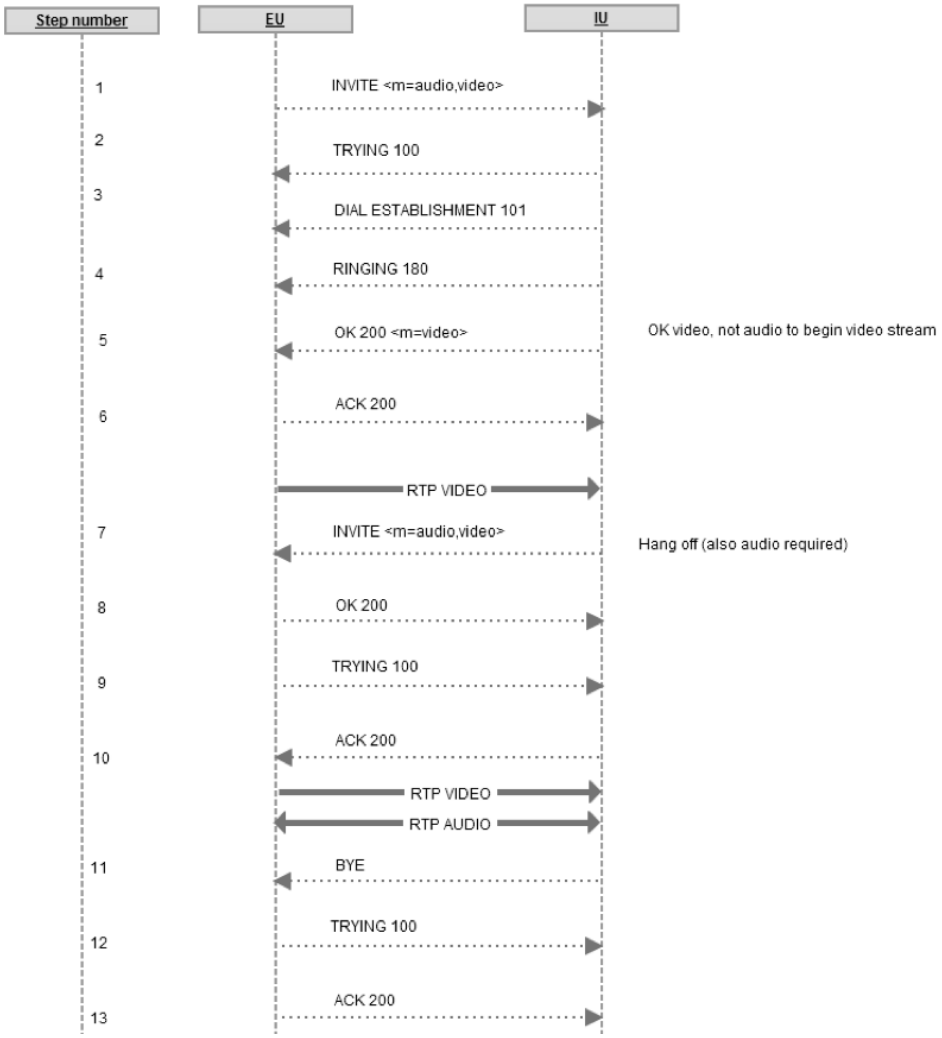


Figura 16: Dialogo per una comunicazione SIP Bticino

Come si può vedere è una classica comunicazione SIP che inizia con una INVITE dal posto esterno (la chiamata dal video-citofono) la quale incorpora le variabili

per avvertire che ci sarà una comunicazione media audio/video. Lo user agent interno risponde con una sequenza composta da TRYING 100, DIAL ESTABLISHMENT 101, RINGING 180 e infine un OK 200 per dare l'ok alla comunicazione, richiedendo tramite parametro SDP il video al posto esterno, il quale una volta ricevuto il messaggio OK invia un messaggio di ack (ACK 200) per dare conferma dell'avvenuta ricezione. A questo punto il dialogo è iniziato, e si inizia tramite protocollo RTP (nel nostro caso gestito dalla libreria GStreamer) la sessione di media stream video perché siamo nel momento in cui nel posto interno viene visualizzato l'interlocutore esterno e si deve decidere se alzare la cornetta e iniziare la comunicazione. In caso positivo viene inviato un secondo INVITE questa volta da parte del posto interno per stabilire una comunicazione audio/video, la quale viene instaurata dopo che il posto esterno invia a sua volta la seria di messaggi OK 200 E TRYING 100 e con la conferma del posto interno tramite il solito ACK 200. Come avvenuto per il primo stream solo video, unidirezionale da esterno a interno, adesso lo stream è audio e video, bidirezionale.
In questo caso specifico l'utente interno ad un certo punto decide di interrompere la chiamata (ma avendo, come già spiegato nel capitolo SIP, entrambi gli user agent la possibilità di essere sia client che server, l'azione di chiusura poteva essere iniziata dall'altro, anche se di solito è nel posto interno che vi è una vera e propria cornetta per la terminazione della chiamata) inviando il messaggio di BYE che inizia il procedimento di chiusura della comunicazione. Il posto esterno invia così il messaggio di TRYING 100 e infine un ACK 200 per confermare la conclusione del dialogo.
Come si può notare dallo schema delle transazioni, è necessario inviare un messaggio TRYING 100 ad ogni messaggio (BYE o INVITE); questo non è una specifica SIP, ma una decisione personale di BTicino

Tutte le sessioni SIP sono point-to-point (anche in una situazione realistica di un impianto BTicino, lì l'unica differenza potrebbe essere la negoziazione della comunicazione), e tutto è sviluppato per oscurare l'indirizzo IP e tenere compatibili gli indirizzamenti sia su bus che via ethernet; nel nostro caso invece, non avendo un sistema di DataBase da cui ottenere eventualmente il matching tra indirizzi IP e posizione nella rete, si andranno ad utilizzare questi ultimi direttamente.

A livello di messaggio, il contenuto definito da SDP può essere vuoto o contenere diversi parametri. Principalmente vengono usati i parametri "*v*", "*o*", "*s*, "*c*, "*t*", "*a*", "*m*". Il parametro "a" viene usato per definire un attributo specifico per BTicino, il BTkind. Questo attributo va a definire il tipo di chiamata che verrà eseguita. Per esempio può descrivere una video chiamata, oppure una telefonata, un call switch, una chiamata interna o di piano (a seconda della morfologia della rete) e, tra le varie tipologie, differenziare il tipo di suoneria.

Il codice in Figura 17 mostra un intero messaggio SIP utilizzato per la prima INVITE della sequenza di negoziazione della comunicazione.

```
INVITE sip:389@192.168.50.127:5060 SIP/2.0
Via: SIP/2.0/UDP
192.168.50.221;rport;branch=z9hG4bK0Ha1SZUX6Xr1m
Max-Forwards: 70
From: <sip:24@192.168.50.221:5060>;tag=0eBpcDecga1tj
To: <sip:389@192.168.50.127:5060>
Call-ID: RhcJdC7UxYbSvepenAbi389
CSeq: 41686205 INVITE
Contact: <sip:24@192.168.50.221:5060>
User-Agent: sofia-sip/1.12.11
Allow: INVITE, CANCEL, BYE, ACK, OPTIONS, MESSAGE, REFER,
NOTIFY
Supported: timer, 100rel
Content-Type: application/sdp
Content-Disposition: session
Content-Length: 288

v=0
o=24 7597650163974703198 4425513679697971981 IN IP4
192.168.50.221
s=A conversation BT kind 1
c=IN IP4 192.168.50.221
t=0 0
a=BT kind:1
a=PAN TILT:0
a=UNIT TO CALL:12
m=audio 7078 RTP/AVP 0
a=rtpmap:0 PCMU/8000/1
m=video 5050 RTP/AVP 26
a=rtpmap:26 JPEG/90000
a=sendonly
```

Figura 17: messaggio SIP, INVITE

La prima parte rappresenta l'header, e cioè tutte quelle informazioni volte ad identificare gli attori della transazione in atto (mittente, ricevente, id sequenza, tipo di user agent, messaggi permessi). L'indirizzo è specificato in maniera particolare, viene aggiunta infatti anche il numero della porta come suffisso all'indirizzo di base.
La seconda parte invece è il cuore del messaggio e rappresenta la descrizione della sessione (tramite SDP) che appunto tramite gli opportuni parametri va a definire quello che sarà il contesto del dialogo. In questo caso si può notare il parametro (obbligatorio) BT kind, il destinatario identificato da UNIT TO CALL e i parametri di media (descritti da m) audio e video.

4.2.2 Android

Nei sistemi finora realizzati da Bticino che si servono di una comunicazione su canale IP (instaurata tramite SIP), il device interno è, come già detto più volte, un apparecchio che deve essere in grado di visualizzare lo stream audio/video proveniente dall'esterno (oltre alle altre funzioni che può compiere) ed essere raggiungibile dal canale di trasmissione. Uno smartphone device può tranquillamente soddisfare tali requisiti. Ha infatti un'interfaccia wireless per collegarsi alla rete (LAN), uno schermo + altoparlanti per gestire il flusso audio e/o video in entrata, e un microfono + videocamera per registrare un flusso audio e/o video in uscita.

La parte di connessione alla rete è demandata al sistema operativo, l'utente non deve fare altro che connettersi come avviene normalmente, sarà poi l'applicazione a riconoscere quale IP vi è stato assegnato (in un contesto abitativo, come avviene con i device videocitofonici fissi, potrà essergli dedicato un indirizzo IP fisso invece che assegnato tramite DHCP, oppure si potrà definire un apparecchio di smistamento della chiamata in arrivo come nodo zero dell'unità abitativa, il quale conoscerà i vari indirizzi all'interno, e il posto esterno raggiungerà (e invierà i messaggi) a quello specifico indirizzo.

La scelta su quale libreria utilizzare per l'implementazione è caduta su PJSIP, in quanto è una libreria usata per creare un'applicazione piuttosto complessa di comunicazione VoIP: CSipSimple, e sembra abbastanza ben supportata, è una libreria scritta in linguaggio C, e quindi compatibile e usabile su altre

piattaforme, su Android è possibile utilizzarla per mezzo di NDK e JNI. La libreria MJsip sebbene possa essere interessante, è carente nella documentazione e ferma da un paio d'anni nello sviluppo. Jain-SIP potrebbe essere una valida alternativa a PJSIP, anche se si basa sulle librerie javax.*, le stesse dello stack SIP di Android, il quale però oltre ad essere ad uno stato iniziale di sviluppo, è troppo ad alto livello e non permette certe personalizzazioni, a noi necessarie, nel flusso dei messaggi SIP; è infatti pensato per creare semplici client VoIP o chiamate vocali gestibili senza tante righe di codice.

4.2.3 PJSIP

Come Accennato nella presentazione della libreria, PJSip è uno stack SIP scritto in C. Anche se in realtà utilizzare la terminologia "Libreria PJSIP" non è del tutto corretto, in quanto PJSIP rappresenta solo una delle librerie presenti (vedi figura). In particolare l'insieme è composto da:

- ***PJSIP***: rappresenta uno stack SIP che supporta un insieme di features ed estensioni del protocollo SIP.
- ***PJLIB***: rappresentante la libreria a cui le restanti si appoggiano. Si occupa di garantire funzionalità di base (es. L'astrazione rispetto al sistema operativo sottostante). Può essere considerata una replica della libreria libc con l'aggiunta di alcune features come la gestione dei socket, funzionalità di logging, gestione thread, mutua esclusione, semafori, critical section, funzioni di timing, gestione eccezioni ecc.
- ***PJLIB-UTIL***: anch'essa come PJLIB è una libreria di appoggio, ma a differenza della prima implementa funzioni più complesse utili per la crittografia come gli algoritmi: SHA1, MD5, HMAC, CRC32.
- ***PJMEDIA***: libreria il cui scopo è la gestione ed il trasferimento dei dati multimediali
- ***PJSUA***: rappresenta il livello più alto fungendo da wrapper verso le altre librerie, fornendo un'interfaccia ad alto livello (fruibile anche tramite linea di comando).

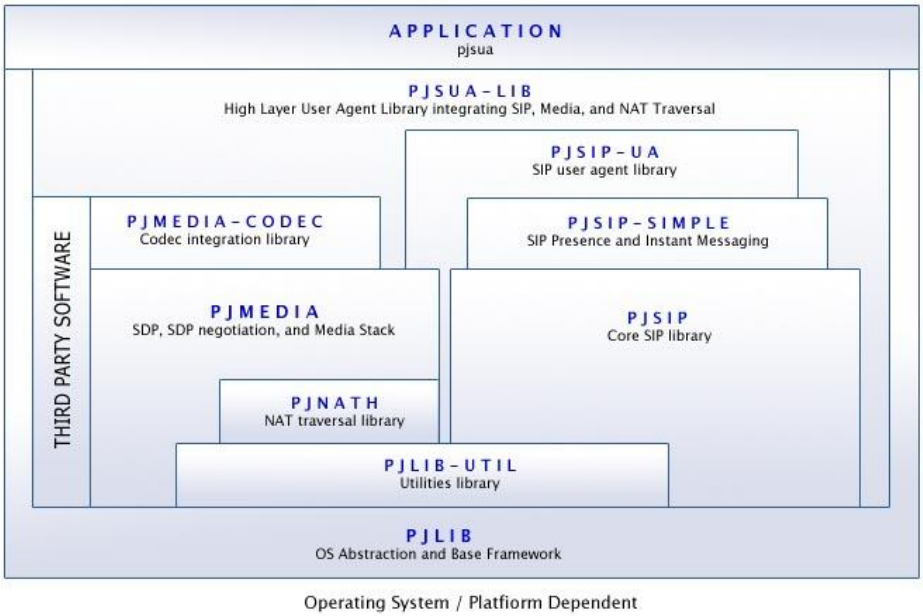


Figura 18: dipendenze stack PJSIP e librerie

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 centro cardine di questa architettura è rappresentato dal **Sip Endpoint**, che si occupa dei seguenti compiti:

- 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;

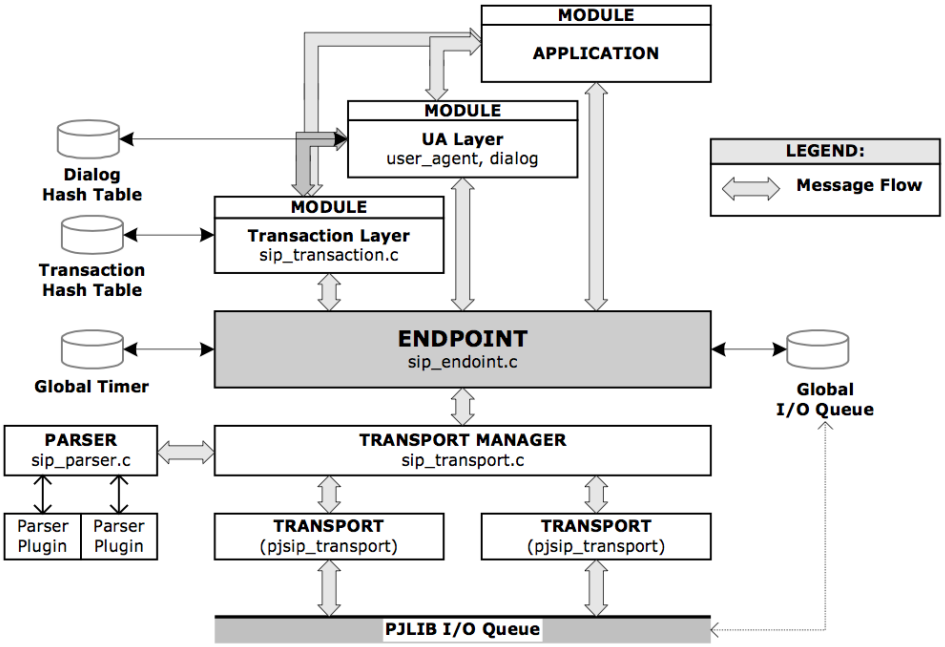


Figura 19: Diagramma di collaborazione moduli

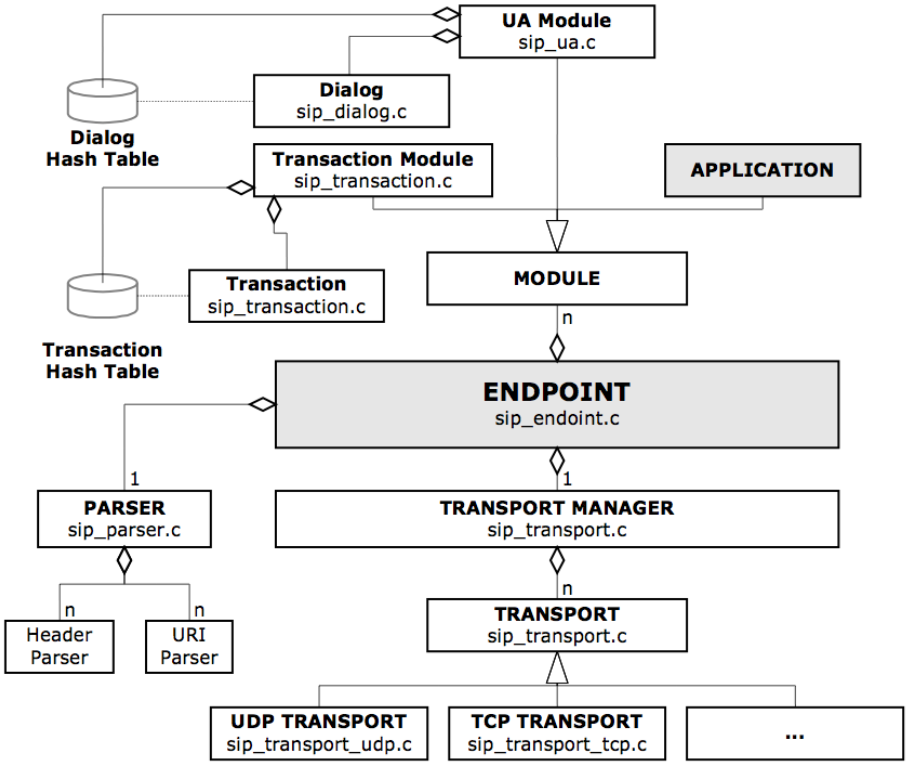


Figura 20: Class Diagram PJSIP

Il **Transport** ha come compito principale l'invio e la ricezione dei messaggi provenienti dalla rete. In generale è possibile dire che il livello di transport ha come fulcro il Transport Manager, Il cui scopo è gestire la creazione di tutti i transport necessari, oltre a offrire servizi possibili come per esempio instradare i pacchetti provenienti dai vari transport per poi passarli all'endpoint. Si occupa di trovare il transport adeguato per l'invio dei messaggi in base al tipo di messaggio da spedire e all'indirizzo remoto, gestisce le factories dei trasporti e gestisce direttamente la vita dei transport basandosi su un sistema di conteggio delle reference ed un timer di inattività. La libreria pjsip alloca un solo tran sport manager per endpoint. Il transport manager normalmente non risulta visibile alle applicazioni che devono utilizzare le funzioni esposte dall'endpoint.

Il compito del modulo **Transaction** è quello di attuare le giuste fasi di transazione in base ai messaggi SIP in entrata e/o uscita: possibile ritrasmissione messaggi di INVITE o REGISTER in mancanza di una risposta dei primi, oltre ad avere delle API necessarie a comunicare all'endpoint i messaggi in uscita e delle funzioni di callback necessarie al monitoraggio delle trasmissioni.

Il concetto dato dallo **User Agent** è quello della creazione, distruzione ed identificazione delle sessioni (INVITE, REGISTER, SUBSCRIBE/NOTIFY) necessarie ad implementare correttamente le fasi di una comunicazione SIP.

DataBuffer: ogni messaggio ricevuto viene passato attraverso i vari componenti software incapsulato in una struttura, anziché in come messaggio semplice, questa struttura contiene informazioni aggiuntive che riguardano il messaggio come ad esempio l'istante temporale di ricezione, o l'indirizzo IP del mittente del messaggio stesso. La dichiarazione dei buffer è presente nel file *pjsip/sip_transport.h*, in tale file viene descritto anche il transmit data buffer che è il buffer che viene usato per l'invio dei messaggi.

PJSIP prevede un meccanismo di **callback** che hanno origine nel tran sport manager, che effettua il parsing del messaggio, transitano dell'endpoint e poi vengono propagate da esso verso le altri parti del sistema. Tale librerie si appoggia sull'utilizzo di un modello framework dei modui ed una gestione degli header.

Il framework dei moduli è il principale mezzo per distribuire i messaggi SIP tra le componenti software in una applicazione PJSIP, e nelle librerie stesse. Esso è basato su un semplice meccanismo: per i messaggi in ingresso, l'endpoint distribuisce il messaggio a tutti i moduli, a partire dal modulo con la priorità più alta, finchè ogni modulo non finisce di processare il messaggio. Per i messaggi in uscita, l'endpoint distribuisce i messaggi ai moduli, sempre seguendo l'ordine per priorità, prima che questi vengano trasmessi sull'interfaccia di rete. In questo modo i moduli possono ognuno aggiungere delle modifiche al messaggio.

È possibile inoltre 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. Di seguito viene proposta la struttura base di un nuovo 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 21: Dichiarazione di un modulo

PJSIP si basa su di un semplice concetto astratto di gerarchia, in cui ad ogni modulo è associato un valore che indica la sua priorità. Questo valore specifica l'ordine con la quale i moduli vengono chiamati: più sarà basso il valore, maggiore priorità avrà il modulo. Di seguito vengono riportate le dichiarazioni standard delle priorità dei vari 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 22: Priorità dei moduli

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.

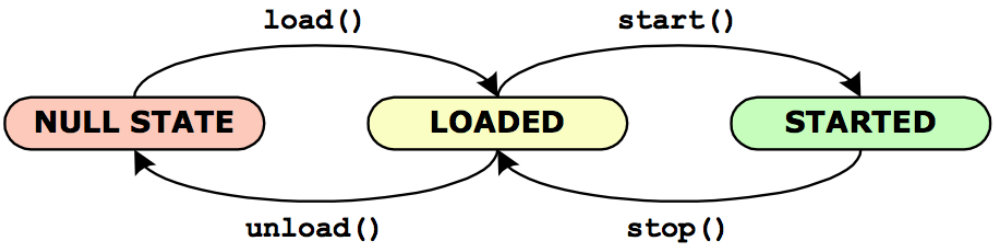


Figura 23: Diagramma degli stati dei moduli

Quando un nuovo messaggio arriva, esso è rappresentato e ricevuto dal buffer dei messaggi (attraverso *pjsip_rx_data*), il Transport manager fa il parsing e mette la struttura dati del messaggio nel receive buffer e passa il messaggio all'endpoint. Quest'ultimo distribuisce il messaggio ricevuto nel buffer ad ogni modulo chiamando la funzione callback on_rx_request() o on_rx_response() partendo dal modulo con priorità maggiore fino all'ultimo. A quel punto l'endpoint finisce di ridistribuire il messaggio. Il seguente diagramma mostra

una possibile sequenza di chiamate a cascata dei vari moduli, i quali si passano i dati del messaggio.

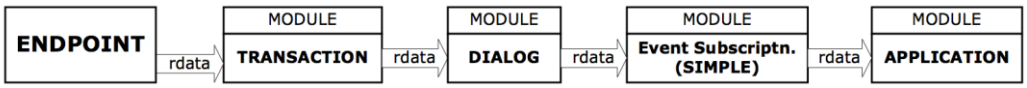


Figura 24: Callback a cascata dei moduli

Avendo fatto una rapida panoramica della gestione generale dei moduli all'interno della libreria, passiamo ora ad analizzare le varie componenti utilizzabili da un'applicazione (nel nostro caso Android) volta ad integrare lo stack SIP.

URI STRUCTURE

Il primo passo per una comunicazione SIP è senza dubbio l'indirizzamento, e per averlo occorre creare innanzitutto un URI, ci viene in aiuto la struttura pjsip sip_uri, che ci permette di avere una serie di parametri obbligatori e facoltativi per la creazione di un indirizzo. Nel nostro caso, occorre anche impostare il numero di porta, che normalmente è facoltativo.

```
struct pjsip_sip_uri
{
pjsip_uri_vptr *vptr; // Pointer to virtual function table.
pj_str_t user; // Optional user part.
pj_str_t passwd; // Optional password part.
pj_str_t host; // Host part, always exists.
int port; // Optional port number, or zero.
pj_str_t user_param; // Optional user parameter
pj_str_t method_param; // Optional method parameter.
pj_str_t transport_param; // Optional transport parameter.
int ttl_param; // Optional TTL param, or -1.
int lr_param; // Optional loose routing param, or 0
pj_str_t maddr_param; // Optional maddr param
pjsip_param other_param; // Other parameters as list.
pjsip_param header_param; // Optional header parameters as
list.
};
```

Figura 25: Dichiarazione di una struttura SIP URI

HEADER MESSAGE

In PJSIP tutti campi dell'header condividono delle proprietà come un tipo, un nome, un nome abbreviato e un tabella delle funzioni virtuali; così facendo possono essere trattati uniformemente dallo stack. PJSIP definisce la structure ***pjsip_hdr*** che contiene appunto le proprietà comuni condivise da tutti i campi header. I campi standard di PJSIP sono dichiarati dalla classe <pjsip/sip_msg.h>, e ogni campo specifica di solito una API per poterlo manipolare; per esempio l'API per creare un campo è convenzionalmente chiamato come il nome del capo stesso e il suffisso _create(), per esempio **pjsip_via_hdr_create()** viene utilizzato per l'istanza psip_via_hdr.

Alcuni SIP headers (come per esempio Require, Contact, etc) possono essere raggruppati in un singolo campo header e separati dalla virgola:

```
Contact: <sip:alice@sip.example.com>;q=1.0,
<tel:+442081234567>;q=0.5
Via: SIP/2.0/UDP proxy1.example.com;branch=z9hG4bK87asdks7,
SIP/2.0/UDP
proxy2.example.com;branch=z9hG4bK77asjd
```

Figura 26: SIP Header

E quando il parser incontrerà la stringa (Pjsip non supporta gli array per header complessi) splitterà gli header mantenendo l'ordine di apparizione in un array dedicato.

MESSAGE

Per i messaggi vi è la struttura pjsip_msg che rappresenta entrambi i tipi: request e response. La funzione **pjsip_msg* pjsip_msg_create(pj_pool_t *pool, pjsip_msg_type_e type);** serve a creare una request o una response in base al tipo (type).

Di seguito riportiamo la dichiarazione tipo di un messaggio.

```
enum pjsip_msg_type_e
{
PJSIP_REQUEST_MSG, // Indicates request message.
PJSIP_RESPONSE_MSG, // Indicates response message.
};
struct pjsip_request_line
{
pjsip_method method; // Method for this request line.
pjsip_uri *uri; // URI for this request line.
};
struct pjsip_status_line
{
int code; // Status code.
pj_str_t reason; // Reason string.
};
struct pjsip_msg
{
/** Message type (ie request or response). */
pjsip_msg_type_e type;
/** The first line of the message can be either request line
for request
* messages, or status line for response messages. It is
represented here
* as a union.
*/
union
{
/** Request Line. */
struct pjsip_request_line req;
/** Status Line. */
struct pjsip_status_line status;
} line;
/** List of message headers. */
pjsip_hdr hdr;
/** Pointer to message body, or NULL if no message body is
attached to
* this mesage.
*/
pjsip_msg_body *body;
```

Figura 27: dichiarazione di un messaggio SIP

Per ricercare o aggiungere un header all'interno del messaggio, si utilizzano le funzioni **pjsip_hdr* pjsip_msg_find_hdr_by_name(pjsip_msg *msg, const pj_str_t *name, pjsip_hdr *start);** e **pjsip_msg_add_hdr(pjsip_msg *msg, pjsip_hdr *hdr);**

Per gestire al meglio i vari tipi di messaggi, Pjsip mette a disposizione una serie di status code relativi ai codici SIP (come 100 per il trying, 200 per OK, 404 per not found ecc).
Inoltre, si possono creare dei parametri (per esempio nell'header) non standard tramite la struttura:

```
struct pjsip_param
{
PJ_DECL_LIST_MEMBER(struct pjsip_param); // Generic list
member.
pj_str_t name; // Param/header name.
pj_str_t value; // Param/header value.
};
```

Figura 28: dichiarazione dei parametri non-standard

Per quanto riguarda il body message, è rappresentato dalla structure pjsip_msg_body, la quale è dichiarata nel file <pjsip/sip_msg.h> (il medesimo dell'header), e vi sono varie API per manipolarli; come per esempio **pjsip_iscomposing_create_body,** oppure **pjsip_iscomposing_parse.**
Per esempio la funzione **pjsip_print_text_body** può servire per gestire quei messaggi con il corpo solamente testuale.

Per gestire messaggi più complessi e non testuali (per quanto riguarda il body), occorre servirsi di una sottolibreria di Pjmedia che si occupa unicamente di SDP. PJsip mette a disposizione una serie di API (un vero e proprio frame work basato sulle specifiche RFC 3264 sul modello offerta/risposta di SDP) per gestire la negoziazione dei parametri media e in particolare di tutta le session description. Il framework si basa su una generica negoziazione SDP (pjmedia_sdp_neg), e il dialog invite fornisce l'integrazione tra l'offerta e risposta di SDP e il protocollo SIP, interpretando il corpo dei messaggi come per esempio INVITE, ACK, ecc e traducendoli in negoziazioni SDP.

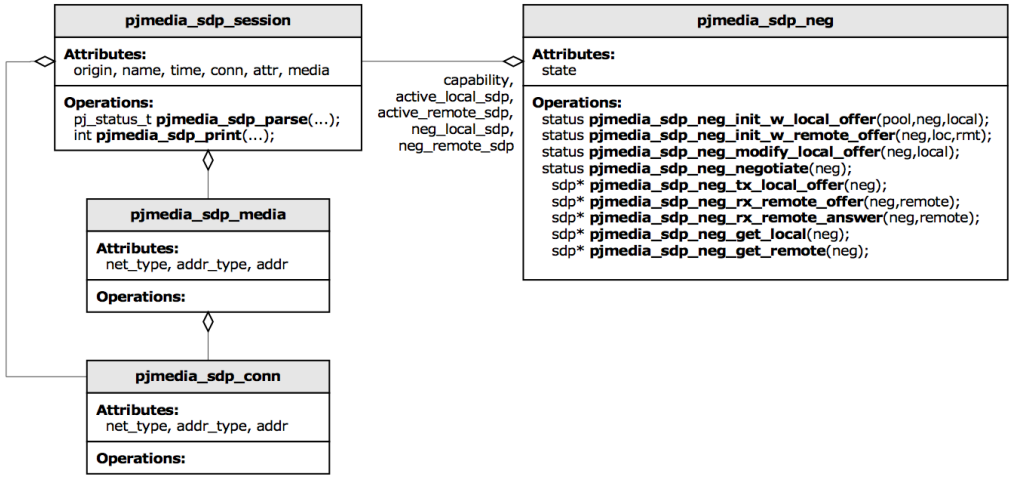


Figura 29: Class Diagram del negoziatore SDP

La già nominata ***pjmedia_sdp_neg*** è formata da tre strutture SDP:

- ***initial_sdp***: questo SDP è passato dal negoziatore durante la creazione e il suo contenuto non può essere cambiato durante la sessione generalmente.
- ***active_local_sdp***: contiene un SDP locale dopo essere stato negoziato con il remoto. Il dialog deve usare questo per iniziare il local media.
- ***active_remote_sdp***: contiene l'SDP corrente usato dal peer remoto.

In generale gli stati di una transizione SDP di offerta e risposta avviene come nello schema seguente.

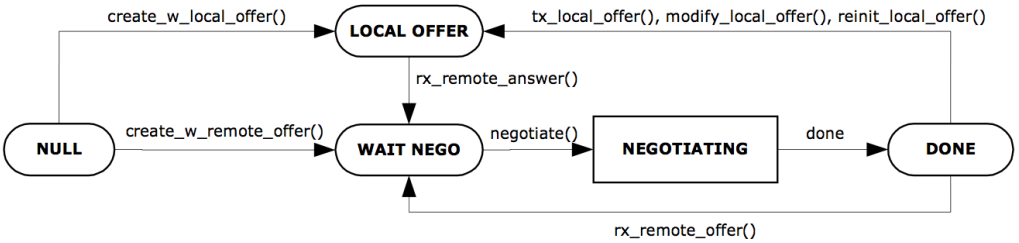


Figura 30: State Diagram di una sessione offerta/risposta SDP

Sebbene esistano questi meccanismi semiautomatici di negoziazione tramite SDP, può essere necessario, creare manualmente un body message per inserire e gestire eventuali parametri personalizzati. In questo caso si utilizza direttamente la struttura **pjmedia_sdp_session.**

Al suo interno troviamo tutti quei campi che possono andare appunto a descrivere la sessione e quindi possono essere lo User, il session ID, la versione, il tipo, l'indirizzo, i vari parametri (come si vede in figura) o, s, t, c ecc.

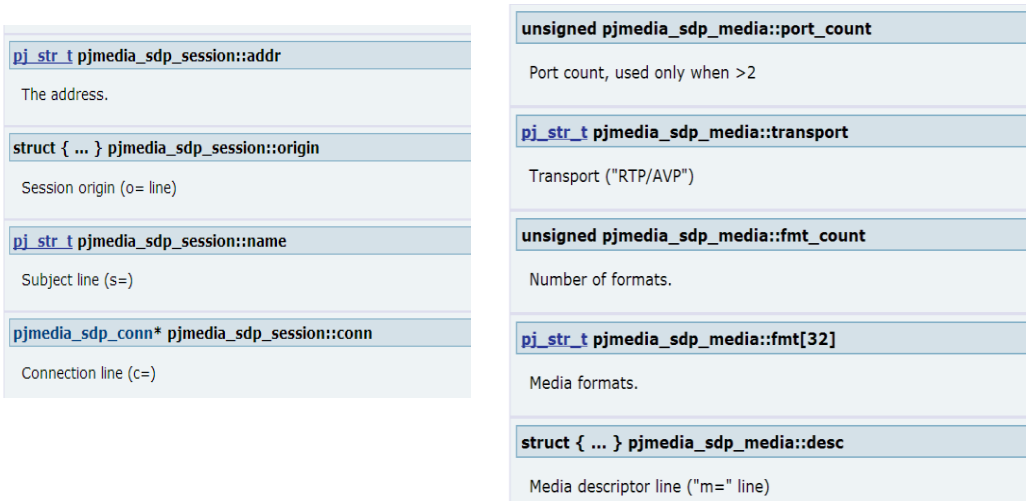


Figura 31: Attributi e Strutture pjmedia

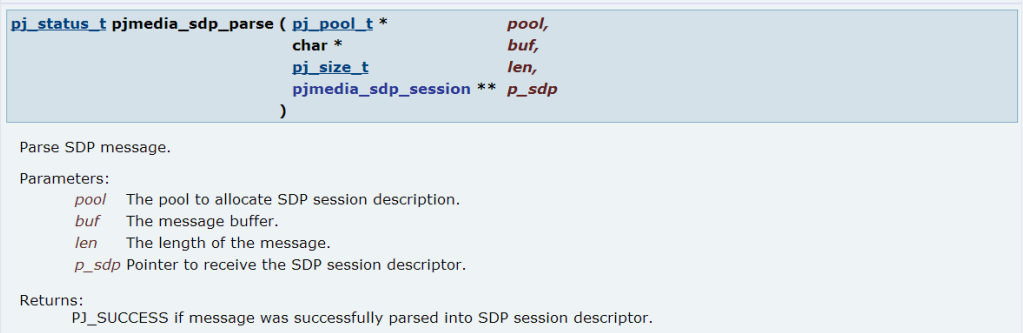
Inoltre sono presenti due campi array per gestire gli attributi e i parametri media, molto importanti per uno sviluppo Bticino in quanto e in questi due campi che si concentrano le personalizzazioni (si pensi alla variabile a: BTkind) e le richieste diverse di risoluzione e formato video. Entrambi le sotto-strutture hanno diverse funzioni per la gestione degli attributi come i classici create, insert, clone, find e remove; inoltre la struttura per i media gestisce tutti quei parametri che possono essere per esempio il tipo audio/video, il tipo di trasporto rtp/avp, il formato, il media descriptor ecc.

Abbiamo parlato dei messaggi e da cosa sono composti, ora trattiamo di come andare ad analizzarli, e cioè dei parser messi a disposizione dallo Stack. Quello principale è senza dubbio il find_msg, il quale controlla i pacchetti in arrivo nel buffer e vede se ci sono dei messaggi SIP validi; quando ciò accade viene indicata la grandezza del messaggio nella variabile *msg_size.* Controllata la validità del messaggio SIP, questo viene passato alla funzione **pjsip_msg* pjsip_parse_msg(pj_pool_t *pool, char *buf, pj_size_t size, pjsip_parser_err_report *err_list);** la quale ritorna un messaggio e la funzione

pjsip_msg* pjsip_parse_rdata(char *buf, pj_size_t size, pjsip_rx_data *rdata);

Secondariamente al controllo e all'estrazione del messaggio, vi sono i parser di URI e header che sono rispettivamente **pjsip_parse_uri** e **pjsip_parse_hdr** la quale fa il parsing del contenuto dell'header inline secondo l'header type *hname.*

Per quanto riguardo il parsing del body message SDP, ci viene in aiuto la funzione **pjmedia_sdp_parse,** la quale restituisce (se il processo ha avuto successo), un descrittore SDP. Vi sono poi altre funzioni utili per il parsing SDP come per esempio la validazione di tale parte del messaggio (sdp_validate) e il confronto tra due SDP (sdp_session_cmp).



```
pj_status_t pjmedia_sdp_parse ( pj_pool_t *              pool,
                                char *                   buf,
                                pj_size_t                len,
                                pjmedia_sdp_session **   p_sdp
                              )
```

Parse SDP message.

Parameters:

pool	The pool to allocate SDP session description.
buf	The message buffer.
len	The length of the message.
p_sdp	Pointer to receive the SDP session descriptor.

Returns:
PJ_SUCCESS if message was successfully parsed into SDP session descriptor.

Figura 32: Parser SDP

INVIO E RICEZIONE

Una struttura base di tutto lo stack SIP è il pjsip_rx_data (e poi rispettivamente il pjsip_tx_data per l'invio), la quale viene passata tra i vari componenti di PJsip al posto di un semplice messaggio rappresentato da una stringa. Il pjsip_tx_data viene utilizzato appunto quando si vuole inviare un messaggio e una volta creato viene passato al relativo data buffer.

Per inviare e ricevere i messaggi sulla rete, viene utilizzato il livello di trasporto (il diagramma delle classi è mostrato in figura), di cui attore principale è il Transport Manager (pjsip_tpmgr). Esso si occupa del tempo di vita del trasporto tramite un contatore e un idle timer, gestisce le factories del tran

sport, riceve i pacchetti, parsa i pacchetti e li distribuisce all'endpoint; instrada i messaggi sip basandosi sul tipo di trasporto e sull'indirizzo remoto e crea dinamicamente dei tran sport qualora non fossero disponibili per inviare un messaggio ad una nuova destinazione. Il tran sport manager è invisibile dall'applicazione ed è unico per un endpoint, il quale fornisce per l'appunto le funzioni tramite la sua interfaccia.
Le funzioni principali relative al livello di trasporto sono l'acquisizione tramite **pjsip_endpt_acquire_transport** e l'invio con **pjsip_transport_send.**
È possibile estendere il layer per creare differenti tipi di trasporto, e tutte le references fanno parte del file <pjsip/sip_transport.h>.

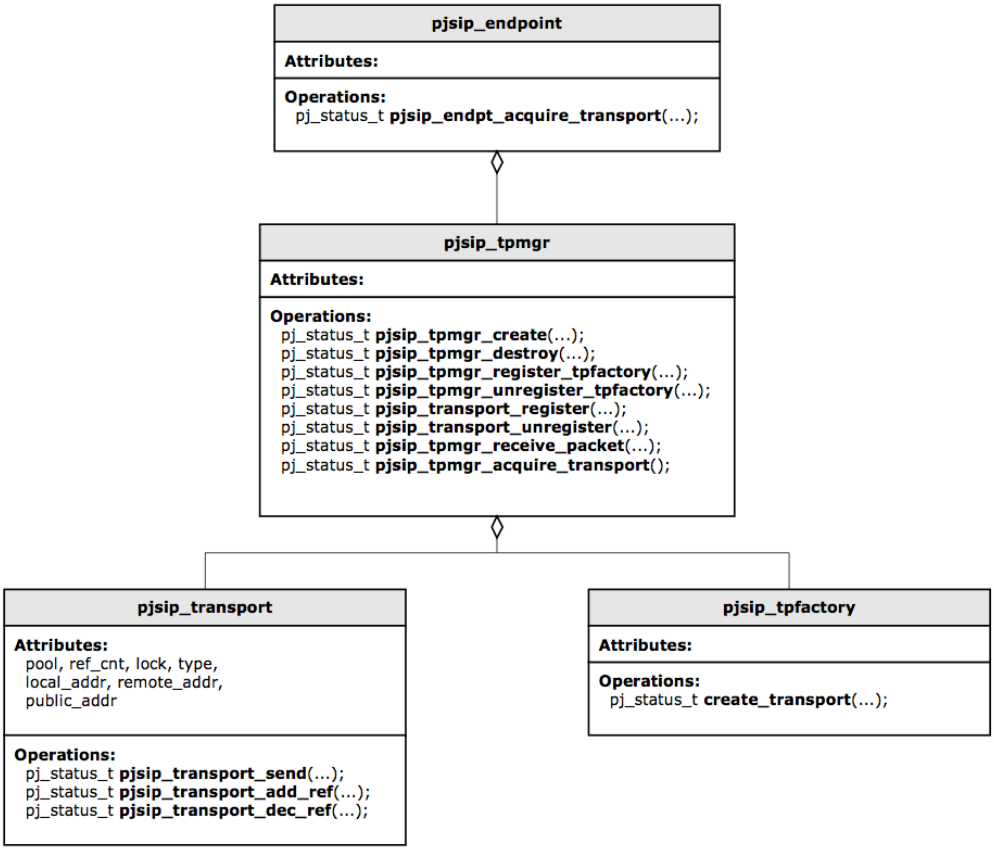


Figura 33: Class Diagram del livello di Traporto

La gestione degli eventi di invio e ricezione stateless è possibile gestirlo tramite callback e ve ne sono una per ogni tipologia di messaggio SIP, e cioè **on_rx_request() e on_rx_response()** rispettivamente per le richieste e per le

risposte. Per le risposte vi è la funzione **pjsip_endpt_create_response(), mentre** per le richieste invece vi sono **pjsip_endpt_create_request()**, **pjsip_endpt_create_request_from_hdr(), pjsip_endpt_create_ack()**, oppure **pjsip_endpt_create_cancel().** Se invece si vuole creare una richiesta (o una risposta) manualmente, si deve fare affidamento alla funzione **pjsip_endpt_create_tdata()** per creare un buffer di trasmissione, poi creare un messaggio con la **pjsip_msg_create()**, aggiungere i campi header tramite la **pjsip_msg_add_hdr()** e usare tutte le altre funzioni di creazione del messaggio (e body message) viste nei paragrafi precedenti). Esiste poi un altro sistema più ad alto livello per creare i messaggi e lo tratteremo in seguito (tramite il livello dialog).
Una volta avuto il messaggio, è pronto per essere inviato, ma non prima di avere "acquisito" il mezzo di trasporto con la **pjsip_endpt_acquire_transport(),** e quindi successivamente inviarlo con la **pjsip_transport_send();** Anche se queste funzioni sono molto a basso livello per essere usato direttamente, vengono però usate per inviare le richieste stateless assieme a **pjsip_endpt_send_request_stateless(),** che ovviamente ha il suo duale per le risposte nella funzione **pjsip_endpt_send_response().** Entrambi i tipi di messaggi si avvalgono di funzioni di supporto come la creazione di tdata, la comunicazione con l'endpoint e il getter dell'indirizzo (solo per citarne alcune). Qui di seguito un breve esempio della creazione di una request.

```
void my_send_request(){
pj_status_t status;
pjsip_tx_data *tdata;
status = pjsip_endpt_create_request( endpt, // endpoint
method, // method
"sip:bob@example.com", // target URI
"sip:alice@thishost.com", // From:
"sip:bob@example.com", // To:
"sip:alice@thishost.com", // Contact:
NULL, // Call-Id
0, // CSeq#
NULL, // body
&tdata ); // output
[...]
status = pjsip_endpt_send_request_stateless( endpt,
```

Figura 34: Invio Stateless di una Request

Per effettuare trasmissioni si utilizza anche (soprattutto per tipi stateful, ma non solo) la struttura pjsip_transaction, che è situata nel file <pjsip/sip_transaction.h>. Normalmente un ciclo di vita di una transazione segue questi step:

- Creata tramite **pjsip_tsx_endpt_create_uac()** / **pjsip_tsx_create_uas()**.
- L'applicazione chiama la fuzione **pjsip_tsx_recv_msg()** per passare l'iniziale il messaggio iniziale della request
- Quando l'applicazione vuole inviare una request o una response usando una transaction, viene chiamata la funzione **pjsip_tsx_send_msg()**
- La transaction automaticamente cambia lo stato appena il messaggio viene passato o il timer scaduto e viene notificato tramite la callback **on_tsx_state()**
- Infine la transaction viene automaticamente distrutto una volta raggiunto lo stato PJSIP_TSX_STATE_TERMINATED, tramite la funzione **pjsip_tsx_terminate().**

Le transazione hanno solo due tipi di timer: quello per le ritrasmissioni e il timout, ed entrambi sono impostati automaticamente dalla transazione secondo il tipo di transazione (UAS o UAC), di trasporto (affidabile o non affidabile) e il metodo (INVITE o non-INVITE).

Le funzioni principale per le transazioni sono **pjsip_tsx_create_uas** per inizializzare lo UAS, e **pjsip_tsx_send_msg** con **pjsip_tsx_recv_msg** rispettivamente per inviare e ricevere i messaggi (i parametri di entrambe sono i medesimi e cioè il puntatore alla transazione *tsx e il puntatore ai dati della transazione *tdata).

Alcuni funzionamenti di una transazione possono essere i seguenti (per inviare in modo state full una risposta):

- In modo complesso, vedi Figura 35.

```
static pj_bool_t on_rx_request( pjsip_rx_data *rdata )
{
pj_status_t status;
pjsip_transaction *tsx;
pjsip_tx_data *tdata;
// Create and initialize transaction.
status = pjsip_tsx_create_uas ( endpt, NULL, rdata, &tsx );
// Pass in the initial request message.
pjsip_tsx_recv_msg(tsx, rdata);
// Create response
status = pjsip_endpt_create_response( endpt, rdata, 200, NULL
/*OK*/, &tdata);
// The response message is good to send, but you may modify
it before
// sending the response.
// Send response with the specified transaction.
pjsip_tsx_send_msg( tsx, tdata );
return PJ_TRUE;
```

Figura 35: Invio Stateful di una response (complesso)

- In maniera più semplice, vedi Figura 36:

```
static pj_bool_t on_rx_request( pjsip_rx_data *rdata )
{
pj_status_t status;
// Respond to the request statefully
status = pjsip_endpt_respond( endpt, NULL, rdata,
200, NULL /* OK */, NULL, NULL, NULL);
return PJ_TRUE;
}
```

Figura 36: Invio Stateful di una response (semplice)

Per quanto riguarda inviare una richiesta invece, è possibile farlo manualmente, oppure con la funzione pjsip_endpt_send_request().

```
extern pjsip_module app_module;
void my_send_request()
{
pj_status_t status;
pjsip_tx_data *tdata;
pjsip_transaction *tsx;
// Create the request.
status = pjsip_endpt_create_request( endpt, ..., &tdata );
// You may modify the message before sending it.
...
// Create transaction.
status = pjsip_endpt_create_uac_tsx( endpt, &app_module,
tdata, &tsx );
// Send the request.
status = pjsip_tsx_send_msg( tsx, tdata /*or NULL*/);
}
static void on_tsx_state( pjsip_transaction *tsx, pjsip_event
*event )
{
pj_assert(event->type == PJSIP_EVENT_TSX_STATE);
PJ_LOG(3,("app", "Transaction %s: state changed to %s",
tsx->obj_name, pjsip_tsx_state_str(tsx->state)));
```

Figura 37: invio Statefull di una request

USER AGENT

Pjsip fornisce un User Agent base per facilitare la gestione dei dialoghi SIP (dialogs) come per esempio il contatore di sessione, il call-ID, il contact header (con from, to, ecc), la sequenca Cseq nelle transazioni ecc.
Il dialog non conosce l'effettivo stato della sessione, e neanche quando è stata stabilita o disconnessa, ma bensì il dialog inizia con un'attività ben precisa, e quando il contatore di sessione raggiunge lo zero e l'ultima transazione è terminata esso viene distrutto.

Parlando dapprima di un uso a basso livello, si possono individuare alcune funzioni utili per gestire il dialog e cioè la creazione (uac e uas) tramite **pjsip_dlg_create_uac** e **pjsip_dlg_create_uac,** poi **pjsip_dlg_inc_session** (e la duale dec per il decremento) per incrementare il numero di sessioni nel dialogo; e forse le più significative per uno stack SIP, le funzioni di creazione e

invio dei messaggi (sempre rispettivamente separate per request e response): **pjsip_dlg_create_request, pjsip_dlg_send_request.**
Nel seguente esempio viene usato un dialog API a basso livello e mostra come creare/inizializzare il dialog in ricezione, e rispondere con un RINGING 180 (ovviamente è solo un esempio, occorrerebbe gestire anche eventuali errori).

```
pj_bool_t on_rx_request(pjsip_rx_data *rdata)
{
if (rdata->msg->line.request.method.id == PJSIP_INVITE_METHOD
&&
pjsip_rdata_get_dlg(rdata) == NULL)
{
pjsip_dialog *dlg;
pjsip_transaction *tsx;
pjsip_tx_data *tdata;
struct app_dialog *app_dlg;
status = pjsip_dlg_create_uas( pjsip_ua_instance(), rdata,
NULL, &dlg);
status = pjsip_dlg_add_usage( dlg, &app_module, NULL );
pjsip_dlg_inc_session(dlg);
status = pjsip_dlg_create_response( dlg, rdata, 180, NULL
/*Ringing*/, &tdata);
status = pjsip_dlg_send_response( dlg,
pjsip_rdata_get_tsx(rdata), tdata);
dlg->mod_data[app_module.id] = pjsip_rdata_get_tsx(rdata);
return PJ_TRUE;
}}
```

Mentre nel successivo viene mostrato come inviare un ACK.

```
static void send_ack(pjsip_dialog *dlg,pjsip_rx_data *rdata){
pjsip_tx_data *tdata;
// Create ACK request
status = pjsip_dlg_create_request( dlg, &pjsip_ack_method,
rdata->msg_info.cseq->cseq, &tdata );
// Send the request.
status = pjsip_dlg_send_request ( dlg, tdata, NULL );
}
```

Figura 38 a, b: Dialog in ricezione, Invio di un ACK

Più ad alto livello invece, troviamo il cosiddetto “dialog invite session”, il quale può essere usato dall’applicazione per gestire le sessioni INVITE (incluso il

controllo del SDP). L'invite session è progettato per essere completamente astratto dal basic dialog, e quindi non ha bisogno di usare codeste API. È implementato in una libreria statica separata: pjsip-ua, e deve essere inclusa l'header <pjsip-ua/sip_inv.h> o alternativamente tutto <pjsip-ua.h> per usare le varie funzionalità.

L'invite session state fornisce particolari callback per notificare all'applicazione il progress della sessione, e si avvale di particolari descriptors per indicare i vari stati (PJSIP_INV_STATE_CALLING, PJSIP_INV_STATE_INCOMING, per esempio).

Nella figura seguente si può vedere lo state diagram dell'Invite Session.

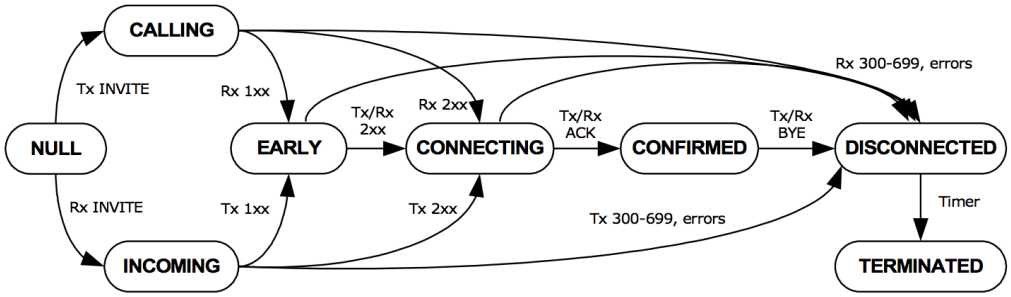


Figura 39: State diagram dell'Invite Session

Per le chiamate in uscite, occorre creare un dialog UAC tramite la funzione **pjsip_dlg_create_uac(),** e successivamente l'invite sessione per il suddetto dialog: **pjsip_inv_create_uac().** Analogamente sono presenti le funzioni per le richieste in arrivo, e in quel caso occorre prima verificare se la richiesta può essere o meno accettata tramite **pjsip_inv_verify_request()** la quale verifica i parametri supported, require e il body.

PJSIP può gestire tramite invite session dialog i messaggi di INVITE, BYE, ACK, CANCEL, UPDATE, PRACK, anche se può comunque avvalersi delle API del base dialog per creare ed inviare request e response per messaggi diversi dai sopracitati. Estendendo il supporto ad altri tipi di messaggi, è importante ricordarsi di assegnare una giusta priortià (tramite PJSIP_MOD_PRIORITY_APPLICATION).

La struttura **pjsip_inv_callback** viene chiamata quando qualcosa nella sessione è cambiato, per esempio lo stato (attraverso on_state_changed), oppure una nuova sessione è stata inizializzata (on_new_session); oppure ancora quando

viene ricevuta una nuova offerta (on_rx_offert), e l'applicazione richiama la funzione pjsip_inv_set_sdp_anser() per rispondere alla negoziazione del SDP.

Per creare l'iniziale INVITE request per la sessione, viene utilizzata la funzione **pjsip_inv_invite(pjsip_inv_session *inv, pjsip_tx_data **tdata);** dove la richiesta viene messa nell'argomento tdata se è stato creato con successo; inoltre se sono specificati i parametri media quando è stata creata l'invite session, questa funzione inserisce un'offerta SDP nella richiesta INVITE in uscita (altrimenti il body non conterrà nessun messaggio SDP). L'altra funzione molto importante è la **pjsip_inv_answer(pjsip_inv_session *inv,**
int st_code, const pj_str_t *st_text, const pjmedia_sdp_session *local_sdp, pjsip_tx_data **tdata); che come dice il nome crea un messaggio di risposta all'INVITE request inviato in precedenza.

5 SCENARIO OPENWEBNET

5.1 ANALISI

Lo scopo principale di questo scenario è la realizzazione di un'applicazione per i dispositivi Android, in grado di comunicare con un video-citofono, attraverso un web server proprietario Bticino, per ricevere un flusso video in tempo reale.

È possibile inoltre, effettuare una chiamata da posto interno al posto esterno, eventualmente per utilizzare la telecamera come videosorveglianza, va notato, però, che se si dovesse effettuare una chiamata da posto esterno contemporaneamente ad una "osservazione" da qualsiasi posto interno, quest'ultima risulterebbe avere priorità più bassa e quindi cesserebbe in favore della comunicazione standard. Questo ovviamente perché è una funzione aggiuntiva, ma quella principale rimane pur sempre quella di video-citofonia, la quale ha la precedenza e quindi un'osservazione dall'interno non può tenere occupato il posto esterno.

La comunicazione tra web server F454 e i dispositivi ad esso collegati (device Android e video-citofono) avviene per mezzo del protocollo Open Web Net (*vedi il relativo capitolo "*Open Web Net*"*).

Il terminale è in connessione LAN (nel nostro caso via Wi-Fi, quindi wireless) con il web server. Nel progetto analizzeremo una configurazione unicast e solo in seguito si potrà pensare ad una gestione multi-cast, e quindi ad una trasmissione del video a tutti i dispositivi collegati.

Lo scenario che ci troviamo ad analizzare è il seguente:

> *Nel momento in cui riceviamo una "chiamata" dal videocitofono, il nostro device Android dovrà mostrare una notifica che ci avvisa della chiamata in arrivo. All'apertura dell'applicazione, l'utente potrà decidere di visualizzare o non visualizzare il video; in caso positivo, il web server invierà al device il flusso video che verrà visualizzato sullo schermo del dispositivo. A questo punto, possiamo rifiutare o accettare*

la chiamata, e in questo caso avviare lo scambio audio tra i device in comunicazione.

5.1.1 Componenti e specifiche tecniche

In questa sezione descriveremo i componenti, le specifiche e i protocolli che bisogna conoscere per poter capire al meglio il capitolo relativo all'implementazione del prototipo.

Video citofono

Il videocitofono è un impianto citofonico in cui è integrato un sistema di monitoraggio video.

L'unità esterna, oltre alla pulsantiera e alla sezione audio comprendente il microfono e l'altoparlante, include anche una telecamera in bianco e nero o a colori. Vicino alla telecamera sono presenti alcuni LED a raggi infrarossi (a cui la telecamera B/N è sensibile) per illuminare la scena e consentire la visione della scena con qualunque condizione di luce ci sia all'esterno.

Ogni unità interna è dotata di un monitor B/N o colore su cui è visibile l'immagine ripresa dalla telecamera. Il monitor è costituito da uno speciale tubo catodico in cui la superficie fluorescente ha profilo semiparabolico ed il pennello elettronico giunge lateralmente, colpendo la superficie dallo stesso lato da cui l'immagine è visualizzata. Questo allo scopo di ottenere un apparecchio quanto più sottile possibile. Nei modelli recenti, a questo tipo di monitor viene solitamente preferito un display LCD.

Il sistema video si attiva solo nel momento in cui un utente preme un pulsante di chiamata, e si disattiva dopo un tempo prefissato. Per ovvi motivi di privacy, l'immagine è visibile solamente nell'unità interna dell'interno chiamato.

Web Server F454

È un web server audio/video utilizzato per il controllo, sia locale che remoto, delle applicazioni *MY HOME* tramite pagine web dedicate.

Il Web Server può essere utilizzato anche come gateway per la gestione dell'impianto tramite dispositivi quali PC e smartphone e per la configurazione virtuale utilizzando il software dedicato (vedi "*Impostazione del web server F454*").

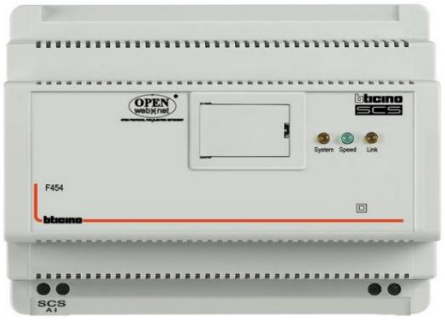


Figura 40: web server F454

Questo dispositivo consente di operare su un impianto MY HOME, installato in casa o nell'ufficio. La connessione può essere effettuata tramite modem e/o rete dati (cablata o wireless). L'utente, utilizzando un Personal computer, uno smartphone o un tablet con browser può collegarsi localmente o remotamente con il Web Server tramite pagine Web personalizzabili ed effettuare le seguenti operazioni:

- Supervisione e comando degli impianti Automazione, Gestione energia e Termoregolazione;
- Supervisione dell'impianto Antifurto mediante ricezione di messaggi di stato ("impianto in allarme" o "nessuna segnalazione di allarme"). È possibile inoltre ricevere un messaggio e-mail con immagini allegate, alla propria casella di posta, quando vengono segnalati eventi verificatisi nell'impianto antifurto;
- Supervisione dell'impianto elettrico mediante ricezione di messaggi di intervento del dispositivo Stop&Go.

Inoltre è possibile effettuare una connessione audio e video real-time con le telecamere presenti nell'impianto videocitofonico. In particolare è possibile:

- Vedere l'immagine tramessa dalla telecamera selezionata (con la possibilità da PC di intervenire sulla qualità dell'immagine, l'inquadratura e lo zoom);

- Ascoltare l'audio registrato dal microfono della telecamera selezionata (sorveglianza acustica da remoto);
- Inviare la propria voce all'altoparlante associato alla telecamera selezionata (comunicazione audio bidirezionale real-time da Personal computer);
- Ascolto e visione dei messaggi registrati con la funzione "Segreteria videocitofonica". I messaggi audio e le immagini possono anche essere inviate mediante e-mail ad un indirizzo di posta elettronica.

Open Web Net

Nell'ultimo decennio BTicino ha investito nella ricerca e nello sviluppo di impianti domotici, applicabili sia a realtà residenziali che industriali.

Per permettere a chiunque abbia conoscenze informatiche di linguaggi ad alto livello, di poter interagire con i sistemi e di poter costruire funzioni innovative, è stato studiato e implementato un linguaggio di comunicazione con il nome di Open Web Net, dove OPEN sta per *Open Protocol for Electronic Networks*.

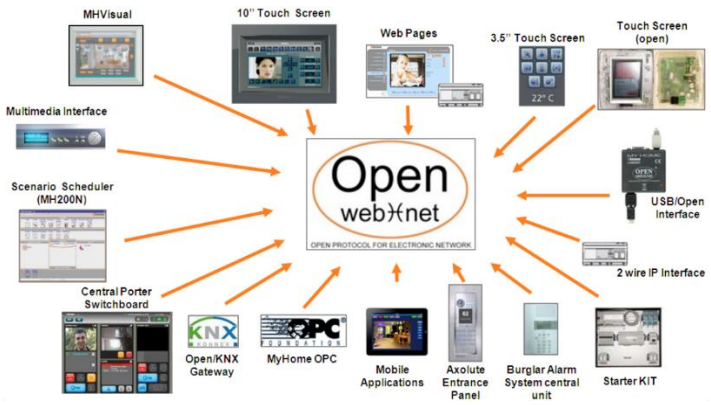


Figura 41: Open Web Net

L'Open Web Net è un linguaggio grazie al quale è possibile scambiare dati ed inviare comandi tra un'unità remota e il sistema My Home BTicino. Il protocollo è pensato per essere indipendente dal mezzo di comunicazione utilizzato, considerando come requisito minimo la possibilità di poter utilizzare toni DTMF

sulla normale linea telefonica PSTN. Attualmente, i dispositivi che utilizzano tale protocollo sono i web server, il comunicatore telefonico e l'attuatore telefonico.

Questo linguaggio è stato pensato anche per permettere l'integrazione con funzioni di altri marchi oppure per permettere a dispositivi come PC, smartphone e tablet di comunicare con l'impianto My Home in remoto.

L'Open Web Net è stato introdotto per fornire un livello astratto che permette la supervisione e il controllo dei sistemi My Home concentrandosi sulle funzioni senza curarsi dei dettagli di dell'installazione e senza dover per forza conoscere la tecnologia SCS.

Sintassi

Un messaggio del protocollo OPEN è composto da caratteri appartenenti al seguente insieme di caratteri:

{0,1,2,3,4,5,6,7,8,9,*,#}

Il messaggio inizia con il carattere '*' e termina con la coppia di caratteri '##'. Il carattere '*' viene anche usato come separatore tra i vari tag. Un messaggio OPEN è quindi così strutturato:

****Tag1*Tag2*Tag3*…*TagN##***

Un "*Tag*" è composto solo dai caratteri che appartengono all'insieme {*0,1,2,3,4,5,6,7,8,9,*,#*} e non può contenere la coppia di caratteri "*##*". Un tag può anche essere omesso.

Open di comando

Il messaggio Open, che ha tale funzione, è così strutturato:

****CHI*COSA*DOVE##***

- ***Tag CHI***: individua la funzione dell'impianto domotico interessata al messaggio OPEN in questione.
- ***Tag COSA***: individua l'azione da compiere (e.g. ON luci, OFF luci, dimmer al 20%, tapparelle SU, tapparelle GIU…). Per ogni CHI viene stabilita una tabelle dei COSA. Questo tag può anche contenere dei parametri (facoltativi) specificati in questo modo:

COSA#PAR$_1$#PAR$_2$#...#Par$_n$

- ***Tag DOVE***: individua l'insieme degli oggetti interessati al messaggio (zona, gruppo di oggetti, ambiente specifico, singolo oggetto, intero sistema).
  Per ogni CHI viene specificata una tabella dei DOVE. Anche questo tag può contenere dei parametri (facoltativi) specificati in maniera analoga al tag COSA.

Open di richiesta stato
Il messaggio Open di richiesta stato è così strutturato:

****#CHI*DOVE ##***

Viene inviato dal client nella sessione comandi/azioni per richiedere informazioni sullo stato di un singolo oggetto, di un insieme di oggetti o di un intero sistema. Il server risponde a questa richiesta inviando uno o più messaggi Open di stato. La risposta deve terminare con un messaggio di ACK (o di NACK in caso di problemi o se l'oggetto/i di cui si richiede lo stato non è presente sul sistema).

Open di richiesta valore / grandezza
Il messaggio OPEN di richiesta grandezze è così strutturato:

****CHI*DOVE*GRANDEZZA##***

Viene utilizzato dal client nelle sessioni di tipo comandi per richiedere informazioni sul valore di un grandezza di un singolo oggetto, di un insieme di oggetti o di un intero sistema.

Il server risponde ad una richiesta inviando uno o più messaggi OPEN di valore grandezza così strutturati:

****CHI*DOVE*GRANDEZZA*VAL$_1$*…*VAL$_n$##***

Il numero di campi VAL dipende dalla GRANDEZZA richiesta. Il messaggio di risposta è seguito dal messaggio di ACK. Se non vi è seguito alla richiesta con una risposta o si verifica un errore nella frame di richiesta grandezza segue un messaggio di NACK.

Messaggi Open particolari

Oltre ai messaggi di comando esistono dei messaggi particolari che vengono trasmessi all'interno del flusso comunicativo come il messaggio di ACK e di NACK:

- ***Messaggio di ACK***. Il messaggio Open di Acknowledge ha la seguente sintassi:

  ***#*1##**

  Indica che il messaggio Open, inviato dal Client e ricevuto dal Server, è sintatticamente e semanticamente corretto. Inoltre viene utilizzato come messaggio terminatore quando la risposta ad un messaggio Open preveda l'invio di uno o più messaggi in sequenza (richiesta stato o richiesta grandezze).

- ***Messaggio di NACK***. Il messaggio OPEN di Not-Acknowledge (NACK) è

  ***#*0##**

  Indica che il messaggio Open, inviato dal Client e ricevuto dal Server, è semanticamente o sintatticamente errato. Inoltre viene utilizzato come messaggio terminatore quando la risposta ad un messaggio Open preveda l'invio di uno o più messaggi in sequenza (richiesta stato o richiesta grandezze). In questo caso, il client deve considerare non validi i messaggi ricevuti prima del NACK.

Sessioni di Comunicazione

I gateway TCP-IP offrono il server Open Web Net su porta 20000. Le fasi che possiamo individuare per instaurare una connessione sono tre: connessione, identificazione, comunicazione.

Il client OPEN può instaurare due tipologie di sessioni:

- Sessione comandi (azioni): utilizzata per inviare comandi, richiedere lo stato; richiedere e impostare la dimensione.
- Sessione eventi: usata dal Client Open per leggere tutto quello che succede sul bus dell'impianto domotico in modo asincrono.

Figura 42: sessione di comunicazione

La fase di identificazione varia se nel Server OPEN è configurato un range di indirizzi IP dal quale è possibile instaurare una connessione senza l'utilizzo della password OPEN [8]. Tale situazione è tipica nella realizzazione di software per la gestione dell'impianto personalizzati.

Sessione comando/azione

Subito dopo aver instaurato una connessione TCP-IP tra la macchina che richiede il servizio (il client) e la macchina che offre il servizio (il server), il flusso che si instaura è osservabile nella Figura 43.

Sessione eventi

Subito dopo aver instaurato una connessione TCP-IP tra la macchina che richiede il servizio (il client) e la macchina che offre il servizio (il server), il flusso che si instaura è osservabile nella Figura 43.

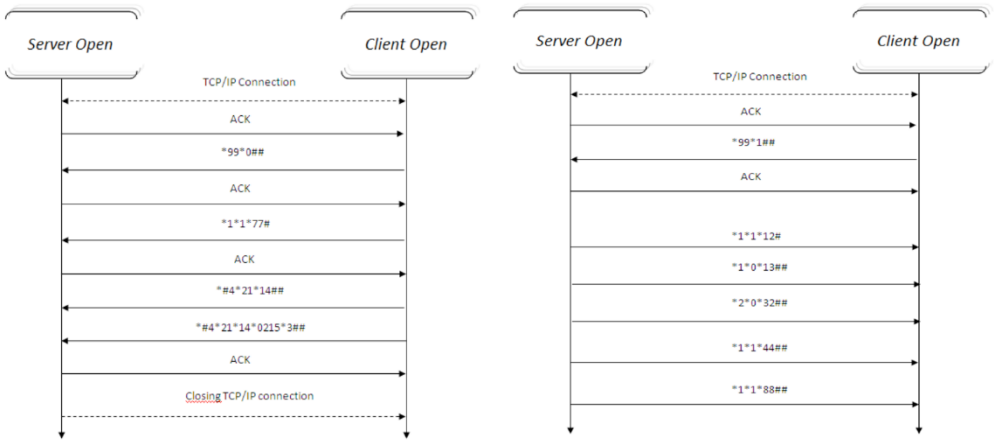


Figura 43: sessione comandi e sessione eventi

BTCommLib

BTCommLib è una nuova libreria Java/Android che ognuno può utilizzare per connettersi, monitorare e gestire l'impianto MyHome Bticino attraverso il gateway compatibile con OpenWebNet.

Struttura della Libreria

La libreria BTCommLib, come mostrato nella Figura 44, è un insieme di componenti Java costruite attorno ad un componente principale chiamato "*communication library*".

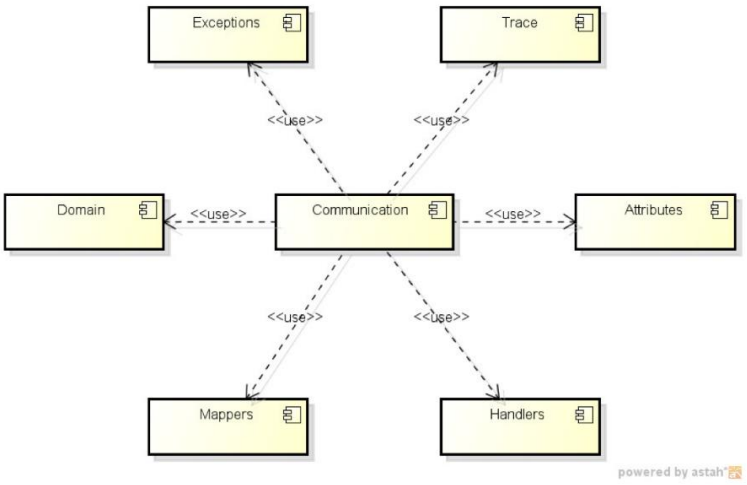


Figura 44: struttura della libreria

Ogni componente è responsabile di uno specifico sottoinsieme di funzionalità, che la libreria deve sostenere, e che saranno descritte nei seguenti paragrafi:

- ***Communication***: questo è il componente principale che la libreria fornisce in quanto è responsabile per le principali interfacce dei servizi pubblici che ogni applicazione deve utilizzare per poter lavorare con il gateway compatibile con OpenWebNet.
  Tra le principali interfacce principali fornite da questo componente c'è il BTCommMgr, un componente che si occupa del canale di comunicazione specifico e la BTCommChan, una classe astratta che fornisce i servizi di comunicazione supportati all'applicazione finale.

- ***Attributes***: è una sorta di componente interno ausiliario responsabile di fornire classi di annotazione che verranno usate per “decorare” il codice dell’applicazione finale in modo che la libreria sarà in grado di identificare le informazioni correlate all’ OpenWebNet, impacchettarle e spacchettarle in accordo con il protocollo e inviarle dal layer applicazioni al gateway e viceversa.
- ***Domain***: ha il compito di definire le classi che si occupano delle rappresentazioni di messaggi di alto livello e di basso livello, il formato ad alto livello è una sorta di messaggio equivalente OpenWebNet che è più vicino al dominio di livello applicazione e potrebbe anche essere totalmente indipendente dalla rappresentazione dei frame OpenWebNet di basso-livello.
- ***Mappers***: grazie alla componente mapper, un messaggio di alto livello e di basso livello possono essere considerati equivalenti in quanto ogni mapper sarà in grado di trasformare un messaggio di alto livello in un basso livello e viceversa.
- ***Exeptions***: contiene le classi di gestione delle eccezioni che la libreria utilizza per sollevare specifiche eccezioni a livello applicazione.
- ***Handlers***: la componente handler fornisce un insieme di gestori specifici progettati per prendersi cura dei messaggi e degli eventi provenienti dal livello applicazione o dal gateway OpenWebNet. L'obiettivo principale di tali handler è tradurre i frame "puri" OpenWebNet nei corrispondenti messaggi di alto livello o in eventi e viceversa.
- ***Trace***: attualmente, la libreria si basa su API Java di logging per il tracciamento dei servizi, *package java.util.logging*.

Principali funzionalità

In questa sezione, attraverso degli esempi di codice, mostreremo le principali funzionalità che questa libreria mette a disposizione, in modo da capire meglio come BTCommLib lavora e come essa fornisce servizi che possono essere utilizzati per sviluppare applicazioni, compatibili con MyHome Bticino, per la piattaforma Android. In particolare mostreremo come connettersi ad un Gateway OpenWebNet, mandare comandi al dispositivo dell’impianto e ricevere eventi da esso.

Impostazione dei parametri di connessione

Impostare i parametri di connessione è necessario per essere in grado di connettersi al gateway, e per fare questo bisogna definire: la modalità di connessione (RO=ReadOnly, RW=ReadWrite), l'indirizzo IP e il numero della porta di un dato gateway OpenWebNet.

```
private static final String _IP = "IP";
private static final String _PORT = "PORT";
private static final String _ MODE = "MODE";
private static final String _IP_VALUE = "192.168.1.107";
private static final String _PORT_OPEN_VALUE = "20000";
private static final String _MODE_VALUE = "RW";
```

Figura 45: setting parametri di connessione

Setting gestiore di eventi e notifica

La creazione di un gestore permette alla libreria di mappare gli eventi OpenWebNet in un dato oggetto della classe evento dell'applicazione. Se l'applicazione non fornisce una classe evento, può essere usata la versione presente di default.

```
try{
  _myOpenChan = cmdMgr.getCommChan(BTChanTypes.Open);
  _myOpenChan.handleEvent(BTOpenMsgType.EventStatus.name(),
     DefaultMsgEventStatus.class);
. . .
}
```

Figura 46: setting gestore di eventi e notifica

Connessione con il gateway

La connessione a gateway avviene grazie al codice mostrato in *figura xxx, connessione al gateway*:

```
_connPar.setConnPar (_IP, _IP_VALUE);
_connPar.setConnPar (_PORT, _PORT_OPEN_VALUE);
_connPar.setConnPar (_MODE, _MODE_VALUE);

_myOpenChan.connect(_connPar);
```

Figura 47: instaurare connessione al gateway

Fondamentalmente, la libreria fornisce due differenti modalità di connessione: ReadOnly che permette solo di ricevere gli eventi di notifica dal gateway monitorato, mentre ReadWrite permette di inviare comandi e ricevere gli eventi di notifica.

Invio di un frame al gateway

Una volta che è disponibile un'istanza di un canale di connessione OpenWebNet, l'applicazione può iniziare ad inviare comandi e ricevere gli eventi correlati, in accordo con il formato dei messaggi OpenWebNet:

```
_myOpenChan.Send("*1*1*11##", out outResult);
```

Figura 48: invio di un frame

Gestione notifiche ed eventi

In accordo con il set-up del canale di connessione OpenWenNet, gli eventi di notifica giungono dal impianto gestito (gateway) e le notifiche di errore verranno gestite dai servizi dell'applicazione *notifyEvent* e *notifyError*.

```
_myOpenChan.addListener(new IBTCommNotify(){
      public void notifyEvent(Object highLevelEventMsg){
            Logger.info ("OnEventReceived: " +
            highLevelEventMsg.getClass().getName());
      }
      public void notifyError(BTCommErr errorMsg){
            Logger.info ("OnErrorReceived: " +
            errorMsg.getErrEnum() );
      }}
```

Figura 49: gestione notifiche ed eventi

5.2 PROGETTAZIONE

5.2.1 Casi d'uso

Per capire al meglio come sviluppare e comprendere i requisiti del progetto sono stati presi come riferimento alcuni casi d'uso. Il caso d'uso in informatica è una tecnica usata nei processi di ingegneria del software per effettuare in maniera esaustiva e non ambigua la raccolta dei requisiti al fine di produrre software di qualità. Vengono utilizzati per l'individuazione e la registrazione dei requisiti funzionali scrivendo come un sistema possa essere utilizzato per consentire agli utenti i raggiungere i loro obiettivi.

Attori principali:

Nome: *Utente Interno*

Descrizione: è solitamente il proprietario del device o comunque uno degli abitanti dell'abitazione, è colui che utilizza l'applicazione.

Aspettative-Obiettivi:

- Ricevere chiamate dall'esterno dell'abitazione
- Effettuare sessioni di videosorveglianza

Nome: *Utente Esterno*

Descrizione: è colui che effettua la chiamata dall'esterno dell'abitazione attraverso il videocitofono.

Aspettative-Obiettivi:

- Effettuare chiamate dall'esterno dell'abitazione

Caso d'uso UC1: Chiamata in arrivo

Livello: Obiettivo utente

Attore primario: Utente Interno

Attore secondario: Utente Esterno

Parti interessate e interessi:

- Utente Interno: riceve una chiamata e comunica con l'interlocutore.
- Utente Esterno: vuole effettuare una chiamata verso il posto interno per comunicare.

Pre-condizioni:

- Il web server deve essere in funzione connesso correttamente alla rete domestica.
- Il videocitofono deve essere funzionante e connesso alla rete domestica.
- Il device deve essere acceso e connesso correttamente alla rete domestica.
- L'applicazione deve essere installata
- Il servizio di ricezione degli eventi dell'applicazione deve essere in esecuzione e non essere stato interrotto dall'utente.

Garanzie di successo: i due utenti comunicano tra loro e lo scambio dei flussi video e/o audio avviene correttamente senza grossi problemi.

Scenario principale di successo:

- L'utente che si trova all'esterno dell'abitazione effettua una chiamata al posto interno attraverso il videocitofono.
- Sullo smartphone viene visualizzata una notifica che avvisa l'utente principale che è stata effettuata una chiamata verso la sua abitazione.
- L'utente clicca sulla notifica.
- Sullo smartphone viene lanciata l'applicazione che visualizza la schermata di chiamata in arrivo.
- L'utente guarda il video proveniente dal videocitofono e, attraverso l'apposito tasto, accetta la chiamata.
- Avviene lo scambio audio/video.
- La chiamata termina.

Possibili scenari negativi:

- L'utente che utilizza l'applicazione non clicca sulla notifica.
- L'utente che utilizza l'applicazione decide di non accettare.

Caso d'uso UC2: Videosorveglianza

Livello: Obiettivo utente

Attore primario: Utilizzatore dell'applicazione

Parti interessate e interessi:

- Utente Interno: vuole effettuare una sessione di videosorveglianza utilizzando la telecamera del videocitofono.

Pre-condizioni:

- Il web server deve essere in funzione connesso correttamente alla rete domestica.
- Il videocitofono deve essere funzionante e connesso alla rete domestica.
- Il device deve essere acceso e connesso correttamente alla rete domestica.
- L'applicazione deve essere installata

Garanzie di successo: l'utente effettua la sessione di videosorveglianza ricevendo il video sul proprio device per una durata di 60 secondi.

Scenario principale di successo:

- L'utente decide di voler effettuare una sessione di videosorveglianza.
- L'utente apre l'applicazione installata sul proprio dispositivo.
- L'utente avvia la funzionalità di videosorveglianza attraverso l'apposito tasto.
- Cliccando sul tasto di avvio, inizia lo streaming del video proveniente dalla telecamera del videocitofono.
- Trascorsi i 60 secondi viene mostrato un popup che avvisa l'utente della fine del tempo a disposizione.
- L'applicazione torna alla schermata principale

Lo schema seguente riassume l'analisi degli attori e delle loro interazioni in uno Use Case Diagram. L'unità esterna infatti interagisce solo durante la chiamata

iniziale e lo stream Audio/video, l'unità interna invece, si rende responsabile di visualizzare la notifica (gestita dal sistema con un controllo in background delle chiamate in corso) e decide se accettare e rifiutare la chiamata, che poi inizierà lo stream audio/video con l'unità esterna.

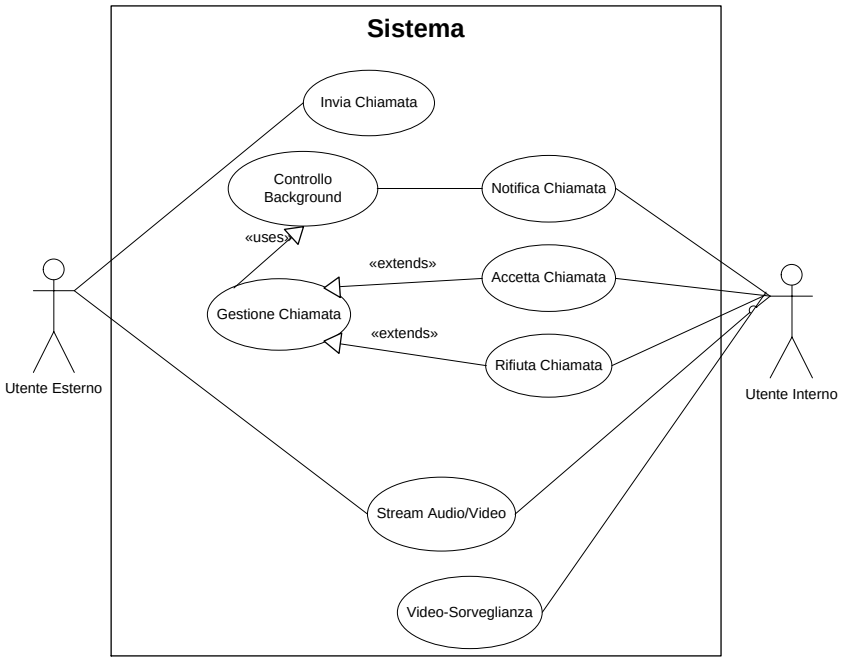


Figura 50: Use Case Diagram

5.2.2 Activity Diagram

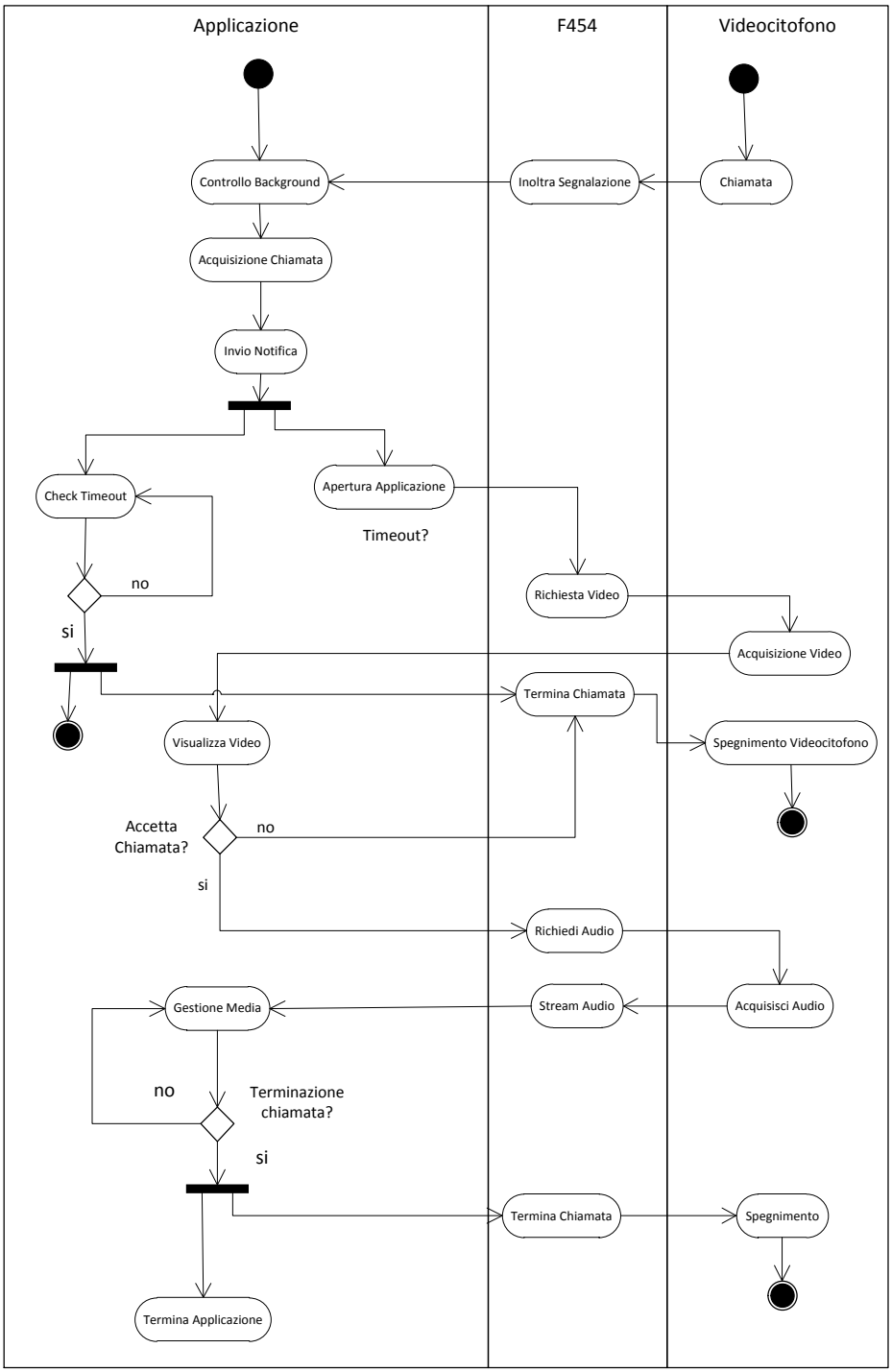


Figura 51: activity diagram chiamata

Il diagramma in Figura 51 mostra il processo completo di chiamata, includendo al suo interno tutte le possibili variazioni come, ad esempio, rifiutare o accettare la chiamata in arrivo.

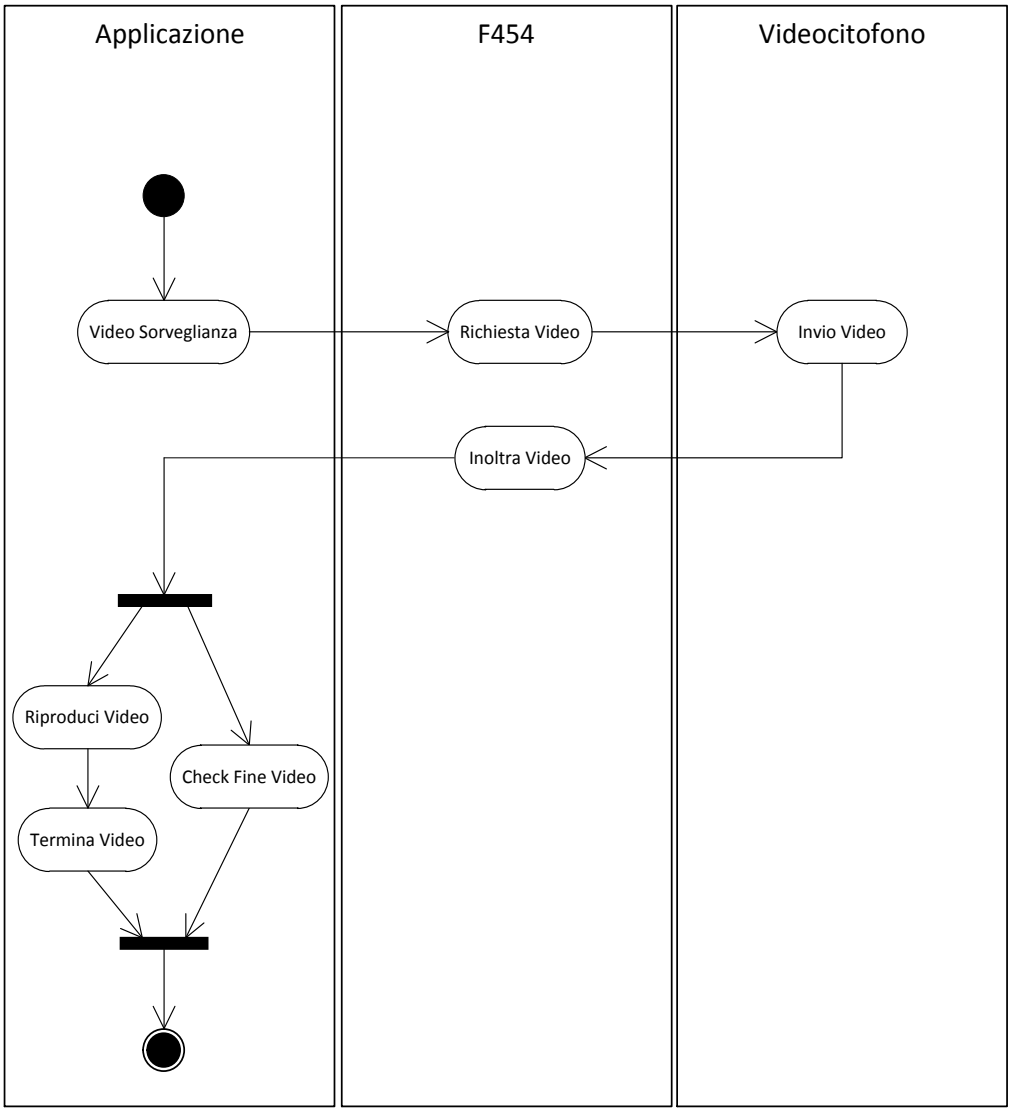


Figura 52: activity diagram Video Sorveglianza

Il diagramma delle attività in Figura 52 mostra, invece, il processo che viene eseguito durante una sessione di video sorveglianza.

5.2.3 Dependence e Class Diagram

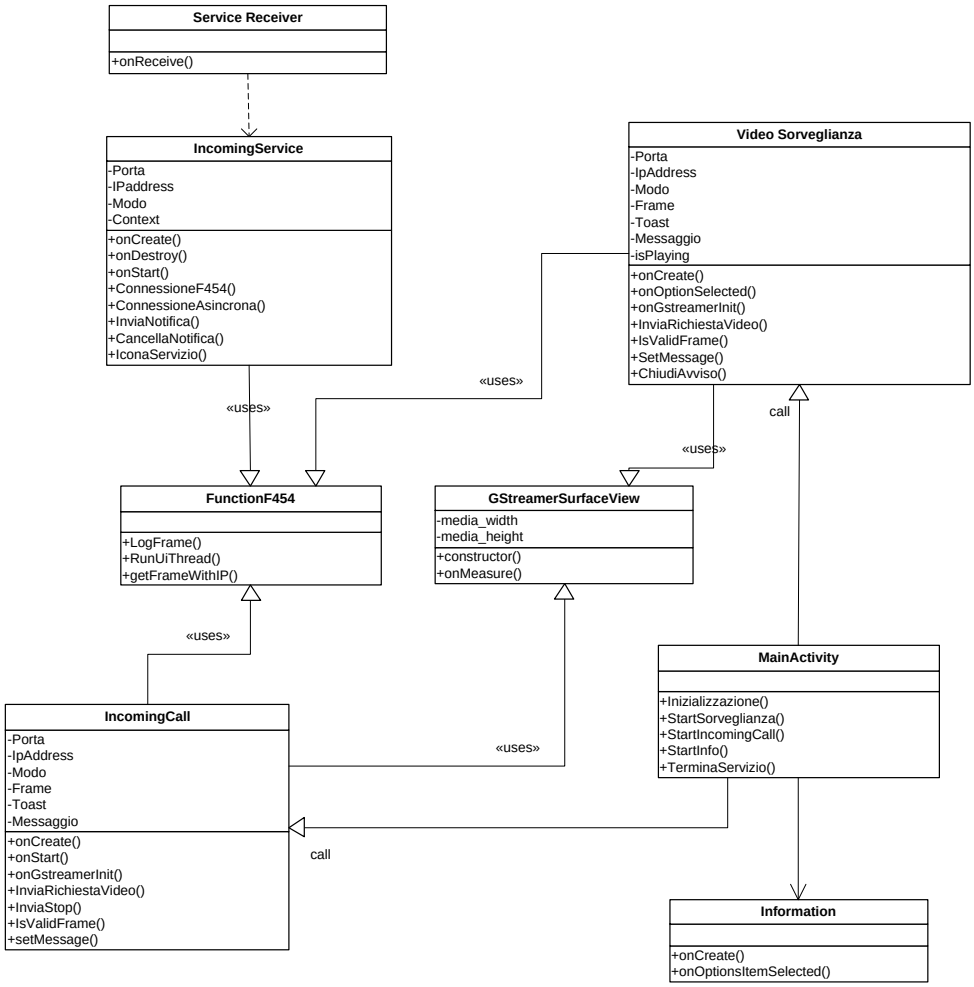


Figura 53: class diagram dell'applicazione

Il Class Diagram precedente, mostra la struttura dell'applicazione e la relazione fra i vari componenti. L'attività principale è la **MainActivity**, che si occupa di inizializzare l'interfaccia grafica della homepage dell'applicazione, il menu, e richiamare a sua volta le altre attività in base alle User Experience. Le attività da essa richiamate sono rappresentate dalle classi:

- ***Information***: utilizzata per mostrare alcune informazioni dell'applicazione.
- ***VideoSorveglianza***: si occupa di gestire la richiesta del video da parte dell'utente, senza che vi sia una vera e propria azione di chiamata esterna.

- ***IncomingCall***: utilizzata per la gestione della chiamata dopo la ricezione della notifica di “Chiamata in Arrivo”, invia infatti il frame di richiesta del video da mostrare all’utente e inizializza il contenitore view.
- ***IncomingService***: classe che, grazie al ***Service Receiver***, riesce a rimanere perennemente attiva, e gestisce in background l’arrivo delle chiamate, inoltre si connette al web server F454 tramite un’apposita funzione.

Vi sono infine due classi le quali forniscono servizi e funzioni alle altre:

- ***FunctionF454***: funzioni di servizio per il web server.
- ***GStreamerSurfaceView***: funzioni utili per la gestione di GStreamer.

5.2.4 Sequence Diagram

Per comprendere al meglio i due use-case descritti precedentemente, analizzeremo ora i loro *sequence diagram*, riproducendo le sequenze complete di un processo di chiamata e di video sorveglianza.

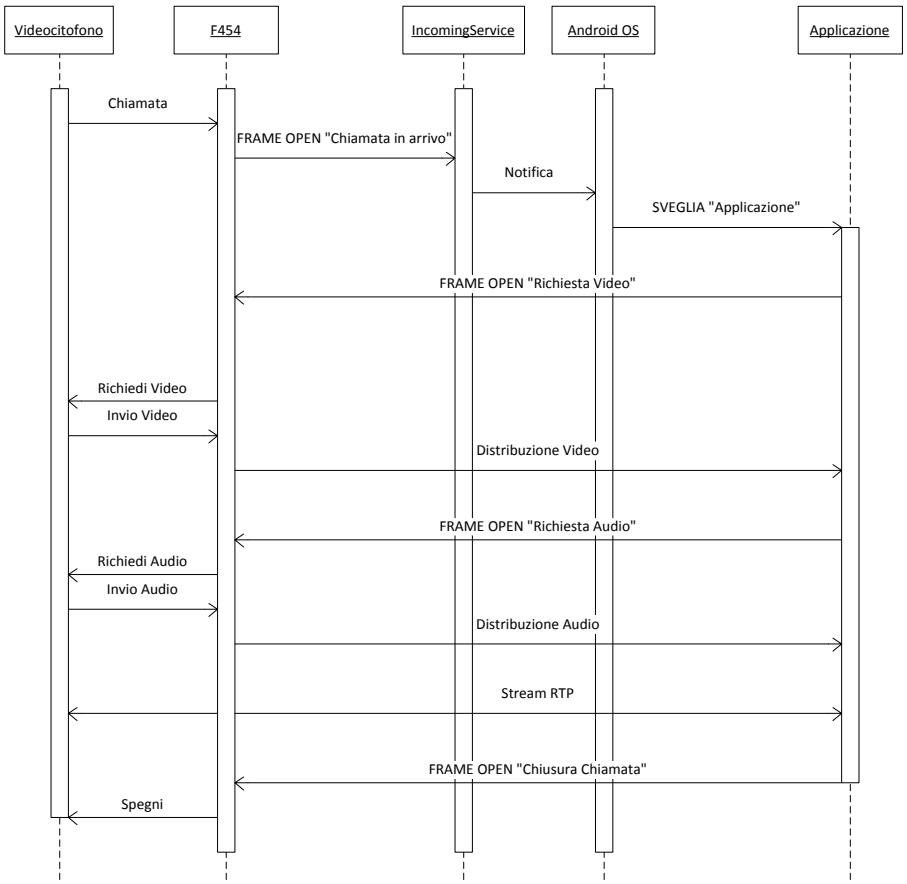


Figura 54: sequence diagram chiamata completa

Il primo diagramma inizia con la chiamata dal posto esterno, e termina con la chiusura tramite applicazione (posto interno). Il web server F454 invia il frame Open riferito alla chiamata in corso, il quale viene interpretato dal servizio “IncomingService”, il quale invia una notifica sul telefono, dove si provvede a risvegliare/aprire l’applicazione corrispondente. A questo punto l’applicazione richiede il segnale video da mostrare all’utente, la segnalazione viene inviata al web server che acquisisce tale stream e lo spedisce sul canale di ritorno.

L'utente a questo punto visualizza il video e decide di accettare la chiamata, in questo modo si avvia la comunicazione bidirezionale audio. Ad un certo punto l'utente termina la chiamata, e al web server viene inviato un comando Open per la chiusura e il videocitofono viene spento.

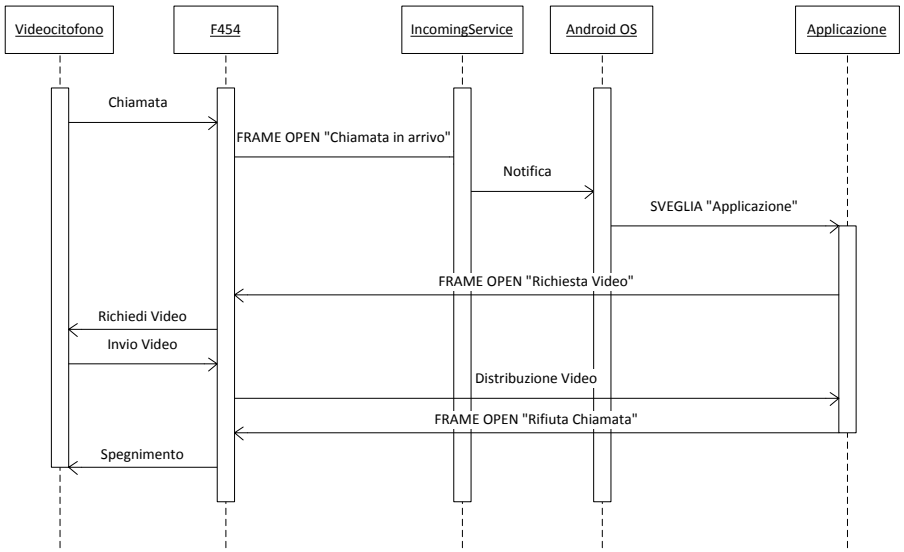


Figura 55: dettaglio sequence diagram chiamata rifiutata

Il sequence diagram di una chiamata rifiutata, differisce dal precedente solo nella parte successiva alla richiesta video effettuata dall'applicazione. In questo caso infatti, l'utente rifiuta la chiamata, e viene immediatamente inviato il frame command di chiusura del videocitofono al web server, il quale a sua volta provvede allo spegnimento.

Può accadere inoltre, che l'utente non si accorga della notifica di chiamata in corso. Quando questo accade, viene impostato un timeout utilizzato per terminare in modo automatico la richiesta. Nel prossimo diagramma viene mostrato proprio questo, scaduto il tempo il web server invia in automatico il frame necessario per lo spegnimento del videocitofono.

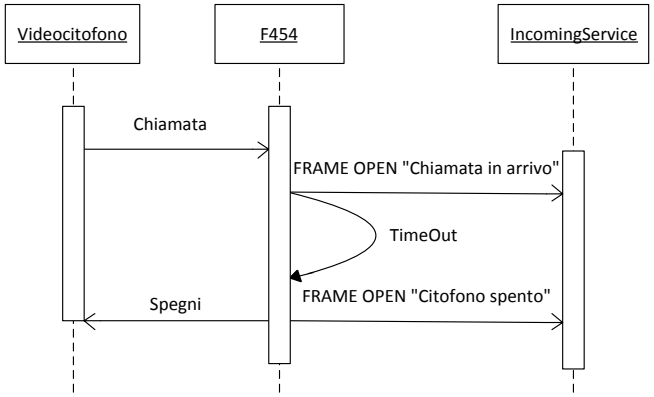


Figura 56: dettaglio sequence diagram chiamata persa

Per l'attività di video sorveglianza è, invece, l'applicazione che inizia la sequenza con una richiesta video inviata al web server. A ricezione di tale richiesta il posto esterno trasmette il flusso video al web server che lo inoltra all'applicazione. Scaduto il timeout impostato dal sistema, il videocitofono viene spento e il web server segnala all'applicazione la chiusura della comunicazione.

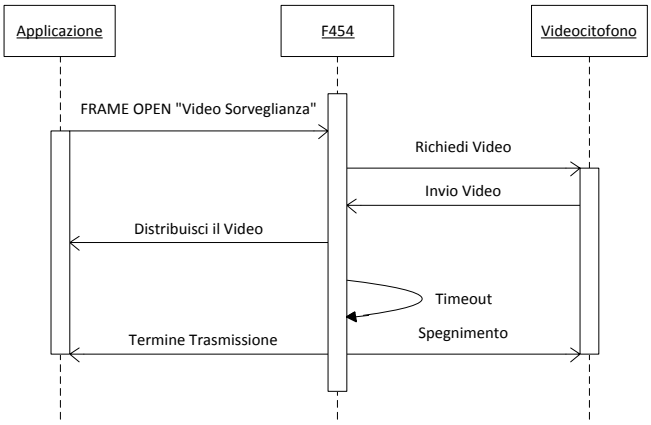


Figura 57: sequence diagram video sorveglianza

5.3 IMPLEMENTAZIONE

Come anticipato nell'analisi dello scenario, nel realizzare il prototipo dell'applicazione abbiamo dovuto considerare alcune limitazioni dovute al fatto che il web server in nostro possesso è ancora un prototipo che non presenta ancora tutte le funzionalità e le caratteristiche che avrà in futuro (vedi tabella riassuntiva sotto).

F454	Prototipo	Versione definitiva
Segnali in ingesso	---	Audio
Segnali in uscita	Video	Audio / Video
Formati supportati	MJpeg 320x240 25fps H264 320x240 25fps	MJpeg 320x240 25fps H264 320x240 25fps

Figura 58: differenze tra prototipo e versione definitiva

A causa di queste limitazioni, il nostro prototipo non presenterà lo scambio del flusso audio ma si limiterà, quindi, alla sola ricezione del flusso video, risultando leggermente diverso rispetto allo scenario analizzato in precedenza.

> *Nel momento in cui riceviamo una "chiamata" dal videocitofono, l'applicazione sul device dovrà mostrare una notifica che ci avvisa della chiamata in arrivo. All'apertura dell'applicazione, l'utente visualizzerà il video e deciderà di accettare o meno la chiamata. Un'altra funzione fruibile dall'utente potrà essere quella di effettuare sessioni di videosorveglianza, richiedendo dal device il video proveniente dall'esterno.*

5.3.1 Impostazione del web server F454

Collegamento alla rete domestica

Come prima cosa, per poter testare l'applicazione e impostare il server, si è reso necessario collegare alla rete domestica la strumentazione fornitaci da Bticino.

Gli strumenti che dobbiamo andare a connettere sono:

- Web server F454, di proprietà di Bticino.
- Prototipo Video-citofono, di proprietà di Bticino.
- Bus SCS, di proprietà di Bticino.
- Alimentatore, di proprietà di Bticino.
- Cavo di Alimentazione.
- Cavo Ethernet.

Figura 59: strumenti utilizzati

La prima cosa da fare è collegare tra di loro il videocitofono e il web server F454, tramite il bus SCS.

Come passo successivo, dobbiamo connettere il web server al modem/router, attraverso il cavo ethernet, in modo che sia tutto connesso alla rete domestica. Come ultimo passaggio, non resta che connettere il video-citofono e il web server alla linea elettrica, per mezzo dell'alimentatore e dei cavi di alimentazione.

Software di gestione di Bticino e impostazioni di base

Per impostare correttamente il web server F454 occorre utilizzare il software di gestione dato in dotazione da Bticino: TiF454 versione 1.0.

Per caricare/scaricare un'impostazione sul/dal web server, occorre che il dispositivo sia già collegato alla rete LAN (deve avere però già un'impostazione caricata per dotarlo di indirizzo IP; di solito c'è la configurazione standard, altrimenti si può collegare direttamente al computer sempre con un cavo RJ45) a cui è collegato anche il computer configuratore, oppure collegarsi direttamente con un connettore USB (il web server ha una porta mini USB).

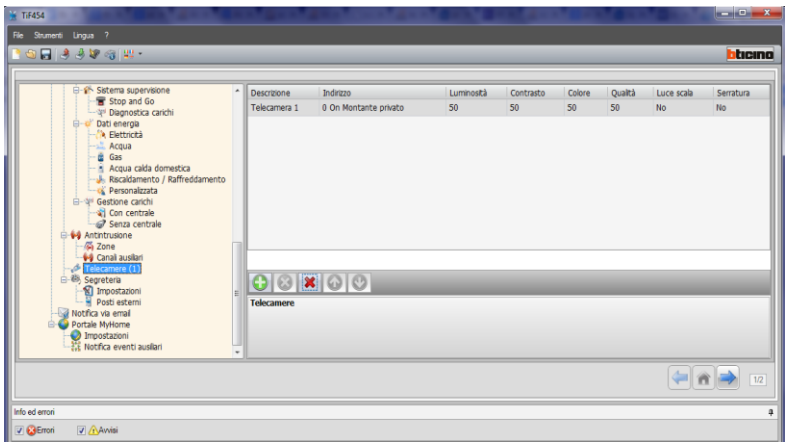


Figura 60: interfaccia grafica programma di gestione TiF454

Per prima cosa, occorre fornire al web server un indirizzo IP (se si vuole cambiare quello dato di default), cambiando eventualmente la rete di appartenenza, il gateway e la subnetmask. Fatto questo passaggio, saremmo in grado di "raggiungere" il web server attraverso la rete di cui fa parte. Se è già presente una configurazione, è possibile scaricarla per modificarla in seguito (e ricaricarla), altrimenti si aggiungono i vari dispositivi tra quelli a scelta nel menu ad albero sul lato sinistro, impostando i vari valori in base ai dispositivi corrispondenti. Per il prototipo nello scenario2, abbiamo a disposizione una telecamera, per cui ne abbiamo aggiunta una (come si vede anche in figura), dandole come indirizzo 0 (sul montante privato) e valori predefiniti per quanto riguarda luminosità ecc. Finita l'impostazione dei dispositivi (in questo scenario semplificato c'è solo la telecamera, ma in un sistema reale possono essercene diversi) si prosegue con il caricamento delle impostazioni sul dispositivo. Quando le impostazioni sono caricate, il web server sa riconoscere quei dispositivi che manualmente sono stati aggiunti, e quindi, accedendo alla pagina web del F454, è possibile monitorare tutti questi dispositivi. Inoltre, il web server è raggiungibile dalle varie applicazioni (e dispositivi) all'indirizzo a

cui è stato assegnato nel primo passaggio della configurazione. Quando una nuova funzionalità viene aggiunta da Bticino, il software di gestione viene utilizzato anche per installare il nuovo firmware, e permettere quindi l'utilizzo di tali funzionalità (ed interpretare correttamente così i vari nuovi comandi "open web net").

5.3.2 Connessione tra Android ed il web server F454

La prima cosa abbiamo implementato nella nostra applicazione è la connessione tra il device Android e il web server Bticino F454.

Per realizzarla abbiamo utilizzato la libreria BTCommLib, personalizzata in modo che si adattasse ai nostri scopi e che fornisse tutte le funzionalità a noi necessarie.

Come prima cosa, abbiamo scaricato la libreria e l'abbiamo importata all'interno della nostra applicazione, per mezzo delle seguenti import:

```
import bticino.btcommlib.communication.BTChanTypes;
import bticino.btcommlib.communication.BTCommChan;
import bticino.btcommlib.communication.BTCommChanPar;
import bticino.btcommlib.communication.BTCommErr;
import bticino.btcommlib.communication.BTCommMgr;
import bticino.btcommlib.communication.IBTCommNotify;
import bticino.btcommlib.domain.highlevel.*;
import bticino.btcommlib.domain.util.BTOpenMsgType;
import bticino.btcommlib.exceptions.LibException;
import bticino.btcommlib.trace.BTLibLogger;
```

Figura 61: import libreria BTCommLib

Il secondo passaggio prevede la dichiarazione e il setting dei parametri di connessione, utilizzate in seguito per connettersi al gateway e/o per modificare la modalità di connessione.

```
public static final String IP_ADDRESS = "192.168.1.35";
public static final String PORT_NUM = "20000";
public static final String MONITOR = "RW";
public static String OPEN_CMD = "";
```

Figura 62: setting parametri di connessione

Come è facile osservare, l'indirizzo IP, il numero della porta e la modalità di connessione (MONITOR) vengono settati a priori, in quanto stabiliti dalle impostazioni del web server (come vedremo in seguito). Per quanto riguarda la variabile OPEN_CMD (che rappresenta il comando open da inviare al server), vedremo nel capitolo successivo come questa sia dinamica e dipenda dall'indirizzo IP del device utilizzato.

Il passaggio successivo è quello di instaurare la connessione attraverso il metodo connectionToF454 (per i dettagli vedere il codice nell'Allegato). Questo avviene all'interno del metodo OnStar() del servizio che verrà eseguito in background (vedi il capitolo "Android Service").

```
public void onStart(Intent intent, int startid) {
      Toast.makeText(this, "My Service Started",
      Toast.LENGTH_LONG).show();
      Log.d(Variables.TAG, "onStart");
      ServiceIcon(_ThisContext);
      connectionToF454();
```

Figura 63: inizializzazione tasto di connessione

Il metodo ServiceIcon() serve per creare un'icona persistente, che verrà visualizzata nella barra di sistema, verrà trattato nel dettaglio all'interno del capitolo relativo al Service.

```
_connPar.setConnPar(getString(R.string.IP), _ipAddressValue);
_connPar.setConnPar(getString(R.string.PORT), _portValue);
_connPar.setConnPar(getString(R.string.MODE), _modeValue);

new ConnectionAsynk().execute();
```

Figura 64: esecuzione della connessione

Lo snippet sopra mostra un estratto di codice del metodo connectionToF454(), più precisamente, esegue l'impostazione dei parametri di connessione (indirizzo IP, numero della porta e modalità di connessione) e l'esecuzione della connessione al gateway, attraverso un task asincrono messo a disposizione da Android, del quale è possibile vederne una porzione di codice sotto (Figura 65).

Questa classe consente di eseguire operazioni in background e di pubblicare i risultati sul thread dell'interfaccia utente senza dover manipolare i thread e/o gli handlers.

```
public class ConnectionAsynk extends AsyncTask<Activity,
Integer, BTCommErr> {
protected BTCommErr doInBackground(Activity... actpar) {
  Variables.logger.info("SONO NEL ASYNKTASK");
  return Variables._myOpenChan2.connect(Variables._connPar);
}
...
```

Figura 65: ConnectionAsynk

Lo snippet sotto, invece, mostra parte del codice contenuto nel metodo disconnect_click(); in particolare è la chiamata al metodo disconnect(), che esegue immediatamente la disconnessione, chiudendo il canale di comunicazione aperto durante la connessione.

```
_myOpenChan.disconnect();
```

Figura 66: chiusura del canale

5.3.3 Scambio messaggi con il web server

Una delle particolarità di questo scenario sta proprio nello scambio dei messaggi tra il device e il web server F454. Questa infatti, avviene attraverso il protocollo Open Web Net (vedi il capitolo Open Web Net). Per questo motivo, abbiamo concordato con i tecnici di Bticino l'utilizzo di un nuovi frame Open, dei comandi che potessero riassumere al loro interno tutte le informazioni necessarie per richiedere il video al server e, in seguito, di permettere al server stesso di inviarlo al device corretto.

Il frame più importante che prendiamo in esame è il seguente:

****chi*cosa#IP1#IP2#IP3#IP4#PORTA*dove##***

Analizzandolo nel dettaglio, tag per tag, ci accorgiamo che, grazie a questo frame, forniamo al server tutte le informazioni necessarie; questo permette quindi di fare tutto quello che ci serve in un unico comando. Al suo interno, infatti, abbiamo:

- Il tag **CHI**. Indica la funzione dell'impianto domotico alla quale è indirizzato il comando OPEN, nel nostro caso il video-citofono; quindi questo tag avrà valore:
  - **6** per la videochiamata.
  - **7** per la videosorveglianza.
- Il tag **COSA**. Individua l'azione da compiere, nel nostro caso l'invio del video mandato dal video-citofono; abbiamo due possibili alternative:
  - Richiesta video **MJpeg**: in questo caso, il tag COSA sarà al valore **4** per la videochiamata e **31** per la videosorveglianza.
  - Richiesta video **H.264**: invece nel caso si voglia questo tipo di video, il tag COSA sarà settato al valore **13** per la videochiamata e **32** per la videosorveglianza.
- Il tag **DOVE**. Individua l'insieme degli oggetti interessati al messaggio, in questo caso la telecamera del citofono; quindi sarà **4000** per la videosorveglianza, mentre per la videochiamata questo valore va omesso**.**

I due tag seguenti, invece, vengono utilizzati per comunicare al server dove indirizzare il flusso video richiesto, e questo avviene per mezzo dell'indirizzo IP e del numero della PORTA:

- Il tag **PORTA**. Indica il numero della porta sulla quale il server invia il flusso video; abbiamo deciso di utilizzare la porta **20000**, quella predefinita del web server f454.
- I tag **IP**. Indica l'indirizzo IP al quale il server invia il flusso video, cambia in base al device che utilizzato.
  Attraverso il metodo *getFrameWithIP* (vedi Figura 67), da noi creato, catturiamo l'indirizzo IP del device che stiamo utilizzando e creiamo il frame, in modo da automatizzare l'operazione e non dover ogni volta modificare manualmente l'indirizzo IP.

```
private String getFrameWithIp(int command){
WifiManager
wifiMan=(WifiManager)_theContext.getSystemService
(Context.WIFI_SERVICE);
WifiInfo wifiInf = wifiMan.getConnectionInfo();
int ipAddress = wifiInf.getIpAddress();
String _ip = String.format("%d#%d#%d#%d", (ipAddress &
0xff),(ipAddress >> 8 & 0xff),(ipAddress >> 16 &
0xff),(ipAddress >> 24 & 0xff));

 switch(command){
      case 1:
      _frameWithIp = "*6*32#"+_ip+"#20000*4000##";
      break;
      case 2:
      _frameWithIp = "*7*13#"+_ip+"#20000*##";
      break;
  }

 return _frameWithIp;
```

Figura 67: snippet getFameWithIP()

Come si può notare, l'indirizzo IP viene fornito direttamente in formato ***IP1#IP2#IP3#IP4*** e non nel formato ***IP1.IP2.IP3.IP4***, in modo da poter essere immediatamente inserito nel frame.
Invece i tag CHI, COSA, DOVE e PORTA, come detto in precedenza, sono statici e quindi assegniamo loro i valori prestabiliti, osservabili anche nel codice.

Differenze tra i due frame utilizzati

A questo punto possiamo analizzare nel complesso i due frame visti precedentemente, per mostrarne le differenze nel loro utilizzo.

Il primo frame è quello utilizzato per la videosorveglianza:

****6*32#IP1#IP2#IP3#IP4#20000*4000##***

Come visto nell'analisi nel dettaglio delle parti, grazie a questo comando sarà possibile richiedere al web server l'invio, all'indirizzo IP specificato, del flusso

video proveniente dalla videocamera posizionata sul videocitofono. Particolarità di questo comando è quella di poter accendere la videocamera.

Il secondo è quello utilizzato invece per la gestione della chiamata, nel nostro caso abbiamo solo la trasmissione del video.

7*13#IP1#IP2#IP3#IP4#20000##

Questo frame invece, per quanto simile al precedente, non consente di poter accendere la videocamera, quindi per poter ricevere il video è necessario che la videocamera del videocitofono sia già attivata, attivazione che solitamente avviene con l'avvio della chiamata dall'esterno.

5.3.4 Integrazione di GStreamer in Android

In questo capitolo tratteremo dell'utilizzo di GStreamer all'interno del progetto Android. Attraverso alcuni snippet di codice, mostreremo i passaggi significativi che abbiamo svolto per poter includere e utilizzare questa libreria all'interno della nostra applicazione.

Il codice Java

MainActivity.java, l'attività principale

Come primo passo, è necessario includere all'interno del progetto il pacchetto contenente le classi necessarie al funzionamento della libreria GStreamer, questo è possibile per mezzo della seguente *import*:

```
import com.gstreamer.GStreamer;
```

Figura 68: snippet import GStreamer

Un altro passo fondamentale è permettere a Java di chiamare codice C, e questo avviene attraverso i metodi nativi, come quelli osservabili nello snippet sotto:

```
private native void nativeInit();
private native void nativeFinalize();
private native void nativePlay();
private native void nativePause();
private static native boolean nativeClassInit();
private native void nativeSurfaceInit(Object surface);
private native void nativeSurfaceFinalize();
```

Figura 69: snippet metodi nativi

Questi comunicano a Java che esistono dei metodi con questo nome che possono essere compilati senza problemi. Sta al programmatore assicurarsi che, a runtime, questi metodi siano accessibili. Questo è ottenibile con il codice mostrato in seguito.

I primi bit di codice che vengono effettivamente eseguiti sono gli "inizializzatori" statici delle classi:

```
static {
      System.loadLibrary("gstreamer_android");
      System.loadLibrary("ScenarioDue");
      nativeClassInit();
}
```

Figura 70: snippet librerie statiche

Il codice sopra carica libgstreamer_andorid.so, che contiene tutti i metodi di GStreamer, e libscenariodue.so che contiene il codice C che analizzeremo in seguito. Inoltre chiamiamo anche il metodo nativeClassiInit(), dichiarato in precedenza.

Dopo il caricamento, vengono eseguiti tutti i metodi delle librerie JNI_OnLoad(). Fondamentalmente registrano i metodi nativi che queste librerie espongono. La libreria GStreamer espone un solo metodo init(), che inizializza GStreamer e registra tutti i plugin.

Successivamente, nel metodo OnCreate() della nostra Activity, inizializzeremo realmente GStreamer attraverso la chiamata GStreamer.init(). Questo metodo richiede il contesto, per questo motivo non può essere chiamato dall'inizializzatore statico visto in precedenza, ma non c'è pericolo nel

richiamarlo più volte in quanto tutte le chiamate, eccetto la prima, vengono ignorate.

```
try {
      GStreamer.init(this);
} catch (Exception e) {
      Toast.makeText(this, e.getMessage(),
      Toast.LENGTH_LONG).show();
      finish();
      return; }
```

Figura 71: snippet inizializzazione GStreamer

In caso l'inizializzazione fallisca, il metodo init() solleverà un'eccezione con i dati forniti dalla libreria GStreamer.

Sempre nel metodo OnCreate(), andremo a settare gli inflate e i listener per gestire le parti dell'interfaccia grafica che andranno ad interagire con la libreria GStreamer:

```
ImageButton surveillance = (ImageButton)
this.findViewById(R.id.button_start_surv);
surveillance.setOnClickListener(new OnClickListener() {
      public void onClick(View v) {
            try{
                  sendVideo();
            } catch (Exception e) {
                  return;
            }
```

Figura 72: inizializzazione elementi della UI

Attraverso il codice sopra, assegniamo ad ogni elemento della UI un metodo che consenta di eseguire l'azione desiderata, dalla richiesta video all'interruzione dei servizi.

```
nativeInit();
```

Figura 73: inizializzazione metodi nativi

Questa chiamata al metodo nativeInit() chiude il metodo OnCreate() e l'inizializzazione del codice Java. I bottoni della UI sono disabilitati, quindi niente avviene affinché il codice nativo non sia pronto e chiamato il metodo onGStreamerInitialized():

```
private void onGStreamerInitialized () {
      Log.i ("GStreamer", "Gst initialized. Restoring state,
playing:" + is_playing_desired);
      if (is_playing_desired) {
         nativePlay();
      } else {
         nativePause();
      }
      final Activity activity = this;
      runOnUiThread(new Runnable() {
            public void run() {
                activity.findViewById(R.id.button_play).setEna
                bled(true);
                activity.findViewById(R.id.button_stop).setEna
                bled(true);
      ...
```

Figura 74: inizializzazione GStreamer

Questo metodo è chiamato dal codice nativo quando il main loop è in esecuzione. Prima di tutto, recupera lo stato di playing desiderato e, in seguito, setta lo stato. Come ultima cosa, riabilita i bottoni della UI.

Visto che Android non fornisce un sistema di windowing (un tipo di interfaccia grafica che implementa il paradigma WIMP), il video sink di GStreamer non può creare una finestra pop-up come avverrebbe in una piattaforma Desktop tradizionale. Per ovviare a questo inconveniente, aggiungeremo al layout principale un widget SurfaceView:

```
SurfaceView sv = (SurfaceView)
this.findViewById(R.id.surface_video);
SurfaceHolder sh = sv.getHolder();
sh.addCallback(this);
```

Figura 75: inizializzazione widget

GStreamerSurfaceView, la SurfaceView personalizzata

Di default, SurfaceView non ha particolari dimensioni, perciò si espande su tutta la superficie disponibile. Anche se questo a volte può essere conveniente, non permette un adeguato livello di controllo. In particolare, quando la superficie non ha la stessa aspect ratio del media, il sink aggiungerà dei bordi neri, che è un lavoro superfluo oltre che uno spreco di risorse.

La sottoclasse della SurfaceView che abbiamo creato sovrascrive (esegue una override) il metodo onMeasure, in modo da riportare le reali dimensioni del media, cosicchè che la surface possa adattarsi ad ogni superficie, rispettando l'aspect ratio del media.

Il codice C

IL METODO JNI_ONLOAD()

Il metodo JNI_OnLoad() è eseguito ogni volta che la JVM (Java Virtual Machine) carica una libreria.

```
jint JNI_OnLoad(JavaVM *vm, void *reserved) {
  JNIEnv *env = NULL;

  java_vm = vm;

  if ((*vm)->GetEnv(vm, (void**) &env, JNI_VERSION_1_4) !=
JNI_OK) {
    __android_log_print (ANDROID_LOG_ERROR, "ScenarioDue",
"Could not retrieve JNIEnv");
    return 0;
  }
  jclass klass = (*env)->FindClass
(env,"tesi/bticino/scenariodue/MainActivity");
  (*env)->RegisterNatives (env, klass, native_methods,
G_N_ELEMENTS(native_methods));

  pthread_key_create (&current_jni_env,
detach_current_thread);

  return JNI_VERSION_1_4;
}
```

Figura 76: snippet JNI_OnLoad()

Questo metodo è fondamentale, in quanto recupera l'ambiente JNI necessario per effettuare le chiamate dei metodi che interagiscono con Java; in seguito individua la classe contenente la parte della UI utilizzando il metodo FindClass(). Infine registra i metodi nativi attraverso la RegisterNatives().

L'array native_methods descrive ogni metodo registrato. Per ogni metodo fornisce il nome Java, il proprio *type signature* e il puntatore alla funzione C che lo implementa:

```
static JNINativeMethod native_methods[] = {
  {"nativeInit", "()V", (void *) gst_native_init},
  {"nativeFinalize", "()V", (void *) gst_native_finalize},
  {"nativePlay", "()V", (void *) gst_native_play},
  {"nativePause", "()V", (void *) gst_native_pause},
  {"nativeSurfaceInit", "(Ljava/lang/Object;)V", (void *)
      gst_native_surface_init},
  {"nativeSurfaceFinalize", "()V", (void *)
gst_native_surface_finalize},
  {"nativeClassInit", "()Z", (void *) gst_native_class_init}
};
```

Figura 77: array dei metodi nativi

Il Metodo gst_native_init(), nativeInit() in Java

Come detto in precedenza, il metodo nativeInit() viene chiamato alla fine della OnCreate():

```
static void gst_native_init (JNIEnv* env, jobject thiz) {
  CustomData *data = g_new0 (CustomData, 1);
  SET_CUSTOM_DATA (env, thiz, custom_data_field_id, data);
```

Figura 78: snippet gst_native_init()

La prima cosa che fa è allocare memoria per i *CustomData* e passare i puntatori alla classe Java attraverso il SET_CUSTOM_DATA, in modo da essere memorizzati.

```
data->app = (*env)->NewGlobalRef (env, thiz);
```

Un puntatore alla classe principale dell'applicazione (MainActivity) viene memorizzato in modo da poter essere chiamato successivamente.

```
pthread_create (&gst_app_thread, NULL, &app_function, data);
```

Infine un thread viene creato e viene avviata l'esecuzione del metodo app_function.

Il metodo app_function()

```
static void *app_function (void *userdata) {
  JavaVMAttachArgs args;
  GstBus *bus;
  CustomData *data = (CustomData *)userdata;
  GSource *bus_source;
  GError *error = NULL;
  GST_DEBUG ("Creating pipeline in CustomData at %p", data);
  data->context = g_main_context_new ();
  g main context push thread default(data->context);
```

Figura 79: snippet app_function(), creazione contesto

Per prima cosa crea il contesto GLib in modo che tutte le GSource siano nello stesso posto. Un nuovo contesto viene creato con g_main_context_new() e in seguito viene posto come predefinito per il thread, grazie al g_main_context_push_thread_defaut().

```
  data->pipeline = gst_parse_launch("udpsrc port=20000
caps=\"application/x-rtp, media=video, clock-rate=90000,
encoding-name=H264\" ! rtph264depay ! ffdec_h264 !
autovideosink", &error);
  if (error) {
    gchar *message = g_strdup_printf("Impossibile costruire
la pipeline: %s", error->message);
    g_clear_error (&error);
    set_ui_message(message, data);
    g_free (message);
    return NULL;
  }
```

Figura 80: snippet app_funtion(), pipeline

Dopo crea la pipeline (che analizzeremo in seguito) attraverso il gst-parse-launch().

```
GST_DEBUG ("Entering main loop... (CustomData:%p)", data);
  data->main_loop = g_main_loop_new (data->context, FALSE);
  check_initialization_complete (data);
  g_main_loop_run (data->main_loop);
  GST_DEBUG ("Exited main loop");
  g_main_loop_unref (data->main_loop);
  data->main_loop = NULL;
```

Figura 81: snippet app_function(), main loop

Infine, viene creato il main loop e approntato per essere eseguito. Prima di entrare nel main loop, viene chiamato il check_inizialization_complete(). Questo metodo controlla se tutte le condizioni necessarie per rendere il codice nativo “ready” vengono rispettate.

Il metodo gst_native_finalize(), nativeFinalize() in Java

```
static void gst_native_finalize (JNIEnv* env, jobject thiz) {
  CustomData *data = GET_CUSTOM_DATA (env, thiz,
custom_data_field_id);
  if (!data) return;
  GST_DEBUG ("Quitting main loop...");
  g_main_loop_quit (data->main_loop);
  GST_DEBUG ("Waiting for thread to finish...");
  pthread_join (gst_app_thread, NULL);
  GST_DEBUG ("Deleting GlobalRef for app object at %p", data-
>app);
  (*env)->DeleteGlobalRef (env, data->app);
  GST_DEBUG ("Freeing CustomData at %p", data);
  g_free (data);
  SET_CUSTOM_DATA (env, thiz, custom_data_field_id, NULL);
  GST_DEBUG ("Done finalizing");
}
```

Figura 82: snippet gst_native_finalize()

Questo metodo è chiamato in Java nella OnDestroy, quando l'attività deve essere distrutta. La native Finalize permette di istruire il main loop GLib di arrestarsi con g_main_loop_quit(). Questo metodo restituisce immediatamente, e il loop main terminerà appena possibile.

Il file android.mk

Il file Android.mk viene scritto per descrivere le sorgenti al build system, più specificatamente:

- Il file è in realtà una piccola parte di un Makefile GNU che può essere "parsato" una o più volte dal build system.
- La sintassi del file è progettata per permettere di raggruppare le sorgenti in "moduli".
    - Librerie condivise: possono essere installate/copiate nei pacchetti dell'applicazione.
    - Librerie statiche: usate per generare librerie condivise.

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE    := ScenarioDue
LOCAL_SRC_FILES := ScenarioDue.c
LOCAL_SHARED_LIBRARIES := gstreamer_android
LOCAL_LDLIBS := -llog -landroid
include $(BUILD_SHARED_LIBRARY)

include $(CLEAR_VARS)

LOCAL_MODULE    := Incoming
LOCAL_SRC_FILES := Incoming.c
LOCAL_SHARED_LIBRARIES := gstreamer_android
LOCAL_LDLIBS := -llog -landroid
include $(BUILD_SHARED_LIBRARY)

ifndef GSTREAMER_SDK_ROOT
ifndef GSTREAMER_SDK_ROOT_ANDROID

endif
GSTREAMER_SDK_ROOT        := $(GSTREAMER_SDK_ROOT_ANDROID)
endif
GSTREAMER_NDK_BUILD_PATH  := $(GSTREAMER_SDK_ROOT)/share/gst-
android/ndk-build/
include $(GSTREAMER_NDK_BUILD_PATH)/plugins.mk
GSTREAMER_PLUGINS         :=
$(GSTREAMER_PLUGINS_CODECS_RESTRICTED)
$(GSTREAMER_PLUGINS_CORE) $(GSTREAMER_PLUGINS_SYS)
$(GSTREAMER_PLUGINS_EFFECTS) $(GSTREAMER_PLUGINS_NET)
$(GSTREAMER_PLUGINS_CODECS)
GSTREAMER_EXTRA_DEPS      := gstreamer-interfaces-0.10
gstreamer-video-0.10
include $(GSTREAMER_NDK_BUILD_PATH)/gstreamer.mk
```

Figura 83: codice android.mk

Il makefile osservabile sopra consente di aggiungere ad Android il supporto a GStreamer.

5.3.5 La pipeline di GStreamer

L'idea alla base della concezione di GStreamer è la pipeline: questo termine (in inglese significa “tubatura”) è usato per indicare un insieme di elementi collegati uno in serie all'altro. Gli elementi con i quali costituire una pipeline

sono di tre tipi: source (sorgente), filter (filtro), sink (pozzo, ad indicare un elemento finale, senza uscite). Ogni applicazione che utilizzi GStreamer contiene almeno una pipeline.

Analizzeremo ora, la pipeline utilizzata all'interno della nostra applicazione prototipo:

```
udpsrc port=20000 caps=\"application/x-rtp, media=video,
clock-rate=90000, encoding-name=H264\" ! rtph264depay !
ffdec_h264 ! autovideosink
```

Figura 84: pipeline utilizzata

Gli elementi principali sono:

- ***udpsrc***: permette di ricevere i dati sulla rete attraverso UDP.
- ***port***: permette di impostare il numero della porta sulla quale riceviamo il flusso dei media, nel nostro caso è la porta 20000.
- ***caps***: è una stringa che permette di specificare il tipo di media accettabile ed eseguibile, nel nostro caso un flusso RTP, le principali componenti sono:
  - *media*: indica il tipo di media da eseguire, in questo caso ci aspettiamo un media di tipo video.
  - *clock-rate*: indica il clock-rate del video, nel nostro caso 90000.
  - *encoding*: indica la codifica, che deve avere il media in arrivo.
- ***rtph264depay***: estrae il video H.264 a partire dai pacchetti RTP.
- ***ffdec_h264***: indica che tipo di codex utilizzare, nel nostro caso, come ampiamente detto, utilizziamo H.264.
- ***autovideosink***: consente di individuare automaticamente il video sink.

5.3.6 Il File AndroidManifest.xml

L'AndroidManifest.xml [9] è il file che definisce i contenuti e il comportamento della nostra applicazione: all'interno di questo file sono elencate le Activity e i Service dell'applicazione, con i permessi che necessita per funzionare correttamente.

Ogni progetto Android include un AndroidManifest.xml memorizzato nella directory principale del progetto: in questo file XML si possono inserire nodi per ogni elemento (Activity, Service, Content Provider e così via) che compone la nostra applicazione, impostando i permessi per determinare come questi interagiscono l'un l'altro e con le altre applicazioni.

Quando l'applicazione viene installata, il gestore dei pacchetti concede o non concede i privilegi, a seconda di come li abbiamo configurati nell'AndroidManifest.xml.

Analizziamo ora il nostro AndroidManifest, ponendo particolare attenzione sulle parti più importanti.

```
<uses-sdk   android:minSdkVersion="14"
            android:targetSdkVersion="18"/>
```

Figura 85: snippet Manifest, versione OS

Come abbiamo già visto nel capitolo "Distribuzione Android", abbiamo deciso che la nostra applicazione dovrà essere compatibile con le versioni di Android superiori alla 4.0. Questo è reso possibile grazie al codice osservabile sopra che, per mezzo di *android:minSdkVersion* impostato al valore "14", indica che l'applicazione è compatibile solo con versioni di Android che possiedono, almeno, API di livello 14 quindi dalla versione Ice-Cream Sandwich (4.0).

L'opzione *android:targetSdkVersion*, invece, indica il target con il quale viene compilata l'applicazione, in pratica quale versione usiamo per i test.

```
<uses-feature android:glEsVersion="0x00020000"/>
```

Figura 86: snippet Manifest, OpenGL ES

Questa opzione permette di includere all'interno della nostra applicazione il supporto ad OpenGL ES. Nello specifico, il suo utilizzo si è reso necessario per poter visualizzare il video sul dispositivo.

Android:glEsVersion indica la versione supportata dall'applicazione; nel nostro caso, il valore "0x00020000" indica la versione 2.0 di OpenGL ES.

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.WAKE_LOCK"
/>
<uses-permission
android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission
android:name="android.permission.ACCESS_WIFI_STATE"/>
<uses-permission
android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
```

Figura 87: snippet Manifest, permessi

La funzionalità del file AndroidManifest più importante è sicuramente la gestione dei permessi. Nel nostro caso, date le funzionalità richieste dall'applicazione, si è reso necessario includere diversi permessi [10] che analizziamo di seguito:

- ***INTERNET***: questo permesso consente di poter utilizzare la connessione Internet.
- ***WAKE_LOCK***: consente di utilizzare il PowerManager WakeLocks, in modo da tenere attive alcune funzionalità anche a schermo spento o quando il device è in sleep; nel nostro caso lo abbiamo utilizzato per mantenere attivo il Wi-Fi, anche con lo schermo spento.
- ***ACCESS_NETWORK_STATE***: consente di accedere alle informazioni della rete.
- ***ACCESS_WIFI_STATE***: consente di accedere alle informazioni della rete wireless.
- ***RECEIVE_BOOT_COMPLETE***: consente a all'applicazione di ricevere il messaggio di *ACTION_BOOT_COMPLETED,* che viene trasmesso solo dopo che il sistema ha terminato l'avvio.

5.3.7 Android Service, IncomingService

Il servizio è un componente dell'applicazione che può eseguire operazioni in background, ma che non fornisce un'interfaccia grafica.

Abbiamo implementato un servizio chiamato *IncomingService,* in modo che l'applicazione possa rimanere perennemente in ascolto, così da sollevare notifiche al verificarsi di determinati eventi, come ad esempio l'arrivo di una chiamata da posto esterno.

Prima di tutto, abbiamo inizializzato e avviato il servizio all'interno del metodo onCreate() dell'attività principale dell'applicazione.

```
intent_service = new Intent(this, IncomingService.class);
startService(intent_service);
```

Figura 88: avvio del servizio

Una volta avviato, il servizio eseguirà la connessione con il web server F454, attraverso la chiamata del metodo connectionToF454() all'interno della onStart() del servizio, in modo che la connessione rimanga sempre attiva fino alla richiesta di disconnessione, che avviene solo quando l'utente decide di arrestare l'applicazione.

Una particolarità del nostro servizio è che viene inizializzato ed avviato al termine del boot del dispositivo. Questa caratteristica permette di poter così ricevere notifiche dall'applicazione senza doverla necessariamente avviare, ma sarà il servizio ad eseguire una "autostart" ogni qualvolta che viene riacceso il dispositivo.

Questo è possibile per mezzo della classe *ServiceReceiver*, da noi creata, che estendendo la classe *BroadcastReceiver,* permette di ricevere messaggi inviati in broadcast dal Sistema Operativo.

```
if("android.intent.action.BOOT_COMPLETED".equals(intent.getAc
tion())) {
 Intent pushInt = new Intent(context, IncomingService.class);
 context.startService(pushInt);
```

Figura 89: BroadcastReceiver

Il codice sopra mostra un estratto della suddetta classe, in particolare vediamo il codice che stabilisce l'azione da compiere (avviare il Servizio) alla ricezione

del messaggio desiderato. In particolare, attendiamo il messaggio di BOOT_COMPLETED. Oltre a questo, è necessario richiedere attraverso il manifesto di Android, come vedremo in seguito, il permesso di sistema *RECEIVE_BOOT_COMPLETED*.

5.3.8 La gestione delle notifiche

Un'altra funzionalità del Servizio è la gestione delle notifiche. Come detto in precedenza, ci troveremo di fronte a due principali tipi di notifiche: la prima strettamente legata al verificarsi di un evento (la chiamata da posto esterno) mentre l'altra tipologia di notifica è legata all'esecuzione del servizio.

Vediamo prima di tutto come creare una notifica e come gestire le impostazioni principali.

Per creare una notifica si devono utilizzare due classi: NotificationBuilder e NotificationManager.
Si usa un'Istanza di Notification Builder per definire le proprietà della status bar (icona, messaggio esteso e altre impostazioni particolari). Il Notification Manager è un servizio di sistema di Android che serve per eseguire e gestire le Notifiche; è per questo necessario istanziare il Notification Manager.
Il collegamento tra Notification e NotificationManager viene implementato attraverso il metodo getSystemService() e, in seguito, quando occorre notificare qualcosa all'utente, si passa l'oggetto Notification tramite il metodo notify().

Prima di tutto, otteniamo un reference al NotificationManager:

```
mNotificationManager =
NotificationManager)getSystemService(Context.NOTIFICATION_SER
VICE);
```

Figura 90: referenza al NotificationManager

Ora possiamo creare la notifica attraverso la manipolazione del Notification Builder.

```
notificationBuilder.setContentTitle("Nuova Chiamata");
notificationBuilder.setContentText("Rispondi alla chiamata");
notificationBuilder.setTicker("Stanno Citofonando");
notificationBuilder.setSmallIcon(R.drawable.ic_stat_ingoing);
```

Figura 91: impostazioni principali delle notifiche

Con setContentTitle e setContentText impostiamo il titolo e il testo della notifica.

Con setTicker, invece, impostiamo il testo che compare nella barra di stato quando appare la notifica, mentre, con setSmallIcon(), settiamo l'icona che viene mostrata insieme alla notifica.

Attraverso il codice mostrato nella Figura 92, creiamo in pending intent che viene lanciato quando viene premuta la notifica. In pratica, indichiamo l'azione da compiere, che nel nostro caso consiste nell'avviare l'activity.

```
Intent notInt = new Intent(_Contesto, MainActivity.class);
PendingIntent contentIntent =
PendingIntent.getActivity(_Contesto,0, notInt, 0);
```

Figura 92: creare il Pending Intent

Per quanto riguarda, invece, la creazione di una notifica persistente, una notifica che rimane nella barra di stato per tutta la durata dell'esecuzione del servizio e che non può essere rimossa, dobbiamo, sempre attraverso in Notification Builder, modificare il flag nel seguente modo:

```
notificationBuilder.setOngoing(true);
```

Figura 93: notifica persistente

5.4 PROTOTIPO

In questa sezione analizzeremo i due principali casi d'uso dell'applicazione, in modo da mostrarne degli esempi di utilizzo.

Inoltre, includeremo un piccolo manuale utente, che permetta di comprendere al meglio tutte le funzionalità dell'applicazione e conoscere nei dettagli le varie parti che la compongono, con particolare attenzione all'interfaccia grafica.

5.4.1 Scenari d'uso

Analizzeremo ora i due casi d'uso introdotti nel capitolo "Progettazione", osservando cosa avviene sullo smartphone, valutando le azioni che l'utente può compiere e mostrandone l'interfaccia grafica.

Scenario "Chiamata in arrivo"

In questo scenario ipotizziamo di ricevere una chiamata dal posto esterno.

Presupponiamo che una persona, all'esterno della nostra abitazione, abbia effettuato una chiamata al nostro appartamento attraverso il videocitofono.

A questo punto, grazie al servizio in esecuzione in background sul dispositivo, visualizzeremo una notifica nella barra di stato che ci avvisa della chiamata in arrivo, vedi Figura 94.

Figura 94: notifica chiamata in arrivo

Figura 95: centro notifiche aperto

Aprendo il centro notifiche del dispositivo (Figura 95), è possibile vedere tutte le informazioni fornite dalla notifica ricevuta e, cliccando su di essa, si apre la sezione dell'applicazione dedicata alla chiamata, osservabile nella figura subito sotto.

Figura 96: chiamata in arrivo

L'utente visualizzerà una schermata, all'interno della quale visualizzerà tre diverse sezioni:

- La prima, posizionata in alto (Figura 97), è dedicata ai tasti che serviranno per gestire, in futuro, funzionalità aggiuntive, come per esempio l'apertura del cancello.

Figura 97: funzionalità aggiuntive

- Al centro, troviamo la sezione dedicata all'esecuzione del video (vedi Figura 98)

Figura 98: sezione video

- In fondo, troviamo i due tasti dedicati alla videochiamata (vedi Figura 99). Quello verde serve per accettare la chiamata e iniziare lo scambio audio, quello rosso serve per rifiutare la chiamata e chiudere l'esecuzione.

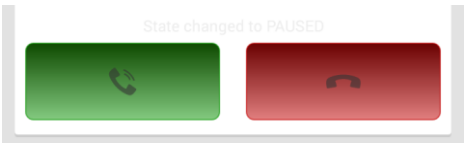


Figura 99: tasti chiamata

Ipotizzando di aver accettato la chiamata, inizierà lo scambio audio tra videocitofono e smartphone fino a quando, scaduti i 30 secondi a disposizione, verrà interrotta mostrando all'utente l'apposito popup (in Figura 100).

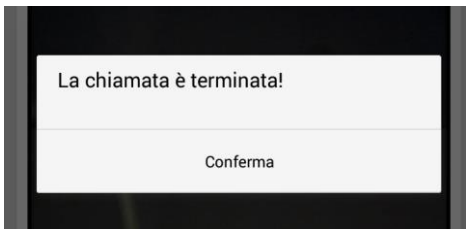


Figura 100: popup di fine chiamata

Cliccando sul tasto “Conferma” verrà chiusa l’applicazione e sarà possibile ricevere una nuova chiamata.

Scenario “Videosorveglianza”

In questo secondo scenario, ipotizziamo di essere un’utente che vuole effettuare una sessione di videosorveglianza.

Al lancio dell’applicazione viene mostrata sullo schermo del nostro dispositivo la schermata principale, vedi Figura 101:

Figura 101: schermata principale dell'applicazione

Da qui, attraverso il tasto “Video Sorveglianza”, è possibile accedere alla funzionalità desiderata e si aprirà una diversa schermata, vedi Figura 102:

Figura 102: schermata Video Sorveglianza

All'apertura della schermata, di Video Sorveglianza l'utente visualizzerà un'interfaccia semplice e di facile comprensione:

- Una sezione dedicata alla visualizzazione del video (inizialmente nero).
- Un tasto che permette di accendere la videocamera del videocitofono.
- Tornare alla schermata principale per mezzo dell'apposito tasto di back posizionato in alto a sinistra, vedi Figura 103.

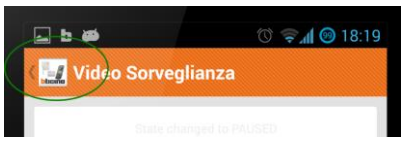


Figura 103: tasto di back

Ipotizziamo ora che l'utente abbia deciso di avviare videocamera. Il video, a causa delle impostazioni della videocamera, viene eseguito per 60 secondi; al termine di questo lasso di tempo l'utente viene avvisato, attraverso un apposito popup, dell'interruzione del video: vedi Figura 104.

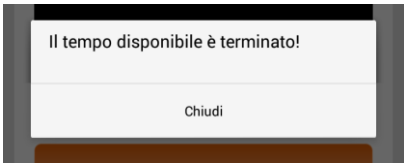


Figura 104: popup di avviso di fine videosorveglianza

Cliccando infine sul tasto "Conferma" l'utente viene reindirizzato alla schermata principale.

5.4.2 Manuale Utente

Scopo di questo manuale non è solo quello di permettere all'utente finale di conoscere tutte le funzionalità dell'applicazione, ma anche quello di descrivere l'applicazione in ogni sua parte, in modo che l'utente possa sfruttare a pieno l'applicazione, in ogni momento e anche se poco esperto e poco avvezzo alla tecnologia.

Illustreremo l'applicazione nei suoi dettagli, a partire dall'installazione, cercando di simulare il più possibile l'esperienza d'uso dell'utente.

Requisiti Minimi

L'unico requisito che richiede l'applicazione riguarda la versione di Android minima richiesta. Infatti l'applicazione non è compatibile con le versioni di Android inferiori alla 4.0.

Installazione

Scaricare l'applicazione

Essendo ancora in fase di beta, l'applicazione non è ancora disponibile sul *Google Play Store*. Per scaricarla, quindi, è necessario andare al seguente link e scaricare l'apk (il file che contiene l'applicazione installabile con un semplice click).

https://github.com/repocode/tesi2013

Schermate installazione e permessi

Dopo aver scaricato l'apk, possiamo proseguire con l'installazione sul device desiderato.

Solitamente, salvo diverse disposizioni dell'utente, i file vengono scaricati nella cartella "Download" del dispositivo; spostatevi nella vostra cartella "Download" e cercate il file "scenarioOpenWebNet.apk", cliccate su di esso e inizierà l'installazione.

Nella prima schermata che apparirà, osservabile in Figura 105.a, vengono mostrati i permessi richiesti dall'applicazione e richiede la conferma da parte dell'utente:

Data conferma, si avvierà l'installazione (solitamente richiede pochi secondi). Al termine della stessa, potrete decidere di avviare l'applicazione o continuare con altre operazioni, Figura 105.b.

Figura 105: installazione

A questo punto l'applicazione è installata sul vostro dispositivo e apparirà la sua icona in mezzo a quelle delle altre applicazioni.

Figura 106: icona dell'applicazione

Descrizione delle aree dell'applicazione

Proseguiamo ora con l'analisi delle sezioni principali dell'applicazione e illustrandone le varie funzionalità.

Figura 107: schermata principale

- ***Area 1***, ***notifiche***: in questa sezione verranno mostrate le notifiche dell'applicazione (le analizzeremo nel dettaglio più avanti nell'apposita sezione).
- ***Area 2***, ***menù***: qui potrai accedere a tutte le funzionalità dell'applicazione, semplicemente cliccando sulla funzione desiderata, vedi sezione dedicata al menù per i dettagli.
- ***Area 3***, ***funzionalità di base e avvisi***: in questa sezione dell'applicazione potrai accedere alle funzioni principali e ricevere alcuni avvisi dall'applicazione.

Primo avvio e servizio in background

Vediamo ora cosa succede all'avvio dell'applicazione. Cliccando sull'icona, si aprirà l'applicazione e apparirà la schermata principale, vedi Figura 107.

A questo punto, l'applicazione mostrerà i primi due avvisi che ci comunicano che l'installazione è avvenuta con successo e tutte le funzionalità sono attive:

Figura 108: avvisi all'avvio

Il primo avviso, a sinistra nella Figura 108, mostra la scritta "IncomingService Avviato" che ci indica che il servizio in background è attivo, mentre il secondo, a destra, ci dice che la connessione con il web server è avvenuta con successo.

Il Menù

Grazie al menù dell'applicazione, Figura 109, è possibile accedere a tutte le funzionalità messe a disposizione da essa. Analizziamole nel dettaglio:

Figura 109: il menù

- ***Esci***: consente di uscire dall'applicazione in maniera sicura e senza causare possibili malfunzionamenti.
- ***Video Sorveglianza***: permette all'utente di accedere alla sezione dell'app adibita al servizio di video sorveglianza.

- ***Informazioni***: permette all'utente di raggiungere la parte dell'applicazione che consente di reperire tutte le informazioni riguardanti l'applicazione.
- ***Chiudi***: consente di uscire dall'applicazione, ma allo stesso tempo interrompe il servizio in background, disabilitando così la possibilità di ricevere notifiche.

Notifiche

Analizziamo ora le notifiche che l'applicazione può inviare. Come detto in precedenza, le notifiche in Android vengono visualizzate nella parte alta dello schermo, Figura 110, e, nel nostro specifico caso, abbiamo due tipi di notifiche che andremo ora a considerare:

Figura 110: area notifiche di Android

- ***Notifica di servizio***: questo tipo di notifica, vedi Figura 111, è legata all'esecuzione del servizio in background dell'applicazione. Essa, infatti, appare fin da subito e sta ad indicare che il servizio è correttamente in esecuzione e l'applicazione è pronta a ricevere le chiamate provenienti dal videocitofono. Osserviamo l'icona e la versione "estesa" della notifica di servizio in Figura 111.

Figura 111: notifica di servizio

- ***Notifica di chiamata in arrivo***: questa notifica, invece, è molto più importante, in quanto consente di accedere alla funzionalità principale dell'applicazione, la ricezione delle videochiamate dal videocitofono esterno. L'icona e la versione "estesa" della notifica le possiamo vedere in Figura 112, mentre nella Figura 113 possiamo osservare il messaggio dato dalla notifica nel momento stesso dell'arrivo della chiamata.

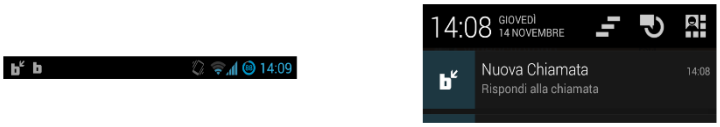


Figura 112: notifica di chiamata in arrivo

È importante osservare che, a causa dei tempi di attivazione del videocitofono, questa notifica si auto elimina dopo 30 secondi e quindi non sarà più possibile accedere ad essa scaduto questo lasso di tempo.

Figura 113: chiamata in arrivo

Cliccando sulla notifica, che viene così eliminata, si viene reindirizzati alla sezione dell'applicazione adibita alla gestione della chiamata in arrivo. Inoltre la notifica può essere eliminata con un semplice slide, vedi Figura 114, ma questo equivale ad ignorare/rifiutare la chiamata.

Figura 114: eliminare la notifica

Video Sorveglianza

Ora analizziamo cosa succede in caso si voglia eseguire una sessione di video sorveglianza. Per accedere a questa funzionalità abbiamo due differenti possibilità:

- Dalla schermata principale dell'applicazione attraverso il tasto arancione "Video Sorveglianza".

- Attraverso il menù, sempre selezionando l'opzione "Video Sorveglianza".

Osserviamo e analizziamo nel dettaglio la schermata dedicata alla video sorveglianza, Figura 115.

Figura 115: schermata di Video Sorveglianza

Abbiamo due sezioni principali, la prima (in alto) consente di visualizzare il video richiesto, mentre la seconda contiene il tasto, Figura 116, che consente di avviare la sessione di video sorveglianza della durata di 60 secondi.

Figura 116: tasto avvio sessione

Durante la visione del video non sarà possibile abbandonare la schermata di video sorveglianza, se infatti si prova a tornare indietro, attraverso gli appositi tasti, verrà mostrato un avviso che vi invita ad "attendere".

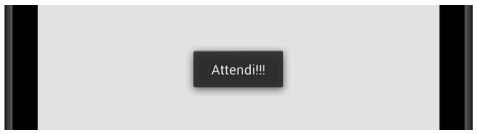


Figura 117: avviso di attesa

Al termine dei 60 secondi, un popup vi avviserà che la sessione è terminata e vi consentirà di tornare alla schermata principale semplicemente cliccando sul tasto “Chiudi”.

Figura 118: popup di fine sessione

Chiamata in arrivo

La sezione più importante da analizzare è quella che andiamo a osservare adesso: la schermata di “Chiamata in arrivo”.

Figura 119: schermata "Chiamata in Arrivo"

Essa è raggiungibile solo cliccando sulla *notifica di chiamata in arrivo* e non direttamente dall'applicazione.

Come è possibile vedere dalla Figura 119, essa è composta di 3 parti principali:

- *Funzioni supplementari*: in questa sezione sono presenti dei tasti che consento di eseguire azioni aggiuntive come, per esempio, aprire cancelli e portoni o accendere le luci del videocitofono.
- *Fruizione video*: la parte della schermata adibita alla riproduzione del video.
- *Tasti di chiamata*: contiene i tasti dedicati alle funzioni di chiamate. Come nei più classici telefoni abbiamo un pulsante verde, che consente di accettare la chiamata e di iniziare lo scambio audio, e un pulsante rosso, che permette di rifiutare la chiamata e tornare alle operazioni che si stavano svolgendo in precedenza.

La durata della chiamata sarà di 30 secondi (limitazione dovuta al web server), terminato questo lasso di tempo, verrà mostrato un popup (Figura 120) che vi avviserà che la chiamata è terminata e vi consentirà di tornare all'operazione che stavate eseguendo semplicemente cliccando sul tasto "Chiudi".

Figura 120: popup di fine chiamata

Servizio in background

Abbiamo parlato spesso di un servizio in background. Questo è avviato dall'applicazione e consente di ricevere notifiche quando una chiamata è in arrivo.

È possibile vedere le proprietà di questo servizio attraverso le impostazioni dello smartphone Android, andando nella sezione applicazioni in esecuzione e cliccando sull'icona della nostra applicazione, Figura 121.

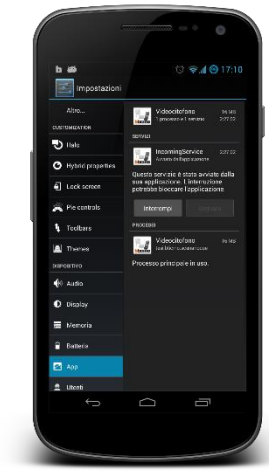


Figura 121: servizio in background

È possibile disattivare questo servizio scegliendo dal menù l'opzione "Chiudi".

Figura 122: opzione "Chiudi" nel menù

Sconsigliamo di compiere questa azione in quanto, chiudendo il servizio, non sarà possibile ricevere notifiche, vedi Figura 123. È possibile, comunque, riattivarlo semplicemente riavviando l'applicazione.

Figura 123: popup di chiusura servizio

Uscire dall'applicazione

Uscire dall'applicazione è un'operazione molto semplice da eseguire. È possibile infatti uscire attraverso il tasto "Home" di Android o in alternativa, scelta da noi consigliata, uscire attraverso l'opzione "Esci" presente, in alto, nel menù dell'applicazione.

Figura 124: opzione "Esci" nel menù

6 STUDI AGGIUNTIVI

6.1 GESTIONE DELL'AUDIO

Come abbiamo visto nel capitolo relativo all'implementazione, il prototipo sviluppato non consente lo scambio di messaggi audio tra il videocitofono e il device Android che esegue l'applicazione realizzata. Infatti, il processo di chiamata termina con la visualizzazione del video e non dà la possibilità di instaurare la comunicazione tra i due utenti.
Questa mancanza è dovuta principalmente alle limitazioni del web server in nostro possesso, che come anticipato, essendo un prototipo, non consente di eseguire alcune funzionalità come, appunto, la ricezione e l'invio di flussi audio.

Lo scambio di un flusso audio in tempo reale (RTP) tra dispositivi così diversi (smartphone e videocitofono), è una procedura complessa, che coinvolge diversi fattori e che proveremo ora ad analizzare.

Figura 125: schema semplificato per lo scambio audio

In questa sezione, non tratteremo di ciò che avviene a livello di interazione tra il web server F454 e il videocitofono, ma ci limiteremo ad analizzare i processi e le azioni necessarie affinché possa avvenire uno scambio audio RTP tra il device Android ed il web server.

6.2 PASSI PER IMPLEMENTARE AUDIO

Analizzeremo ora, con alcuni esempi di codice, i passi principali necessari per implementare uno scambio audio in tempo reale.
Affronteremo questa breve analisi considerando due differenti approcci:

1. Invio e ricezione del flusso audio, utilizzando solamente le API messe a disposizione da Android.
2. Invio e ricezione del flusso audio, utilizzando come supporto alle API di Android, la libreria di gestione dei media GStreamer.

6.2.1 Invio e ricezione audio attraverso le API di Android

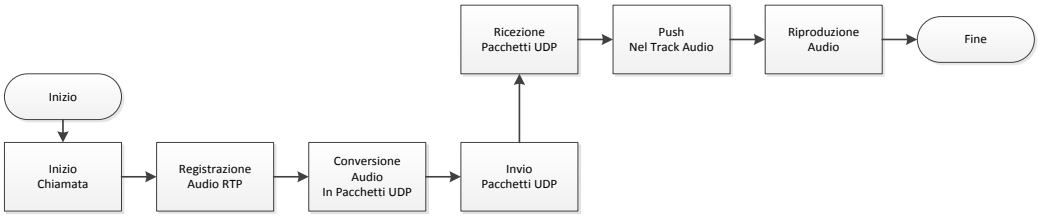


Figura 126: gestione audio con le API di Android

CATTURA DELL'AUDIO E INVIO AL WEB SERVER TRAMITE UDP

Per prima cosa, dobbiamo catturare l'audio da inviare al web server; in pratica, la voce della persona che parla dal cellulare. Questo è possibile grazie alla classe ***AudioRecord*** [11] disponibile in Android.
Questa classe consente di gestire le risorse audio e soprattutto consente di registrare l'audio proveniente dalle periferiche hardware di input presenti sul device, nel nostro caso il microfono.

Vediamo ora le parti principali da implementare in modo da poter, nel momento opportuno, registrare la voce dell'utente che sta utilizzando il device e di inviarla al web server.

Creiamo il nostro elemento AudioRecord, fondamentale per la registrazione:

```
AudioRecord = new AudioRecord (int audioSource,
                  int sampleRate,
                  int channelConfig,
                  int audioFormat,
                  int bufferSize)
```

Figura 127: creazione elemento AudioRecord

Analizziamo i parametri nel dettaglio:

- ***audioSource***: la sorgente dalla quale registreremo. Per mezzo del valore **MediaRecorder.AudioSource.MIC** andremo a settare come fonte audio il microfono.
- ***sampleRate***: il rate di campionamento espresso in Hertz. Stando al sito degli sviluppatori [11], l'unico rate che garantisce compatibilità con tutti i dispositivi è **44100Hz**.
- ***channelConfig***: descrive la configurazione dei canali audio. Anche in questo caso utilizzeremo il valore che garantisce compatibilità con tutti i dispositivi, quindi sarà **AudioFormat.CHANNEL_IN_MONO**.
- ***audioFormat***: il formato in cui codificheremo il nostro file audio. Possiamo scegliere tra 16 e 8 bit; noi useremo, sempre per una questione di compatibilità, la versione a 16 bit quindi: **AudioFormat.*ENCODING_PCM_16BIT***.
- ***bufferSize***: la dimensione totale (in byte) del buffer in cui i dati audio vengono scritti durante la registrazione. Utilizzeremo getMinBufferSize() per determinare questo parametro.

```
AudioRecord audioRecord = new
AudioRecord(MediaRecorder.AudioSource.MIC,
                        44100,
                        AudioFormat.CHANNEL_IN_MONO,
                        AudioFormat.ENCODING_PCM_16BIT,
                        minBufferSize);
```

Figura 128: creazione elemento AudioRecord con parametri

Come detto pocanzi, la creazione del bufferSize avviene per mezzo del codice osservabile in Figura 129, all'interno della quale, i parametri utilizzati sono del tutto simili a quelli descritti sopra.

```
int minBufferSize = AudioRecord.getMinBufferSize(11025,
                  AudioFormat.CHANNEL_CONFIGURATION_MONO,
                  AudioFormat.ENCODING_PCM_16BIT);
```

Figura 129: creazione MinBufferSize

Adesso possiamo iniziare a registrare l'audio attraverso il microfono. Basterà utilizzare il metodo:

```
audioRecord.startRecording();
```

Figura 130: inizio registrazione

A questo punto, il nostro metodo, sta catturando l'audio attraverso il microfono. Vediamo adesso come inviarlo al nostro webserver.
Creiamo delle variabili che ci serviranno per gestire la conversione in pacchetti UDP ed aiutarci nell'invio.

```
int bytes_read = 0;
int bytes_count = 0;
byte[] buffer = new byte[BUF_SIZE];
```

Figura 131: inizializzazione elementi di supporto

E inizializziamo due elementi che ci serviranno per la creazione del socket:

```
InetAddress addr = InetAddress.getByName("192.168.1.35");;
DatagramSocket sock = new DatagramSocket();
```

Figura 132: creazione DatagramSocket

- ***addr***: un elemento ***InetAddres***, al quale assegneremo l'indirizzo IP del web server F454.
- ***sock***: creiamo un elemento ***DatagramSocket*** che crea un Socket che ci consente di inviare e ricevere ***DatagramPacket*** secondo i paradigmi del protocolo UDP.

A questo punto, siamo in grado di convertire e inviare il flusso audio. Un esempio lo possiamo vedere in Figura 133.

```
while(chiamata attiva){
      bytes_read = audio_recorder.read(buf, 0, BUF_SIZE);
      DatagramPacket pack = new DatagramPacket(buffer,
                        bytes_read, addr, AUDIO_PORT);
      sock.send(pack);
      bytes_count += bytes_read;
}
```

Figura 133: invio pacchetti

All'interno del ciclo while, che sarà eseguito finché la chiamata sarà attiva, avverranno le seguenti operazioni:

- Viene aggiornato il numero di bytes letti attraverso il metodo ***read*** della classe *AudioRecord*.
- Viene creato l'elemento DatagramPacket che verrà effettivamente inviato al web server. Come possiamo vedere, viene inizializzato passando come parametri: il buffer creato in precedenza, il numero di bytes letti, l'indirizzo IP e il numero della porta.
- Viene inviato il DatagramPacket.
- Viene aggiornato il conteggio dei byte inviati.

RICEZIONE DELL'AUDIO TRAMITE UDP

Vediamo, brevemente, come implementare il metodo di ricezione del flusso di pacchetti UDP.

```
AudioTrack track = new AudioTrack(AudioManager.STREAM_MUSIC,
                  SAMPLE_RATE,
                  AudioFormat.CHANNEL_CONFIGURATIO_MONO,
                  AudioFormat.ENCODING_PCM_16BIT, BUF_SIZE,
                  AudioTrack.MODE_STREAM);
track.play();
```

Figura 134: creazione elemento AudioTrack

Come prima operazione, dobbiamo inizializzare un elemento AudioTrack [12] che ci consente di gestire ed eseguire le singole risorse audio. I parametri, che andremo a settare, sono del tutto simili a quelli visti per l'inizializzazione dell'elemento di AudioRecord visto precedentemente, ad eccezione di:

- ***AudioManager.STREAM_MUSIC***: consente di eseguire il file audio come se fosse un file musicale.
- ***AudioTrack.MODE_STREAM***: modalità di creazione in cui i dati audio sono trasmessi da Java a livello nativo come la riproduzione audio.

Avviamo, poi, l'esecuzione dell'elemento creato attraverso il metodo *play()*.

Ora siamo pronti a ricevere ed eseguire l'audio:

```
DatagramSocket sock = new DatagramSocket(AUDIO_PORT);
byte[] buf = new byte[BUF_SIZE];

while(chiamata attiva){
      DatagramPacket pack = new DatagramPacket(buf,
BUF_SIZE);
      sock.receive(pack);
      track.write(pack.getData(), 0, pack.getLength());
}
```

Figura 135: ricezione pacchetti

Creaimo l'elemento DatagramSocket, che ci consentirà di ricevere i DatagramPacket e Il buffer di supporto per la ricezione.

All'interno del ciclo *while*, che, analogamente al ciclo visto per l'invio, rimane attivo durante tutta la chiamata, eseguiremo le seguenti operazioni:

- Creiamo il DatagramPacket al quale, grazie al metodo *receive(),* attribuiremo iterativamente l'elemento ricevuto.
- Per mezzo del metodo write() eseguiamo una push sull'elemento AudioTrack permettendo, così, la riproduzione del flusso audio ricevuto.

6.2.2 Invio dell'audio attraverso GStreamer

Vediamo, ora, come inviare il flusso audio attraverso una pipeline GStreamer.

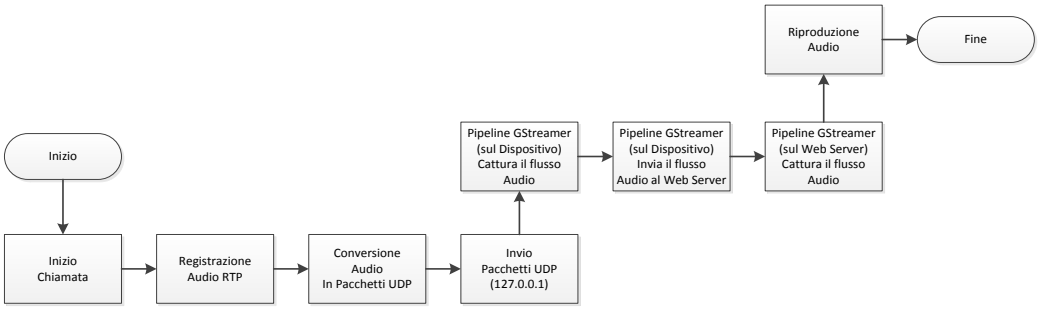


Figura 136: gestione audio tramite GStreamer

Ipotizzando di aver già registrato e convertito l'audio RTP in pacchetti UDP, come visto in precedenza, dobbiamo ora costruire una pipeline che sia capace di ricevere localmente questi pacchetti e, successivamente, inviarli al nostro web server. Analizziamo separatamente le due diverse parti della pipeline:

Parte di ricezione:

```
udpsrc port=PORTA_1 caps="application/x-rtp, media=audio,
payload=8, clock-rate=8000, encoding-name=PCMA" ! queue !
rtpbin. recv_rtp_sink_0
 rtpbin. buffer-mode=RTP_JITTER_BUFFER_MODE_BUFFER !
rtppcmadepay ! alawdec ! alsasink
```

Figura 137: porzione di ricezione della pipeline

Analizziamo gli elementi principali nel dettaglio:

- ***udpsrc***: permette di ricevere i dati sulla rete attraverso il protocollo UDP.
- ***port***: permette di impostare il numero della porta sulla quale riceviamo il flusso audio.
- ***caps***: è una stringa che permette di specificare il tipo di media accettabile ed eseguibile, nel nostro caso un flusso RTP.
    - *media*: indica il tipo di media da eseguire; in questo caso, ci aspettiamo un media di tipo audio.
    - *clock-rate*: indica il clock-rate del video; nel nostro caso 80000.

  - *encoding*: indica la codifica che deve avere il media in arrivo; in questo caso PCMA.
- ***rtpbin***: consente di utilizzare sessioni RTP, creando e utilizzando pacchetti RTCP. Nel nostro caso utilizziamo:
  - *recv_rtp_sink_0*: indichiamo che utilizziamo la sessione rtpbin di invio numero 0.
  - *buffer-mode*: controlliamo l'algoritmo di buffering. Noi useremo *RTP_JITTER_BUFFER_MODE_BUFFER*, algoritmo standard.
- ***rtppcmadepay***: consente di estrarre audio PCMA dai pacchetti RTP.
- ***alawdec***: converte il flusso da 8 a 16bit.

Parte di invio:

```
rtpbin audiotestsrc ! queue ! audioconvert ! alawenc !
rtppcmapay pt=8 ! rtpbin.send_rtp_sink_0
rtpbin.send_rtp_src_0 ! multiudpsink
clients="IP_ADDRESS:PORTA_2" sync=false async=false
```

Figura 138: porzione di invio della pipeline

Analizziamo, anche io questo caso, gli elementi principali nel dettaglio:

- ***audiotestsrc***: crea il segnale audio ad una determinata frequenza.
- ***multiudpsink***: permette di inviare il flusso audio attraverso la rete, tramite il protocollo UDP che, grazie all'attributo ***clients,*** consente di impostare l'indirizzo IP e la porta dell'host di destinazione.
- ***Audioconvert***: consente di convertire l'audio in diversi formati.
- ***rtppcmapay:*** codifica l'audio PCMA in pacchetti RTP.
- ***Sync:*** abilita o disabilita la sincronizzazione con il clock.
- ***Async:*** abilita o disabilita la modifica asincrona dello stato.

6.3 CONNETTIVITÀ ESTERNA

In questo capitolo, verrà effettuata un'analisi e una prima progettazione riguardanti lo sviluppo di una funzione molto importante per il sistema, e per l'applicazione in particolare. Ci focalizzeremo sulla possibilità dell'applicazione di ricevere le chiamate effettuate dal posto esterno anche in ambito extra abitativo, e quindi non collegata alla rete domestica, bensì alla rete di telefonia (e internet 2g, 3g ecc). Uscendo quindi dall'ambito di una rete LAN, subentrano tanti problemi di connessione e persistenza della comunicazione, della banda utilizzabile dal segnale audio (o audio/video); di cui il più importante è senza dubbio la localizzazione del dispositivo, dato che non esiste un IP fisso con cui è possibile raggiungerlo come invece accade negli scenari analizzati precedentemente.

6.3.1 Tecnologie Utilizzabili

Proxy Server SIP

I server SIP sono elementi essenziali in una rete che autorizza i terminali SIP a scambiarsi messaggi, registrare la posizione dell'utente e "muoversi" attraverso la rete. I SIP server infatti, si occupano di instradare i vari messaggi, e assumono anche il ruolo di autenticazione utente e di policy di sicurezza. Vi sono tre tipi standard di SIP server: Proxy, Redirect e Registrar.

Il registrar server accetta richieste di registrazione e memorizza le informazioni in un "location service" per il dominio di cui fa parte e gestisce. Le richieste sono generalmente generate da clients per stabilire o rimuovere un mapping tra il loro indirizzo SIP e l'indirizzo a cui vogliono essere contatti. Il location service, usato come detto per il processo di registrazione (e recupero) delle informazioni, è un database delle location, può risiedere su una macchina remota oppure sulla stessa macchina, e può essere contattato usando qualsiasi protocollo appropriato (per esempio LDAP).

Il redirect server riceve le richieste SIP e risponde con dei messaggi 3xx (redirection), direttamente al client per contattare un alternativo set di indirizzi

IP; questo nuovo gruppo di indirizzi sono contenuti nell'header del messaggio di risposta. I messaggi di re direzione possono indicare uno spostamento temporaneo o permanente, la necessità di usare un Proxy o un servizio alternativo.

Il proxy server SIP instrada le richieste allo User Agent Server e le risposte allo User Agent Client. Entrambe devono essere processate per stabilire l'instradamento ed eventualmente modificare la richiesta prima del forwarding, le risposte seguono lo stesso procedimento ma nell'ordine inverso. Lo standard SIP permette ai proxy servers di validare le richieste, autenticare a loro volta gli utenti, risolvere gli indirizzi, controllare e gestire eventuali loop ed altre azioni relative all'instradamento. Un proxy server è progettato per essere il più trasparente possibile agli user agents. Le specifiche SIP inoltre definiscono due tipi di Proxy SIP: stateful e stateless.

Un proxy stateless è un semplice forwarder di messaggi, quando riceve una richiesta, la processa senza salvare il contesto della transazione, facendo in modo che una volta che il messaggio è stato inviato, il proxy non tiene tracchia della gestione di quel messaggio. Questo tipo di procedimento serve per migliorare le performance e la scabilità, ma ovviamente a qualche conseguenza, un proxy stateless non può associare risposte con richieste inoltrate perché, appunto, non ha alcuna informazione a riguardo, e l'applicazione proxy non può sapere se la transazione ha avuto successo o no. Non può inoltre associare ritrasmissioni di richieste e risposte.

Un proxy stateful, invece, processa le transazioni piuttosto che singoli messaggi, le quali possono essere di tipo server e client. Questo tipo di server è consapevole dello stato delle transizioni e della cronologia dei messaggi; può quindi, per esempio, identificare una ritrasmissione e inoltrare il messaggio. Oltre ad identificare tali ritrasmissioni, può generarle in caso di perdita dei messaggi. Dovendo gestire la cronologia dei messaggi, le performance sono leggermente limitate (soprattutto per quanto riguarda throughput e uso della memoria). Questi server però, sono indispensabili per operazioni quali forking, cioè inviare più volte lo stesso messaggio, accounting, e supporto al NAT traversal.

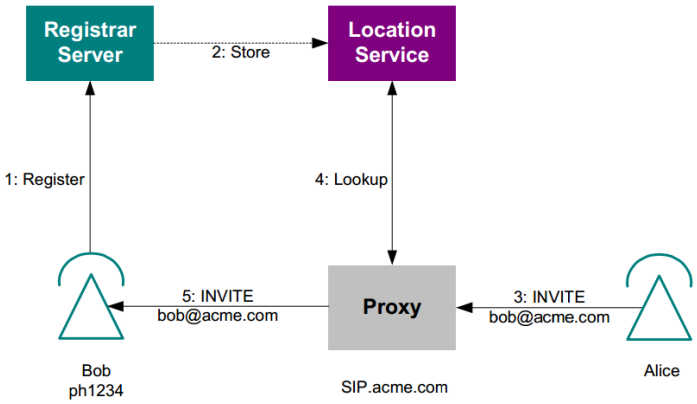


Figura 139: esempio generale server SIP

Prima di effettuare le richieste di routing, un server SIP (proxy o redirect) ha bisogno di validare il messaggio. Innanzitutto deve passare il controllo di sintassi, deve essere ben formattato, e la URI deve avere uno schema ben preciso. Il client può indicare qualche estensione SIP nel campo proxy, e il server deve avere il supporto corrispondente per processare il messaggio in maniera corretta. Ultima cosa è l'autenticazione, se il messaggio non contiene delle credenziali, questo processo ovviamente fallisce. L'autenticazione SIP può avvenire anche in un sistema multi-livello, coinvolgendo differenti server proxy.

Effettuata l'autenticazione, occorre determinare la destinazione alla quale il messaggio deve essere inoltrato; questo processo può avvenire in due maniere: determinando il target-set oppure tramite una risoluzione DNS.

NAT, STUN/TURN

NAT (Network Address Translation) è una tecnologia usata comunemente da firewall e router per permettere a più dispositivi in una LAN (e quindi con IP privati) di condividere un singolo indirizzo IP pubblico. In questo modo, possono essere indirizzati e raggiunti attraverso internet, dato che un indirizzo IP privato può essere acceduto solo all'interno della LAN di riferimento. Per fare ciò, occorre quindi una traduzione tra indirizzo pubblico e privato nel punto in cui la rete LAN si connette ad Internet. Il passaggio del traffico attraverso il NAT è chiamato "NAT Traversal".

Esiste un problema generale utilizzando NAT e VoIP. Per esempio, utilizzando una connessione SIP, durante la registrazione, viene passato l'indirizzo IP del dispositivo, ma questo è un indirizzo privato (della LAN a cui è connesso), quindi il Service Provider non sarà in grado di inviare messaggi SIP al telefono. Inoltre, la connessione per lo stream audio, è generalmente su una porta differente da quella utilizzata per iniziare e terminare una chiamata; il router NAT permette le connessioni in uscita, ma blocca quelle in entrata, questo significa che potrebbe essere inviato il messaggio per aprire un media stream, ma il device in remoto non sarà in grado di aprire la connessione a causa del blocco NAT.

STUN (Simple Traversal of UDP through NATs, ridefinito come Session Traversal Utilities for NAT) è un protocollo di supporto ai dispositivi nascosti da un firewall o router NAT. Permette ad un dispositivo di essere raggiunto tramite il suo indirizzo IP pubblico (ed eventualmente mascherato da un servizio NAT), opera sulle porte TCP e UDP e si serve di record DNS per trovare i server STUN associati ad un dominio.
Un client STUN è un'entità che genera le richieste, può operare su un end system oppure su un elemento di rete come può essere un server. Un Server STUN è l'entità che riceve le richieste e invia le risposte. Permette ai vari client di recuperare il loro indirizzo pubblico, il tipo di NAT e la porta Internet ad esso associato collegata alla porta locale; queste informazioni poi possono essere utilizzare per una comunicazione UDP tra il client e il povider VoIP.
Il protocollo è stato riscritto con il RFC 5389 e chiamato STUN-bis. Dopo quest'ultima revisione, non è più una soluzione stand alone, ma definisce una serie di strumenti e meccanismi per effettuare il NAT traversal. Alcuni attributi come per esempio RESPONSE-ADDRESS, SOURCE-ADDRESS, CHANGED-ADDRESS, sono stati deprecati.

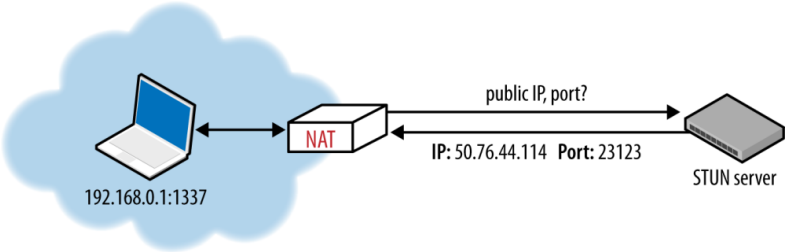


Figura 140: funzionamento STUN server

Un'estensione del protocollo STUN-bis è il protocollo **TURN**, il quale si serve di un relay server, e dato che è molto costoso (in termini di banda) deve essere fornito dal provider. Il protocollo **ICE** fornisce un meccanismo per entrambi gli endpoints per scoprire il percorso ottimale da usare per il traffico media. TURN utilizza il protocollo binario STUN-bis, ma definisce nuove richieste, nuovi attributi e nuove funzionalità, e a differenza di quest'ultimo, è un protocollo stateful. Questo significa che definisce le sessioni, e ognuna ha il suo ciclo di vita e il proprio contesto.

Google Cloud Message Service

Per inviare semplici dati (informazioni) dal proprio server ad un'applicazione Android, Google mette a disposizione un servizio gratuito che permette di inviare un messaggio (fino a pochi KB per i payload) all'applicazione per notificarle che ci sono nuovi dati da poter recuperare dal server, oppure direttamente per un servizio di messaggistica istantanea. Questo servizio prende il nome di Google Cloud Message [13] e va a sostituire il precendente Android Cloud to Device Messaging (C2DM). Utilizzando il GCM Cloud Connection Server (CCS) si possono ricevere messaggi dal device dell'utente, e una cosa molto importante, l'applicazione non ha bisogno di essere in funzione per ricevere il messaggio: il sistema "sveglia" l'applicativo attraverso il broadcast Internet. Per i devices precedenti alla versione 3.0 di Android è necessario impostare gli account Google, mentre non è richiesto per le versioni dalla Jelly Bean in poi.

Ora vediamo brevemente come funziona il meccanismo. IL GCM connection server prende i messaggi dal server (sviluppato esternamente) e invia i messaggi all'applicazione Android (client app) del dispositivo precedentemente abilitata dal GCM. Al giorno d'oggi sono fornite connessioni server per http e XMPP. Il server esterno mette in coda e memorizza i messaggi, inviandoli quando il dispositivo è online. L'applicazione client riceve i messaggi, e per farlo deve essere registrata con il GCM e ricevere un registration ID, utilizzando XMPP come connection server, inoltre, ha la possibilità di inviare messaggi di ritorno al server in upstream.

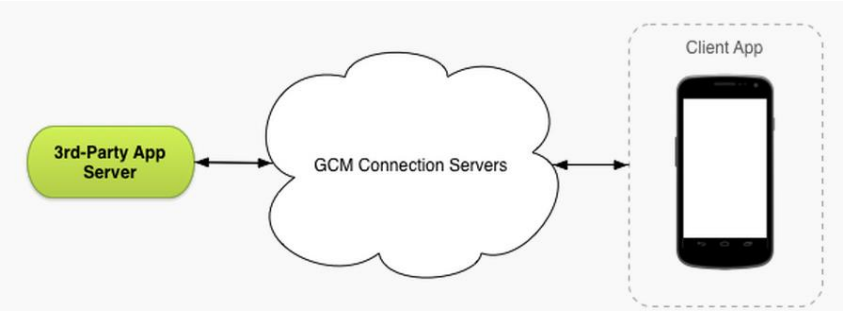


Figura 141: Comunicazione Google Cloud Messaging

PJSIP Stack

Per quanto riguarda un'eventuale implementazione in un'applicazione client, abbiamo già trattato e analizzato la struttura e le funzioni dello stack SIP PJSIP. Per il protocollo STUN (e relativi TURN e ICE) esso offre il supporto per la gestione della comunicazione e del problema del NAT traversal attraverso la libreria PJNATH [14]. Esiste inoltre all'interno dello stack, un frame work per l'autenticazione sia da parte del client che lato server. Questa autenticazione supporta il digest authentication HTTP, ma può essere esteso aggiungendo altri schemi. Le API per l'autenticazione sono dichiarate in <pjsip/sip_auth.h>. La struttura base è pjsip_cred_info, dove sono descritte tutte le credenziali (anche multiple) dell'utente client. Un esempio di autenticazione tramite le funzioni messe a disposizione dal frame work potrebbe essere il seguente:

```
pjsip_auth_client_session auth_sess;
void init_auth(pj_pool_t *session_pool){
pjsip_cred_info cred;
pj_status_t status;
cred.realm = pj_str("sip.example.com");
cred.scheme = pj_str("digest");
cred.username = pj_str("nomeutente");
cred.data_type = PJSIP_CRED_DATA_PLAIN_PASSWD;
cred.data = pj_str("secretpassword");
status = pjsip_auth_client_init( &auth_sess,session_pool, 0);
status = pjsip_auth_set_credentials( &auth_sess, 1, &cred );}
void send_request(pjsip_tx_data *tdata){
pj_status_t status;
status = pjsip_auth_client_init_req( &auth_sess, tdata );
status = pjsip_endpt_send_request( endpt, tdata, -1, NULL,
&on_complete);}
static void on_complete( void *token, pjsip_event *event ){
int code;
pj_assert(event->type == PJSIP_EVENT_TSX_STATE);
code = event->body.tsx_state.tsx->status_code;
if (code == 401 || code == 407) {
pj_status_t status;
pjsip_tx_data *new_request;
status = pjsip_auth_client_reinit_req( &auth_sess, endpt,
event->body.tsx_state.src.rdata,
tsx->last_tx,
&new_request);
if (status == PJ_SUCCESS)
status=pjsip_endpt_send_request(endpt,new_request,-1,NULL,
&on_complete);
Else PJ LOG(3,("app","Authentication failed!!!"));
```

Figura 142: autenticazione PjSIP

Come anticipato, la libreria PJNATH fornisce quelle strutture e funzioni per gestire il problema del NAT traversal. Vi sono API per inviare e ricevere pacchetti UDP, risoluzioni NDS SRV per il server STUN, e mantenimento della sessione STUN. Per il protocollo TURN sono fornite le funzioni per scoprire il server corrispondente, per l'autenticazione (usando il metodo long-term di STUN), la gestione delle allocazione (compreso l'aggiornamento), la creazione dei permessi e la ricezione dei dati. Vi sono due moduli differenti riguardanti TURN: il client session, il quale è un oggetto indipendente dal trasporto, contiene il core logic per la gestione della sessione TURN (credenziali,

allocazione, server, permessi), e il client transport, che fornisce un wrapper per il client session, per gestire tutte quelle operazioni utili al protocollo TURN.

6.3.2 Possibile Progettazione

Per realizzare il sistema completo di chiamata verso un dispositivo mobile, anche quando questo non è connesso alla rete LAN, occorre pensare, progettare e realizzare un sistema più articolato e ricco di componenti. Vengono, infatti, introdotte problematiche diverse che riguardano la connessione e il raggiungimento del dispositivo che, è possibile aggirare, solo grazie ad alcune tecniche e infrastrutture particolari.

Il problema iniziale è la gestione dell'inoltro della chiamata. Il web server, aggiornato e con la funzione di chiamata diretta attivata, deve sapere se il dispositivo è collegato alla rete domestica e, quindi, poterlo contattare direttamente tramite indirizzo IP e numero di porta con un comando OpenWebNet; oppure se questo è scollegato, poterlo raggiungere solo mediante la rete Internet (qualora ovviamente il dispositivo fosse acceso e collegato in rete).
Dato che il web server ha già un programma di configurazione, una soluzione potrebbe essere aggiungere una classe di dispositivi collegabili alla rete MyHome, così come accade per telecamere, posti esterni, posti interni e qualsiasi altro dispositivo dotato di un'interfaccia di rete (e non solo, ma in questo ambito ci stiamo occupando di indirizzamenti IP). In questo modo, al momento della scelta di utilizzare un determinato smartphone, per la ricezione di chiamate videocitofoniche viene configurato il dispositivo all'interno della rete, assegnandogli quindi un IP statico che lo identifichi all'interno del sistema domestico. Così facendo, alla ricezione di una chiamata, il web server non dovrà far altro che effettuare un controllo su tale indirizzo IP, e nel caso fosse individuato attivo all'interno della rete, inviargli direttamente i messaggi e lo stream multimediale. Nel caso in cui, invece, non vi sia risposta da parte dell'IP registrato, il web server si dovrebbe occupare di contattare un dispositivo esterno, che faccia da tramite, da intermediario, per raggiungere il dispositivo attraverso Internet.

Avendo analizzato il protocollo SIP, ed essendo già sviluppato il centralino con tale protocollo, una valida idea potrebbe essere l'utilizzo di un Proxy Server SIP per la registrazione e, successiva autenticazione, dei vari dispositivi (e utenti) coinvolti nel sistema.
Per esempio, l'applicazione sul dispositivo potrebbe essere configurata come un normale client VoIP, inserendo delle credenziali che appartengono al singolo utente, e così facendo si comunica al Proxy Server la propria posizione nella rete Internet. Analogamente, il web server, quando volesse comunicare con un account utente registrato nella propria rubrica (che potrebbe risiedere sia al suo interno, oppure sul portale MyHome), non dovrebbe fare altro che collegarsi a tale SIP server utilizzando come indirizzo quello dell'utente. Compito del Server è, in seguito alla ricezione della richiesta di comunicazione, associare tale indirizzo utente all'indirizzo IP pubblico ottenuto dal dispositivo connettendosi alla rete Internet.

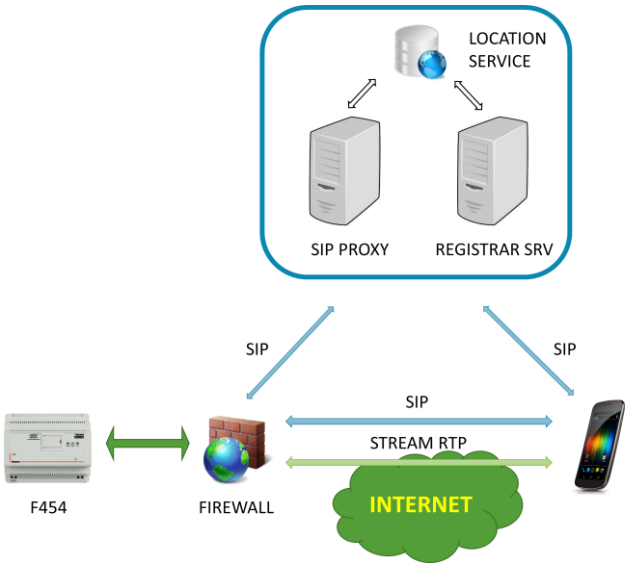


Figura 143: Design Schema del Sistema

Grazie al Proxy Server, quindi, è possibile instradare correttamente i messaggi SIP. Può però vedersi necessaria l'introduzione di un altro sistema: lo STUN server. In diverse reti infatti, può essere presente un NAT (visto precedentemente), indi per cui non può essere passato direttamente il proprio indirizzo IP privato. Tramite i servizi messi a disposizione dal server STUN, è possibile ricavare il proprio indirizzo IP pubblico e la morfologia dell'eventuale

maschera NAT che nasconde il dispositivo, in questo caso il web server, all'ambiente esterno. Questo server potrebbe non essere necessario se si volesse utilizzare il portale MyHome, il quale potrebbe includere la funzionalità di individuazione dell'abitazione. Un ultimo problema da affrontare è la comunicazione in entrata, la quale può, però, risultare bloccata e, quindi, impedire una corretta fruizione dei contenuti multimediali. Per gestire questo inconveniente, si utilizza il protocollo STUN (con TURN e ICE) grazie al quale è possibile mantenere aperta una connessione proveniente dall'interno, nel nostro caso, ancora una volta proveniente dal web server.

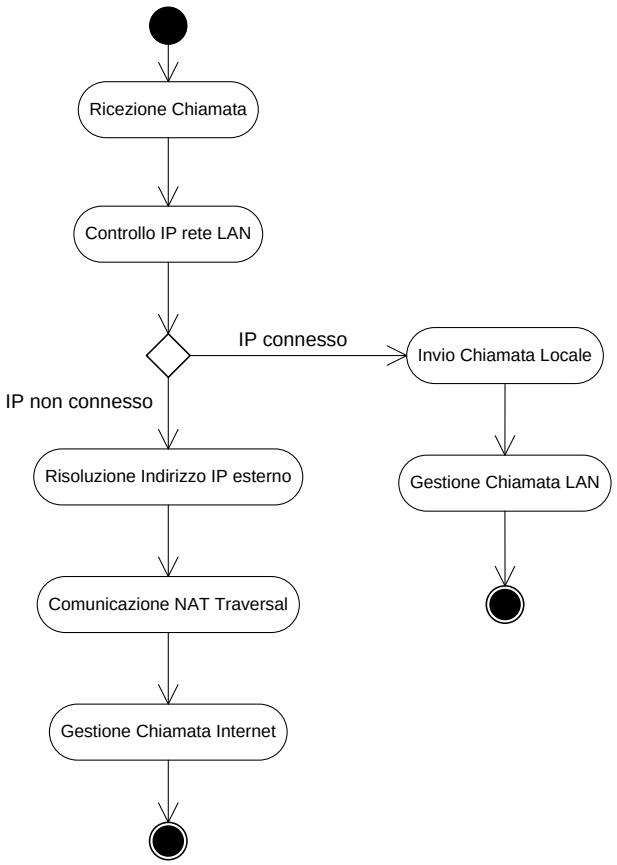


Figura 144: Diagramma Attività Controllo IP Chiamata

Una soluzione alternativa per la risoluzione dell'indirizzo del device mobile potrebbe essere rappresentata dall'utilizzo del servizio GCM di Google. Pensando allo scenario "prototipato", non vi è alcuna gestione della comunicazione tramite SIP. Benché si possa utilizzare SIP sul web server e

utilizzare comunque un proxy server con autenticazione per recuperare l'IP del dispositivo mobile, il servizio di Google permette, anche, un'integrazione maggiore all'interno di un'applicazione Android.
Registrando un account Google infatti, è possibile inviare da un server dei messaggi ad un dispositivo associato, ovviamente, qualora questo fosse collegato alla rete Internet.
Il server in questione potrebbe essere lo stesso del portale MyHome, il quale conosce già l'indirizzo a cui accedere per contattare il sistema domotico nell'abitazione. Utilizzando una comunicazione XMPP inoltre, è possibile ricevere risposte dall'applicazione in direzione del server. Con questa architettura, quindi, si può inoltrare una notifica di chiamata direttamente utilizzando il servizio Google Cloud Messaging e fare in modo che, quando l'utente accetta la chiamata, l'applicazione invii il proprio indirizzo IP al server. Una volta raccolti i dati dei due endpoint, sarà possibile effettuare una chiamata e uno scambio di dati RTP. Anche in questa situazione, ovviamente, sarà necessario avvalersi del protocollo STUN (TURN e ICE) per i messaggi in entrata al sistema domestico, e per mantenere aperta la connessione iniziata dall'interno con la chiamata del web server F454 al server esterno.

7 CONCLUSIONI

Lo scopo principale del nostro lavoro era quello di analizzare e progettare un sistema in grado di interfacciarsi con un impianto domotico Bticino di videocitofonia, attraverso un dispositivo mobile.

In pratica, abbiamo sviluppato un sistema capace di poter rispondere al citofono in qualsiasi posto dell'abitazione e, non più, solo in prossimità dello stesso.

Abbiamo analizzato due differenti scenari:

- Il primo, strettamente legato al protocollo SIP, sul quale ci siamo limitati, essendo SIP uno standard comune, a compiere un'analisi sulle tecnologie e sulle librerie esistenti. Tra tutte, abbiamo scelto quella che ci è sembrata più adatta ai nostri scopi, PJSIP, e analizzata nel dettaglio. Infatti, il flusso della connessione SIP da realizzare per connettersi al centralino Bticino non segue al 100% lo standard, ma presenta delle particolarità che hanno reso la scelta relativamente complessa.
- La particolarità del secondo scenario, invece, riguarda l'utilizzo del protocollo OpenWebNet. Questo, realizzato da Bticino, consente di comunicare con gli elementi facenti parte del sistema MyHome attraverso dei comandi alfanumerici, che abbiamo dovuto analizzare prima di poterli utilizzare.

Abbiamo, in seguito, sviluppato un prototipo dell'applicazione, seguendo le linee guida dello scenario che utilizza il protocollo OpenWebNet. Questo perché ci è sembrata la scelta più innovativa e più interessante, rispetto a quella di sviluppare un'applicazione SIP. Infatti, è già possibile trovare moltissime applicazioni di domotica che utilizzano il protocollo SIP, mentre sono davvero poche quelle che utilizzano il protocollo di Bticino.

Nel prototipo, un'applicazione Android compatibile con versioni superiori alla 4.0, abbiamo incluso quelle che, secondo noi, erano alcune delle funzionalità maggiormente innovative, oltre ad alcune espressamente richieste da Bticino. Per esempio:

- Connessione con il web server F454.
- Invio e ricezione di comandi OpenWebNet, per mezzo di una “Monitor”.
- Scambio di flussi video, grazie a GStreamer.
- Esecuzione di un processo persistente in background in ascolto di eventi provenienti dal web server.
- Gestione delle notifiche.

Essendo un prototipo, ovviamente, non sono presenti tutte le funzionalità che la renderebbero, altrimenti, un’applicazione pronta ad essere commercializzata. Infatti, in futuro, oltre a ottimizzazioni legate al codice e al consumo delle risorse, si potrebbero implementare funzionalità aggiuntive come: la gestione dell’audio, l’implementazione di un maggior numero di notifiche (per esempio per le chiamate perse), l’esecuzione concorrenziale su più device, il controllo dello stato della rete e della connessione.

Volevamo, infine, evidenziare il fatto che lavorare con una grande azienda come Bticino, ci sia servito da stimolo e ci abbia insegnato molte cose e permesso di poter approcciare al lavoro in maniera molto professionale e puntuale.

8 BIBLIOGRAFIA

[1] S. Liang, The Java Native Interface. Programmer's Guide and Specification, Addison-Wesley, 1999.

[2] Google, Android Open Source Project, «Building for devices» 2 Agosto 2012. URL: http://source.android.com/source/building-devices.html.

[3] L. Whitney, «Samsung Galaxy Tab 8.9 to ship with Honeycomb 3.1» 26 Settembre 2012. URL: http://reviews.cnet.com/8301-197367-20111881-251.html.

[4] GalaxyTabs.com, «Explanation and Examination of TouchWiz UX aka TouchWiz 4.0» 23 Marzo 2011. URL: http://www.galaxytabs.com/2011/03/explanation-and-examination-of-touchwiz-ux-aka-touchwiz-4-0/.

[5] Google, «Dashboard, distribuzioni Android, terzo trimestre 2013» Ottobre 2013. URL: http://developer.android.com/about/dashboards/index.html.

[6] Google Inc., «Android NDK, Google Developer» URL: http://developer.android.com/tools/sdk/ndk/index.html.

[7] G. Developer, «GStreamer Good Plugins 1.0 Plugins Reference Manual» Luglio 2013. URL: http://gstreamer.freedesktop.org/data/doc/gstreamer/head/gst-plugins-good-plugins/html/index.html.

[8] My Open Staff, Indirizzi IP Autorizzati, 2011.

[9] Google Developers, «Android Manifest, Introduction» Ottobre 2013. URL: http://developer.android.com/guide/topics/manifest/manifest-intro.html.

[10] Google Developers, «Android Manifest, Permission» Ottobre 2013. URL:

http://developer.android.com/reference/android/Manifest.permission.html.

[11] Google Developers, «AudioRecord» 15 Novembre 2013. URL: http://developer.android.com/reference/android/media/AudioRecord.html.

[12] Google Developers, «AudioTrack» 15 Novembre 2013. URL: http://developer.android.com/reference/android/media/AudioTrack.html.

[13] G. Developers, «Google Cloud Messaging» 15 Novembre 2013. URL: http://developer.android.com/google/gcm/gcm.html.

[14] Teluu Inc, «PJNATH - Open Source ICE, STUN, and TURN Library» URL: http://www.pjsip.org/pjnath/docs/html/index.htm.

[15] V. Silva, Pro Android Games, Apress, 2009.

[16] Bticino, OpenWebNet Java/Android Library CookBook, 2011.

[17] J.L.Ryan, Electronic And Communication Engineering Journal, Home Automation, 2009.

[18] M. Carli, Android 4. Guida per lo sviluppatore, APOGEO, 2013.

[19] V. Dimitar e F. Ivailo, Service Gateway Architecture For A Smart Home IEEE, 2002.

[20] A. Cole e B. Tran, Home Automation To Promote Independent Living In Elderly Populations, 2002.

[21] Casa Tecnica: Rivista di domotica e soluzioni per una casa migliore, BSB Editori s.r.l..

[22] Echelon, «Smart Energy Starts Here» URL: www.echelon.com.

[23] Bluetooth, «Bluetooth Technology Website» URL: www.bluetooth.com.

[24] EMS, «CANopen Protocol,» URL: www.canopen.com.

[25] «JINI,» URL: www.jini.org.

[26] «X-10,» URL: www.x-10.org.

[27] KNX Association, «KNX Association Official WebSite» URL: http://www.knx.org/it/.

[28] «A reference guide to all things VOIP» 11 Novembre 2013. URL: http://www.voip-info.org/.