UNIVERSITÀ DEGLI STUDI DI PARMA
FACOLTÀ DI INGEGNERIA
CORSO DI LAUREA SPECIALISTICA IN
INGEGNERIA DELLE TELECOMUNICAZIONI

Progettazione e realizzazione di un applicativo Push-To-Talk con segnalazione SIP per terminali Symbian

Relatore:
Chiar.mo Prof. **Luca Veltri**

Correlatore:
Ing. **Andrea Barontini**

Tesi di laurea di
Alessandro Gilardi

Anno Accademico 2004/05

Sommario

1. Introduzione

Già da diversi anni nel settore delle telecomunicazioni si assiste, in parallelo, alla diffusione capillare della rete Internet e ad un enorme sviluppo della telefonia mobile, sia per quanto riguarda il numero di utenti sia in riferimento ai sistemi e alle reti di telecomunicazioni. Solo da poco tempo, tuttavia, gli operatori del settore si stanno avviando con decisione verso uno scenario caratterizzato dalla convergenza tra il servizio di trasporto dati mediante protocollo IP e il tradizionale servizio telefonico su reti dedicate, sia fisse che mobili. Lo studio di soluzioni che permettano di integrare le reti IP con la rete telefonica classica, sia a livello di infrastruttura che dal punto di vista dei servizi forniti agli utenti finali, ha condotto alla definizione di ciò che viene abitualmente indicato come VoIP, ovvero di sistemi che consentano il trasporto di comunicazioni in tempo reale su reti a pacchetto basate su IP, e in primo luogo su Internet.

Le motivazioni che stanno spingendo gli utenti alla scelta di utilizzare Internet anche per le normali chiamate telefoniche sono principalmente di natura economica: i prezzi delle telefonate internazionali sono notevoli, mentre in Internet la tariffazione non è legata alla distanza. Oltre a ciò, l'uso di sistemi VoIP consente agli utenti di spostarsi tra diversi terminali mantenendo lo stesso identificativo (analogamente a quanto avviene per la posta elettronica), di crittografare facilmente i dati scambiati e di usufruire di servizi (quali ad esempio videochiamate e conferenze) che si affiancano alla classica chiamata vocale tra due interlocutori.

Dal punto di vista dei gestori, invece, il trasporto del traffico voce su reti IP consente di agevolare la gestione dell'infrastruttura (una singola rete che trasporta diversi tipi di traffico può essere controllata e sviluppata con maggiore semplicità rispetto a varie reti distinte), di utilizzare le reti con maggiore efficienza (evitando, ad esempio, di impegnare banda e risorse durante le fasi di silenzio di una comunicazione vocale) e di introdurre facilmente servizi innovativi (che possono consentire di conquistare nuove categorie di clienti e di incrementare l'*average revenue per user*). D'altra parte, i prezzi degli apparati di rete necessari a gestire il traffico voce su IP sono in costante

diminuzione, e la qualità del servizio offerto, benché non sia ancora giunta al livello garantito dalle reti dedicate, sta aumentando progressivamente.

Un caso concreto in cui la convergenza descritta viene realizzata su vasta scala è costituito dalla Release 5 del sistema UMTS: all'interno delle relative specifiche, infatti, è previsto che la dorsale della rete, utilizzata per trasferire sia le informazioni di segnalazione che quelle di traffico, sia interamente basata sul protocollo IP. Per garantire un adeguato supporto ai servizi multimediali in tempo reale anche su reti a pacchetto basate su IP, all'interno di queste viene introdotto un nuovo dominio di rete detto IMS (*IP Multimedia Subsystem*), che comprende tutti gli elementi di rete utili per la fornitura di tali servizi. Mediante questo sistema gli operatori mobili, oltre a godere dei vantaggi descritti in precedenza, possono fornire agli utenti non solo l'accesso alla rete ma anche servizi multimediali a valore aggiunto precedentemente offerti da service provider indipendenti.

Un esempio dei servizi in questione è rappresentato dal cosiddetto *Push-to-talk over Cellular*, grazie al quale due o più utenti, mediante appositi terminali UMTS, possono comunicare tra loro in modalità half-duplex, secondo lo schema tipico dei comuni walkie-talkie. Questo schema, descritto in maggiore dettaglio nel seguito, prevede che in un determinato istante ognuno degli interlocutori possa solamente parlare oppure ascoltare, che per parlare l'utente debba premere un tasto e tenerlo premuto per tutta la durata del messaggio, e che le parole dell'utente attivo possano essere ricevute quasi istantaneamente da tutti gli altri. Un'implementazione di tale servizio è stata sviluppata all'interno del progetto *UniPR-Ptt*, che è oggetto della presente tesi.

1.1. Obiettivo e struttura della tesi

L'obiettivo principale del presente lavoro di tesi consiste nella progettazione e nella realizzazione, in linguaggio C++, di un particolare applicativo VoIP, chiamato *UniPR-Ptt*, che permette di effettuare una comunicazione in modalità push-to-talk tra due smartphone dotati di sistema operativo Symbian. Lo scambio dei flussi di dati audio tra i due terminali, collegati ad Internet mediante la rete UMTS, avviene tramite pacchetti RTP trasportati mediante il protocollo UDP. L'instaurazione e l'abbattimento della comunicazione sono invece realizzati mediante il protocollo SIP.

Per raggiungere l'obiettivo indicato è necessario innanzi tutto prendere confidenza con le proprietà fondamentali del sistema operativo Symbian e con le peculiarità del linguaggio C++ quando viene utilizzato per sviluppare applicativi destinati a tale sistema operativo; successivamente occorre esaminare le caratteristiche dei protocolli che dovranno essere utilizzati, in modo da comprenderne le potenzialità e le modalità d'uso. In seguito, dopo aver definito con esattezza le specifiche del sistema (particolarmente per quanto riguarda il modello di push-to-talk da implementare), è possibile procedere allo sviluppo del progetto e alla successiva fase di test.

La struttura del presente documento riflette la scansione temporale descritta: per prima cosa vengono descritte le caratteristiche fondamentali del sistema operativo Symbian e vengono esaminate le particolarità dello stile di programmazione che deve essere seguito quando si intende scrivere codice in linguaggio C++ destinato a terminali basati su tale sistema operativo. In seguito vengono presentati, sempre a livello teorico, i protocolli SIP e SDP, che saranno utilizzati all'interno del progetto per l'instaurazione della comunicazione, e viene esaminato il plug-in fornito da Nokia per lo sviluppo e l'utilizzo di applicativi basati su tali protocolli. Si passa quindi all'analisi delle caratteristiche principali del protocollo RTP, che sarà utilizzato per il trasporto dei dati audio, e del protocollo di controllo ad esso associato, RTCP. Per concludere la premessa teorica al progetto realizzato, vengono poi illustrati i concetti di *push-to-talk* e *push-to-talk over cellular*, presentando alcuni dei possibili modelli di push-to-talk che sono stati individuati, esaminando vantaggi e svantaggi di ciascuno e motivando la scelta di quello che è stato implementato all'interno del progetto *UniPR-Ptt*.

Successivamente si passa alla descrizione del progetto in questione e del relativo sviluppo. In particolare, vengono dapprima riassunte le funzionalità e le modalità d'uso dell'applicativo realizzato; viene quindi descritto l'esempio (fornito da Nokia) dal quale si è partiti per lo sviluppo; in seguito vengono elencate, in ordine cronologico, le modifiche mediante le quali è stato possibile passare da tale esempio all'applicativo *UniPR-Ptt*; infine viene illustrato nei dettagli il codice che è stato scritto per realizzare questo applicativo, esaminando le varie parti che lo compongono e discutendo l'implementazione delle funzionalità principali.

In seguito vengono descritti i test a cui è stato sottoposto l'applicativo realizzato per verificarne la rispondenza agli obiettivi che erano stati prefissati e il corretto funzionamento nelle varie configurazioni in cui può venire utilizzato. Basandosi sui risultati delle prove svolte e sull'analisi delle funzioni che potrebbero essere realizzate

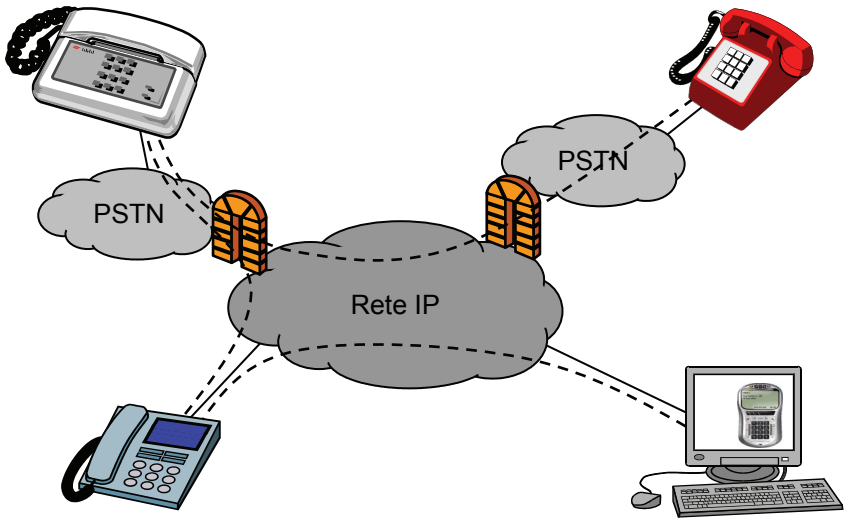

Figura 1. Scenario generale del servizio VoIP

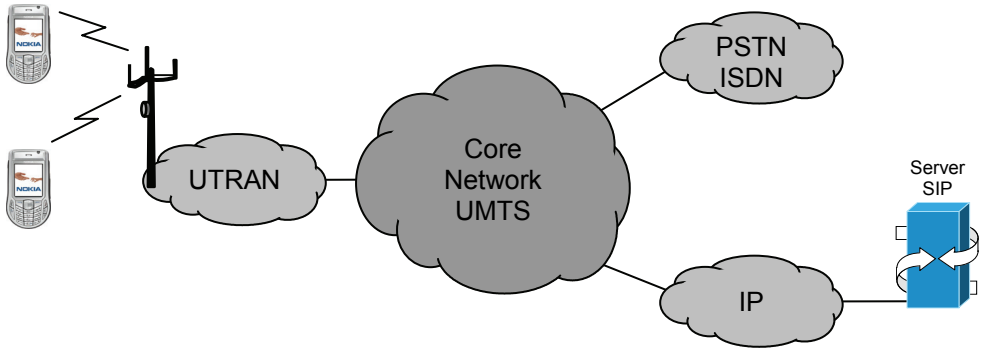

Figura 2. Scenario del progetto *UniPR-Ptt*

modificando o estendendo il codice realizzato, vengono infine proposti alcuni possibili sviluppi futuri del progetto *UniPR-Ptt*.

1.2. Scenario

In termini generali, lo scenario in cui si colloca il progetto realizzato è quello tipico del servizio VoIP, nel quale si utilizza una rete basata su IP per trasportare il traffico relativo ad una comunicazione telefonica tra due o più utenti. Si parla comunemente di VoIP sia quando la chiamata transita da estremo a estremo su reti IP, sia nel caso in cui viene instradata su reti di questo tipo solo per compiere una parte del percorso tra i due terminali, ognuno dei quali può essere collegato a una rete diversa (PSTN, ISDN, TACS, GSM,...), come mostrato in figura 1.

Più in particolare, nel caso specifico a cui si riferisce il progetto *UniPR-Ptt* (mostrato in figura 2), i terminali (detti *Mobile Station* o MS) di entrambi gli utenti sono collegati all'*UMTS Terrestrial Radio Access Network* (UTRAN). Questa è la rete di accesso del sistema UMTS, comprendente i nodi (stazioni base e server) necessari per consentire agli utenti di connettersi alla rete e di trasmettere e ricevere dati attraverso l'interfaccia radio. L'*Access Network*, a sua volta, è collegata alla *Core Network*, che

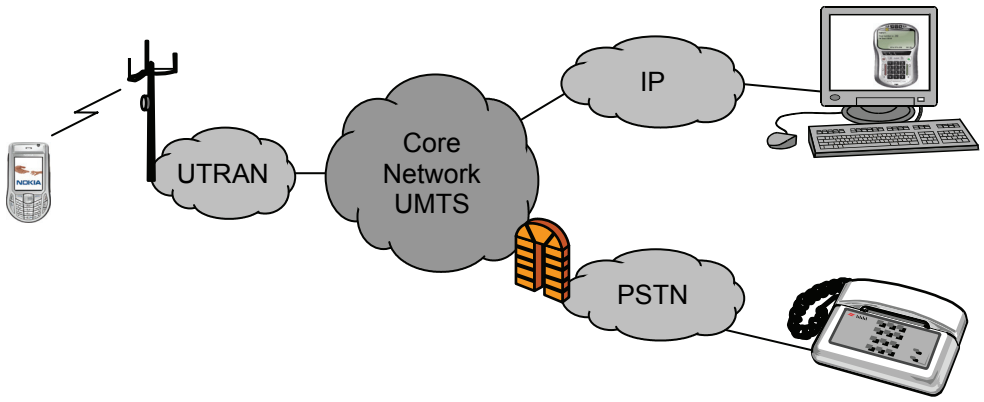

Figura 3. Scenario relativo a possibili sviluppi futuri

trasporta i dati (di traffico e di segnalazione) verso altri terminali, appartenenti alla stessa rete o a reti differenti. Queste possono essere non solo reti UMTS di altri operatori, ma anche reti telefoniche tradizionali (PSTN o simili) o reti basate su IP (Internet su tutte).

Sfruttando le interconnessioni della Core Network UMTS con reti esterne, è anche possibile realizzare uno scenario (mostrato in figura 3 e lasciato come sviluppo futuro del presente lavoro) in cui uno degli smartphone UMTS è rimpiazzato da un terminale SIP, costituito eventualmente da un softphone in esecuzione su un comune PC. Introducendo un apposito gateway che metta in comunicazione la rete IP con la rete telefonica classica, sarebbe addirittura possibile (come mostrato nella stessa figura) comunicare direttamente con un normale telefono.

2. Strumenti utilizzati

Per la realizzazione del progetto *UniPR-Ptt* sono stati utilizzati diversi strumenti, intesi sia come sistemi e protocolli di comunicazione che come sistemi operativi e relativi linguaggi di programmazione. Prima di procedere alla stesura del codice sono state studiate le caratteristiche principali di ognuno di tali strumenti; queste verranno esposte all'interno del presente capitolo, in modo da evidenziare le modalità con cui gli strumenti in questione sono stati impiegati all'interno del progetto complessivo.

2.1. Symbian

Il progetto *UniPR-Ptt* è stato sviluppato espressamente per essere utilizzato su dispositivi mobili dotati di sistema operativo Symbian 8.0. Nel seguito verranno perciò esaminate alcune delle caratteristiche specifiche di tale sistema operativo e dello stile di programmazione che deve essere adottato quando si intende scrivere un programma destinato a terminali che lo supportano.

2.1.1. Caratteristiche generali

Fino a qualche anno fa i telefoni cellulari erano apparecchi relativamente semplici, e quindi non richiedevano sistemi operativi particolari o complesse piattaforme per lo sviluppo di applicativi: ogni produttore poteva utilizzare un proprio sistema operativo, senza che ciò creasse particolari problemi agli utenti o agli sviluppatori. Da qualche tempo, invece, si è assistito a una rapida evoluzione di tali dispositivi, alcuni dei quali hanno raggiunto una complessità tale da rendere persino difficile catalogarli come semplici telefoni. Questi dispositivi, detti più specificamente *smartphone*, dispongono della possibilità di eseguire non solo le applicazioni create e preinstallate dal costruttore, ma anche altre sviluppate da terzi e installate dall'utente. Essi necessitano perciò di sistemi operativi complessi e il più possibile standardizzati, nonché di una piattaforma affidabile e versatile per agevolare lo sviluppo di applicativi da parte di terzi.

Per la creazione di un sistema operativo con le opportune caratteristiche, nel 1998 alcuni dei principali produttori di smartphone (Ericsson, Nokia, Matsushita, Motorola, Psion, Siemens, Sony) hanno creato il consorzio indipendente Symbian, il cui logo è mostrato in figura 1. Tale consorzio ha prodotto un sistema operativo standard specifico per terminali mobili, chiamato anch'esso Symbian. La realizzazione di un nuovo sistema operativo si è resa necessaria perché creare una versione ridotta di uno esistente per PC o espanderne uno progettato per terminali più semplici avrebbe portato a compromessi inaccettabili.

Figura 1. Il logo di Symbian

Il sistema operativo Symbian è stato progettato in modo da essere abbastanza flessibile per poter essere usato su dispositivi che differiscono gli uni dagli altri per quanto riguarda hardware, interfacce utente e interfacce di rete. A causa di tale varietà di applicazioni, la versione 6 di Symbian è stata suddivisa in tre categorie, ognuna delle quali specifica per una determinata tipologia di dispositivi: la versione *Quartz* è stata pensata per PDA con touch screen, la versione *Crystal* per terminali con tastiera QWERTY, la versione *Pearl* per smartphone con tastiera numerica. A partire dalla versione 7 questa suddivisione è stata rimossa, in modo da lasciare ai produttori dei dispositivi ampia libertà di scelta per quanto riguarda l'interfaccia utente.

Per sviluppare software destinato al sistema operativo Symbian è possibile utilizzare Java2 Micro Edition (J2ME) con Mobile Information Device Profile (MIDP), ma operando in questo modo si soffrono varie limitazioni e lo sviluppo di alcuni tipi di applicazioni risulta difficoltoso se non impossibile. In particolare, è difficile creare applicativi che richiedono l'accesso all'hardware del dispositivo o ad altri software installati, o che necessitano di prestazioni elevate. In tali casi è possibile e conveniente sviluppare applicazioni in maniera nativa per il sistema operativo Symbian. Quest'ultima strada è stata quella scelta per lo sviluppo del progetto *UniPR-Ptt*. Come si può vedere dalla figura 2, le applicazioni native dispongono di un accesso diretto al sistema operativo, mentre le applicazioni Java possono accedervi in maniera limitata unicamente attraverso le opportune API Java.

2.1.2. C++ per Symbian

Il progetto *UniPR-Ptt* è stato scritto in linguaggio C++, utilizzando gli strumenti specifici per lo sviluppo di applicazioni destinate a terminali mobili dotati di sistema operativo Symbian. Tali strumenti si traducono, dal punto di vista del programmatore, in alcune particolarità che differenziano lo stile di programmazione che deve essere seguito rispetto a quello tipico del C++ standard; queste particolarità sono state introdotte principalmente

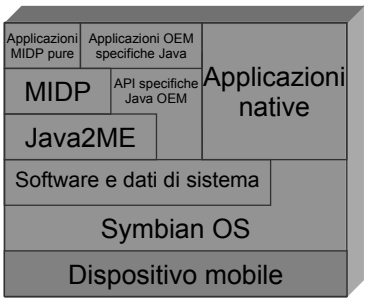

Figura 2. Architettura software di un tipico dispositivo mobile

per tenere conto del fatto che i programmi dovranno essere eseguiti su terminali con ridotte capacità di elaborazione e memorizzazione. Il progetto è stato sviluppato facendo uso della *Series 60 Developer Platform*, un'estensione del sistema operativo Symbian che ne estende le funzionalità offerte agli sviluppatori, rendendo disponibili in particolare una libreria per la gestione dell'interfaccia grafica e un emulatore (figura 3) su cui gli sviluppatori possono testare il funzionamento dei propri applicativi prima di caricarli su un dispositivo reale.

Figura 3. L'emulatore di terminale Series 60

L'ambiente di sviluppo utilizzato è stato il Borland C++BuilderX Mobile Edition (un'estensione del C++Builder apposita per piattaforme Symbian Series 60), nella versione 1.5, in abbinamento all'SDK fornito da Nokia appositamente per agevolare la creazione di applicazioni destinate ai propri dispositivi. L'applicativo è stato testato sia sull'emulatore incluso nell'SDK che su un terminale Nokia 6630.

Come si è accennato, lo stile di programmazione in C++ per terminali Symbian Series 60 è simile a quello tradizionale, ma comprende diversi concetti specifici del particolare ambiente su cui le applicazioni dovranno essere eseguite, che tengono conto delle limitazioni di quest'ultimo per quanto riguarda in particolare la quantità di memoria disponibile. Per ovviare a tali limitazioni, sono state introdotte politiche particolarmente efficienti per la gestione delle risorse di sistema, alle quali gli sviluppatori devono attenersi. Nel seguito si elencano alcune delle differenze più evidenti tra la programmazione in C++ standard e quella per terminali Series 60.

Innanzi tutto, scompare la funzione `main()`: per scrivere anche il programma più semplice è necessario implementare diverse classi, nessuna delle quali contiene una funzione assimilabile alla `main()` tipica della normale programmazione in C/C++. Ogni applicazione deve tipicamente includere almeno le seguenti classi:

- **Application**: questa classe costituisce il principale punto di ingresso dell'applicazione, ed ha lo scopo fondamentale di indicare al sistema operativo l'identificativo univoco (detto UID) dell'applicazione stessa. Tipicamente la classe Application ha un'implementazione quasi standard, e la sua struttura rimane praticamente invariata per qualunque progetto.
- **AppUi**: questa classe agisce da collegamento tra l'interfaccia grafica dell'applicazione e gli eventi generati dall'utente, quali ad esempio la pressione di particolari tasti. La classe può contenere al suo interno le funzioni delegate alla gestione di tali eventi oppure può richiamare apposite funzioni di gestione.
- **AppView**: questa classe definisce le modalità (anche più di una) con cui i dati devono essere rappresentati all'utente. La rappresentazione dei dati può avvenire mediante uno qualunque dei widget offerti dalle librerie per la GUI comprese nella Series 60 Developer Platform SDK.
- **Document**: questa classe fornisce un sistema per memorizzare i dati di un'applicazione e contiene le funzioni che servono a manipolare questi dati. Il vantaggio di utilizzare una classe per la memorizzazione dei dati separata dalla classe che si occupa della loro rappresentazione è che in questo modo ogni variazione dei dati si riflette simultaneamente su tutte le differenti visualizzazioni degli stessi.

Un'altra differenza rispetto al C++ standard è che in Symbian non si fa uso della Standard Template Library disponibile normalmente in C++. Infatti, il C++ per Symbian definisce un proprio insieme di classi template, che devono essere utilizzate al posto di quelle tradizionali.

Infine, i tipi di dato fondamentali utilizzati in C++ per Symbian sono differenti da quelli tipici del C++ standard: ad esempio, il tipo di dato normalmente utilizzato per contenere un numero intero non è `int` ma `TInt`. L'uso dei tipi di dato standard rimane possibile ma è sconsigliato.

Come si è detto, nello scrivere programmi per terminali mobili, che dispongono di risorse limitate, è necessario porre particolare attenzione ad evitare sprechi di memoria. Il sistema utilizzato in Symbian per prevenire la perdita di porzioni di memoria si basa sull'uso del cosiddetto *cleanup stack*. Il suo principio di funzionamento prevede che, se all'interno di una funzione si crea una variabile allocata dinamicamente nello heap, un puntatore a questa variabile venga inserito nel cleanup stack, mediante l'invocazione della funzione `CleanupStack::Push()`. Se la funzione, per qualunque motivo, non dovesse terminare correttamente, in assenza del cleanup stack il puntatore all'area di memoria creata verrebbe cancellato e quindi non sarebbe più possibile liberare tale area. In Symbian, invece, in caso di errori il sistema operativo esamina automaticamente il cleanup stack e provvede a distruggere tutte le variabili in esso contenute e a liberare le corrispondenti aree di memoria. Se la funzione termina senza errori, è necessario rimuovere manualmente i puntatori presenti nel cleanup stack mediante le funzioni `CleanupStack::Pop()`, che rimuove solamente il puntatore ma non distrugge

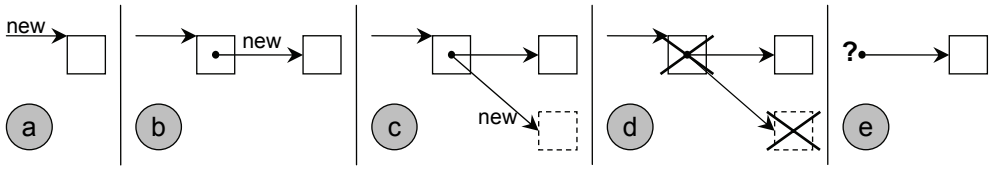

Figura 4. a) viene creato un oggetto composto; b) all'interno del relativo costruttore si crea un primo oggetto; c) la creazione di un secondo oggetto fallisce; d) l'oggetto composto non viene costruito; e) rimane allocata un'area di memoria non referenziata da alcun puntatore e quindi non deallocabile

l'oggetto, o `CleanupStack::PopAndDestroy()`, che distrugge anche l'oggetto puntato.

Il cleanup stack viene utilizzato, in particolare, nella fase di costruzione degli oggetti composti, che in C++ per Symbian segue regole particolari. Infatti, a causa della scarsità delle risorse disponibili sui dispositivi portatili, non è raro che il processo di costruzione di un oggetto composto fallisca, per l'impossibilità di allocare la memoria necessaria ad uno degli oggetti di cui esso si compone, dopo che una parte del processo di costruzione è già stata completata. In questo caso, come mostrato in figura 4, la memoria già allocata rimarrebbe occupata, e non ci sarebbe alcun modo di liberarla. Per questo motivo, per creare gli oggetti composti si utilizza un processo di costruzione in due fasi e si fa uso di un cleanup stack in cui inserire gli oggetti che vengono via via creati, in modo che questi possano essere distrutti se la costruzione dell'oggetto composto non va a buon fine. In teoria è anche possibile non seguire queste regole e costruire tutti gli oggetti mediante i normali costruttori C++, ma tale modo di procedere è vivamente sconsigliato. Il metodo a due fasi suggerito prevede invece che il costruttore standard della classe (che deve essere presente per compatibilità con la normale sintassi del C++) non esegua alcuna particolare operazione: quando è chiamato, viene solamente allocata la memoria necessaria per contenere un oggetto della classe in questione. La creazione di tutti gli oggetti membri della classe avviene in un'altra funzione, chiamata tipicamente[1] `ConstructL()`.

Per costruire un oggetto di una determinata classe, occorre richiamare una particolare funzione appartenente alla classe stessa, denominata `NewL()`. Essa, in genere, include entrambe le fasi della costruzione: al suo interno viene dapprima richiamato il costruttore standard della classe, che alloca un'opportuna quantità di memoria; quindi, se la costruzione è terminata con successo, un puntatore all'oggetto creato viene inserito nel cleanup stack. In seguito viene richiamata la `ConstructL()`, che si occupa della costruzione degli oggetti membri della classe: in tal modo, se la costruzione di uno di tali oggetti dovesse fallire, si avrebbe comunque un sistema per

[1] I nomi qui indicati per le varie funzioni sono quelli suggeriti dallo standard, ma gli sviluppatori possono utilizzare i nomi che preferiscono. Tuttavia, per semplificare la lettura del codice e la compatibilità con altri applicativi, è vivamente consigliato attenersi alle denominazioni standard.

liberare la memoria precedentemente allocata per contenere l'oggetto principale. Se invece tutto procede correttamente, dopo la `ConstructL()` viene richiamata la funzione `CleanupStack::Pop()`, che rimuove il puntatore al nuovo oggetto dal cleanup stack.

Talvolta all'interno di una classe è presente, in aggiunta o in alternativa alla `NewL()`, una funzione chiamata `NewLC()`, che effettua le medesime operazioni ma non rimuove dal cleanup stack il puntatore all'oggetto creato. La costruzione di un oggetto deve essere effettuata mediante tale funzione quando il puntatore all'oggetto creato è contenuto in una variabile automatica. In tal caso, infatti, l'oggetto non deve essere rimosso dal cleanup stack prima che il suo utilizzo sia terminato, poiché altrimenti, se l'esecuzione del programma si dovesse interrompere a causa di errori, la corrispondente area di memoria rimarrebbe allocata.

Oltre a queste peculiarità, il linguaggio C++ per Symbian introduce particolari convenzioni per i nomi delle classi, delle variabili e delle funzioni; il rispetto di queste convenzioni nella scrittura di un programma non è ovviamente obbligatorio, ma semplifica la leggibilità del codice e le verifiche di consistenza. Si elencano nel seguito quelle utilizzate più frequentemente.

- **Classi**:
  - Le classi il cui nome inizia col prefisso "T" non contengono come membro alcun oggetto che debba essere allocato nello heap: per questa ragione, esse non necessitano di distruttori. Un oggetto appartenente a una classe di questo tipo può essere allocato sia nello stack che nello heap.
  - Le classi il cui nome inizia col prefisso "C" ereditano da una particolare classe contenuta all'interno del sistema operativo Symbian, chiamata `CBase`. Stando alla documentazione ufficiale di Symbian, le istanze di tali classi devono essere allocate nello heap, e tutti i relativi dati membri sono automaticamente inizializzati a 0.
  - Le classi il cui nome inizia col prefisso "M" sono quelle per cui è definita unicamente l'interfaccia, ma non l'implementazione. In altre parole, esse consistono unicamente in un file header, in cui sono definite le funzioni virtuali che dovranno essere implementate da ogni classe non astratta che erediti da una di tali classi.
- **Variabili e costanti**:
  - I nomi delle variabili definite all'interno di una classe, ovvero dei dati membri della classe, devono essere preceduti dalla lettera minuscola "i".
  - I nomi delle variabili usate come argomenti di una funzione devono essere preceduti dalla lettera minuscola "a".
  - I nomi delle variabili locali, ovvero il cui ambito di visibilità è limitato a un singolo blocco di codice, non hanno alcun prefisso ed iniziano con una lettera minuscola.
  - I nomi delle costanti devono essere preceduti dalla lettera maiuscola "K".

- **Funzioni**:
  - I nomi delle funzioni devono iniziare con una lettera maiuscola.
  - I nomi di tutte le funzioni la cui esecuzione può interrompersi prima che venga raggiunto il termine della funzione stessa, ad esempio perché tentano di creare un nuovo oggetto per cui possono non essere disponibili risorse oppure perché richiamano altre funzioni che a loro volta possono terminare prima di concludere la normale esecuzione, devono essere seguiti dal suffisso "L".
  - I nomi delle funzioni che creano un oggetto e lo inseriscono sul cleanup stack senza rimuoverlo prima di terminare devono essere seguiti dalla lettera maiuscola "C".
  - I nomi delle funzioni che distruggono l'elemento passato loro come argomento devono essere seguiti dalla lettera maiuscola "D".

Il sistema operativo Symbian, inoltre, fa largo uso di metodi di programmazione asincrona per semplificare la gestione del multithreading da parte degli sviluppatori; gli strumenti per implementare tali concetti sono dunque presenti anche nel linguaggio C++ per Symbian.

In generale, ogni programma per Symbian appare composto da una singola thread; se si agisse realmente in questo modo, tuttavia, non si sfrutterebbero le caratteristiche di multithreading offerte da Symbian e non si ottimizzerebbero i tempi di esecuzione dei processi. Ad esempio, se un programma, a un certo punto dell'esecuzione, ha la necessità di leggere dati da un file, la sua esecuzione risulterebbe bloccata fin quando l'acquisizione dei dati richiesti non fosse terminata, e nel frattempo il sistema non sarebbe in grado di rispondere a eventuali comandi dell'utente. Il sistema operativo Symbian permette allo sviluppatore di evitare questo tipo di inconvenienti senza dover suddividere esplicitamente il programma in diverse thread, mediante l'uso di richieste asincrone e di oggetti attivi.

È previsto infatti che, per effettuare operazioni il cui completamento potrebbe richiedere tempi notevoli, il programma principale possa stabilire una sessione con un apposito server, al quale poi possono essere inviate le richieste di esecuzione dell'operazione in questione. Quando l'elaborazione dell'operazione è completata, sarà il server stesso a notificare l'evento al programma che l'ha richiesta, mediante la chiamata di un'apposita funzione. Il processo server viene eseguito separatamente, e non interferisce con l'esecuzione del programma principale. Nel periodo di tempo che intercorre tra la richiesta al server e la sua risposta, il programma può eseguire normalmente altre operazioni, ma è anche possibile, invocando la funzione `User::WaitForRequest()`, forzare l'attesa del completamento della richiesta prima di proseguire con l'esecuzione. Quest'ultima metodologia comporta una maggiore semplicità di programmazione e può non causare eccessivi problemi se l'operazione richiesta al server termina in breve tempo. Tuttavia, per la maggior parte delle applicazioni, il primo metodo è preferibile, specialmente se la richiesta può richiedere parecchio tempo prima di essere completata.

Questo modo di procedere è implementato in C++ per Symbian attraverso i concetti di oggetti attivi e di active scheduler. Un oggetto attivo è un'istanza di una qualsiasi classe che deriva da `CActive`, una particolare classe definita in Symbian appositamente per la gestione di tali oggetti. Perché un oggetto possa essere definito attivo non è tuttavia sufficiente che la classe a cui appartiene l'oggetto stesso erediti da `CActive`, ma è necessario inserire nella funzione `ConstructL()` della classe derivata una chiamata alla funzione `CActiveScheduler::Add()`, che serve a registrare l'oggetto sull'active scheduler. In seguito, ogni volta che l'oggetto effettua una richiesta asincrona al server, occorre richiamare la funzione `SetActive()`. L'active scheduler mantiene un elenco degli oggetti attivi registrati su di esso; quando uno di questi invoca la `SetActive()`, lo scheduler ne esamina la richiesta e, quando l'elaborazione è completata, notifica l'evento all'oggetto richiedente, la cui elaborazione nel frattempo può proseguire. La notifica dell'avvenuto completamento dell'operazione è effettuata dall'active scheduler mediante la chiamata di un'apposita funzione chiamata `RunL()`, che è definita come *pure virtual* in `CActive` e deve quindi essere obbligatoriamente implementata all'interno della classe a cui appartiene l'oggetto attivo. Un'altra funzione che deve essere definita in ogni classe derivata da `CActive` è la `DoCancel()`; questa viene chiamata dall'active scheduler quando una richiesta pendente è stata annullata. Si noti che né la `RunL()` né la `DoCancel()` dovrebbero mai essere richiamate esplicitamente dall'interno del programma: per annullare una richiesta pendente si usa invece la funzione `Cancel()`, che a sua volta, compiute le necessarie operazioni, richiamerà la `DoCancel()`. Se durante l'esecuzione delle operazioni il server incontra un errore, viene richiamata la funzione `RunError()`, che, a differenza della `RunL()`, possiede un'implementazione di default che prevede di terminare immediatamente il programma; questa tuttavia può essere ridefinita liberamente dallo sviluppatore.

Per concludere, occorre rilevare come il linguaggio C++ per Symbian presenti anche numerose somiglianze con il C++ standard: in particolare, la sintassi dei programmi è quella tipica del C++, e quindi la loro struttura può essere compresa senza eccessive difficoltà anche da chi sia abituato al normale C++, benché diverse tecniche di programmazione siano specifiche della piattaforma.

2.2. SIP e SDP

Per l'instaurazione della comunicazione tra i due interlocutori, nel progetto *UniPR-Ptt* è stato utilizzato il protocollo SIP, in abbinamento a SDP per quanto riguarda la descrizione delle caratteristiche del media che dovrà essere scambiato dai terminali. Nel seguito verranno dunque presentate le caratteristiche fondamentali di tali protocolli. Inoltre, siccome per l'uso di SIP sul terminale utilizzato è necessaria la preventiva

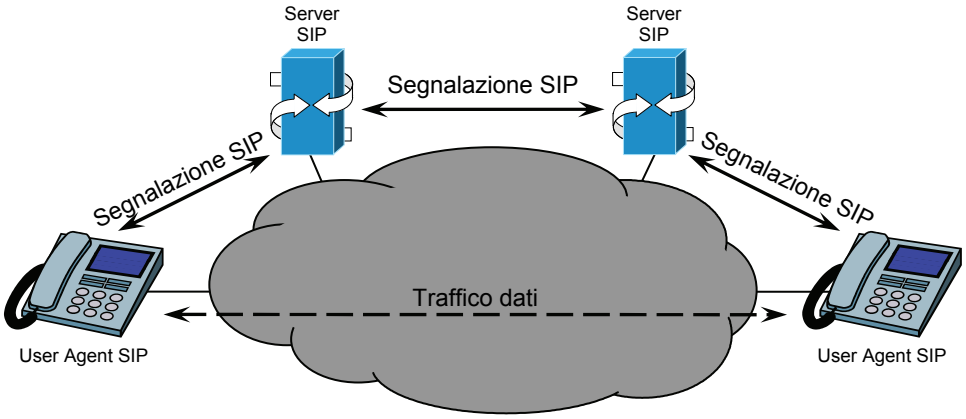

Figura 5. Architettura di base del protocollo SIP

installazione di un apposito plug-in, verranno presentate le funzionalità di quest'ultimo e le relative modalità d'uso.

2.2.1. Descrizione del protocollo SIP

SIP (*Session Initiation Protocol*)[2] è un protocollo di livello applicativo sviluppato a partire dal 1999 da parte del MMUSIC Working Group dell'IETF con lo scopo di fornire gli strumenti necessari per instaurare e gestire una comunicazione tra due o più utenti. Esso, in particolare, permette di iniziare, modificare e terminare una sessione, durante la quale gli utenti possono scambiarsi informazioni multimediali di vario genere: non solo flussi di dati audio, ma anche video, messaggi testuali, informazioni relative a giochi online o ad applicativi di realtà virtuale. Attualmente SIP, insieme ad H.323, è uno dei principali protocolli di segnalazione utilizzati per il servizio di *Voice over IP* (VoIP). SIP, inoltre, è stato scelto come protocollo standard per il supporto delle sessioni multimediali nelle reti 3GPP Release 5 e successive. L'architettura di base del protocollo è schematizzata in figura 5.

Il protocollo SIP, analogamente ad altri quali HTTP o SMTP, è di tipo testuale (ovvero i messaggi scambiati sono costituiti da stringhe di caratteri ASCII) e sfrutta un paradigma di tipo client-server: i messaggi SIP possono essere suddivisi in richieste (inviate dal client al server) e risposte (che viaggiano in senso contrario). Tuttavia, a differenza di quanto avviene per altri protocolli, uno stesso terminale può fungere sia da client (quando invia un messaggio di richiesta) che da server (quando risponde a una richiesta proveniente da un altro terminale). La caratteristica di utilizzare messaggi di tipo testuale semplifica la fase di debug di applicativi basati su SIP. Esso, inoltre, è stato progettato in modo da essere altamente modulare, facilmente estendibile e indipendente

[2] Inizialmente definito nella RFC 2543, resa poi obsoleta dalla RFC 3261. La versione di SIP attualmente in uso è la 2.0.

dal protocollo di trasporto utilizzato: i messaggi SIP possono infatti essere trasportati sia da TCP che da UDP, in quanto SIP comprende meccanismi per rendere affidabile la comunicazione anche utilizzando protocolli di trasporto di per sé non affidabili. Questi meccanismi si basano sulla ritrasmissione dei pacchetti: se non si riceve risposta a un messaggio di richiesta entro un tempo stabilito, lo stesso messaggio viene trasmesso nuovamente, e così via finché non si riceve la risposta opportuna o finché non viene raggiunto un numero di ritrasmissioni prestabilito.

Figura 6. Esempio di hardphone SIP

Le funzioni principali del protocollo SIP sono le seguenti:

- Registrazione di un utente, a seguito della quale l'utente in questione può essere temporaneamente associato a un particolare terminale, in modo che i messaggi diretti al suo indirizzo SIP possono essere inviati al terminale che sta attualmente utilizzando;
- Instaurazione di una sessione:
  - Verifica della disponibilità dell'interlocutore;
  - Definizione delle caratteristiche della sessione;
  - Descrizione delle capacità dei terminali;
  - Negoziazione dei parametri della comunicazione;
- Modifica dei parametri di una sessione aperta;
- Termine di una sessione;
- Deregistrazione di un utente.

Gli elementi che costituiscono l'architettura di una rete SIP sono i seguenti:

- **User Agent (UA)**: terminale mediante il quale l'utente può scambiare messaggi SIP con un opportuno server o con un altro terminale dello stesso tipo. In pratica uno user agent può essere un dispositivo esteticamente simile a un normale telefono (detto *hardphone* e mostrato in figura 6) oppure un applicativo software che gira su un comune personal computer (detto *softphone* e mostrato in figura 7). Uno user agent può sia effettuare una chiamata (agendo quindi da client) che accettare, rifiutare o redirigere una chiamata ad

Figura 7. Esempio di softphone SIP

esso diretta (comportandosi da server). Per instaurare una sessione, gli UA che vi prenderanno parte possono scambiarsi direttamente gli opportuni messaggi SIP o (come avviene tipicamente) fare uso di uno o più server SIP.

- **Server SIP**: sistema che può ricevere ed elaborare le richieste di registrazione degli utenti, mantenere le informazioni sulla loro attuale localizzazione, inoltrare i messaggi provenienti dagli User Agent o rispondere a questi ultimi. I server SIP si dividono, a seconda della funzione, in, *proxy*, *redirect* e *registrar*. Uno stesso server, tuttavia, può appartenere, a seconda della situazione, a più di una categoria.
  - **Server proxy**: riceve i messaggi SIP di richiesta da un client e li inoltra, per conto di quest'ultimo, al nodo successivo all'interno della rete[3]. Il messaggio in ingresso può anche essere inoltrato, in maniera sequenziale o in parallelo, a diverse destinazioni. Oltre a questa funzionalità di base, un server proxy può effettuare la ritrasmissione dei messaggi in modo da rendere la comunicazione affidabile, nonché implementare funzioni relative alla sicurezza quali autenticazione, autorizzazione, controllo dell'accesso alla rete, ecc. Un server proxy può essere stateless oppure stateful; in quest'ultimo caso il concetto di stato si può riferire alla singola transazione o all'intera chiamata.
  - **Server redirect**: riceve i messaggi SIP di richiesta e fornisce al client che li ha inviati l'indirizzo del successivo nodo a cui questi devono essere spediti, ma non li inoltra direttamente.
  - **Server registrar**: memorizza l'indirizzo IP del terminale che l'utente sta attualmente utilizzando, in modo da consentirgli di spostarsi da un terminale all'altro restando comunque raggiungibile. Tipicamente ad un server registrar è associato un proxy e/o un redirect.
- **Gateway**: funge da interfaccia tra una rete basata su SIP e un'altra che utilizza, per le stesse operazioni, un diverso protocollo di segnalazione (quale ad esempio H.323 o SS7, oppure la segnalazione d'utente della rete PSTN). Le funzionalità implementate devono perciò comprendere quanto meno la traduzione dei messaggi SIP nei corrispondenti messaggi appartenenti al protocollo in uso sull'altra rete e viceversa. Alcuni gateway possono anche terminare il piano dati, effettuando la transcodifica dei media, in modo da adattarne il formato a quello in uso sulla rete di destinazione (che può essere, ad esempio, la rete telefonica classica).

Come si è detto, il protocollo SIP suddivide i messaggi in richieste e risposte. Ognuna di queste due categorie di messaggi è caratterizzata da un formato ben preciso. I messaggi SIP di richiesta, in particolare, sono composti da una *request line*, da diversi campi di header e da un eventuale body (che può contenere qualunque tipo di dati e che non è gestito direttamente da SIP). All'interno della *request line*, in particolare, è indicato il metodo, ovvero il tipo di richiesta:

[3] Un server proxy si comporta quindi contemporaneamente sia da server che da client.

- **REGISTER**, utilizzato per richiedere la registrazione dell'utente su un server SIP e la memorizzazione delle relative informazioni.
- **INVITE**, utilizzato per invitare un utente a partecipare a una sessione; all'interno di tale messaggio è possibile inserire una descrizione dei parametri che caratterizzeranno la sessione, utilizzando (in genere) il protocollo SDP, descritto in seguito. Inoltre, mandando un messaggio di INVITE quando la sessione è già stata aperta, è possibile variarne le caratteristiche.
- **ACK**, utilizzato per confermare la ricezione della risposta relativa ad un INVITE e terminare la fase di handshake.
- **CANCEL**, utilizzato per annullare una richiesta pendente.
- **BYE**, utilizzabile sia da chi ha spedito l'INVITE sia da chi l'ha ricevuto per terminare la sessione.
- **OPTIONS**, utilizzato per interrogare un server in modo da determinarne le capacità.

I messaggi di risposta invece, analogamente a quanto avviene in HTTP, iniziano con una *status line*, la quale contiene al suo interno un codice numerico di tre cifre che identifica il tipo di risposta. La prima di queste tre cifre rappresenta la classe della risposta:

- **1XX**: *Provisional* (la richiesta è stata ricevuta e la sua elaborazione è in corso);
- **2XX**: *Success* (la richiesta è stata ricevuta e accettata);
- **3XX**: *Redirection* (per completare la richiesta sono necessarie ulteriori operazioni);
- **4XX**: *Client Error* (la richiesta contiene errori di sintassi o non può essere soddisfatta dal server scelto);
- **5XX**: *Server Error* (il server ha incontrato errori nel tentativo di soddisfare una richiesta apparentemente corretta);
- **6XX**: *Global Failure* (la richiesta non può essere soddisfatta da alcun server).

In SIP è previsto che ogni utente sia identificato da un indirizzo (URL SIP) simile al normale indirizzo e-mail: il formato degli indirizzi è infatti del tipo `sip:nomeutente@dominio`. L'utente, se lo desidera, può registrare il proprio indirizzo su un server registrar, associandolo all'indirizzo IP del terminale che sta utilizzando. Per iniziare una sessione è quindi possibile specificare unicamente l'indirizzo SIP dell'utente chiamato, che verrà mappato dal server nell'opportuno indirizzo IP, a cui verranno poi inviati tutti i messaggi diretti all'utente in questione. Un utente, come si è accennato, può anche essere registrato contemporaneamente su più dispositivi, nel qual caso un server proxy inoltrerà a tutti questi dispositivi i messaggi ad esso diretti, mentre un server redirect restituirà gli indirizzi di tutti.

D'altra parte, la comunicazione può anche essere instaurata direttamente tra i due user agent, senza passare attraverso alcun server, così come mostrato in figura 8. In questo caso, l'utente che vuole iniziare la sessione invia all'altro una richiesta di tipo INVITE, nel cui header specifica (tra l'altro) il proprio indirizzo SIP, quello del destinatario, l'oggetto della comunicazione e il tipo di dati trasportati nel corpo del messaggio stesso. Tipicamente, tali dati descrivono i formati di media supportati dal

terminale che ha spedito il messaggio di INVITE; per rappresentare queste informazioni è possibile utilizzare qualunque metodo, ma tipicamente si fa uso del protocollo SDP, che sarà oggetto del prossimo paragrafo.

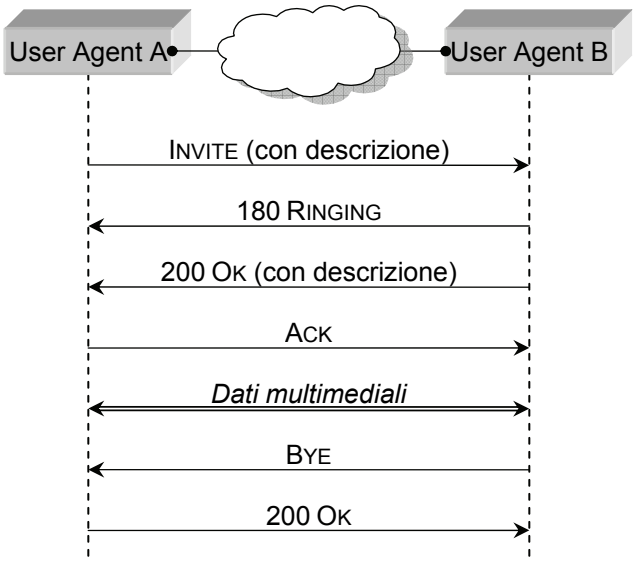

Figura 8. Comunicazione diretta tra due user agent

Se tutto procede correttamente, il terminale che riceve la richiesta di INVITE risponde con un messaggio (di classe *Provisional*) 180 RINGING. Se l'utente accetta la chiamata, viene quindi inviata la risposta (di classe *Success*) 200 OK; nel body di tale messaggio, il terminale chiamato indica quali tra i formati supportati dal chiamante sono stati scelti e verranno quindi utilizzati per la comunicazione. Anche in questo caso, benché non vi sia alcun obbligo in tal senso, è prassi comune rappresentare le necessarie informazioni mediante il protocollo SDP. Al momento della ricezione di quest'ultimo messaggio, il primo terminale invia il messaggio di ACK[4], che, a differenza di tutti gli altri tipi di richieste SIP, non prevede alcuna risposta.

Il messaggio di ACK conclude la fase di instaurazione della comunicazione, al termine della quale è possibile iniziare a scambiare dati nel formato stabilito durante l'handshake. Infine, per terminare la sessione, uno qualunque dei terminali può inviare all'altro la richiesta SIP di BYE, alla quale l'altro terminale risponde con un messaggio di 200 OK.

Tipicamente, tuttavia, lo scambio di messaggi SIP non avviene direttamente tra due terminali, ma coinvolge uno o più server di vario tipo. In figura 9 è schematizzato il caso in cui un utente utilizza un server registrar per associare il proprio indirizzo SIP all'indirizzo IP del terminale che intende utilizzare nell'immediato futuro. In questo scenario, per prima cosa l'utente invia al server una richiesta di REGISTER, in cui indica semplicemente il proprio indirizzo SIP e l'indirizzo IP del terminale che intende utilizzare; il server, dopo aver memorizzato le informazioni in un apposito *location server* (che non è un elemento dell'architettura SIP) risponde alla richiesta con un messaggio di 200 OK.

[4] Tale messaggio, come si è visto, secondo la terminologia propria di SIP è considerato come una richiesta, ma di fatto costituisce il riscontro affermativo alla precedente risposta di 200 OK.

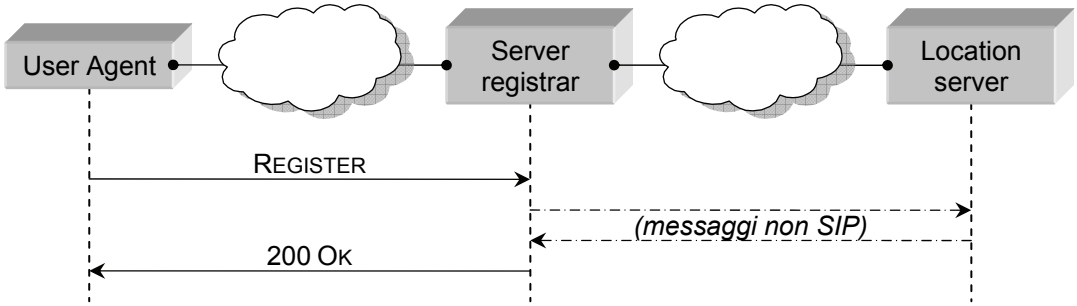

Figura 9. Registrazione di un profilo SIP

In seguito, se il server registrar è associato a un proxy, come mostrato in figura 10, per comunicare con un utente registrato su di esso è sufficiente specificarne l'indirizzo SIP in un messaggio di INVITE diretto al server; quest'ultimo si occuperà di inoltrare il messaggio all'indirizzo su cui l'utente è registrato, informando il primo terminale di tale operazione mediante un 100 TRYING. Anche i messaggi SIP successivi, fino al 200 OK, transiteranno attraverso il server; a partire dal relativo ACK, invece, i dati potranno essere inviati direttamente da un terminale all'altro. In particolare, i flussi di dati multimediali saranno scambiati direttamente tra i due terminali, a meno di forzarne il passaggio attraverso il server.

La figura 11 schematizza invece il caso in cui al server registrar è associato un redirect: ora, al momento della ricezione del messaggio di INVITE diretto a un utente

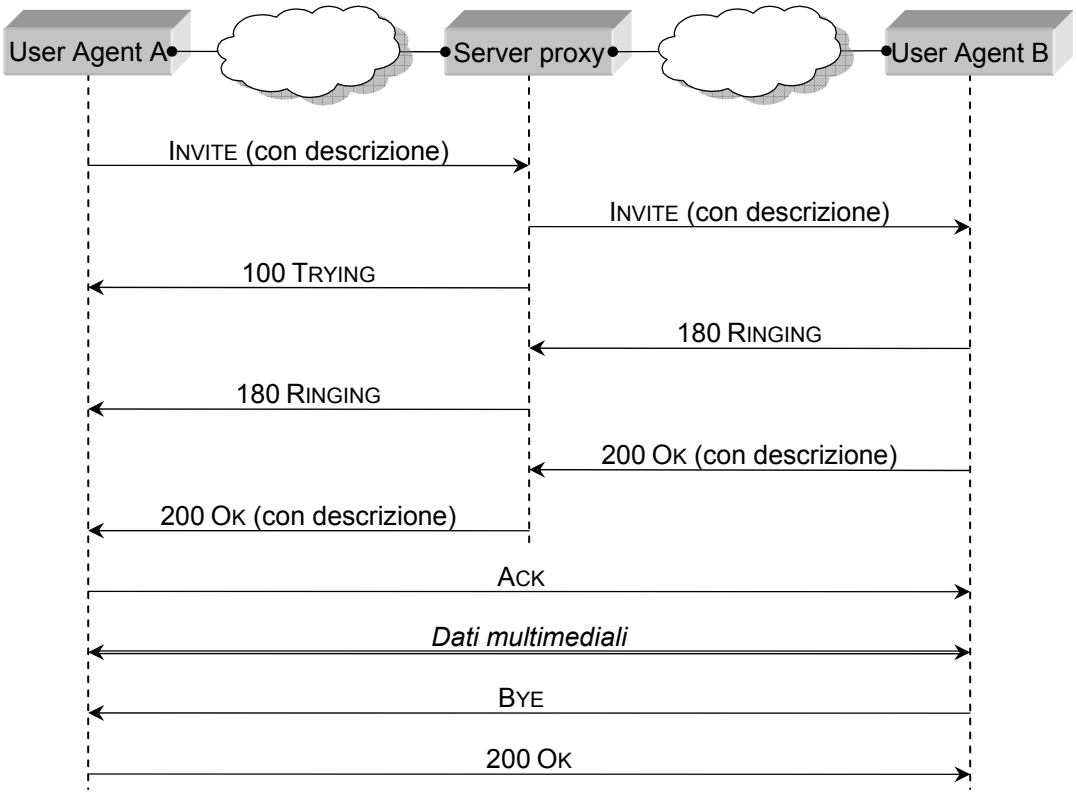

Figura 10. Comunicazione mediante server proxy

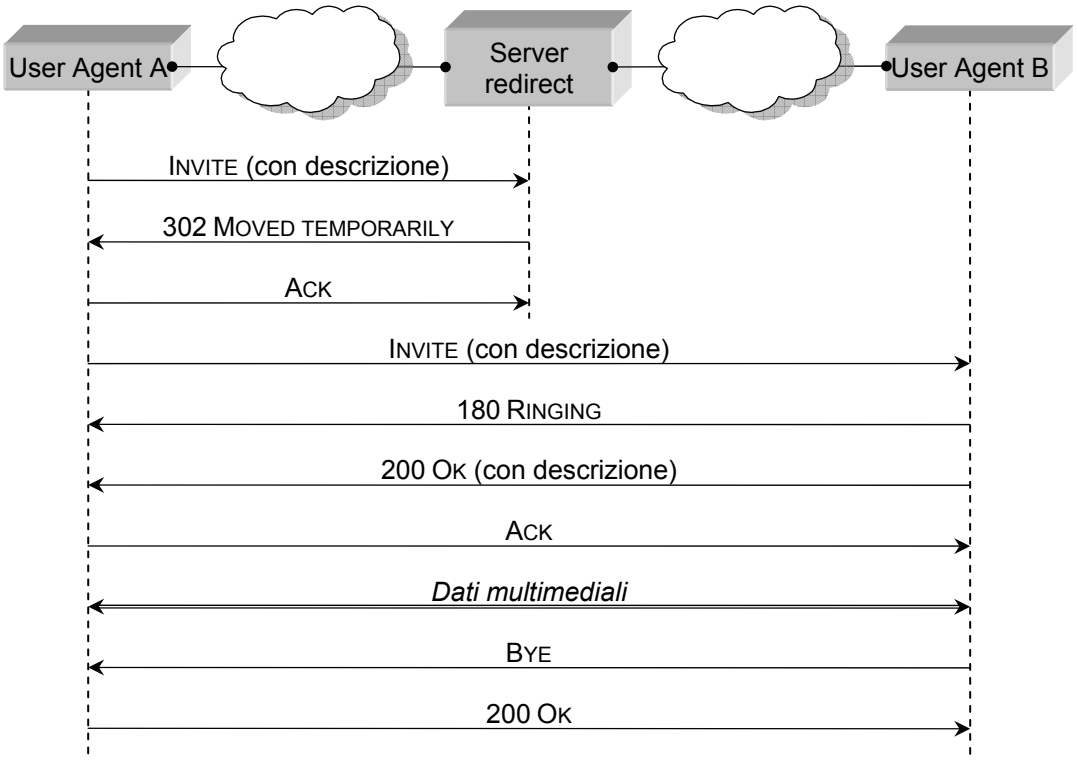

Figura 11. Comunicazione mediante server redirect

registrato, il server risponde con un messaggio di tipo 302 MOVED TEMPORARILY, all'interno del quale è indicato l'indirizzo IP a cui l'utente dovrà essere contattato. Il chiamante ha quindi la possibilità di inviare nuovamente l'INVITE all'interlocutore, e la comunicazione può proseguire in maniera assolutamente analoga al caso in cui la sessione viene instaurata direttamente tra i due user agent.

È altresì prevista la possibilità di utilizzare in cascata diversi server; in questo caso, ognuno dei server attraversati dai messaggi SIP può essere sia un proxy che un redirect. Per evitare la formazione di cicli chiusi, ogni volta che un server proxy inoltra un messaggio di richiesta deve decrementare un apposito contatore in esso contenuto; se il valore di questo diviene pari a 0, il server scarta il pacchetto e invia al mittente una risposta di tipo 483 TOO MANY HOPS. Inoltre, per fare in modo che le risposte seguano esattamente lo stesso percorso delle relative richieste, ogni proxy modifica i messaggi SIP di richiesta inserendo il proprio indirizzo in un apposito campo; il destinatario del messaggio invierà quindi la risposta all'ultimo degli indirizzi indicati, e così via fino a tornare allo user agent che ha generato la richiesta. Si noti come il flusso media segua, in generale, un percorso totalmente distinto rispetto al traffico di segnalazione, e come questo percorso non possa essere stabilito né conosciuto mediante SIP.

Come esempi di messaggi SIP si riportano i seguenti, inclusi all'interno della RFC che definisce il protocollo e relativi a una richiesta di tipo INVITE e alla relativa risposta di tipo 200 OK[5]:

```
INVITE sip:bob@biloxi.com SIP/2.0
Via: SIP/2.0/UDP
  pc33.atlanta.com;branch=z9hG4bK776asdhds
Max-Forwards: 70
To: Bob <sip:bob@biloxi.com>
From: Alice <sip:alice@atlanta.com>;tag=1928301774
Call-ID: a84b4c76e66710@pc33.atlanta.com
CSeq: 314159 INVITE
Contact: <sip:alice@pc33.atlanta.com>
Content-Type: application/sdp
Content-Length: 142
```

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP server10.biloxi.com;
  branch=z9hG4bKnashds8;received=192.0.2.3
Via: SIP/2.0/UDP bigbox3.site3.atlanta.com;
  branch=z9hG4bK77ef4c2312983.1;received=192.0.2.2
Via: SIP/2.0/UDP pc33.atlanta.com;
  branch=z9hG4bK776asdhds;received=192.0.2.1
To: Bob <sip:bob@biloxi.com>;tag=a6c85cf
From: Alice <sip:alice@atlanta.com>;tag=1928301774
Call-ID: a84b4c76e66710@pc33.atlanta.com
CSeq: 314159 INVITE
Contact: <sip:bob@192.0.2.4>
Content-Type: application/sdp
Content-Length: 131
```

Uno dei problemi legati all'instaurazione della sessione mediante il protocollo SIP, riscontrato anche durante la realizzazione del progetto *UniPR-Ptt*, è costituito dall'attraversamento dei NAT (*Network Address Translator*). Questi sistemi vengono utilizzati estensivamente per ovviare al problema del limitato numero di indirizzi IPv4 disponibili. È prassi comune, infatti, non assegnare ad ogni terminale un indirizzo IP pubblico, ma utilizzare all'interno di ogni sottorete un certo insieme di indirizzi privati, in modo che lo stesso indirizzo possa venir assegnato a diversi host collegati a sottoreti distinte. In questo modo, tuttavia, gli host appartenenti a una rete privata non possono scambiare dati direttamente con l'esterno della sottorete; per consentire la

[5] Quelli riportati sono solamente gli header SIP dei messaggi; nel corpo degli stessi, come indicato nel campo *Content-Type*, è presente la descrizione della sessione, effettuata tramite il protocollo SDP e opaca a SIP.

comunicazione è necessario inserire, all'interno del nodo che mette in comunicazione la rete privata con quella pubblica, un server NAT. Questo associa temporaneamente ad ogni coppia indirizzo/porta privata una coppia indirizzo/porta pubblica, e sostituisce, negli header IP e TCP/UDP di ognuno dei pacchetti uscenti, l'indirizzo IP e la porta originari con quelli attualmente assegnati al terminale, compiendo l'operazione opposta per i pacchetti entranti.

Come si è visto, per l'instaurazione della sessione tramite SIP un terminale deve indicare l'indirizzo IP al quale essere contattato; se il terminale dispone di un indirizzo IP privato, quindi, non potrà che inserire quest'ultimo. Ma così facendo i messaggi ad esso diretti saranno spediti ad un indirizzo privato, e quindi, a meno che non provengano dall'interno della stessa sottorete a cui è collegato il terminale, non potranno giungere a destinazione. Il server NAT, infatti, non elabora i dati relativi a protocolli dello strato applicativo quali SIP e SDP. Per risolvere il problema è dunque necessario inserire un ALG (*Application Level Gateway*) che conosca l'indirizzo pubblico e la porta assegnati dal NAT al terminale e li sostituisca a quelli, privati, da esso indicati all'interno dei messaggi SIP. Il sistema scelto per la soluzione del problema sarà descritto con maggiore dettaglio nel seguito.

2.2.2. Descrizione del protocollo SDP

Il protocollo SIP, come si è visto, realizza un sistema mediante il quale gli interlocutori possono accordarsi sulle caratteristiche dei media scambiati, ma non fornisce direttamente gli strumenti per descrivere tali caratteristiche: è infatti previsto che questo compito venga svolto inserendo le opportune informazioni all'interno del corpo dei messaggi SIP. Queste informazioni vengono esaminate solamente dagli user agent, mentre i server SIP non le elaborano in alcun modo; il formato utilizzato per la descrizione può dunque, in teoria, essere definito arbitrariamente dagli sviluppatori. Tuttavia è prassi comune, per garantire una vasta interoperabilità, seguire le regole definite nel protocollo SDP.

SDP (*Session Description Protocol*)[6], a dispetto del nome, non è un vero e proprio protocollo, ma è piuttosto un linguaggio scritto appositamente per la descrizione delle caratteristiche dei flussi media che verranno scambiati all'interno di una sessione multimediale. Tra queste caratteristiche, le principali sono l'identificativo della sessione, l'indirizzo e la porta a cui dovranno essere inviati i dati, i tipi di media coinvolti (audio, video, ecc.) e i relativi formati di codifica.

Anche SDP, come SIP, è un protocollo di tipo testuale: un messaggio SDP è costituito da una sequenza di linee di testo, dette campi. Ogni campo inizia con una lettera (che ne identifica il tipo) seguita dal segno di uguale; dopo di questo possono

[6] Definito nella RFC 2327 del 1998.

essere indicati uno o più parametri (dipendenti dal particolare tipo di campo) separati l'uno dall'altro da uno spazio; per terminare un campo si utilizza la sequenza di caratteri CRLF. All'interno del messaggio, i vari campi devono essere inseriti secondo un ordine ben preciso (in modo da semplificare la rivelazione di errori e l'elaborazione); solo alcuni, tuttavia, devono essere obbligatoriamente presenti in ogni messaggio SDP, mentre altri sono opzionali. Si elencano di seguito i possibili campi SDP, associando a ognuno la descrizione presente nella RFC che definisce il protocollo e indicando i campi obbligatori col pallino pieno e quelli opzionali col pallino vuoto:

- **v** (protocol version)
- **o** (owner/creator and session identifier)
- **s** (session name)
  - **i** (session information)
  - **u** (URI of description)
  - **e** (email address)
  - **p** (phone number)
  - **c** (connection information)
  - **b** (bandwidth information)
- **t** (time the session is active)
  - **r** (repeat times)
  - **z** (time zone adjustments)
  - **k** (encryption key)
  - **a** (session attribute lines)
- **m** (media name and transport address)
  - **i** (media title)
  - **c** (connection information)
  - **b** (bandwidth information)
  - **k** (encryption key)
  - **a** (media attribute lines)

Alcuni campi, come si vede, possono essere presenti più di una volta: questo perché le informazioni in essi contenute possono essere relative a tutti i media oppure specifiche per ognuno dei flussi che compongono lo stream.

Un esempio di messaggio SDP, così come riportato sulla RFC che definisce tale protocollo, è il seguente:

```
v=0
o=mhandley 2890844526 2890842807 IN IP4 126.16.64.4
s=SDP Seminar
i=A Seminar on the session description protocol
u=http://www.cs.ucl.ac.uk/staff/M.Handley/sdp.03.ps
e=mjh@isi.edu (Mark Handley)
c=IN IP4 224.2.17.12/127
t=2873397496 2873404696
a=recvonly
```

```
m=audio 49170 RTP/AVP 0
m=video 51372 RTP/AVP 31
m=application 32416 udp wb
a=orient:portrait
```

Figura 12. Il plug-in SIP nel menu principale

2.2.3. Plug-in e server SIP Nokia

Per poter instaurare connessioni tramite il protocollo SIP fra terminali Symbian Series 60 che non dispongono di un supporto nativo a tale protocollo è necessario installare preventivamente un apposito plug-in, che viene fornito dalla casa produttrice del terminale stesso. Nel caso in questione, poiché lo smartphone a disposizione (un Nokia 6630), contrariamente a modelli più recenti, non integra un modulo per SIP, è stato necessario scaricare e installare il plug-in fornito da Nokia e giunto, al momento dell'inizio del progetto, alla versione 3.0. Lo stesso plug-in è disponibile anche per l'emulatore del terminale, in modo da consentire agli sviluppatori di testare il funzionamento di applicazioni basate su SIP prima della loro esecuzione su uno smartphone reale.

L'installazione e la configurazione del plug-in sull'emulatore, alquanto macchinose, prevedono che l'utente, dopo aver installato l'SDK per Series 60, disabiliti l'utilizzo di IPv6 (convertendo il file di configurazione dell'emulatore in un file di testo, modificandolo opportunamente e riconvertendolo), installi il plug-in SIP (lanciando un particolare eseguibile) e modifichi le opzioni Ethernet dell'emulatore in modo da attivare la modalità promiscua e da impostare l'indirizzo IP che verrà utilizzato[7] e il gateway di default.

Una volta installato il plug-in sullo smartphone o sull'emulatore, nel menu principale compare una nuova icona ad esso associata (figura 12); selezionandola, l'utente ha la possibilità di impostare uno o più profili SIP, che verranno poi utilizzati, mediante opportune chiamate a funzioni, all'interno di tutte le applicazioni basate su SIP. Non è quindi necessario specificare i vari parametri in ognuna di queste applicazioni, ma è sufficiente scegliere di volta in volta il profilo che si intende utilizzare.

I parametri che si possono impostare sono:

- Provider (nome del profilo)
- Modo d'uso servizio (IETF o IMS)
- Punto di accesso predefinito (ovvero provider di default per la connessione dati)

[7] Si noti che l'indirizzo dell'emulatore deve essere differente da quello della macchina che lo ospita.

- Nome utente pubblico
- Uso della compressione (abilitato o disabilitato)
- Modalità di registrazione (attiva sempre o su richiesta)
- Uso della protezione (abilitato o disabilitato)
- Server proxy
  - Indirizzo server proxy
  - Area (realm)
  - Nome utente
  - Password
  - Consenti loose routing (sì o no)
  - Protocollo di trasporto
  - Porta
- Server di registrazione (registrar)
  - Indirizzo server di registrazione
  - Area (realm)
  - Nome utente
  - Password
  - Protocollo di trasporto
  - Porta

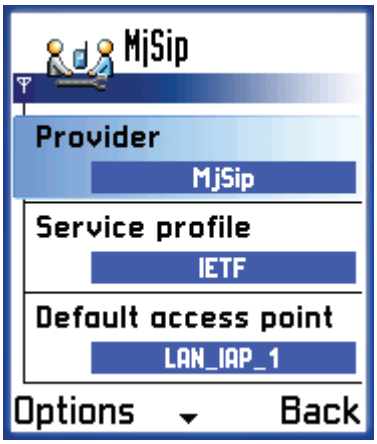

Figura 13. Il plug-in SIP

Una schermata del plug-in SIP è mostrata in figura 13.

Dal punto di vista dello sviluppatore, per la creazione di applicativi che impieghino lo stack SIP è possibile utilizzare le apposite funzioni presenti nella API installata come parte integrante del plug-in SIP. Questa comprende tutte le funzioni necessarie per la gestione dell'handshake SIP, ovvero per l'invio e la ricezione di richieste, risposte, messaggi di errore ecc.

Assieme al plug-in SIP, Nokia fornisce anche un server SIP minimale, mostrato in figura 14, che può essere lanciato su una delle due macchine su cui è in esecuzione un emulatore di terminale o su una terza macchina; questo agisce sia da server proxy che da server registrar, e inoltre può memorizzare su file e mostrare a video i messaggi SIP che transitano attraverso di esso.

Per utilizzare il server SIP Nokia è necessario solamente specificare l'indirizzo o gli indirizzi IP e le porte TCP e UDP su cui il server si mette in attesa. Inoltre, per agevolare lo sviluppo di applicazioni basate su SIP, è possibile forzare il server a rispondere alle richieste in arrivo non come farebbe un normale server, ma inviando messaggi stabiliti dall'utente (BUSY, NOT FOUND, FORBIDDEN ecc.) in modo da simulare i differenti scenari che potrebbero presentarsi.

Il semplice server SIP descritto ha tuttavia diverse limitazioni: in particolare, non è possibile utilizzarlo per l'instaurazione di sessioni tra due terminali attestati su reti private diverse tra loro. Questo perché il server utilizza come indirizzi dei terminali quelli pubblicizzati dai medesimi nel campo *Via* del messaggio SIP di REGISTER e non

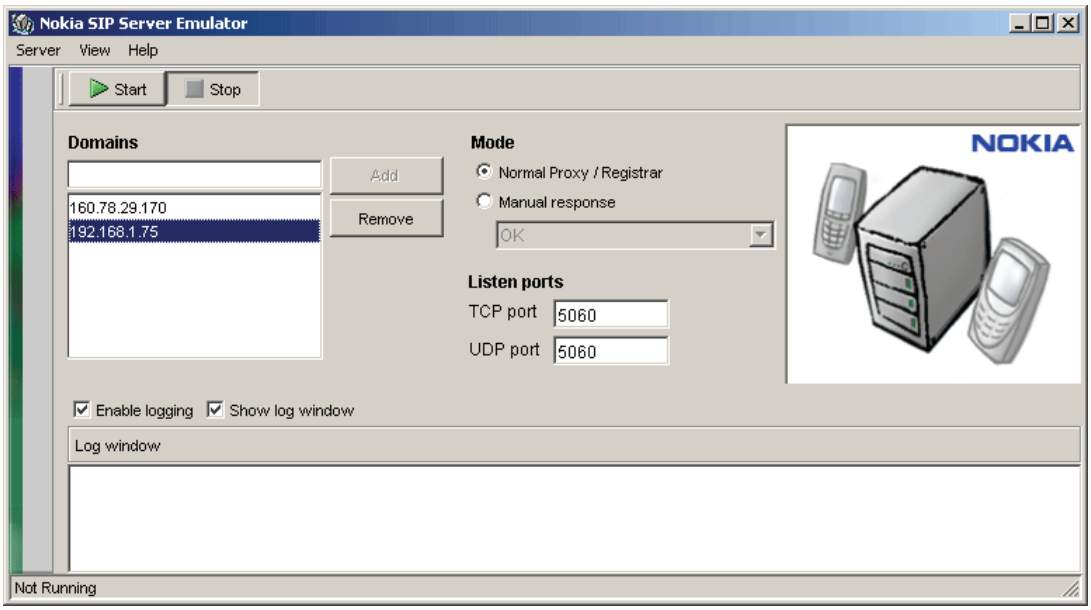

Figura 14. Il server SIP Nokia

quelli da cui arriva effettivamente il messaggio: se il terminale è attestato su una rete privata, esso pubblicizzerà il proprio indirizzo privato, e quindi, se questo indirizzo è fornito all'altro terminale, la comunicazione tra i due risulterà impossibile salvo nel caso particolare in cui entrambi i terminali siano collegati alla stessa rete privata. Siccome il codice dell'emulatore di server SIP non è reso disponibile da Nokia, si è reso necessario l'uso di un server SIP differente, come verrà spiegato in seguito.

2.3. RTP e RTCP

Terminata l'instaurazione della comunicazione mediante SIP, è possibile iniziare lo scambio di dati multimediali. Tipicamente per il trasporto di questo tipo di dati in tempo reale si utilizza il protocollo RTP; questo è stato implementato anche all'interno del progetto *UniPR-Ptt*.

Il protocollo RTP (*Real-time Transport Protocol*)[8] definisce le funzioni necessarie per il trasporto di informazioni multimediali in tempo reale tra due o più terminali. RTP è considerato un protocollo appartenente allo strato di trasporto, anche se in realtà non è un protocollo di trasporto completo ma ne utilizza altri, estendendone le funzionalità. In teoria RTP non dipende dal particolare protocollo di trasporto sottostante, ma tipicamente le trame RTP vengono trasportate all'interno di pacchetti UDP, in quanto l'uso di TCP risulta impossibile per applicazioni in tempo reale: TCP prevede infatti

[8] Originariamente definito nella RFC 1889 del 1996, resa obsoleta nel 2003 dalla RFC 3550.

funzioni (quali il controllo di congestione e la ritrasmissione dei dati) che non risultano di alcuna utilità per il trasporto di dati in tempo reale, e d'altra parte non supporta trasmissioni multicast ma unicamente comunicazioni punto-punto. UDP, però, non realizza alcune delle funzioni necessarie per la trasmissione di flussi multimediali in tempo reale, quali l'inserimento all'interno dei pacchetti dei numeri di sequenza o di informazioni di sincronizzazione. Per questo motivo, è necessario introdurre al di sopra di UDP un protocollo (quale appunto RTP) che realizzi le funzionalità utili per lo scopo desiderato.

In particolare, RTP definisce i metodi per indicare, all'interno di ognuno dei pacchetti contenenti dati multimediali, il formato dei dati trasportati, il numero di sequenza del pacchetto, l'istante temporale in cui questo dovrà essere riprodotto e un identificativo della sessione di cui il pacchetto fa parte. Questo protocollo, tuttavia, non garantisce che i dati trasmessi arrivino a destinazione senza errori e nell'ordine corretto, e richiede che le operazioni di frammentazione e ricostruzione dei pacchetti siano svolte da protocolli sottostanti; inoltre, venendo elaborato unicamente dai terminali e non dai nodi intermedi, RTP non permette di riservare risorse per la comunicazione. Il protocollo RTP può essere utilizzato per trasportare ogni tipo di media in qualunque formato di codifica, e può venire usato sia per comunicazioni punto-punto che per trasmissioni multicast. In ogni caso, RTP supporta unicamente flussi unidirezionali: se la comunicazione è bidirezionale occorre cioè definire due flussi RTP differenti e indipendenti l'uno dall'altro. Inoltre, se la comunicazione comprende diversi media, ognuno di questi deve essere trasportato in un diverso flusso RTP.

Il protocollo RTP si basa sul concetto di sessione, che indica univocamente l'associazione tra un insieme di utenti che comunicano tra loro; una sessione è definita dall'indirizzo IP (unicast o multicast) di destinazione e da una coppia di porte, la più bassa delle quali viene utilizzata per i dati relativi al protocollo RTP, mentre la più alta si usa per le relative informazioni di controllo, scambiate mediante RTCP. Quest'ultimo, che verrà descritto più avanti, è il protocollo di controllo associato a RTP.

Nel caso di comunicazioni unicast, i flussi RTP vengono scambiati direttamente tra i due terminali che partecipano alla sessione; invece, se le sessioni comprendono più di due utenti, è possibile sia inviare direttamente i dati all'indirizzo multicast che identifica tutti i destinatari, sia utilizzare un apposito server che effettui l'inoltro dei pacchetti a tutti i relativi indirizzi unicast, come mostrato in figura 15. Il protocollo RTP prevede infatti la presenza di sistemi detti *RTP relay*, che hanno appunto la funzione di ricevere in ingresso i pacchetti

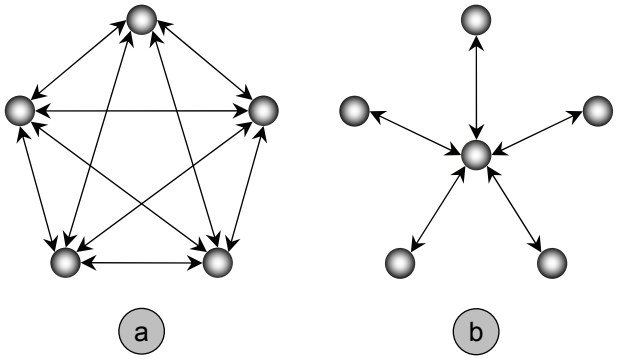

Figura 15. Scambio di dati tra più utenti senza relay (a) e con relay (b)

provenienti da una o più sorgenti, effettuare le eventuali trasformazioni necessarie e inoltrarli verso una o più destinazioni. Questi sono necessari non solo per risparmiare banda nel caso di sessioni unicast che coinvolgano più di due interlocutori, ma anche per attraversare eventuali firewall o per effettuare la transcodifica dei flussi nel caso in cui non tutti gli utenti utilizzino gli stessi codec. Gli RTP relay possono essere suddivisi in *mixer* (che ricevono in ingresso vari flussi e li miscelano in modo da crearne uno unico in uscita, effettuando le necessarie transcodifiche e sincronizzando i flussi) e *translator* (che operano su singoli flussi variandone unicamente il formato di codifica).

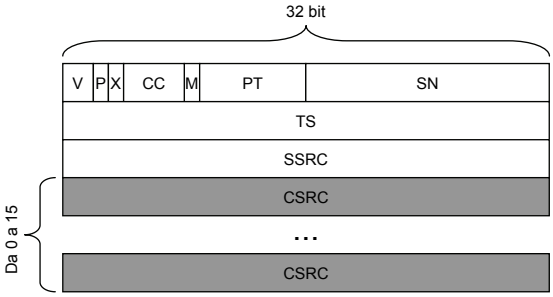

Figura 16. L'header RTP

Il protocollo RTP prevede che ogni pacchetto contenente dati multimediali venga fatto precedere dall'header schematizzato in figura 16[9]. I campi che compongono l'header RTP sono i seguenti:

- **V** (Version, 2 bit): versione del protocollo RTP utilizzata; la versione attualmente in uso è la 2.
- **P** (Padding, 1 bit): se posto a 1, significa che il pacchetto contiene al termine uno o più byte di padding; il loro numero è indicato nell'ultimo dei byte di padding.
- **X** (eXtension, 1 bit): se posto a 1, significa che l'header base è seguito da un'estensione, nella quale si possono inserire informazioni riguardanti nuove funzioni implementabili in aggiunta a quelle previste dal protocollo.
- **CC** (CSRC Count, 4 bit): indica il numero di campi CSRC (v. oltre) contenuti all'interno dell'header SDP.
- **M** (Marker, 1 bit): se posto a 1, indica che il contenuto del pacchetto si riferisce a un evento significativo, quale ad esempio il termine di una trama all'interno dello stream.
- **PT** (Payload Type, 7 bit): identifica il formato dei dati contenuti nel payload del pacchetto RTP. I codici corrispondenti ai formati più comuni sono definiti dall'IETF[10], ma è anche possibile specificarne altri in maniera dinamica associando al formato di codifica che si utilizza uno dei codici non assegnati ad alcun formato. Questa operazione deve però essere effettuata mediante protocolli diversi da RTP; in particolare può essere compiuta inserendo l'attributo *rtpmap*, seguito dal codice scelto e dal formato che si vuole associare a quel codice, all'interno del campo *media attribute* dell'header SDP contenuto nei messaggi SIP scambiati durante

[9] In figura sono rappresentati con sfondo bianco i campi che devono essere obbligatoriamente presenti, e con sfondo grigio quelli opzionali.

[10] Originariamente nella RFC 1890 del 1996, resa poi obsoleta nel 2003 dalla RFC 3551.

l'handshake. Questa procedura è necessaria, ad esempio, per il formato AMR, che sarà utilizzato nel progetto *UniPR-Ptt*.

- **SN** (Sequence Number, 16 bit): indica il numero di sequenza del pacchetto relativamente alla sessione corrente. Questo campo viene incrementato di 1 ogni volta che si spedisce un pacchetto RTP, a partire da un valore iniziale generato in modo casuale, e può servire al ricevitore per rivelare eventuali perdite di pacchetti o per rimettere nel giusto ordine pacchetti ricevuti fuori sequenza.
- **TS** (Time Stamp, 32 bit): rappresenta l'istante in cui deve iniziare la riproduzione del pacchetto RTP in esame. Tale valore temporale non è però indicato esplicitamente: ogni volta che si spedisce un pacchetto, infatti, questo campo viene incrementato del numero di campioni, codificati nel formato scelto, che sono contenuti nel pacchetto stesso. Conoscendo la frequenza con cui vengono generati i campioni, da questa informazione è possibile risalire all'istante in cui il pacchetto deve essere riprodotto. Anche in questo caso, il valore iniziale del campo è scelto in maniera casuale.
- **SSRC** (Synchronization SouRCe, 32 bit): identifica univocamente la sorgente (intesa sia come terminale che come relay) dei dati contenuti all'interno del pacchetto. Questo campo è composto da un numero scelto a caso, ad ogni sessione, da ognuno dei nodi che trasmettono dati, in modo che ognuno di tali nodi abbia un diverso identificativo. Esso viene poi mantenuto inalterato per l'intera durata della sessione.
- **CSRC** (Contributing SouRCe, 32 bit): questo campo, che può non essere presente oppure essere ripetuto da una a 15 volte, viene inserito dai mixer RTP per indicare l'SSRC di ognuno dei flussi tributari che, combinati insieme, hanno dato origine al blocco di dati contenuti nel pacchetto. Le sorgenti dei tributari, a loro volta, possono essere sia terminali che mixer. Il numero di CSRC presenti nell'header RTP può essere determinato, come si è visto, mediante il campo *CSRC Count*.

Come si è accennato, il protocollo di controllo associato a RTP è RTCP (*RTP Control Protocol*)[11]. L'uso di RTCP in abbinamento a RTP non è obbligatorio (e infatti tale protocollo non è stato inserito nel progetto *UniPR-Ptt*), ma spesso può risultare utile. Le sue funzioni principali consistono nel fornire a tutti coloro che partecipano alla sessione una stima della qualità del servizio percepita da un utente, nel sincronizzare media differenti (tipicamente audio e video) provenienti da una stessa sorgente, nell'identificare i partecipanti a una comunicazione di tipo multicast e nel permettere un controllo della sessione.

Per svolgere queste funzioni, RTCP prevede che tutti i partecipanti a una sessione trasmettano a intervalli regolari (la cui durata dipende dal numero dei partecipanti) particolari pacchetti contenenti informazioni di controllo. Alle varie tipologie di

[11] RTCP è definito all'interno delle stesse RFC che riguardano RTP.

informazioni trasportate corrispondono diverse categorie di pacchetti: un pacchetto RTCP è infatti composto da un header simile a quello utilizzato da RTP seguito da un payload specifico per il particolare tipo di pacchetto. Il protocollo RTCP definisce i seguenti tipi di pacchetto:

- **Sender Report**: questo pacchetto viene creato dagli utenti che stanno trasmettendo i flussi multimediali per inviare a coloro che li ricevono informazioni o statistiche riguardanti i flussi medesimi, come ad esempio il numero di byte trasmessi, i timestamp ecc.
- **Receiver Report**: questo pacchetto, duale rispetto al precedente, viene creato dagli utenti che stanno ricevendo i flussi multimediali per inviare a coloro che li trasmettono informazioni riguardanti la qualità del flusso ricevuto, come ad esempio la percentuale di pacchetti persi, il jitter, il ritardo ecc. Gli utenti che ricevono questi pacchetti possono (a loro discrezione) variare le caratteristiche del flusso di dati trasmesso in modo da adattarlo alle condizioni attuali del canale di comunicazione: ad esempio, se la frazione di pacchetti che vengono persi è troppo elevata, il mittente può ritenere che vi sia una congestione e quindi può decidere di ridurre il bitrate.
- **Source Description**: questo pacchetto viene utilizzato per trasportare informazioni addizionali riguardanti un utente, le quali possono essere utilizzate per associare tra loro i vari flussi RTP inviati (con SSRC diversi) dalla stessa sorgente.
- **Bye**: questo pacchetto viene utilizzato dall'utente che sta trasmettendo i dati per comunicare la sua decisione di abbandonare la sessione.
- **Application Specific**: questo pacchetto, come suggerisce il nome, trasporta informazioni di controllo specifiche della particolare applicazione utilizzata per l'invio e la ricezione dei flussi.

Per concludere, è da notare come né RTP né RTCP includano meccanismi per fornire una determinata qualità del servizio o per riservare le risorse necessarie alla comunicazione. Inoltre questi protocolli non garantiscono che i pacchetti inviati siano ricevuti nella giusta sequenza e allo stesso ritmo con cui sono stati emessi, ma si limitano a fornire al destinatario le indicazioni necessarie per effettuarne la riproduzione agli istanti corretti; quest'ultimo, però, non è tenuto a prendere necessariamente in considerazione tali indicazioni. Infine, il controllo di congestione non è svolto direttamente ma è demandato all'applicativo che sta trasmettendo i dati, il quale può decidere autonomamente se e come affrontare eventuali problemi.

2.4. PTT e PoC

In generale, si definisce come *Push-to-talk* (abbreviandolo in PTT) un metodo di comunicazione tipico dei sistemi che operano in modalità half-duplex, caratterizzato dal fatto che un utente, per poter parlare, deve premere (ed eventualmente tenere premuto) un apposito pulsante presente sul proprio terminale. Durante questa fase il segnale viene

trasmesso agli altri terminali impegnati nella conversazione, ma non è possibile ricevere contemporaneamente i segnali emessi da questi ultimi. Per poter ascoltare eventuali comunicazioni provenienti dagli altri utenti, è necessario interrompere la trasmissione rilasciando il pulsante.

Il metodo descritto è stato implementato non solo negli *walkie-talkie* ma anche in alcuni tra i primi sistemi di telefonia mobile[12], a causa delle sue limitate richieste in termini di banda di trasmissione e di capacità di elaborazione. Negli ultimi anni diversi produttori di dispositivi mobili, tra i quali Nokia, hanno proposto un'implementazione di tale sistema di comunicazione sui propri terminali, naturalmente offrendola come caratteristica supplementare alle normali comunicazioni in full-duplex. In tali casi il segnale vocale viene trasmesso sulle reti dati mobili GPRS o UMTS, e il servizio prende il nome di *Push-to-talk over Cellular* o PoC.

I principali vantaggi per gli utenti consistono nella possibilità di effettuare facilmente conferenze, ovvero comunicazioni a cui partecipino più di due interlocutori, anche aggiungendone di nuovi a una sessione già in corso, e nel sistema di tariffazione basato sulla quantità di dati scambiati anziché sulla durata complessiva della sessione: nei periodi di tempo in cui nessun interlocutore trasmette dati, la tariffazione si interrompe ma la connessione resta attiva, in modo che non appena uno degli utenti intende ricominciare a parlare tutti gli altri lo possano ascoltare quasi istantaneamente[13]. Inoltre, rispetto a una normale chiamata vocale su rete GSM, i tempi di instaurazione della connessione sono più rapidi e il sistema è molto più flessibile. Tale flessibilità potrebbe essere sfruttata, ad esempio, per utilizzare codec audio con prestazioni migliori (e quindi bitrate inferiori) rispetto a quelle tipiche dello standard GSM, oppure metodi di crittografia più sicuri, o ancora per trasmettere, in aggiunta o in alternativa al segnale vocale, brevi messaggi di testo o piccoli file.

Dal punto di vista degli operatori, il supporto delle funzionalità di Push-to-talk può rappresentare un modo per ampliare e differenziare la gamma dei servizi offerti mediante un sistema efficiente, semplice ed economico da implementare ma che, essendo complementare e non sostitutivo rispetto ai classici servizi di telefonia in full-duplex, può permettere di rispondere alle esigenze di nuove categorie di clienti e di supportare nuovi scenari operativi. Questo sistema appare adeguato in particolare per tutte le situazioni in cui è richiesta la comunicazione istantanea tra più utenti distribuiti in un'ampia area geografica, che debbano tenersi in contatto per un tempo elevato ma con lunghi periodi di inattività. Le categorie di utenti a cui il servizio si rivolge sono corrieri, distributori, autonoleggi, aziende di trasporto pubblico, lavoratori che operano all'interno di grandi complessi industriali, porti, aeroporti, ospedali, cantieri, centri

[12] Un esempio è lo standard MTS (*Mobile Telecommunication System*), introdotto nel 1946 da parte di AT&T e Southwestern Bell nella città si Saint Louis, Missouri.

[13] In realtà, tra l'istante in cui un utente parla e quello in cui le sue parole vengono ricevute dagli interlocutori può passare un certo tempo, che dipende dall'operatore di rete utilizzato.

commerciali ecc., ma anche escursionisti, gruppi di teenager, famiglie, giocatori on line, o comunque gruppi di persone che, per lavoro o per svago, abbiano la necessità di condividere informazioni con semplicità e rapidità.

Il servizio di *Push-to-talk over Cellular* costituisce una delle applicazioni comprese nell'*Internet Multimedia Subsystem* (IMS), un dominio definito dal 3GPP all'interno della Release 5 dell'UMTS con lo scopo di fornire servizi integrati di trasporto di voce e dati su reti a pacchetto basate su IP.

Per avere un esempio delle modalità d'uso del sistema, se ne può analizzare l'implementazione proposta da Nokia. Questa prevede che ogni utente, quando è disposto a partecipare ad una sessione, si registri preventivamente su un opportuno server. Fatto ciò, per avviare una conversazione in modalità Push-to-talk, è sufficiente selezionare un contatto o un gruppo da un elenco definito in precedenza (per mezzo del quale è anche possibile verificare lo stato attuale degli altri utenti) e tener premuto un apposito tasto finché non sente un tono; a quel punto, sempre tenendo premuto lo stesso tasto, l'utente può iniziare a parlare, mentre rilasciandolo può ascoltare le comunicazioni provenienti dagli altri partecipanti alla sessione. La chiamata ha inizio quasi immediatamente, senza neppure bisogno di attendere che il chiamato prema alcun tasto: si suppone infatti che la registrazione implichi la disponibilità a ricevere chiamate. Se più utenti chiedono di poter parlare nello stesso tempo, viene soddisfatta la richiesta giunta per prima. Per il supporto del servizio di Push-to-talk, gli operatori devono inserire all'interno della rete appositi server che gestiscano le operazioni di registrazione e presence, di inizio e fine della sessione e di scambio dei dati, soprattutto per quanto riguarda il controllo delle sessioni che coinvolgono più di due utenti. Attualmente nessuno degli operatori mobili che operano sul territorio italiano supporta tale servizio.

2.4.1. Modelli di push-to-talk

Prima di realizzare, nel progetto *UniPR-Ptt*, un applicativo che permettesse la trasmissione di dati audio in modalità push-to-talk, è stato necessario fissarne con esattezza le specifiche, e in particolare stabilire quale modello di push-to-talk dovesse essere implementato. Infatti una volta deciso che la comunicazione avviene in modalità half-duplex è ancora possibile definire le regole con cui si determina quale degli utenti abbia diritto a parlare. Sono stati identificati cinque differenti modelli di push-to-talk, e di ognuno sono stati studiati vantaggi e svantaggi.

In tutti i modelli creati si è stabilito, per semplicità, che la comunicazione possa avvenire solamente tra due interlocutori; l'eventuale estensione al caso in cui alla stessa sessione partecipino più di due utenti viene lasciata come possibile sviluppo futuro. Inoltre, nella realizzazione di tutti i modelli si è deciso che, prima di iniziare la chiamata, gli utenti si debbano registrare su un opportuno server, manifestando così la propria disponibilità a prendere parte a una sessione. In seguito, come mostrato in figura 17, se un utente intende iniziare a parlare con un altro, manda a quest'ultimo un

apposito messaggio di invito. L'utente che riceve l'invito ha la possibilità di accettarlo o rifiutarlo; se lo accetta può iniziare immediatamente a parlare, mentre l'utente che ha inviato il messaggio si pone inizialmente in ricezione. Le differenze tra i vari schemi riguardano invece le fasi in cui la possibilità di trasmettere passa da un utente all'altro.

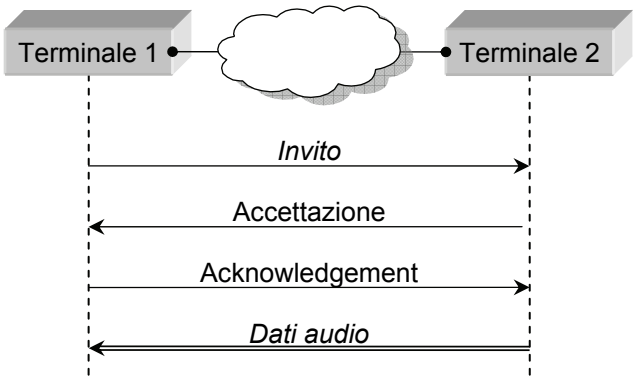

Figura 17. Inizio della comunicazione

Il primo modello esaminato, schematizzato in figura 18, prevede che l'utente che si trova in fase di trasmissione, premendo un apposito pulsante sul proprio dispositivo, interrompa la trasmissione ed invii all'interlocutore un messaggio in cui lo informa dell'intenzione di iniziare una fase di riproduzione; il terminale dell'interlocutore, al momento della ricezione di tale messaggio, a sua volta interrompe la riproduzione dei dati audio ricevuti, invia all'altro terminale un messaggio di acknowledgement e subito dopo inizia a trasmettere il segnale proveniente dal microfono. La pressione del pulsante sul dispositivo che si trova in fase di ricezione non ha alcun effetto.

Questo modello presenta il vantaggio di essere molto simile a quanto avviene normalmente durante una conversazione telefonica, nonché quello di permettere, mediante il meccanismo di acknowledgement, la verifica dell'effettiva raggiungibilità dell'interlocutore ad ogni cambio di stato. D'altra parte, l'utente che si trova in fase di ricezione non può in alcun modo iniziare la trasmissione finché l'interlocutore non decide di interrompere l'invio dei dati; inoltre non possono esistere intervalli di tempo in cui nessuno degli utenti trasmette i propri dati, il che può portare a sprechi di banda; infine, con questo modello risulta difficile estendere la comunicazione a più di due interlocutori, poiché non sarebbe banale decidere quale di essi debba iniziare a parlare allorché l'utente precedentemente attivo termina la trasmissione.

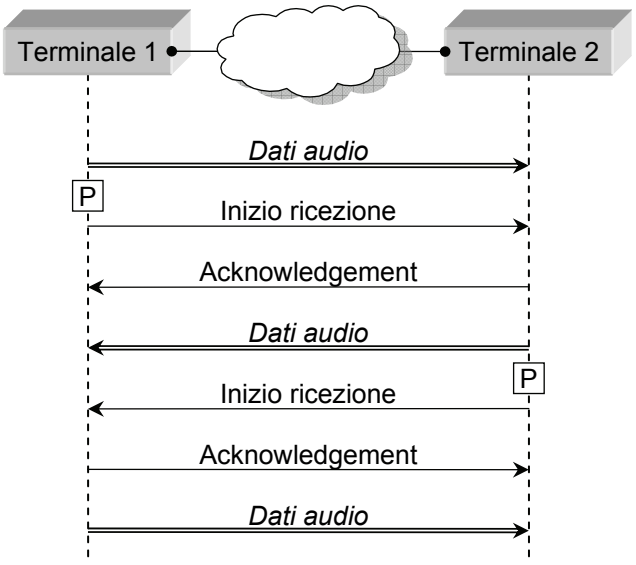

Figura 18. Primo modello di PTT

Il secondo modello, duale rispetto al precedente e schematizzato in figura 19, prevede che il controllo della sessione appartenga all'utente che si trova in fase di ricezione: questi infatti, quando desidera parlare, può

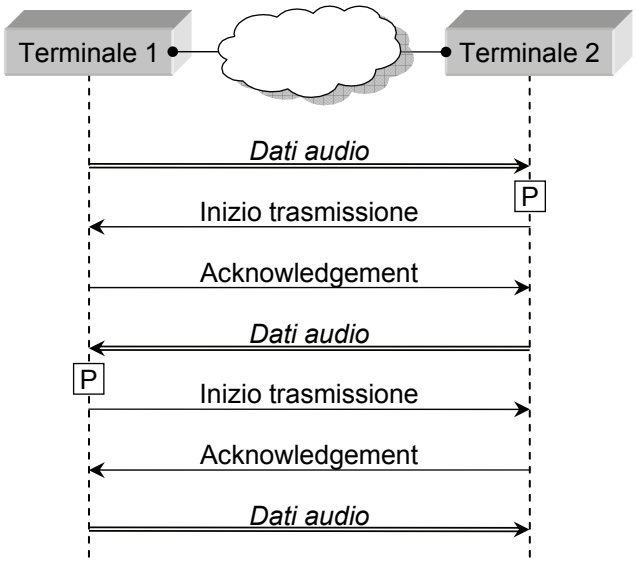

Figura 19. Secondo modello di PTT

premere un tasto e iniziare così a trasmettere i propri dati, mentre l'interlocutore interrompe l'invio e si pone in ricezione. Anche in questo caso, come nel precedente, la variazione di stato prevede l'invio di un messaggio diretto all'altro terminale e la ricezione di un acknowledgement da quest'ultimo.

Questo modello, più vicino del precedente al concetto di "push-to-talk" (nel senso che l'utente preme il pulsante prima di iniziare a parlare anziché dopo aver finito) presenta il vantaggio di consentire all'utente di trasmettere i propri dati quando lo ritiene opportuno e non quando l'interlocutore glielo permette; mediante l'introduzione di un opportuno server, ora potrebbe anche essere possibile supportare comunicazioni che coinvolgano più di due interlocutori, in modo che in qualsiasi momento uno di questi parli mentre gli altri ascoltano; inoltre, l'uso dei messaggi di acknowledgement permette di verificare l'effettivo stato dell'interlocutore prima di modificare lo stato del sistema. L'impossibilità di gestire situazioni in cui nessuno degli utenti parla, con gli sprechi di banda che ciò comporta, nonché l'impossibilità per l'utente attualmente attivo di interrompere la trasmissione a suo piacimento sono i principali svantaggi di questo modello.

Il terzo schema esaminato (rappresentato in figura 20) costituisce una sintesi dei precedenti: in questo caso, infatti, sia l'utente che sta parlando sia quello che sta ascoltando, premendo un tasto sul terminale possono modificare contemporaneamente sia il proprio stato che quello dell'interlocutore.

Questo modello, benché maggiormente flessibile rispetto a quelli descritti in precedenza, rimane comunque più simile a quello di una normale telefonata che non a quello tipico di un sistema realmente push-to-talk. In quanto tale, questo schema mantiene il

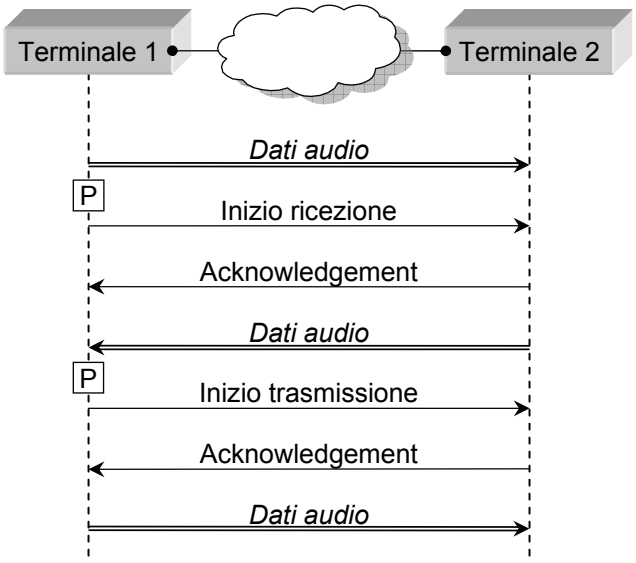

Figura 20. Terzo modello di PTT

principale difetto di quelli precedentemente analizzati, ovvero l'impossibilità di interrompere l'invio dei dati negli intervalli di tempo in cui nessuno degli interlocutori ha necessità di parlare; inoltre, come nel primo caso esaminato, non è facile estendere la sessione a più di due utenti.

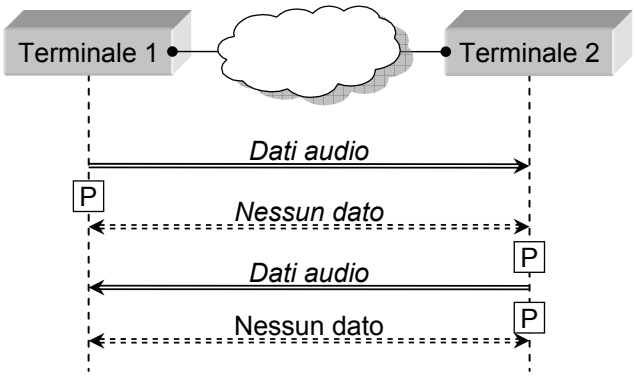

Figura 21. Quarto modello di PTT

Per superare il primo di questi problemi, è stato preso in considerazione un quarto modello, schematizzato in figura 21, in cui ognuno degli interlocutori può modificare il proprio stato indipendentemente da quello dell'altro. In altre parole, se l'utente attualmente in fase di trasmissione intende interrompere l'invio dei dati, premendo un pulsante sul proprio terminale compie questa operazione e inizia una fase di ricezione, senza che l'interlocutore ne venga informato in alcun modo. Analogamente, per terminare la fase di ricezione dei dati provenienti dal terminale dell'interlocutore e iniziare a trasmettere i propri, è necessario unicamente premere il medesimo tasto, senza che ciò provochi alcuna variazione nello stato dell'interlocutore.

Questo schema, tipico di un sistema in push-to-talk, consente a ognuno degli interlocutori di agire indipendentemente dall'altro e permette di risparmiare banda nelle fasi in cui nessuno degli utenti sta parlando. D'altra parte un modello di questo tipo può portare a situazioni in cui entrambi gli interlocutori trasmettono contemporaneamente dati senza che nessuno di essi possa riceverli, e senza che si possano neppure rendere conto di questo stato di cose; inoltre, mancando qualunque meccanismo di conferma della ricezione dei dati, non è possibile sapere se l'interlocutore è ancora connesso o se, per cause indipendenti dalla sua volontà, è divenuto irraggiungibile. Questo modello, comunque, potrebbe essere esteso con semplicità al caso in cui la comunicazione avviene tra più di due utenti; per raggiungere tale scopo sarebbe tuttavia necessario introdurre un server attraverso il quale passino tutti i messaggi scambiati, che dovrebbe essere in grado di inoltrare i dati provenienti da uno dei terminali verso tutti gli altri e di gestire le situazioni in cui due o più utenti vogliano parlare contemporaneamente.

Se però ci si limita a considerare scenari in cui la comunicazione avviene tra due soli interlocutori, è possibile elaborare un quinto modello, schematizzato in figura 22, in cui, analogamente al caso precedente, ognuno degli utenti può variare a piacimento il proprio stato, ma in modo che tale variazione venga quanto meno comunicata all'interlocutore. In questo caso, se colui che si trova in fase di trasmissione ha intenzione di interrompere questa operazione e di iniziare a ricevere l'eventuale segnale proveniente dall'altro terminale, a seguito della pressione dell'apposito tasto si interrompe la trasmissione dei dati audio e si invia un messaggio che segnala l'intenzione di modificare il proprio stato. Il terminale dell'interlocutore, ricevuto tale

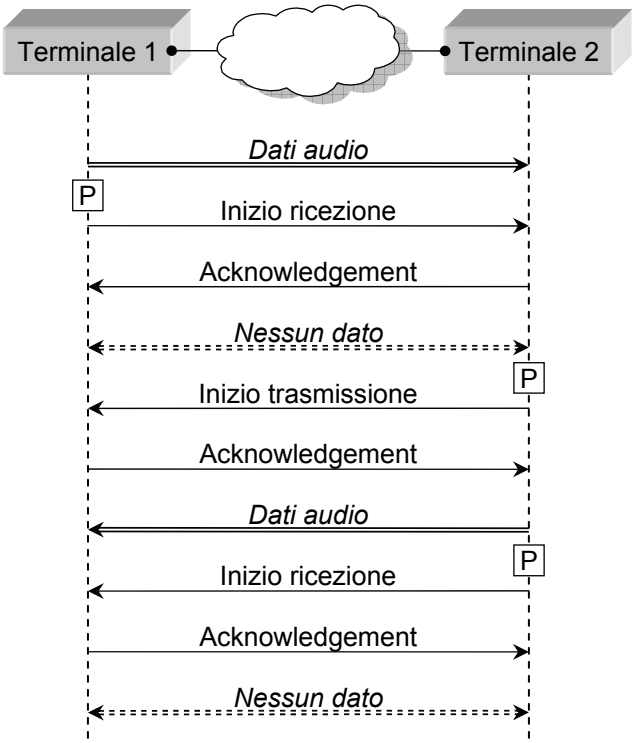

Figura 22. Quinto modello di PTT

messaggio, invia un acknowledgement ma non interrompe l'operazione che stava svolgendo. Solo al momento della ricezione dell'acknowledgement, il primo terminale può iniziare effettivamente l'operazione richiesta. La medesima sequenza di operazioni viene eseguita quando l'utente che si trova in fase di ricezione intende iniziare la trasmissione dei propri dati.

Il principale vantaggio derivante dall'utilizzo dello schema descritto è rappresentato dalla possibilità, per l'utente, di conoscere lo stato del proprio interlocutore: è infatti possibile indicare sul display del terminale se questi si trova attualmente in fase di trasmissione o di ricezione. In base a tale informazione, l'utente può decidere se parlare o ascoltare, senza tuttavia essere obbligato a compiere l'una o l'altra operazione. Inoltre, grazie all'introduzione dei messaggi di acknowledgement, è possibile verificare l'effettiva raggiungibilità dell'interlocutore all'inizio e alla fine di ogni fase di trasmissione: in tal modo non si ha, evidentemente, la garanzia che questi riceva correttamente tutti i dati trasmessi, ma si può comunque ottenere un'utile indicazione su ciò che accade negli istanti in cui si intende variare il proprio stato. Infine, con questo modello viene mantenuta la possibilità di interrompere la trasmissione di dati in ambedue le direzioni negli intervalli di tempo in cui nessuno degli utenti intende parlare.

L'ultimo modello esposto è quello che è stato implementato nel progetto *UniPR-Ptt*; il precedente, che può essere visto come una semplificazione di questo, è stato poi realizzato semplicemente modificando alcune delle funzioni in esso contenute, in modo da agevolarne un eventuale sviluppo futuro che potesse supportare le comunicazioni tra più di due utenti.

3. Descrizione dello sviluppo

Il punto di partenza per la realizzazione del progetto *UniPR-Ptt* è stato l'esempio *Chipflip*, fornito da Nokia in abbinamento al plug-in per SIP con lo scopo di fornire agli sviluppatori un'applicazione pratica delle funzionalità presenti nella API per SIP e di esemplificare le modalità di utilizzo delle stesse. Si è dunque proceduto per prima cosa allo studio dell'esempio in questione; questo è stato poi modificato in modo da inserire le funzioni necessarie all'esecuzione delle nuove operazioni e da rimuovere quelle specifiche dell'esempio e non più necessarie per lo svolgimento dei compiti voluti.

Nel seguito quindi, dopo una rapida presentazione delle funzionalità e delle modalità d'uso dell'applicazione realizzata, verranno mostrate le caratteristiche fondamentali del progetto *Chipflip* e verranno descritte le principali modifiche apportate a tale programma per modificarne le funzionalità, nell'ordine cronologico in cui sono state effettivamente introdotte. Successivamente l'applicativo *UniPR-Ptt* verrà presentato dal punto di vista dello sviluppatore, mostrando come siano state effettivamente implementate le funzioni che il programma svolge e motivando le principali scelte progettuali compiute.

3.1. *Uso del programma* UniPR-Ptt

Scopo del progetto *UniPR-Ptt*, come si è detto, è la realizzazione di un programma che permetta di effettuare lo streaming di dati audio in modalità push-to-talk tra due terminali mobili dotati di sistema operativo Symbian. Dal punto di vista dell'utente, per l'uso del programma è necessario dapprima installare il plug-in Nokia per SIP

Figura 1. *UniPR-Ptt* nel menu principale

(qualora il supporto di tale protocollo non sia già previsto in origine sul terminale), e successivamente utilizzare quest'ultimo per impostare almeno un profilo SIP valido. Terminate queste operazioni preliminari, è possibile installare e lanciare il programma in questione, che si baserà sui profili definiti in precedenza per effettuare la registrazione a un apposito server SIP. Installando il programma, viene aggiunta la corrispondente icona al menu principale del telefono, come mostrato in figura 1.

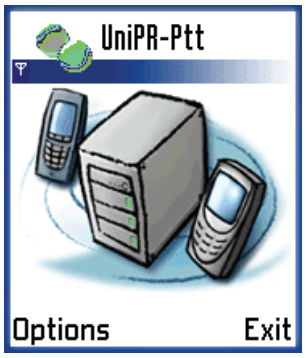

Figura 2. La schermata iniziale di *UniPR-Ptt*

Selezionando tale icona si lancia il programma, trovandosi così di fronte all'interfaccia mostrata in figura 2. Attraverso di essa, in questa fase, è possibile unicamente effettuare la registrazione sul server SIP o terminare il programma tornando al menu principale. Per registrare uno dei profili impostati in precedenza, è necessario entrare nel menu *Opzioni* e selezionare la voce *Registra*, come mostrato in figura 3. Il sistema presenta quindi l'elenco dei profili disponibili (figura 4), dal quale è possibile selezionare quello che si desidera utilizzare nella sessione corrente. Durante l'operazione di registrazione, che può richiedere diversi secondi in quanto comporta l'apertura della connessione alla rete, viene visualizzato il messaggio di attesa mostrato in figura 5. Se la registrazione termina con successo, l'utente è informato mediante un'apposita finestra, mostrata in figura 6.

Figura 3. La voce *Registra* nel menu *Opzioni*

A questo punto l'utente è registrato sul server SIP prescelto e può invitare un altro utente per iniziare con esso una nuova sessione, oppure può essere invitato da chiunque altro ne conosca l'indirizzo SIP e disponga di un programma compatibile. Per iniziare una sessione è necessario selezionare l'opzione *Inizia* nel menu *Opzioni* (v. figura 7) e specificare l'indirizzo SIP dell'utente con cui si vuole parlare nella finestra (mostrata in figura 8) che compare a seguito della selezione. Si noti, per inciso, come il contenuto del menu *Opzioni* vari a seconda dello stato in cui

Figura 4. La lista dei profili SIP disponibili

Figura 5. Registrazione in corso

Figura 6. Registrazione completata

Figura 7. La voce *Inizia* nel menu *Opzioni*

Figura 8. Inserimento dell'indirizzo dell'interlocutore

Figura 9. Invito in corso

si trova il sistema, in modo da rendere di volta in volta disponibili all'utente solamente i comandi che possono essere impartiti in quel particolare stato.

All'utente che ha mandato l'invito viene mostrato il messaggio visualizzato in figura 9, mentre a finché colui che lo riceve viene data la possibilità di accettarlo o di rifiutarlo agendo sugli appositi comandi (v. figura 10). Se quest'ultimo accetta l'invito, al chiamante viene mostrato il messaggio in figura 11; se invece lo rifiuta, il chiamante riceve il messaggio visualizzato in figura 12.

Figura 10. Invito ricevuto

Se l'instaurazione della comunicazione va a buon fine, l'utente chiamato può iniziare a parlare, mentre il chiamante inizialmente si mette in ascolto. In qualsiasi momento, comunque, ognuno degli interlocutori può variare il proprio stato (in altre parole può iniziare a ricevere se era in fase di trasmissione o a trasmettere se era in fase di ricezione) premendo il pulsante *OK* del proprio terminale. A seguito di tale comando, viene interrotta l'operazione (campionamento o riproduzione) che si stava effettuando e viene inviato all'interlocutore un messaggio che indica l'operazione che l'utente intende iniziare. Il terminale dell'altro utente, senza che quest'ultimo debba compiere alcuna operazione, invia al primo un messaggio di acknowledgement, ma non varia il proprio stato; solo una volta ricevuto tale messaggio il primo terminale può

Figura 11. Invito accettato

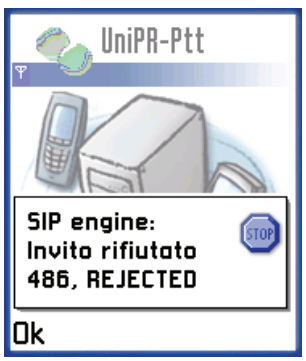

Figura 12. Invito rifiutato

Figura 13. L'utente ascolta, l'altro parla

Figura 14. L'utente parla, l'altro ascolta

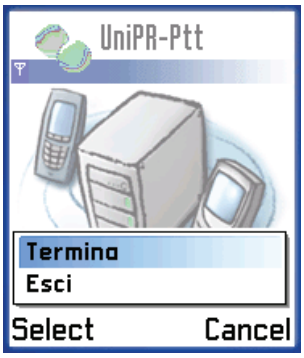

Figura 15. La voce *Termina* nel menu *Opzioni*

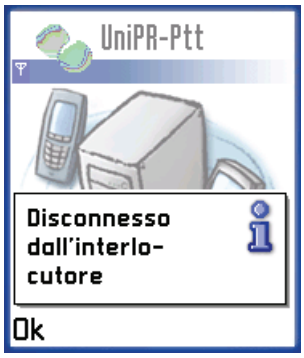

Figura 16. Chiamata terminata

iniziare a compiere l'operazione che l'utente aveva richiesto. Durante la fase in cui gli utenti stanno comunicando l'uno con l'altro, il display del dispositivo mostra a ciascuno di essi sia lo stato proprio che quello dell'interlocutore (figure 13 e 14).

Sia l'utente che ha iniziato la conversazione sia quello che ha ricevuto l'invito possono decidere in qualunque momento di interrompere la comunicazione e disconnettersi dall'interlocutore. Per fare questo è sufficiente selezionare, nel menu *Opzioni*, la voce *Termina* (figura 15), che diviene disponibile solo al momento dell'instaurazione della chiamata. A seguito di questa operazione, entrambi gli utenti ricevono il messaggio mostrato in figura 16 ma rimangono comunque registrati sul server SIP prescelto, e in quanto tali possono iniziare una nuova conversazione, sia nel ruolo di chiamante che in quello di chiamato, seguendo lo stesso procedimento descritto in precedenza.

Figura 17. La voce *Deregistra* nel menu *Opzioni*

Se un utente desidera revocare la propria registrazione sul server SIP corrente, può scegliere il comando *Deregistra* dal menu *Opzioni* (figura 17). Si noti come tale comando sia presente nel menu solo quando l'utente è registrato sul server ma non è impegnato in una chiamata. Terminata la deregistrazione si riceve il messaggio mostrato in figura 18, dopodiché è possibile registrarsi nuovamente sullo stesso server oppure su uno differente, seguendo le istruzioni precedentemente indicate, oppure uscire dal programma selezionando l'apposito comando.

Figura 18. Profilo SIP deregistrato

3.2. *Descrizione di* Chipflip

L'esempio *Chipflip*, da cui si è partiti per lo sviluppo del progetto, consiste in un gioco simile alla dama: due giocatori devono disporre a turno le proprie pedine su una

scacchiera seguendo determinate regole. Queste prevedono che una pedina possa essere piazzata in un certo punto della scacchiera solo se così facendo almeno una pedina dell'avversario si venisse a trovare compresa tra due pedine proprie. In questo caso, tutte le pedine avversarie comprese tra le due proprie cambiano colore e divengono pedine proprie. Se uno dei due giocatori non può effettuare alcuna mossa lecita, il turno passa all'altro. La partita si conclude quando tutte le pedine presenti sulla scacchiera sono di uno stesso colore: in tal caso, il giocatore a cui corrispondevano le pedine di quel colore è il vincitore. La schermata iniziale del gioco è mostrata in figura 19.

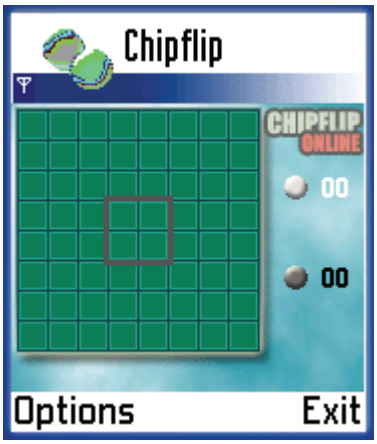

Figura 19. Chipflip

L'implementazione di tale gioco prevede che chiunque intenda prendervi parte definisca dapprima un proprio profilo SIP (mediante il plug-in SIP descritto in precedenza) e quindi lo registri, mediante l' apposito comando presente nel menu a tendina *Opzioni*, su un opportuno server. L'applicazione, quando riceve tale comando, si connette al server SIP prescelto e invia a questo il messaggio di REGISTER.

Dopo che un utente ha effettuato la registrazione, questi, agendo sugli appositi comandi a menu, può invitare un altro utente registrato a giocare, specificandone l'indirizzo SIP; se questo accetta, la partita può iniziare. Anche i messaggi di invito e di accettazione/rifiuto del medesimo sono normali messaggi SIP di INVITE, OK ecc. Terminata la fase di handshake SIP, l'utente che è stato invitato inizia a giocare, decide la posizione in cui collocare la propria pedina utilizzando i tasti direzionali del terminale, e conferma la sua scelta mediante il tasto *OK*. A questo punto il sistema verifica che la mossa sia legale e, in caso affermativo, genera un messaggio contenente la posizione della pedina, che viene inviato all'avversario. Tale messaggio è di tipo testuale, ovvero è costituito da una stringa di caratteri ASCII standard. Esso inizia con il numero d'ordine del messaggio (che servirà per verificare la correttezza dell'acknowledgement), seguito da una barra verticale ("|", codice ASCII 124), dal numero della colonna della scacchiera in cui l'utente intende inserire la propria pedina, da un'altra barra verticale e dal numero della riga di inserimento della pedina.

Il programma in esecuzione sull'altro terminale verifica se la mossa è legale e, in caso affermativo, dispone la pedina avversaria nella posizione voluta, determina le conseguenze della mossa appena compiuta e manda all'altro un messaggio di acknowledgement; quindi il turno passa all'altro giocatore, e così via, finché ci sono mosse possibili o finché uno dei due giocatori, agendo sull'apposito comando a menu, interrompe la partita. Anche il messaggio di acknowledgement è di tipo testuale, ed è costituito dalla scritta "ACK" seguita da una barra verticale, dal numero d'ordine della mossa a cui tale messaggio si riferisce e da un'altra barra verticale. Si noti come, utilizzando un protocollo di trasporto connection oriented come TCP, questi messaggi

siano assolutamente pleonastici, in quanto non fanno che ripetere alcune delle funzionalità già svolte da TCP. Con ogni probabilità, essi sono stati inseriti dagli sviluppatori per consentire anche l'uso di un protocollo di trasporto connectionless quale UDP.

I messaggi contenenti la posizione delle pedine e i relativi acknowledgement sono scambiati direttamente tra i due dispositivi, senza passare attraverso il server SIP, mediante una connessione TCP che viene instaurata al termine dell'handshake SIP. Il messaggio mediante il quale uno dei giocatori comunica all'altro l'intenzione di interrompere il gioco è invece un normale messaggio SIP di BYE. Terminata una partita, l'utente può iniziarne una nuova oppure restare in attesa di un nuovo invito o ancora revocare la propria registrazione mediante un messaggio SIP di REGISTER il cui campo *Expires* ha un valore pari a 0.

L'applicazione si occupa, naturalmente, anche di gestire e comunicare all'utente eventuali messaggi di errore e di variare le informazioni visualizzate sul display, il contenuto dei menu a tendina e le funzioni attribuite ai tasti in funzione dello stato del gioco, nonché di rinnovare la registrazione al server SIP prima della scadenza del periodo di validità.

3.3. *Modifiche apportate a* Chipflip

Per la realizzazione del progetto desiderato, si è pensato di partire dall'esempio *Chipflip* e di effettuare via via le opportune modifiche. Infatti la struttura di base dei due programmi è abbastanza simile: dapprima si ha la fase di handshake SIP, quindi l'utente invitato, che in *Chipflip* effettua la prima mossa, inizia a parlare mentre l'altro si mette in ascolto; la commutazione fra trasmissione e ricezione di dati audio, con la comunicazione del nuovo stato all'interlocutore, è assimilabile all'invio delle informazioni relative a una mossa. La transizione da un'applicazione all'altra è avvenuta attraverso una serie di passi successivi.

➢ Modifiche al file *bld.inf*

Innanzi tutto, per visualizzare e modificare il codice relativo al progetto *Chipflip* utilizzando l'IDE Borland C++BuilderX, è necessario importare il progetto stesso nell'IDE aprendo il relativo file *bld.inf*[1]; così facendo, però, non viene visualizzato nessuno dei file che compongono il progetto, anche se è possibile effettuare la compilazione senza problemi. Il file *bld.inf* originale non fa che includere diversi file aventi lo stesso nome e contenuti nelle cartelle corrispondenti alle varie classi di cui si

[1] Questo file, che deve essere incluso in ogni progetto C++ per Symbian, ha lo scopo di elencare tutte le parti che lo compongono, così da permettere al compilatore di individuare tutti i file necessari.

compone il progetto. All'interno di ognuno di questi, in particolare, alcuni dei file header che lo compongono vengono automaticamente copiati nella directory che contiene i file header appartenenti alle librerie standard.

Il motivo di questa operazione consiste nel fatto che ognuna delle classi di cui si compone il progetto è contenuta in una directory differente, ma talvolta, dall'interno di una classe, è necessario accedere a file header appartenenti a classi diverse. Se non si vuole indicare esplicitamente la cartella in cui sono contenuti i file necessari, dunque, è necessario copiarli in una cartella accessibile da ogni file del progetto, quale appunto quella contenente i file header standard. Dopo aver compilato una prima volta il progetto, e aver quindi copiato i necessari file nella posizione opportuna, per accedere ai sorgenti dall'IDE Borland è necessario modificare il file *bld.inf*. Il contenuto del file originale è il seguente:

```
#include "..\SocketEngine\group\bld.inf"
#include "..\SIPEngine\group\bld.inf"
#include "..\CommunicationChannel\group\bld.inf"
#include "..\ChipflipEngine\group\bld.inf"
#include "..\ChipflipUI\group\bld.inf"
#include "..\ChipflipEComPlugin\group\bld.inf"
```

mentre il contenuto del file così come è stato modificato è

```
PRJ_MMPFILES
..\ChipFlipEComPlugin\group\ChipflipEComPlugin.mmp
..\ChipFlipEngine\group\ChipflipEngine.mmp
..\ChipFlipUI\group\ChipFlip.mmp
..\CommunicationChannel\group\CommunicationChannel.mmp
..\SIPEngine\group\SIPEngine.mmp
..\SocketEngine\group\SocketEngine.mmp
```

Dopo aver effettuato la modifica descritta, è stato possibile procedere con la stesura del progetto *UniPR-Ptt*.

➢ Passaggio da TCP a UDP

Per prima cosa, si è creata una nuova versione del gioco apparentemente identica alla precedente ma tale che i messaggi contenenti le informazioni sulla mossa eseguita e i relativi acknowledgement viaggiassero su UDP. Tale scelta è motivata dal fatto che TCP non è un protocollo di trasporto adatto allo streaming di dati multimediali, in quanto comprende funzionalità (quali il controllo di flusso e il recupero di errore) opportune per il trasferimento di file ma inutili, anzi potenzialmente dannose, per trasmettere media in tempo reale. In tal caso, infatti, non è indispensabile evitare assolutamente le perdite di pacchetti, e comunque la ritrasmissione degli stessi dati dopo un determinato intervallo di tempo non ha alcuna utilità (i dati arriverebbero troppo in ritardo per poter essere riprodotti correttamente) e per di più contribuisce a sovraccaricare la rete, aggravando la situazione.

La variazione del protocollo di trasporto utilizzato da TCP a UDP, apparentemente banale, ha in realtà notevoli implicazioni sulla struttura del programma. Infatti, com'è noto, l'uso di un protocollo di trasporto a stream quale TCP prevede (come schematizzato in figura 20) che uno dei due utenti crei una socket, la colleghi a un indirizzo locale e si metta in attesa di una connessione su di essa; allorché l'altro utente effettua la connessione, questa socket viene rimessa in attesa di connessioni e ne viene creata un'altra, sulla quale poi transiteranno tutti i dati scambiati. Per l'invio dei dati si fa riferimento alla connessione, senza dover specificare l'indirizzo del destinatario. Infine, le due socket vengono chiuse in maniera indipendente dai due utenti.

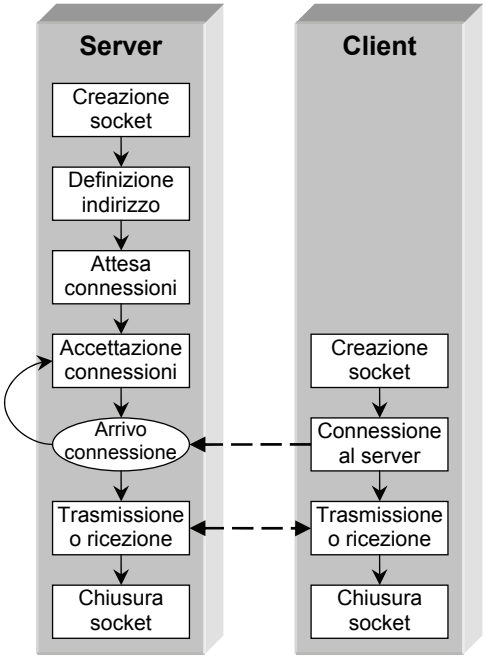

Figura 20. Scambio di dati con un protocollo a stream

Invece in un protocollo a pacchetto quale UDP (figura 21) non esiste il concetto di connessione, ma i dati viaggiano dal mittente al destinatario sotto forma di pacchetti, ognuno dei quali contiene il proprio indirizzo di destinazione. In questo caso ognuno degli utenti crea la socket e la collega a un indirizzo locale, che verrà utilizzato dall'altro utente come indirizzo di destinazione. Scompaiono quindi le operazioni di attesa e di instaurazione della connessione, così come scompare la socket su cui uno dei due utenti attende che l'altro si connetta; inoltre cambiano le primitive da utilizzare per l'invio dei dati, poiché ogni volta che si spedisce un pacchetto è necessario indicarne esplicitamente l'indirizzo di destinazione.

In un primo tempo, tuttavia, per evitare tali ripetizioni e per lasciare il più possibile inalterato il codice, si è pensato di mantenere le funzioni `RSocket::Listen()` e `RSocket::Connect()`, tipiche dei protocolli di trasporto a stream, anche con UDP: ognuno dei due processi avrebbe dovuto mettersi in attesa su una determinata porta, pubblicizzata nell'opportuno messaggio SIP, e successivamente effettuare la "connessione" alla socket aperta dall'altro. Questa possibilità appariva lecita, stando a ciò che si può leggere nella sezione dell'help dell'SDK riguardante la funzione `RSocket::Connect()`: *If a protocol has the KSIConnectionLess flag, then Connect() may be used to set the address for all data sent from the*

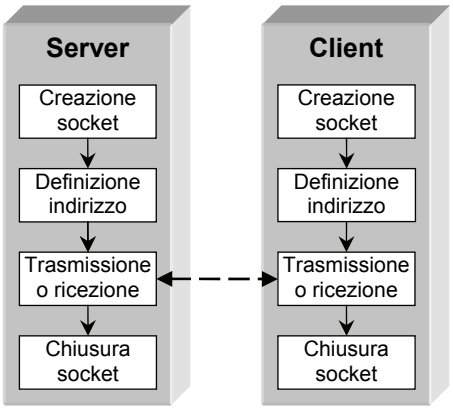

Figura 21. Scambio di dati con un protocollo a pacchetto

socket, in which case Send()/Write() may be used in addition to SendTo(). Si pensava cioè che la chiamata della `Connect()` per una socket UDP significasse unicamente memorizzare l'indirizzo di destinazione, che avrebbe poi dovuto essere utilizzato per tutti gli invii successivi. Tuttavia, ripetuti esperimenti hanno dimostrato che questa strada non era percorribile: l'invio e la ricezione del primo messaggio e del corrispondente acknowledgement[2] andavano a buon fine, mentre il messaggio successivo veniva inviato da uno degli utenti ma non veniva ricevuto dall'altro.

Si è perciò resa necessaria una modifica radicale della modalità in cui viene gestita la socket, con il passaggio allo schema tipico per protocolli connectionless. In questo scenario, entrambi i processi aprono una socket e la collegano alla prima porta disponibile tra quelle comprese nell'intervallo [49152,65535][3]; la porta scelta dal terminale dell'utente che invita l'altro è pubblicizzata nel messaggio SIP di INVITE, mentre quella scelta dal terminale dell'invitato è indicata da questo nel corrispondente 200 OK. Al momento della ricezione di uno di questi messaggi, vengono memorizzati l'indirizzo IP dell'altro terminale e la porta da questo scelta, che saranno indirizzo e porta di destinazione di tutti i messaggi ad esso diretti. Si noti che ognuno dei due programmi è scritto in modo da utilizzare la medesima socket sia per l'invio che per la ricezione dei dati, anche se teoricamente sarebbe possibile usare una socket per l'invio e una differente per la ricezione. Inoltre, nella modalità implementata non è più necessaria la creazione della socket su cui un terminale si poneva in attesa delle connessioni dell'altro.

➢ Introduzione dell'operazione di campionamento

Il passo successivo per la trasformazione del programma è stato l'aggiunta delle funzionalità relative al campionamento dei dati audio dal microfono di uno dei due terminali e all'invio dei dati stessi all'altro terminale. In questa fase dello sviluppo, tuttavia, erano ancora implementate anche le funzionalità proprie del gioco originario. Quando era il turno di un determinato utente, questi, come previsto in *Chipflip*, poteva scegliere dove piazzare la propria pedina utilizzando i tasti direzionali, ma contemporaneamente i suoni provenienti dal microfono del suo terminale venivano campionati e spediti all'altro utente, sulla stessa socket utilizzata per comunicare la posizione della nuova pedina. Quando l'utente attivo premeva il tasto *OK* del proprio terminale, veniva inviato il normale messaggio contenente la posizione della pedina e inoltre veniva interrotta l'operazione di acquisizione dell'audio. Dall'altra parte, i pacchetti contenenti dati audio (riconoscibili dalla lunghezza) erano semplicemente

[2] Col termine "acknowledgement", ora e nel seguito, si intende il messaggio di livello applicativo inviato per confermare la ricezione di uno dei messaggi che indicano l'inizio di una fase di trasmissione o di ricezione.

[3] Le porte comprese in questo intervallo, dette *private* o *dinamiche*, non sono assegnate dalla IANA ad alcuna particolare applicazione ma possono essere utilizzate liberamente dagli sviluppatori.

scartati, mentre quelli che contenevano i dati relativi a una mossa venivano elaborati normalmente, fino all'invio dell'acknowledgement. Terminata la gestione del messaggio relativo alla posizione della nuova pedina, il turno passava all'altro utente, che a sua volta decideva la posizione della successiva pedina e contemporaneamente iniziava a inviare pacchetti di dati audio, e così via.

In questa fase dello sviluppo è stato necessario studiare attentamente le modalità con cui il sistema operativo Symbian permette di gestire l'operazione di campionamento del segnale audio proveniente dal microfono del dispositivo. Il meccanismo utilizzato (che sarà analizzato in maggior dettaglio in seguito) prevede infatti che per iniziare l'operazione di campionamento venga richiamata un'apposita funzione, che si occupa dell'apertura dello stream proveniente dal microfono. Quando l'apertura termina, il sistema operativo richiama un'apposita funzione callback, che deve essere definita all'interno del programma. All'interno di tale funzione, tipicamente, si dà inizio al campionamento dell'audio proveniente dal microfono e alla sua memorizzazione in formato PCM in un buffer; i dati sono campionati su un singolo canale con una frequenza di 8 kHz e ogni campione è rappresentato su 16 bit. È da notare come, qualunque sia la dimensione del buffer che viene passato alla funzione che si occupa del campionamento, di esso vengano comunque riempiti al massimo 320 byte, corrispondenti a 160 campioni, ovvero a un intervallo di tempo pari a 20 millisecondi. In ogni caso, quando il buffer è stato riempito, il sistema operativo richiama un'altra funzione callback, anch'essa obbligatoriamente presente, nella quale tipicamente si effettuano le necessarie azioni sul buffer e si dà inizio a una nuova operazione di lettura, e così via. Per terminare l'operazione di campionamento è necessario richiamare un'altra funzione; quando l'operazione termina, il sistema operativo richiama una terza callback (o almeno la dovrebbe richiamare, anche se in tutte le prove effettuate ciò non è mai avvenuto).

Tornando al progetto, in un primo momento nella funzione richiamata ogni volta che la lettura di un buffer è completata si effettuava l'invio del blocco di dati grezzi e al termine dell'invio si dava inizio a una nuova operazione di campionamento. In seguito ci si è resi conto che, inviando semplicemente i dati codificati in PCM, la banda necessaria per la trasmissione sarebbe stata superiore a quella garantita attualmente in uplink dagli operatori mobili, e quindi è stato necessario introdurre una codifica dei dati più efficiente del semplice PCM. La codifica scelta è stata la AMR a 4,75 kbit/s. Questa opera su blocchi codificati in PCM da 320 byte (e ciò spiega il motivo del limite precedentemente citato) e restituisce trame da 13 byte. Quindi, ogni volta che la callback veniva richiamata dal sistema operativo, si effettuava la conversione dei dati contenuti nel buffer dal formato PCM all'AMR e, se possibile, si trasmetteva una singola trama AMR in un pacchetto UDP; se l'invio non era possibile, la trama veniva scartata. In ogni caso, terminate queste operazioni, si dava inizio a una nuova lettura. Le operazioni di campionamento, transcodifica e invio erano perciò eseguite ciclicamente, in modo sequenziale. Ma Symbian è un sistema operativo multithread, dunque si è pensato di sfruttare tale caratteristica per effettuare in maniera concorrente le operazioni

di codifica e invio di un blocco di dati audio e di campionamento del blocco successivo. A tale scopo, è stato introdotto un array di due buffer da 320 byte ciascuno, e si è fatto in modo che nella callback richiamata quando uno dei due buffer è stato riempito di dati audio si ricominciasse subito a memorizzare sull'altro buffer il segnale campionato e intanto si effettuasse la codifica e si tentasse l'invio del primo.

➢ Introduzione dell'operazione di riproduzione

Successivamente, com'è naturale, è stata introdotta la funzione complementare alla precedente, ovvero la riproduzione sull'altoparlante del dispositivo dei campioni audio ricevuti. Ora, in fase di ricezione, i pacchetti di dati audio non vengono scartati, ma sono decodificati e riprodotti. La fase di attesa dei dati audio in arrivo e della loro riproduzione sul terminale ha inizio non appena l'utente riceve l'acknowledgement relativo al messaggio indicante la volontà di interrompere la trasmissione, e termina al momento della ricezione dello stesso messaggio proveniente dall'altro utente.

Anche l'operazione di riproduzione, così come quella di campionamento, viene gestita dal sistema operativo Symbian mediante un sistema a callback: per iniziare l'operazione occorre richiamare un'apposita funzione che si occupa dell'apertura dello stream diretto verso l'altoparlante del dispositivo; quando l'operazione è completata, il sistema operativo richiama un'apposita callback, che deve essere definita dallo sviluppatore. In questa funzione, tipicamente, se sono già stati ricevuti i primi dati audio da riprodurre, si inizia la riproduzione del primo pacchetto da 320 byte di dati codificati in PCM. Quando tale pacchetto è stato passato agli strati inferiori del multimedia framework per essere riprodotto, il sistema operativo richiama un'altra funzione callback, nella quale tipicamente si verifica che siano presenti altri pacchetti da riprodurre e, in caso affermativo, se ne avvia la lettura, e così via. Quando si desidera interrompere l'operazione di riproduzione è necessario invocare un'altra funzione; al termine dell'operazione, il sistema operativo si occupa di richiamare una terza callback.

Già a questo punto, perciò, sarebbe possibile trasmettere e ricevere dati audio in modalità *push-to-talk*: in questo scenario, infatti, uno degli utenti parla e contemporaneamente decide la posizione della prossima pedina; quando finisce di parlare preme il tasto *OK*, col duplice effetto di spedire il messaggio contenente la posizione scelta per la pedina e di interrompere la trasmissione e iniziare la riproduzione. L'altro utente dapprima si pone in ascolto e riproduce quanto sta arrivando dal primo, quindi, al momento della ricezione della mossa, interrompe la riproduzione dei dati audio e inizia a campionare e trasmettere i propri.

Fino a questo punto, però, sono ancora presenti tutte le funzionalità relative allo svolgimento del gioco: in particolare, l'utente attivo deve scegliere la posizione in cui disporre la pedina, il sistema deve verificare la correttezza della mossa e le sue conseguenze, ecc. Evidentemente, nell'ottica di un sistema che effettui il solo streaming di dati audio, tutte queste funzioni sono superflue, se non fastidiose: perciò si è dovuto provvedere a rimuovere dal progetto le parti che eseguivano funzioni non conformi alle

specifiche del progetto. In questa fase è stata anche modificata l'interfaccia con l'utente, eliminando la visualizzazione della scacchiera e del numero delle pedine dei due giocatori e sostituendole con l'indicazione dello stato (trasmissione o ricezione) proprio e dell'interlocutore. Infine, è stato modificato il formato dei messaggi scambiati tra i due utenti: in essi non compaiono più informazioni riguardanti la posizione della pedina o il numero progressivo della mossa, ma il messaggio inviato all'interlocutore quando si intende smettere di trasmettere e iniziare a ricevere consiste semplicemente nella stringa di testo "Rx", e l'acknowledgement è formato dalla sola stringa "Ack". In questo modo si è quindi realizzato il primo dei modelli di push-to-talk descritti nel paragrafo 2.4.1.

➢ Prima modifica del modello di push-to-talk

Nel programma realizzato fino a questo punto, solo l'utente che sta parlando può decidere quando smettere di parlare e iniziare ad ascoltare, e contemporaneamente può obbligare l'altro utente a iniziare a trasmettere; l'utente che si trova in fase di riproduzione non ha alcun controllo sullo stato della comunicazione. Tale scenario non è quello tipico di un sistema di tipo push-to-talk: in questo caso, infatti, è normalmente previsto che ognuno degli interlocutori possa decidere autonomamente se parlare o ascoltare, senza neppure comunicare all'interlocutore il proprio stato.

Per questo motivo, la modifica successiva è consistita nel permettere a ciascuno degli interlocutori di variare il proprio stato in qualunque momento: se l'utente che è in fase di trasmissione intende iniziare a ricevere, premendo il tasto *OK* sul proprio terminale interrompe l'invio dei dati e spedisce all'altro un messaggio di testo contenente la stringa "Rx", mentre se l'utente che si trova nella fase di ricezione vuole iniziare la trasmissione, premendo il medesimo tasto interrompe la riproduzione e invia all'altro un messaggio contenente la stringa "Tx". In entrambi i casi, il terminale che riceve il messaggio invia automaticamente all'altro un messaggio di acknowledgement, senza modificare il proprio stato. Solo al momento della ricezione dell'acknowledgement il primo terminale inizia effettivamente a compiere la nuova operazione (ricezione o trasmissione). Lo scopo dello scambio di tali messaggi, in questa fase, è di informare ognuno degli utenti, mediante un'opportuna indicazione sul display del proprio dispositivo, dello stato in cui si trova l'interlocutore in quel momento, verificando nel contempo che l'altro sia ancora connesso ed abbia ricevuto correttamente il messaggio. In questo modo si implementa il quinto dei modelli di push-to-talk descritti nel paragrafo 2.4.1.

Come si può notare, in questo modo non si ottiene uno scenario di tipo push-to-talk "puro", in cui ogni interlocutore agisce in totale autonomia. Si può tutavia ricadere facilmente in questo caso (corrispondente al quarto modello descritto) rimuovendo alcune delle funzioni introdotte, specificatamente quelle che si occupano della trasmissione e della ricezione dei messaggi "Tx", "Rx" e "Ack".

➢ Introduzione dei messaggi di keepalive

A questo punto ci si è posti il problema dell'attraversamento dei NAT: tipicamente ad ognuno dei terminali, al momento della connessione alla rete, viene assegnato dinamicamente un indirizzo IP privato, che verrà mappato su un corrispondente indirizzo pubblico ad opera di un opportuno server NAT. I pacchetti inviati da questo terminale viaggeranno sulla rete pubblica e verranno ricevuti dall'altro utente avendo come indirizzo e porta di sorgente quelli del NAT, e al medesimo indirizzo dovranno essere inviati i pacchetti diretti al terminale stesso. Ma un terminale, normalmente, non può conoscere a priori l'indirizzo pubblico che gli verrà assegnato: esso infatti conosce unicamente il proprio indirizzo privato, e può comunicare all'interlocutore solo quest'ultimo. Se, come avviene tipicamente, i due terminali sono attestati su reti private differenti, non è quindi possibile stabilire una comunicazione diretta tra i due utenti inviando i pacchetti di uno all'indirizzo (privato) pubblicizzato mediante SIP/SDP dall'altro. Per di più, come si è accennato, in genere gli indirizzi privati sono assegnati dinamicamente, quindi l'associazione tra un indirizzo privato e il corrispondente indirizzo pubblico è mantenuta dal NAT solo per un breve periodo di tempo (qualche decina di secondi), dopodiché, se non transitano pacchetti relativi a quell'indirizzo pubblico, esso viene reso disponibile a un nuovo utente.

È stato quindi necessario risolvere due problemi: fare in modo che un terminale potesse conoscere il più presto possibile l'indirizzo pubblico assegnato all'altro, così da poter iniziare quanto prima l'invio di dati audio, e obbligare i server NAT a mantenere attiva l'associazione tra l'indirizzo privato assegnato a un utente e il relativo indirizzo pubblico anche durante le fasi in cui questi non invia né riceve dati audio. La soluzione scelta è consistita nell'introdurre dei messaggi di keepalive, costituiti dal solo carattere "k", che vengono inviati nei periodi durante i quali un utente si trova in fase di ricezione, il primo all'inizio di questa fase e i successivi ad intervalli di tempo regolari; da parte dell'altro utente, questi messaggi vengono semplicemente scartati. Il primo di questi keepalive consente a chi sta inviando i pacchetti di conoscere il vero indirizzo (pubblico) a cui questi devono essere spediti perché possano arrivare all'interlocutore anche se quest'ultimo è attestato su una rete privata differente dalla propria, mentre i successivi servono a rinfrescare, sul NAT, l'associazione tra indirizzo privato e indirizzo pubblico di un utente anche se questi attualmente non sta inviando né ricevendo alcunché.

Operando in questo modo, e modificando opportunamente il programma, in teoria sarebbe possibile scambiare direttamente dati audio tra i due terminali, utilizzando il server SIP solo per le operazioni di segnalazione. Ma si è preferito operare in maniera differente, facendo transitare non solo i messaggi SIP ma anche i successivi dati audio attraverso il server, che quindi assume anche la funzione di gateway. Il motivo di questa scelta è che così facendo è possibile registrare su file tutto quanto passa attraverso il server, verificando la qualità dell'audio trasmesso prima che questa possa essere modificata dalle operazioni relative alla ricezione. D'altra parte, operando in questo

modo è anche possibile inviare a uno dei terminali il contenuto di un file precedentemente registrato, verificando la correttezza delle operazioni di ricezione senza che i risultati risentano di eventuali problemi della fase di trasmissione.

Affinché tutti i pacchetti, sia di segnalazione che di traffico, passino attraverso il server, questo potrebbe semplicemente modificare i messaggi SIP/SDP prima di inoltrarli al destinatario, sostituendo il proprio indirizzo a quello del mittente originario all'interno delle richieste SIP, e compiendo l'operazione opposta nelle corrispondenti risposte. In questo modo, ognuno dei terminali sarebbe portato a pensare che l'indirizzo IP del proprio interlocutore sia quello che effettivamente appartiene al server SIP, e quindi, naturalmente, invierebbe tutti i dati a quest'ultimo, che poi dovrebbe inoltrarli verso il legittimo destinatario. Questo modo di procedere, sebbene lecito, non è però particolarmente ortodosso; si è deciso quindi di procedere secondo una diversa metodologia: prima di inoltrare al destinatario la richiesta di INVITE, il server vi inserisce il campo *Record-Route*, mediante il quale forza tutti i messaggi SIP successivi a transitare attraverso di esso. Un metodo simile non è però disponibile per quanto riguarda il traffico dati; per far passare anche questo attraverso il server occorre perciò sostituire l'indirizzo e la porta contenuti all'interno dei pacchetti SDP scambiati mediante i messaggi di INVITE e di 200 OK, come descritto in precedenza.

➢ Gestione degli errori

Fino a questo punto si è ipotizzato che tutti i messaggi inviati da uno degli utenti vengano ricevuti correttamente dall'altro. Come è noto, però, il protocollo UDP non offre alcuna garanzia in merito: è quindi necessario tenere in considerazione l'eventualità che alcuni dei pacchetti inviati possano non arrivare a destinazione, o comunque arrivare corrotti. Se tali pacchetti contengono dati audio, il problema non è particolarmente grave, e non è quindi necessario prendere alcun provvedimento; d'altra parte, lo stack SIP installato nei terminali provvede autonomamente a replicare l'invio delle richieste qualora non sia stata ricevuta una risposta entro un determinato periodo di tempo, senza che né lo sviluppatore né l'utente debbano compiere alcuna operazione particolare. Il problema è tale solo per l'invio dei messaggi che indicano l'inizio della trasmissione o della ricezione, nonché per i relativi acknowledgement. Questi, infatti, non possono andare perduti senza compromettere il funzionamento del programma: la perdita di tali pacchetti verrebbe considerata come una prova che l'interlocutore non è più connesso, mentre in realtà potrebbe semplicemente essere causata da problemi temporanei.

Per ovviare a questo problema, si è fatto in modo che al momento dell'invio di un messaggio "Tx" o "Rx" venga fatto partire un timer, alla scadenza del quale viene ritentata la trasmissione dello stesso messaggio, e così via per un certo numero di tentativi, terminati i quali si suppone che il terminale dell'interlocutore non sia più raggiungibile e quindi si procede con la disconnessione. Se invece, prima di raggiungere il numero massimo di tentativi possibili, si riceve dall'altro terminale un

acknowledgement, si ha la certezza che il messaggio è stato ricevuto correttamente e si può proseguire con le operazioni. Operando in questo modo, è possibile cautelarsi sia contro la perdita di un "Tx" o "Rx" che contro la perdita di un acknowledgement: in entrambi i casi la trasmissione del messaggio verrà ripetuta; se era stato perso il messaggio stesso, il terminale a cui era diretto ne riceve una copia dopo pochi secondi, mentre se si era perso l'acknowledgement il terminale stesso riceve un nuovo messaggio identico al precedente e manda semplicemente un nuovo acknowledgement senza modificare il proprio stato. Il ritardato arrivo di un messaggio di acknowledgement non causa alcun problema: infatti, se l'acknowledgement relativo ad un messaggio giunge con un ritardo eccessivo, nel frattempo il messaggio in questione viene trasmesso una seconda volta, ma al momento della ricezione del primo acknowledgement il sistema passa in uno stato nel quale gli acknowledgement in arrivo vengono scartati.

➢ Introduzione del protocollo RTP

Il passo successivo è consistito nell'aggiunta dell'header RTP a ognuno dei pacchetti di dati audio inviati: poiché l'header RTP comprende, tra gli altri, un campo indicante il numero di sequenza del pacchetto, esaminando tutti i dati che transitano attraverso il gateway è divenuto possibile valutare il numero di pacchetti persi in rapporto a quelli spediti. Il terminale che riceve i dati non fa che scartare l'header ed elaborare unicamente il payload del pacchetto. Ma incapsulando un singolo frame AMR, da 13 byte, in ogni pacchetto RTP, il cui header ha una lunghezza di almeno 12 byte, si introduce un overhead eccessivo: per questo motivo si è pensato di accorpare diversi frame AMR in un unico pacchetto RTP, e di incapsulare quest'ultimo in un pacchetto UDP. Il numero di trame AMR da inserire in ogni pacchetto RTP è fissato a priori, e deve essere stabilito trovando un compromesso tra il ritardo di pacchettizzazione, che aumenta all'aumentare del numero di trame, e l'overhead, che diminuisce. Per includere diversi frame AMR in un unico pacchetto RTP, si è fatto in modo che la callback richiamata quando un blocco di dati audio è stato riempito convertisse il blocco stesso in AMR e lo accodasse ai precedenti per un certo numero di cicli, mentre al ciclo successivo viene anche aggiunto l'header RTP e viene tentato l'invio dell'intero pacchetto. In ogni caso, in fase di ricezione è sufficiente eliminare l'header RTP e spezzare il payload in blocchi da 13 byte, ognuno corrispondente a una trama AMR, che verranno poi riconvertiti in PCM e riprodotti in sequenza.

Come si è detto, esiste tuttavia la possibilità che un pacchetto di dati audio trasmesso da un terminale non venga ricevuto dall'altro, o comunque contenga degli errori e sia quindi scartato da UDP senza essere passato agli strati superiori dello stack protocollare. Inoltre il ritardo con cui i pacchetti giungono a destinazione non può essere controllato, quindi è possibile che al termine della riproduzione di un pacchetto non siano presenti altri dati da leggere. In questi casi l'utente percepisce una brusca interruzione dell'audio emesso dall'altoparlante, che può risultare fastidiosa. Per ovviare a questo inconveniente, è stata introdotta la possibilità di ripetere la riproduzione dello stesso

pacchetto per più di una volta nel caso in cui in memoria non siano presenti altri dati: in altre parole, la callback richiamata quando un blocco di dati audio è stato riprodotto verifica se sono presenti altri blocchi (nel qual caso avvia la riproduzione del primo di questi) oppure se non vi sono nuovi dati pronti per essere riprodotti. In questa eventualità, per un certo numero di volte viene riprodotto l'unico pacchetto in memoria, dopodiché si suppone che effettivamente dall'altra parte sia cessato l'invio di dati, volontariamente o per cause di forza maggiore, e si interrompe la riproduzione. Qualora poi dovesse riprendere la ricezione di dati audio, si darebbe inizio a una nuova operazione di lettura e riproduzione degli stessi.

In un secondo momento si è anche provato a variare dinamicamente il numero di trame AMR da inserire in un pacchetto RTP: l'idea era che un terminale, dopo aver campionato e codificato un frammento di dati audio, verificasse la possibilità di trasmetterlo, e se la trasmissione non era possibile lo accodasse in un buffer per ritentare la trasmissione una volta campionato il frammento successivo. Tuttavia, i risultati sperimentali hanno mostrato che la qualità dell'audio ricevuto in tale modalità era inferiore a quella che si poteva ottenere utilizzando un buffer di lunghezza fissa correttamente dimensionato; inoltre il dimensionamento dinamico del buffer non permette di controllare l'overhead e il ritardo di trasmissione introdotto. Per tali motivi, questo metodo è stato abbandonato.

- **Seconda modifica del modello di push-to-talk**

Infine è stata creata una versione del progetto che opera in modalità pienamente push-to-talk, ovvero in cui ognuno degli utenti non comunica in alcun modo all'interlocutore il proprio stato. Tale modalità può essere utile, ad esempio, nel caso in cui uno dei due interlocutori abbia la possibilità di comunicare in full duplex: in questo caso colui che opera in full duplex può trasmettere e ricevere dati contemporaneamente, mentre l'altro può commutare il proprio stato dalla trasmissione alla ricezione e viceversa in qualsiasi momento, senza che l'interlocutore si accorga di nulla. Inoltre, questa modalità è necessaria per un eventuale futuro supporto di comunicazioni tra più di due utenti.

3.4. Implementazione di UniPR-Ptt

Scendendo a un livello più basso, è possibile esaminare con maggiore dettaglio le funzioni che realizzano le azioni descritte in precedenza e discutere le decisioni che sono state prese di volta in volta per la loro implementazione.

3.4.1. Classi

Innanzi tutto, occorre tener presente che il progetto sviluppato si basa su un esempio (*Chipflip*) realizzato da Nokia e fornito in abbinamento al plug-in per SIP. Tale esempio, in verità piuttosto complesso, prevede una suddivisione delle varie funzioni tra diverse classi, ognuna delle quali si occupa di una particolare area funzionale. Tale suddivisione è stata mantenuta anche nello sviluppo del progetto *UniPR-Ptt*. Nel seguito si elencano dunque tutte le classi che compongono il progetto, descrivendo brevemente le funzionalità svolte da ciascuna di esse[4].

- **`CChipflipEComPlugin`**: implementa un plug-in per testare la funzione `ClientResolver`.
- **`CChipflipEngine`**: in origine comprendeva le funzioni necessarie allo svolgimento del gioco (disposizione delle pedine, invio e ricezione delle mosse ecc.); dopo le modifiche contiene invece le funzioni relative al campionamento dell'audio e alla sua riproduzione, alla gestione dei messaggi SIP inviati e ricevuti e di quelli che informano l'interlocutore della variazione dello stato dell'utente. Questa classe ne contiene al suo interno diverse altre, ognuna deputata allo svolgimento del sottoinsieme di funzioni necessarie quando il sistema si trova in un particolare stato. Tali classi sono:
  - `CGameBaseState`
  - `CGameRegistered`
  - `CGameConnecting`
  - `CGameConnected`
  - `CGameMyTurn`
  - `CGameMakingOwnMove`
  - `CGameOpponentTurn`
- **`MChipflipEngineObserver`**: classe astratta che elenca le funzioni callback richiamate dal sistema operativo a seguito di particolari eventi asincroni relativi allo scambio di dati sulla socket; l'effettiva implementazione di tali funzioni deve essere specificata altrove.
- **`CChipflipTimer`**: classe non presente originariamente in Chipflip, inserita per consentire all'utente che si trova in fase di ricezione di inviare all'altro dei keepalive a intervalli di tempo predefiniti, in modo da tenere aperto un eventuale NAT.
- **`CChipflipApp`**: contiene le funzioni necessarie per creare il documento associato all'applicazione, nel quale, in Symbian, vengono memorizzati i dati ad essa relativi.
- **`CChipflipAppUI`**: questa classe comprende le funzioni relative al controllo dell'interfaccia utente: gestisce i comandi impartiti dall'utente mediante la tastiera,

[4] Come si potrà notare, i nomi delle classi e delle funzioni in esse contenute non sono stati modificati, quindi sono state mantenute le diciture "Chipflip", "Game", "Move" ecc. La ragione di tale scelta è puramente di semplicità nell'implementazione.

visualizza le finestre di dialogo, varia il contenuto dei menu a seconda dello stato del sistema, aggiorna il display a seguito di particolari eventi.

- **`CCancelNoteDialog`**: l'unico scopo di questa classe è la chiusura delle finestre di dialogo in seguito alla pressione dell'apposito pulsante da parte dell'utente.
- **`CChipflipDocument`**: classe contenente la funzione che si occupa di creare l'identificativo univoco dell'applicazione, ovvero il codice che, in Symbian, contraddistingue univocamente una particolare applicazione.
- **`CChipflipGameControl`**: comprende le funzioni per la visualizzazione delle opportune informazioni sul display del terminale; queste ultime in *Chipflip* consistevano nella posizione delle pedine dei due giocatori e nei relativi punteggi, mentre nel progetto finale viene visualizzato lo stato (trasmissione o ricezione) dell'utente e quello del suo interlocutore. In tale classe è inoltre contenuta la funzione che viene richiamata dal sistema operativo a seguito della pressione di un tasto da parte dell'utente, nella quale, a seconda del tasto premuto e dello stato del sistema, vengono richiamate le funzioni che svolgono le opportune operazioni.
- **`CCommunicationChannel`**: in questa classe sono incluse sia le funzioni che vengono richiamate dall'applicazione quando è necessario inviare dati su una socket, sia le callback richiamate a seguito di eventi provenienti dal sistema operativo per informare l'applicazione stessa dell'avvenuta ricezione di dati. I dati in questione possono essere relativi sia alla segnalazione che al traffico generato dalla trasmissione del segnale audio.
- **`MCommunicationChannelObserver`**: classe astratta contenente i prototipi delle funzioni richiamate dal sistema operativo in seguito a particolari eventi o errori relativi ai canali di comunicazione aperti, sia di segnalazione che di traffico; esempi di possibili eventi sono la ricezione di un INVITE o di un pacchetto di dati audio. L'implementazione di tali funzioni dovrà essere realizzata in altre classi che ereditino da questa.
- **`CMethodInOutLogger`**: contiene funzioni utili in fase di debug, che salvano su un apposito file di testo i messaggi relativi alle operazioni che vengono eseguite e ad eventuali codici di errore che vengono restituiti dal sistema.
- **`CSIPEngine`**: implementa le funzioni (alcune delle quali definite in classi astratte) relative all'instaurazione, al mantenimento e all'abbattimento della comunicazione mediante SIP, nonché alla scelta di un profilo SIP tra quelli definiti in precedenza dall'utente.
- **`MSIPEngineObserver`**: classe astratta che contiene i prototipi delle funzioni relative al protocollo SIP, e richiamate sia dall'applicazione che dal sistema operativo a seguito di particolari eventi.
- **`CSIPTransactionExecutor`**: contiene le funzioni che realizzano effettivamente lo scambio di messaggi SIP, ritrasmettendo le richieste se non si ottiene risposta entro un certo tempo e informando l'applicazione dell'avvenuto completamento dell'operazione.

- **`CSIPTransactionStateBase`**: contiene, definendole come *virtual*, le funzioni che si occupano di eseguire le operazioni necessarie per l'invio di messaggi SIP e quelle richiamate a seguito del verificarsi di determinati eventi. Inoltre include diverse altre classi, ognuna delle quali corrisponde a un determinato stato in cui si può trovare il sistema relativamente alla segnalazione tramite SIP. In ognuna di queste classi vengono ridefinite unicamente le funzioni che hanno senso quando il sistema si trova in quel particolare stato. Ad esempio, la funzione che manda il messaggio di INVITE è definita sia nella classe base che in quella corrispondente allo stato in cui l'utente è registrato sul server SIP, ma l'implementazione è diversa nei due casi: nel primo la funzione non fa che restituire un codice di errore, mentre nel secondo crea ed invia il messaggio opportuno. Le classi definite all'interno di `CSIPTransactionStateBase` sono[5]:
  - `CSIPTransactionStateInitalState`
  - `CSIPTransactionStateRegistered`
    - `CSIPTransactionStateInviteUnderProcess`
      - `CSIPTransactionStateCancelling`
      - `CSIPTransactionStateWait183Prack`
      - `CSIPTransactionStateWait183Prack200OK`
      - `CSIPTransactionStateWaitInvite200OK`
      - `CSIPTransactionStateWaitRingingPrack`
      - `CSIPTransactionStateWaitRingingPrack200OK`
      - `CSIPTransactionStateWaitUpdate`
      - `CSIPTransactionStateWaitUpdate200OK`
    - `CSIPTransactionStateWaitInviteAcception`
  - `CSIPTransactionStateSendBye`
  - `CSIPTransactionStateSendMessage`
  - `CSIPTransactionStateSendResponse`
  - `CSIPTransactionStateSessionConnected`
  - `CSIPTransactionStateWaitBYE200OK`
  - `CSIPTransactionStateWaitInviteACK`
  - `CSIPTransactionStateWaitMessage200OK`
- **`CDataTransporter`**: in questa classe originariamente era definito il buffer in cui venivano memorizzati i messaggi in arrivo dal terminale dell'avversario, contenenti le informazioni sulla posizione della pedina piazzata da quest'ultimo o gli acknowledgement inviati; assieme al buffer erano definite anche le funzioni necessarie ad accodarvi i pacchetti. Memorizzando tali messaggi in un buffer era possibile accettare nuovi messaggi mentre era ancora in corso l'elaborazione di quelli ricevuti in precedenza. In *UniPR-Ptt* questo sistema è stato mantenuto

[5] Nel seguito, tramite l'indentazione sono indicate le relazioni di ereditarietà.

unicamente per i messaggi che contengono le informazioni sull'operazione che un utente si appresta a iniziare e per i relativi acknowledgement, mentre i pacchetti contenenti dati audio sono gestiti, seppure in maniera analoga, mediante un buffer definito direttamente nella classe che si occupa della riproduzione del segnale audio.

- **`CSocketEngine`**: contiene le funzioni che realizzano le operazioni necessarie per aprire e chiudere la socket e per iniziare l'invio e la ricezione, attraverso di essa, dei dati audio e dei messaggi che segnalano all'interlocutore l'avvio della fase di trasmissione o di ricezione.
- **`CSocketEngineObserver`**: le funzioni definite in questa classe si occupano di notificare all'applicazione particolari eventi o errori relativi alla comunicazione sulla socket, tra cui in particolare l'avvenuta ricezione di un messaggio dall'interlocutore.
- **`CSocketReader`**: questa classe comprende le funzioni specifiche per iniziare la lettura di dati da una socket e per notificare all'applicazione la ricezione di un pacchetto attraverso la socket stessa.
- **`CSocketTimer`**: in questa classe sono definite le funzioni mediante le quali l'applicazione può avviare un timer, sospenderlo, farlo ripartire ed annullarlo; allo scadere del tempo impostato, viene richiamata un'apposita funzione che si dovrà

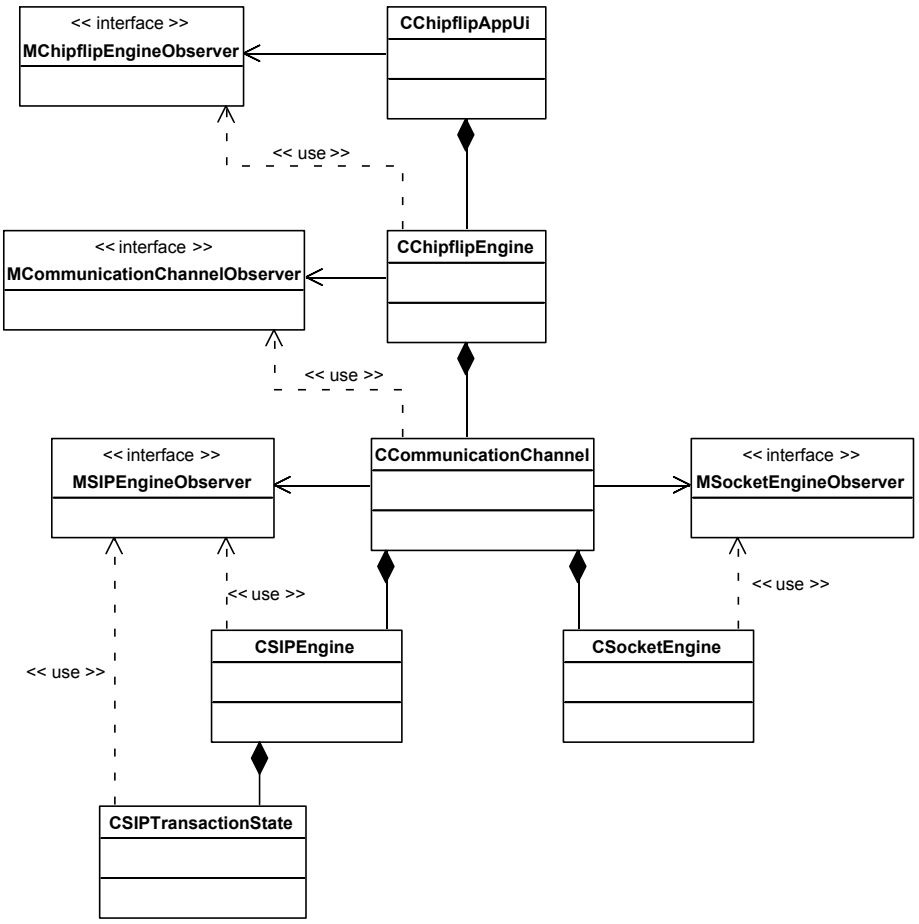

Figura 22. Classi principali di *UniPR-Ptt* e relazioni tra di esse

occupare di gestire l'evento.

- **`CSocketWriter`**: questa classe comprende le funzioni specifiche per iniziare la scrittura di dati su una socket.
- **`CSIPResolvedClient`**: definisce l'interfaccia mediante la quale l'applicazione può lanciare il client SIP e accedere ai dati in esso contenuti.
- **`CExamplePlugin`**: implementa un plug-in per testare la funzione `ClientResolver`.

Uno schema delle relazioni che intercorrono fra le classi principali tra quelle sopra descritte è mostrato in figura 22.

3.4.2. Realizzazione delle funzionalità principali

L'utilità delle diverse classi e delle funzioni in esse contenute si può comprendere studiando le modalità mediante le quali esse vengono utilizzate all'interno del programma per realizzare le varie operazioni necessarie al suo funzionamento. Verranno quindi presentate ora le implementazioni delle funzionalità fondamentali del programma, descrivendo le funzioni di volta in volta chiamate e motivando le scelte progettuali compiute durante la realizzazione del progetto.

➢ Registrazione di un profilo SIP

La figura 23 schematizza la sequenza di chiamate a funzioni necessarie ad effettuare la registrazione su un opportuno server SIP di uno dei profili che l'utente ha precedentemente definito mediante il plug-in SIP. Nel seguito le operazioni mostrate in figura sono descritte con maggiore dettaglio. I numeri tra parentesi quadre nel testo corrispondono a quelli associati, in figura, ad ogni funzione chiamata.

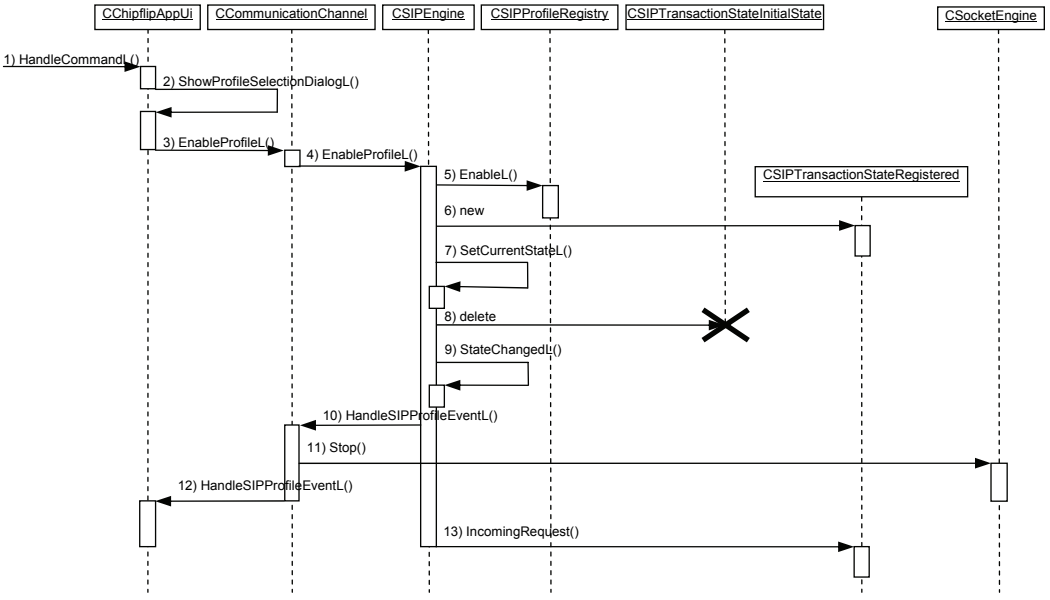

Figura 23. Registrazione di un profilo SIP

La funzione `CChipflipAppUi::HandleCommandL()`, invocata [1] dal sistema operativo a seguito della selezione da parte dell'utente di un comando presente nel menu, se tale comando è quello che avvia l'operazione di registrazione, richiama [2] la `CChipflipAppUi::ShowProfileSelectionDialogL()`. Quest'ultima mostra all'utente i vari profili disponibili, ovvero quelli impostati per mezzo del plug-in SIP, e attende che questi selezioni uno di tali profili, quindi mostra una finestra con cui comunica all'utente che l'operazione è in corso e richiama [3] la `CCommunicationChannel::EnableProfileL()`.

Questa non fa altro che chiamare [4] la funzione `CSIPEngine::EnableProfileL()`, la quale, verificata l'esistenza del profilo scelto, richiama [5] la `CSIPProfileRegistry::EnableL()`, ovvero la funzione della API SIP che abilita tale profilo, effettuandone anche la registrazione sul server SIP. Se l'operazione si conclude senza errori, ovvero se si riceve dal server SIP il messaggio di 200 OK[6], dalla `EnableProfileL()` viene creato [6] un oggetto della classe `CSIPTransactionStateRegistered`, che poi viene passato [7] alla funzione `CSIPEngine::SetCurrentStateL()`. Questa funzione, oltre a eliminare [8] l'oggetto della classe `CSIPTransactionStateInitialState` (corrispondente allo stato precedente) e a memorizzare nella variabile `iCurrentState` il puntatore ricevuto[7], richiama [9] anche la `CSIPEngine::StateChangedL()`, che, nel caso in esame, non compie alcuna operazione.

In seguito si informa il sistema dell'avvenuta registrazione, mediante l'invocazione [10] della funzione `CCommunicationChannel::HandleSIPProfileEventL()` a cui si passa come parametro `ESPOnline`. Questa, se la socket era già connessa, richiama [11] la `CSocketEngine::Stop()`, che annulla qualunque connessione preesistente, e in ogni caso chiama [12] la `CChipflipAppUi::HandleSIPProfileEventL()`, la quale informa l'utente dell'avvenuta registrazione mediante un'opportuna finestra di dialogo. Infine, sempre nella `CSIPEngine::EnableProfileL()`, se c'era un INVITE pendente diretto all'utente che si è appena registrato, questo viene processato mediante chiamata [13] alla funzione `CSIPTransactionStateRegistered::IncomingRequest()`.

---

[6] La ricezione e l'interpretazione del messaggio di risposta non devono essere effettuate esplicitamente dall'applicazione, ma sono incluse tra i compiti della funzione `CSIPProfileRegistry::EnableL()`.

[7] Si noti il particolare meccanismo con cui viene implementata la macchina a stati che rappresenta il sistema dal punto di vista del protocollo SIP: lo stato corrente è in realtà un'istanza di una particolare classe, puntata da `iCurrentState`. In questo modo, è possibile definire nelle diverse classi varie funzioni con lo stesso nome, ognuna da eseguire quando il sistema si trova in un particolare stato, ed eseguire quella corretta semplicemente richiamando `iCurrentState->funzione()`.

➢ Spedizione del messaggio SIP di Invite e ricezione del 200 Ok

La sequenza delle funzioni che vengono chiamate quando un utente registrato su un apposito server SIP ne invita un altro è mostrata in figura 24, con la stessa convenzione utilizzata nel paragrafo precedente (e che sarà utilizzata anche nei successivi). Nel seguito le operazioni necessarie sono descritte in maggiore dettaglio.

Quando l'utente seleziona dal menu il comando *Inizia*, la funzione [1] `CChipflipAppUi::HandleCommandL()` (invocata automaticamente da parte del sistema operativo) richiama [2] la `CChipflipAppUi::EnterSipAddressL()`, la quale crea una finestra di dialogo in cui l'utente può digitare l'indirizzo SIP che desidera invitare; tale indirizzo, che deve iniziare col prefisso *sip:*, viene poi passato direttamente [3] alla funzione `CChipflipEngine::InviteL()`. Questa richiama [4] il metodo `InviteL()` dell'oggetto corrispondente allo stato corrente: come si è detto, infatti, anche all'interno della classe `CChipflipEngine`, così come avviene per `CSIPTransactionStateBase`, sono definite diverse sottoclassi, ognuna delle quali corrisponde a un diverso stato del sistema dal punto di vista della comunicazione tra i due utenti. Esiste poi una variabile `iCurrentState` contenente il puntatore ad un'istanza della classe correntemente attiva, e utilizzabile per richiamare una funzione di tale classe anche senza sapere a priori in che stato si trova attualmente il sistema.

Se lo stato non è "registrato", cioè se l'oggetto puntato da `iCurrentState` non appartiene alla classe `CGameRegistered`, la funzione `InviteL()` non compie alcuna operazione. In caso contrario, all'interno di tale funzione si impostano le due

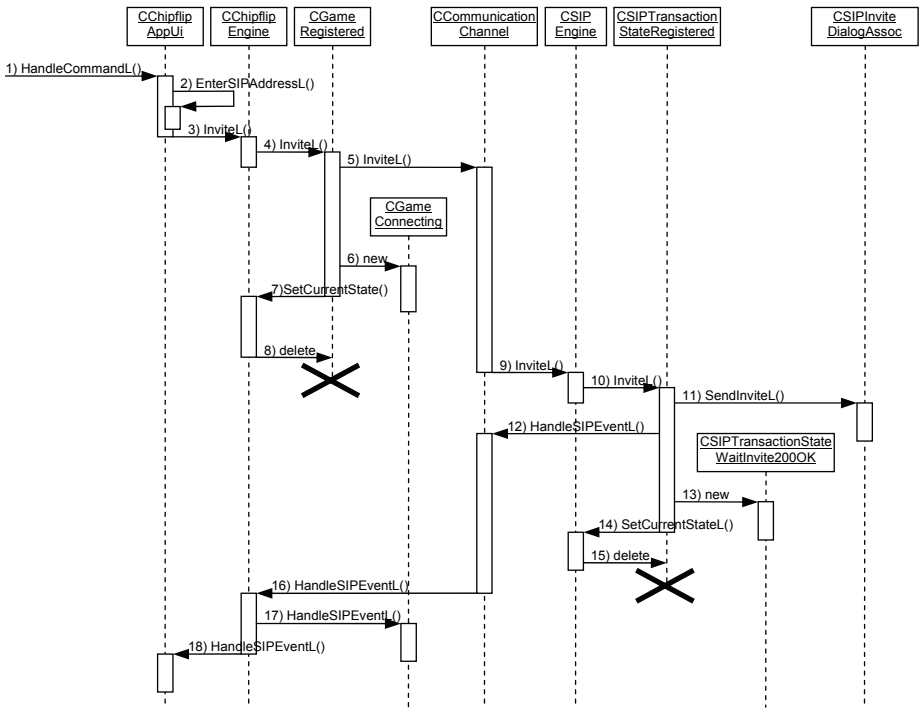

Figura 24. Spedizione dell'Invite

variabili che indicheranno l'operazione che sta svolgendo ognuno degli interlocutori in modo che colui che ha invitato l'altro si metta inizialmente in ascolto mentre colui che è stato invitato possa iniziare a parlare[8], dopodiché si richiama [5] la `CCommunicationChannel::InviteL()` e, se questa termina senza errori, si modifica lo stato del sistema (inteso nel senso descritto in precedenza) da "registrato" a "in corso di connessione" creando [6] un oggetto della classe `CGameConnecting` e sostituendolo a quello della classe `CGameRegistered` mediante la funzione [7] `CChipflipEngine::SetCurrentState()`, che cancella [8] l'oggetto corrispondente allo stato precedente.

La `CCommunicationChannel::InviteL()` a sua volta non fa che richiamare [9] la `CSIPEngine::InviteL()`. Questa verifica che lo stato del sistema rispetto a SIP sia "registrato", nel qual caso richiama [10] la `CSIPTransactionStateRegistered::InviteL()`; in caso contrario, invece, restituisce unicamente un codice di errore.

La funzione `CSIPTransactionStateRegistered::InviteL()` è quella che si occupa effettivamente della creazione e della spedizione del messaggio di INVITE: dapprima determina l'indirizzo IP del terminale in uso e si mette in ascolto su una particolare porta, in modo da poter ricevere i dati audio campionati dall'altro terminale; quindi costruisce l'indirizzo SIP del destinatario a partire dalla stringa inserita dall'utente, e nel contempo verifica che sia un indirizzo SIP valido; in caso contrario la funzione termina immediatamente restituendo un codice di errore. Se invece l'indirizzo inserito era formalmente corretto, vengono costruiti i vari campi che costituiranno gli header SIP e SDP, attingendo alle informazioni fornite dall'utente, a quelle predefinite all'interno dell'applicazione e a quelle ricavate dal terminale in uso. Infine viene tentata la spedizione del messaggio di INVITE così costruito, mediante l'invocazione [11] della funzione `CSIPInviteDialogAssoc::SendInviteL()`, appartenente alla API SIP.

Se l'operazione di invio fallisce viene restituito un codice di errore; in caso contrario, sempre all'interno della `InviteL()` viene chiamata [12] la `CCommunicationChannel::HandleSIPEventL()`, a cui viene passato come parametro `ESInviteSent`, e lo stato del sistema relativo a SIP viene posto a "attesa del 200 OK relativo all'INVITE" creando [13] un oggetto della classe `CSIPTransactionStateWaitInvite200OK` e sostituendolo a quello della classe `CSIPTransactionStateRegistered` mediante la funzione [14] `CSIPEngine::SetCurrentStateL()`, che si occupa anche di distruggere [15] l'oggetto corrispondente allo stato precedente.

[8] Il modello di chiamata descritto è stato scelto per rispecchiare quello che avviene tipicamente nel caso di una normale telefonata, in cui è l'utente chiamato che, sollevata la cornetta, inizia a parlare per primo.

La `CCommunicationChannel::HandleSIPEventL()` non fa che chiamare [16] la `CChipflipEngine::HandleSIPEventL()`, che, a sua volta, richiama le funzioni omonime definite nella classe corrispondente allo stato attuale della comunicazione, ovvero in `CGameConnecting` [17], e nella classe `CChipflipAppUi` [18], sempre con parametro `ESInviteSent`; la prima non compie alcuna operazione, mentre la seconda visualizza una finestra in cui si informa l'utente che il messaggio di INVITE è stato spedito e che si attende una risposta.

A questo punto, se tutto procede correttamente, il terminale da cui è partita la richiesta di INVITE dovrebbe ricevere uno o più messaggi di 100 TRYING, quindi uno o più messaggi di 180 RINGING, e infine il 200 OK. La figura 25 mostra la sequenza delle funzioni richiamate in quest'ultimo caso, che sarà quello di maggior interesse.

Al momento della ricezione di un qualunque messaggio SIP di risposta, il sistema operativo, tramite il plug-in SIP, richiama automaticamente [1] la funzione `IncomingResponse()`, definita come *pure virtual* nella classe `MSIPConnectionObserver` (appartenente alla API SIP) e implementata in `CSIPEngine`. Questa non fa che invocare la funzione omonima definita all'interno della classe corrispondente allo stato corrente della segnalazione. Viene quindi richiamata [2] la `CSIPTransactionStateWaitInvite200OK::`

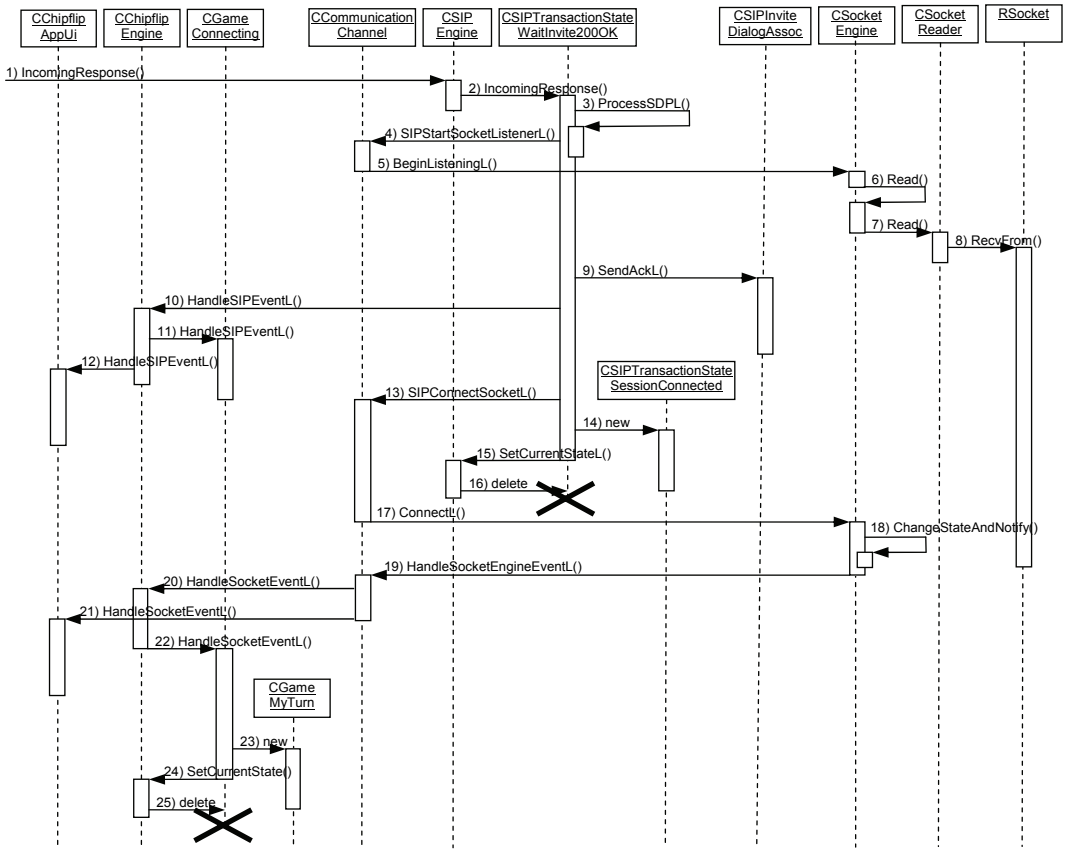

Figura 25. Ricezione del 200 Ok

`IncomingResponse()`, nella quale si esamina il messaggio ricevuto e si compiono le azioni opportune in relazione al tipo di risposta che è stata ottenuta.

Se è stato ricevuto un 100 TRYING, viene richiamata la `CCommunicationChannel::HandleSIPEventL()` con parametro `ESTrying`, che in questo caso si limita a richiamare la funzione `CChipflipEngine::HandleSIPEventL()` passandole il medesimo parametro. Quest'ultima, a sua volta, richiama la `CChipflipEngine::CGameConnecting::HandleSIPEventL` e la `CChipflipAppUi::HandleSIPEventL()`, passando loro il parametro `ESTrying`; ma nessuna di queste funzioni compie alcuna operazione. In definitiva, quindi, la ricezione del 100 TRYING non comporta alcuna variazione dello stato del sistema.

Se invece il messaggio ricevuto è un 180 RINGING, si ha la certezza che l'interlocutore utilizza SIP in modalità IETF (infatti lo stack SIP installato gestisce anche la modalità IMS), quindi si salva tale informazione in un'opportuna variabile. In tal caso non viene però richiamata alcuna funzione.

Se l'interlocutore accetta l'invito, invece, viene ricevuto il messaggio SIP di 200 OK. In questo caso all'interno della `CSIPTransactionStateWaitInvite200OK::IncomingResponse()` viene richiamata dapprima la funzione `CSDPDocument::DecodeLC()`, appartenente alle funzioni di libreria dell'API SIP, che riceve in ingresso l'header SDP del messaggio ricevuto, in formato testuale, e restituisce un puntatore a un'istanza della classe `CSDPDocument`, che contiene le funzioni necessarie ad accedere a ognuno dei campi dell'header stesso. Tale puntatore viene poi passato [3] alla funzione `CSIPTransactionStateWaitInvite200OK::ProcessSDPL()`[9], nella quale i vari campi dell'header SDP vengono utilizzati per impostare i parametri necessari per la successiva instaurazione della comunicazione.

All'interno di questa funzione, originariamente, era previsto che l'utente che invita l'altro si mettesse in attesa di una connessione sulla socket aperta all'indirizzo e alla porta comunicati all'interlocutore mediante l'INVITE, mentre l'utente chiamato, se accettava l'invito, doveva effettuare la connessione TCP su tale socket. Come si è detto, nel progetto *UniPR-Ptt* il protocollo TCP è stato sostituito con UDP, dunque il concetto di connessione non ha più significato; ognuno dei terminali deve perciò, in questa fase, mettersi in attesa di dati sulla socket il cui indirizzo è stato comunicato all'interlocutore tramite SIP. Per tale motivo, a questo punto viene richiamata [4] la `CCommunicationChannel::SIPStartSocketListenerL()`, nella quale si richiama solamente [5] la `CSocketEngine::BeginListeningL()`. In questa funzione, quando il protocollo di trasporto utilizzato era TCP, la socket veniva messa in

[9] In realtà tale funzione appartiene alla classe `CSIPTransactionStateRegistered`, ma la classe `CSIPTransactionStateWaitInvite200OK`, che ne eredita, non la ridefinisce.

attesa di connessioni; usando UDP invece non si fa che scegliere la porta da associare alla socket e richiamare [6] la `CSocketEngine::Read()`, nella quale, se non è gia in corso un'operazione di lettura, si richiama [7] la `CSocketReader::Read()`, che a sua volta avvia l'operazione asincrona di lettura mediante l'invocazione [8] della funzione `RSocket::RecvFrom()` seguita dalla `SetActive()`. La lettura, in questo modo, non è bloccante, ma al momento della ricezione di un pacchetto di dati verranno richiamate dal sistema operativo le opportune callback.

Tornando alla `ProccessSDPL()`, terminate queste operazioni i vari parametri che dovranno comporre il messaggio di ACK vengono convertiti in formato testuale, in accordo con la sintassi definita nel protocollo SDP, dalla funzione di libreria `CSDPDocument::EncodeL()`; terminata questa operazione, il controllo torna alla funzione `CSIPTransactionStateWaitInvite200OK::IncomingResponse()`, che provvede (mediante la funzione [9] `CSIPInviteDialogAssoc::SendAckL()`, definita nella API SIP) a inviare il messaggio SIP di ACK generato in precedenza. In seguito la stessa funzione informa il sistema dell'avvenuta ricezione del 200 OK e dell'invio dell'ACK chiamando [10] la `CChipflipEngine::HandleSIPEventL()` e passandole come parametro `ES200ReceivedAckSent`. Questa, a sua volta richiama prima [11] la `CChipflipEngine::CGameConnecting::HandleSIPEventL()` e poi [12] la `CChipflipAppUi::HandleSIPEvntL()`, sempre con parametro `ES200ReceivedAckSent`. La prima non compie alcuna operazione, mentre la seconda visualizza sul display un messaggio che informa l'utente dell'avvenuta ricezione del 200 OK e dell'invio dell'ACK. Successivamente la funzione `CSIPTransactionStateWaitInvite200OK::IncomingResponse()` chiama [13] la `CCommunicationChannel::SIPConnectSocketL()`, passandole l'indirizzo dell'interlocutore così come è stato ricavato attraverso lo scambio di messaggi SIP. Infine, lo stato del sistema dal punto di vista del protocollo SIP viene modificato in "connesso"[10] creando [14] un oggetto della classe `CSIPTransactionStateSessionConnected` e passandolo [15] alla `CSIPEngine::SetCurrentStateL()`, che lo sostituisce [16] a quello precedente.

Nella `CCommunicationChannel::SIPConnectSocketL()` viene chiamata [17] la `CSocketEngine::ConnectL()`, in cui, non esistendo in UDP il concetto di connessione, non si fa altro che memorizzare l'indirizzo dell'interlocutore, al quale dovranno essere inviati tutti i pacchetti contenenti dati audio, e richiamare [18] la

[10] Utilizzando il protocollo UDP, in realtà, un terminale non è mai veramente "connesso" all'altro: il nome dello stato è stato mantenuto solo per semplicità, intendendo con "connesso" lo stato in cui i due interlocutori hanno concluso l'handshake SIP e possono scambiarsi dati audio.

`CSocketEngine::ChangeStateAndNotify()` passandole come parametro `ESEConnected`. Questa, a sua volta, chiama [19] la `CCommunicationChannel::HandleSocketEngineEventL()`, che chiama [20] la `CChipflipEngine::HandleSocketEventL()` e [21] la `CChipflipAppUi::HandleSocketEventL()`, sempre con parametro `ESEConnected`.

Quest'ultima mostra all'utente un messaggio in cui lo informa dell'avvenuta connessione. La funzione `CChipflipEngine::HandleSocketEventL()`, invece, quando le viene passato tale parametro, se lo stato del sistema è "in corso di connessione" richiama [22] la `CChipflipEngine::CGameConnecting::HandleSocketEventL()`, passandole il medesimo parametro. All'interno di questa, siccome è stato previsto che l'utente che ha mandato il messaggio di INVITE inizialmente si metta in ascolto, lo stato del sistema viene modificato in "ricezione" (creando [23] un oggetto della classe `CGameOpponentTurn` e passandolo [24] alla `CChipflipEngine::SetCurrentState()`, che lo sostituisce [25] a quello preesistente) e viene dato inizio alla ricezione e riproduzione dei dati audio, come descritto nel paragrafo relativo a tale funzionalità.

Come si può notare da quanto esposto in precedenza, all'interno del progetto esistono in parallelo tre differenti accezioni del concetto di *stato del sistema*: una di esse è relativa allo scambio di dati sulla socket (per quest'ultima, possibili stati sono "connesso all'interlocutore", "disconnesso", "attesa acknowledgement" ecc.), la seconda è relativa al protocollo SIP (quindi i possibili stati saranno "non registrato", "in corso di registrazione", "ricevuto INVITE" ecc.), mentre la terza si riferisce alla comunicazione vera e propria (e quindi prevede gli stati di "campionamento", "riproduzione" ecc.). Le rispettive macchine a stati sono implementate all'interno delle

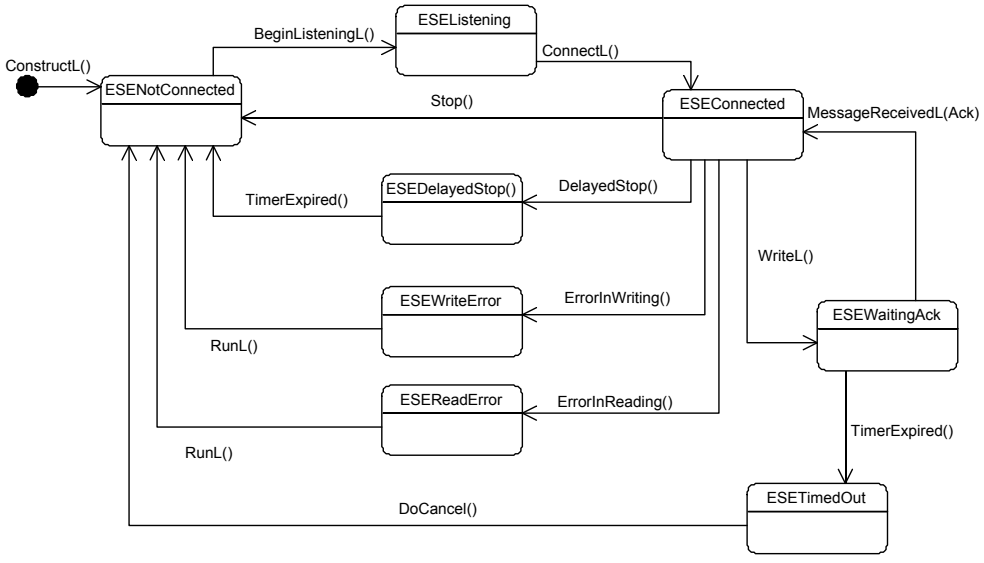

Figura 26. Diagramma di stato relativo alla socket

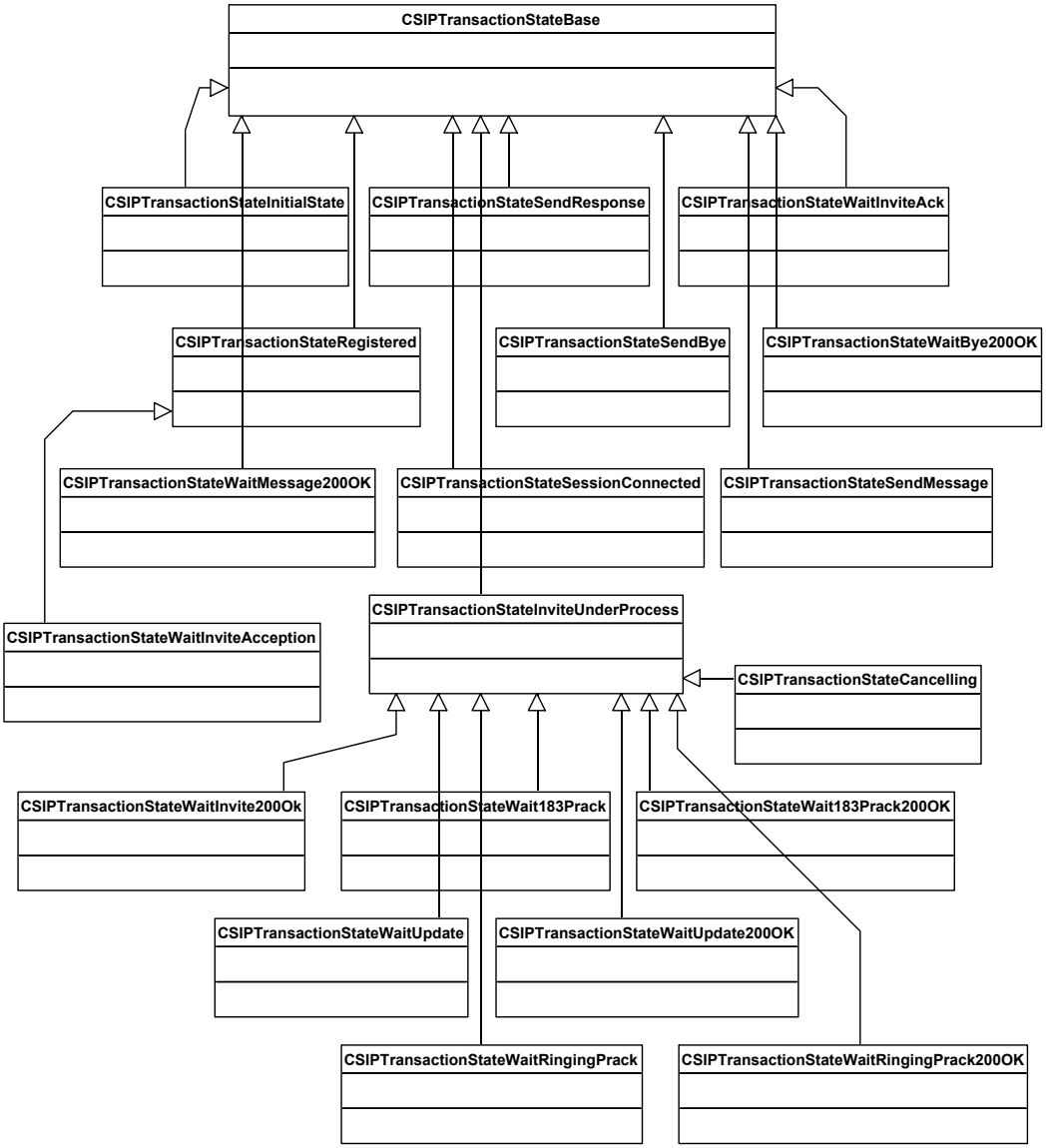

Figura 27. Class diagram degli stati relativi a SIP

classi `CSocketEngine`, `CSIPTransactionStateBase` e `CChipflipEngine`.

Nel primo caso, l'implementazione prevede l'uso di una variabile `iState` associata a un tipo di dato enumerato `TSocketEngineEnum`, comprendente tutti gli stati in cui il sistema può trovarsi; la modifica e la verifica dello stato vengono effettuate semplicemente accedendo a `iState`. Il relativo diagramma di stato è mostrato in figura 26.

Negli altri due casi, invece, ogni stato è associato a una particolare classe contenuta all'interno della principale; lo stato corrente è un'istanza della classe opportuna puntata da `iCurrentState`, e per variare lo stato del sistema occorre distruggere l'oggetto in uso, crearne uno nuovo appartenente alla classe relativa e farlo puntare da

`iCurrentState`. Le classi associate agli stati relativi a SIP sono mostrate in figura 27, mentre quelle relative agli stati della comunicazione sono schematizzate in figura 28.

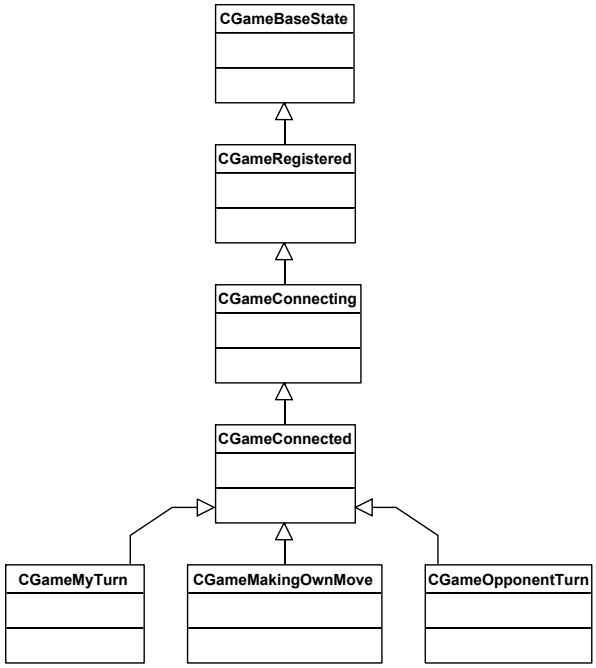

Figura 28. Class diagram degli stati relativi alla comunicazione

➢ Ricezione dell'Invite e accettazione o rifiuto

La figura 29 mostra la sequenza di funzioni chiamate quando un utente riceve da un altro un messaggio SIP di INVITE. Tali funzioni vengono ora descritte in dettaglio.

Quando uno dei due terminali riceve un qualunque messaggio SIP di richiesta, e in particolare un INVITE, il sistema operativo, tramite il plug-in SIP, richiama automaticamente [1] la funzione `IncomingRequest()`, definita come *pure virtual* nella classe `MSIPConnectionObserver` (appartenente al plug-in) e implementata in `CSIPEngine`. Questa non fa che invocare la funzione omonima definita all'interno della classe corrispondente allo stato corrente (considerato dal punto di vista della segnalazione SIP). Se attualmente il terminale è registrato sul server SIP ma non è impegnato in alcuna comunicazione, viene quindi richiamata [2] la `CSIPTransactionStateRegistered::IncomingRequest()`.

All'interno di questa funzione, se il messaggio ricevuto è proprio un INVITE, da questo vengono estrapolati l'indirizzo SIP del mittente, l'oggetto della comunicazione e il tipo di profilo SIP usato dall'interlocutore (IETF o IMS). Se, come nel caso in esame, il profilo in uso è di tipo IETF, viene subito inviato come risposta il messaggio 180 RINGING[11] (sfruttando [3] la funzione `CSIPServerTransaction::SendResponseL()` definita nella API SIP), dopodiché si richiede all'utente se intende o meno accettare l'invito, mediante la funzione [4] `CCommunicationChannel::SIPInviteReceived()`. A seconda del valore restituito da questa verranno prese le opportune decisioni.

[11] Si ricorda che il messaggio 100 TRYING è inviato al terminale che ha effettuato l'invito da parte del server SIP e non dall'altro terminale.

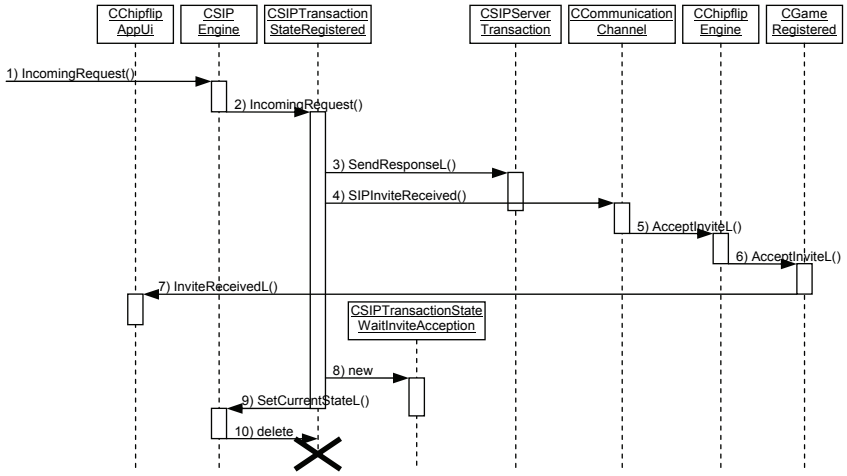

Figura 29. Ricezione dell'Invite

La funzione `CCommunicationChannel::SIPInviteReceived()` non fa che chiamare [5] la `CChipflipEngine::AcceptInviteL()`, la quale richiama la funzione omonima definita all'interno della classe corrispondente allo stato corrente dal punto di vista dello stato della comunicazione. Se l'utente è registrato ma non è connesso all'interlocutore viene richiamata [6] la `CChipflipEngine::CGameRegistered::AcceptInviteL()`, che a sua volta richiama [7] la `CChipflipAppUi::InviteReceivedL()`; altrimenti viene restituito il valore 0.

La funzione `CChipflipAppUi::InviteReceivedL()` mostra all'utente la finestra di dialogo in cui questi può decidere se accettare o rifiutare l'invito e restituisce immediatamente il valore `KMaxTInt`, che viene passato all'indietro alla `CSIPTransactionStateRegistered::IncomingRequest()`.

In questa funzione, se viene ricevuto il valore `KMaxTInt`, viene modificato lo stato della segnalazione SIP da "registrato" in "attesa della decisione dell'utente" creando [8] un oggetto della classe `CSIPTransactionStateWaitInviteAcception` e passandolo [9] alla funzione `CSIPEngine::SetCurrentStateL()`, che lo sostituisce [10] a quello che rappresentava lo stato precedente.

A questo punto l'utente è messo al corrente dell'invito e può decidere se accettarlo o meno, agendo sui corrispondenti comandi. Una volta che ha compiuto questa scelta, viene eseguita la sequenza di funzioni mostrata in figura 30[12]e descritta nel seguito.

Quando l'utente che ha ricevuto l'invito sceglie se accettarlo o meno agendo sui corrispondenti comandi, la finestra di dialogo viene chiusa e viene automaticamente richiamata [1] la funzione callback `DialogDismissedL()`, definita come *pure*

[12] La figura si riferisce, in particolare, al caso in cui l'utente accetta l'invito.

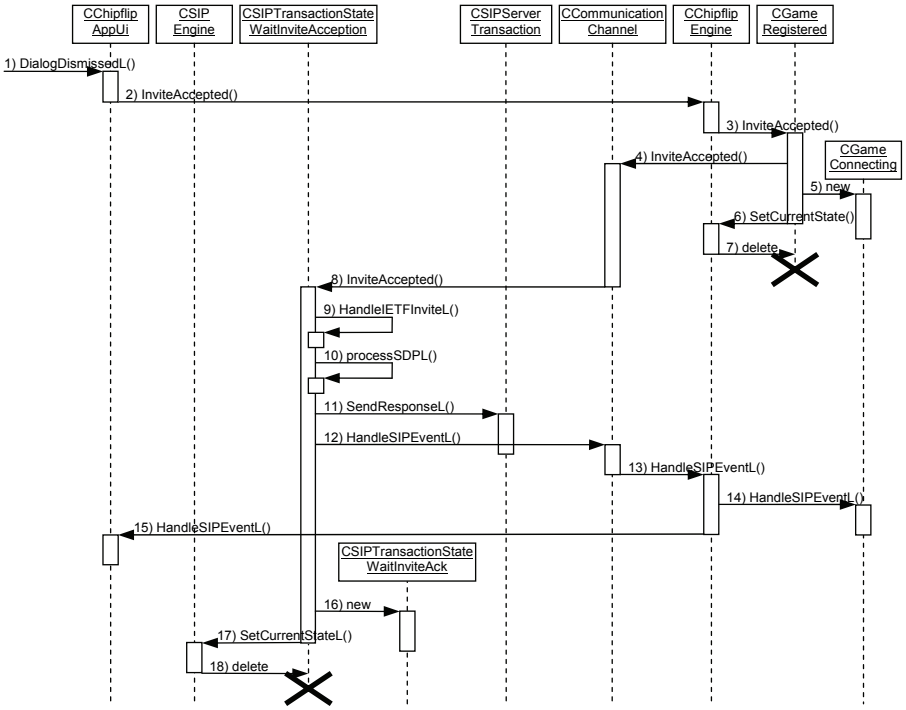

Figura 30. Accettazione dell'Invite

virtual nella classe `MProgressDialogCallback` e implementata nella `CChipflipAppUi`, che ne deriva. Questa, qualunque sia la scelta compiuta dall'utente, richiama [2] la `CChipflipEngine::InviteAccepted()`, passandole però come parametro il valore 1 se questi ha accettato l'invito o il valore 0 se l'ha rifiutato.

La `CChipflipEngine::InviteAccepted()`, come al solito, richiama la funzione omonima definita all'interno della classe corrispondente allo stato attuale del sistema dal punto di vista della comunicazione, passando a quest'ultima il parametro ricevuto. Se tutto va come previsto, questo dovrebbe essere "registrato"; il valore corrispondente alla scelta compiuta dall'utente viene quindi passato [3] alla `CChipflipEngine::CGameRegistered::InviteAccepted()`.

All'interno di quest'ultima funzione, per prima cosa il valore ricevuto viene passato [4] alla `CCommunicationChannel::InviteAccepted()`; quindi, se l'invito è stato accettato, si impostano le variabili che rappresentano lo stato dell'utente e dell'interlocutore in modo che inizialmente chi è stato invitato parli e chi ha mandato l'invito ascolti. Infine lo stato della comunicazione viene modificato in "connessione in corso" creando [5] un oggetto della classe `CGameConnecting` e passandolo [6] alla `CChipflipEngine::SetCurrentState()`, che lo sostituisce [7] a quello che rappresentava lo stato precedente.

La funzione `CCommunicationChannel::InviteAccepted()` non fa che passare il valore che corrisponde alla decisione presa dall'utente alla [8] `CSIPTransactionStateWaitInviteAcception::InviteAccepted()`.

Quest'ultima, sempre nel caso in cui il profilo in uso sia di tipo IETF, esamina la decisione presa dall'utente e invia a chi ha mandato l'INVITE la risposta corrispondente. In particolare, se l'utente ha accettato l'invito si richiama [9] la `CTransactionStateWaitInviteAcception::HandleIETFInviteL()` [13] e, terminate le diverse operazioni che quest'ultima esegue, si varia lo stato del sistema dal punto di vista della segnalazione SIP in "attesa dell'ACK" creando [16] un oggetto della classe `CSIPTransactionStateWaitInviteACK` e passandolo [17] alla funzione `CSIPEngine::SetCurrentStateL()`, che lo sostituisce [18] a quello che rappresentava lo stato precedente. Se invece l'invito è stato rifiutato, si genera il messaggio SIP di 486 REJECTED e lo si spedisce all'altro terminale chiamando la funzione `CSIPServerTransaction::SendResponseL()`, definita all'interno del plug-in SIP, dopodiché si riporta il sistema allo stato "registrato" creando un oggetto della classe `CSIPTransactionStateRegistered` ed assegnandolo a `iCurrentState` mediante la funzione `CSIPEngine::SetCurrentStateL()`.

La funzione `CSIPTransactionStateWaitInviteAcception::HandleIETFInviteL()`, richiamata se l'utente ha accettato l'invito, per prima cosa elabora il pacchetto SDP contenuto (in formato testuale) nel messaggio SIP di INVITE creando a partire da questo un oggetto della classe `CSDPDocument` e passando un puntatore a quest'ultimo alla [10] `CSIPTransactionStateWaitInviteAcception::ProcessSDPL()` [14].

Quest'ultima funzione accetta in ingresso il messaggio SDP ricevuto dall'interlocutore, ne estrapola i dati necessari ad impostare i parametri che serviranno per il successivo scambio di dati (principalmente l'indirizzo IP e la porta a cui inviare i pacchetti) nonché i vari parametri che identificano la comunicazione e definiscono il formato dei dati scambiati; nel contempo la stessa funzione genera, a partire dalle informazioni ottenute e dalle caratteristiche del sistema in uso, il messaggio SDP che verrà inviato in risposta a quello appena ricevuto. Un puntatore a quest'ultimo, restituito dalla `ProcessSDPL()`, viene utilizzato dalla `HandleIETFInviteL()` per inserire il pacchetto SDP appena creato all'interno del messaggio SIP di risposta (200 OK), che viene costruito subito dopo il termine dell'elaborazione del pacchetto SDP ricevuto.

Il messaggio SIP così costruito viene spedito all'interlocutore mediante la funzione [11] `CSIPServerTransaction::SendResponseL()`, appartenente alla API SIP. Se l'operazione di invio va a buon fine, sempre nella `HandleIETFInviteL()` viene richiamata la funzione [12] `CCommunicationChannel::`

[13] In realtà la funzione `HandleIETFInviteL()` è definita unicamente all'interno della classe `CSIPTransactionStateRegistered`, ma la classe `CSIPTransactionStateWaitInviteAcception` eredita da questa.

[14] Anche questa funzione in realtà è definita nella classe `CSIPTransactionStateRegistered`.

`HandleSIPEventL()` passandole il parametro `ES200Sent`. Quest'ultima, in tal caso, non fa che passare il parametro ricevuto alla funzione omonima [13] definita all'interno della classe `CChipflipEngine`.

La `CChipflipEngine::HandleSIPEventL()` a sua volta passa il parametro stesso alle due funzioni con lo stesso nome definite nella classe corrispondente allo stato attuale della comunicazione [14] e nella classe `CChipflipAppUi` [15].

La funzione `CChipflipEngine::CGameConnecting::HandleSIPEventL()` non compie alcuna operazione, mentre la `CChipflipAppUi::HandleSIPEventL()`, se il parametro passatole è `ES200Sent`, mostra all'utente un messaggio in cui lo si avverte dell'avvenuto invio del messaggio 200 OK. Questo messaggio però, tipicamente, resta visibile solo per una frazione di secondo, poiché, l'interlocutore, appena riceve il 200 OK, termina l'handshake rispondendo col messaggio SIP di ACK.

Al momento della ricezione del messaggio di ACK, che costituisce il termine dell'handshake SIP, viene invece eseguita la sequenza di operazioni illustrata in figura 31 e descritta di seguito.

Per prima cosa il programma viene informato della ricezione di un messaggio da parte del sistema operativo, tramite il plug-in SIP, mediante la stessa funzione [1] `IncomingRequest()`[15], definita come *pure virtual* nella classe `MSIPConnectionObserver` (appartenente al plug-in) e implementata in `CSIPEngine`, che era stata richiamata in seguito alla ricezione dell'INVITE. Questa non fa che invocare la funzione omonima definita all'interno della classe corrispondente allo stato corrente (considerato dal punto di vista della segnalazione SIP). Il sistema dovrebbe trovarsi proprio nello stato "attesa dell'ACK": sarà perciò richiamata [2] la funzione `CSIPTransactionStateWaitInviteACK::IncomingRequest()`.

All'interno di questa, se il messaggio ricevuto è proprio l'ACK relativo alla transazione in corso, viene memorizzato l'indirizzo IP dell'interlocutore, vengono richiamate [3] la `CCommunicationChannel::HandleSIPEventL()` con parametro `ESAckReceived` e [8] la `CCommunicationChannel::SIPConnectSocketL()` e, se non si verificano errori, si imposta lo stato dal punto di vista della segnalazione SIP in "connesso" creando [19] un oggetto della classe `CSIPTransactionStateSessionConnected` e passandolo [20] alla `CSIPEngine::SetCurrentStateL()`, che lo sostituisce [21] a quello che rappresentava lo stato precedente.

[15] Si ricorda che il messaggio SIP di ACK non è che una richiesta, con l'unica particolarità di non richiedere alcuna risposta.

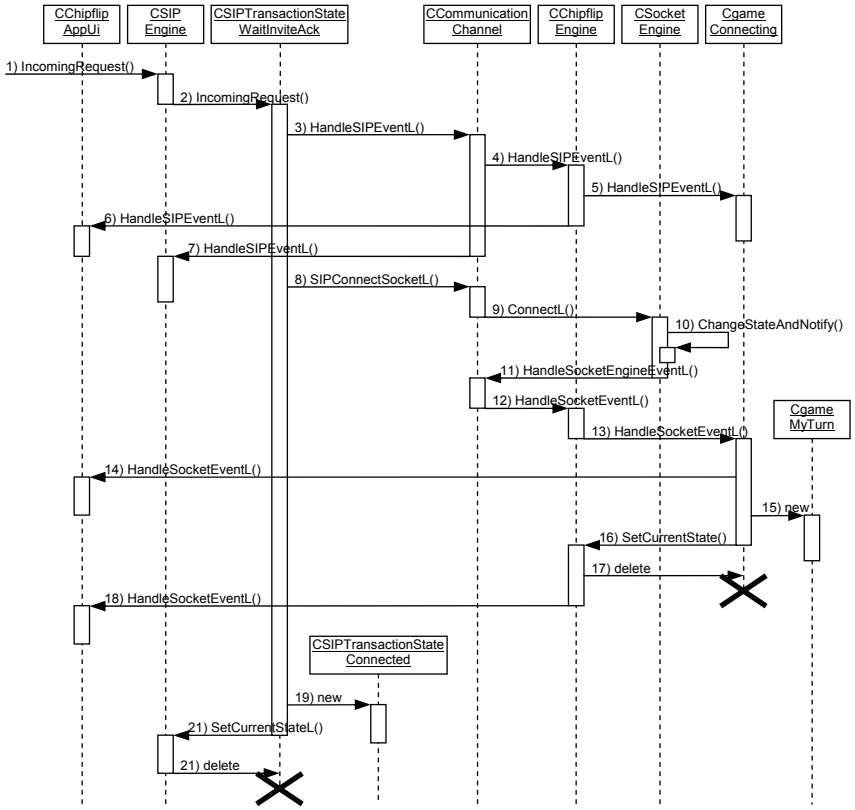

Figura 31. Ricezione del 200 Ok

La funzione `CCommunicationChannel::HandleSIPEventL()` non fa che passare il parametro `ESAckReceived` alla [4] `CChipflipEngine::HandleSIPEventL()`, la quale lo passa alle funzioni omonime appartenenti alla classe relativa allo stato corrente della comunicazione [5] e alla classe `CChipflipAppUi` [6].

Lo stato corrente, dal punto di vista della connessione, dovrebbe essere "connessione in corso", quindi viene richiamata per prima la `CChipflipEngine::CGameConnecting::HandleSIPEventL()`, che non compie alcuna operazione. La funzione `CChipflipAppUi::HandleSIPEventL()`, invece, se il valore ricevuto è `ESAckReceived`, visualizza la finestra di dialogo con cui informa l'utente della ricezione dell'ACK. Ma tale finestra rimane sul display del dispositivo solo per una frazione di secondo, poiché, come si è detto, all'interno della `CSIPTransactionStateWaitInviteAck::IncomingRequest()`, subito dopo il termine della `CCommunicationChannel::HandleSIPEventL()` viene richiamata la `CCommunicationChannel::SIPConnectSocketL()`, che, tra le altre operazioni, informa l'utente della conclusione della fase di connessione.

La `SIPConnectSocketL()` richiama infatti [9] la funzione `CSocketEngine::ConnectL()`. Come si è detto, utilizzando UDP non è necessario compiere l'operazione di connessione, quindi tale funzione non fa che

memorizzare l'indirizzo IP dell'interlocutore e modificare lo stato della socket in "connesso", informandone l'utente, passando [10] alla `CSocketEngine::ChangeStateAndNotify()` il parametro `ESEConnected`. Quest'ultima passa il parametro stesso alla [11] `CCommunicationChannel::HandleSocketEngineEventL()`, che lo inoltra [12] alla `CChipflipEngine::HandleSocketEventL()`. Questa chiama le funzioni aventi lo stesso nome appartenenti alla classe corrispondente allo stato attuale della comunicazione [13] e alla `CChipflipAppUi` [14].

La `CChipflipEngine::CGameConnecting::HandleSocketEventL()`, quando viene richiamata con parametro `ESEConnected`, siccome è stato stabilito che l'utente che ha ricevuto l'Invite inizi a parlare, modifica lo stato del sistema in "trasmissione" (creando [15] un oggetto della classe `CGameMyTurn` e passandolo [16] alla `CChipflipEngine::SetCurrentState()`, che lo sostituisce [17] a quello che corrispondeva allo stato precedente) e inizia l'operazione di invio dei dati audio campionati, come descritto nel paragrafo relativo a tale funzionalità.

Invece la funzione `CChipflipAppUi::HandleSocketEventL()`, se le viene passato il parametro `ESEConnected`, come si è detto rimpiazza la finestra aperta poco prima con una che informa l'utente della conclusione dell'handshake SIP.

➢ Campionamento e invio dei dati audio

Al termine della fase di handshake SIP, l'utente che ha ricevuto l'invito può iniziare immediatamente a parlare, mentre colui che l'ha invitato si pone in ascolto. Ciò significa che l'applicativo in esecuzione sui due terminali dovrà in un caso campionare il segnale audio proveniente dal microfono e trasmetterlo, mentre nell'altro sarà necessario ricevere i dati in arrivo e mandarli in riproduzione sull'altoparlante. Verrà ora descritta l'implementazione della prima di queste operazioni. Le funzioni principali in essa coinvolte sono altresì rappresentate in figura 32.

Come si è accennato, quando si conclude l'instaurazione della connessione tramite SIP, viene richiamata la funzione `CChipflipEngine::CGameConnecting::HandleSocketEventL()`, passandole il parametro `ESEConnected`. Questa, se l'utente del terminale su cui sta girando l'applicazione era quello che aveva ricevuto l'invito, modifica lo stato in "trasmissione", creando [1] un nuovo oggetto della classe `CGameMyTurn` e assegnandolo a `iCurrentState` attraverso la funzione `SetCurrentState()`.

In questa circostanza è possibile notare una delle particolarità relative al metodo di costruzione in due fasi tipico del linguaggio C++ per Symbian: la classe `CGameMyTurn` eredita da `CGameConnected`, quindi nel momento in cui si costruisce un'istanza della classe `CGameMyTurn` viene automaticamente richiamato il costruttore standard di `CGameConnected`. Ma la classe `CGameConnected` prevede che la costruzione avvenga col metodo in due fasi, quindi il costruttore standard non

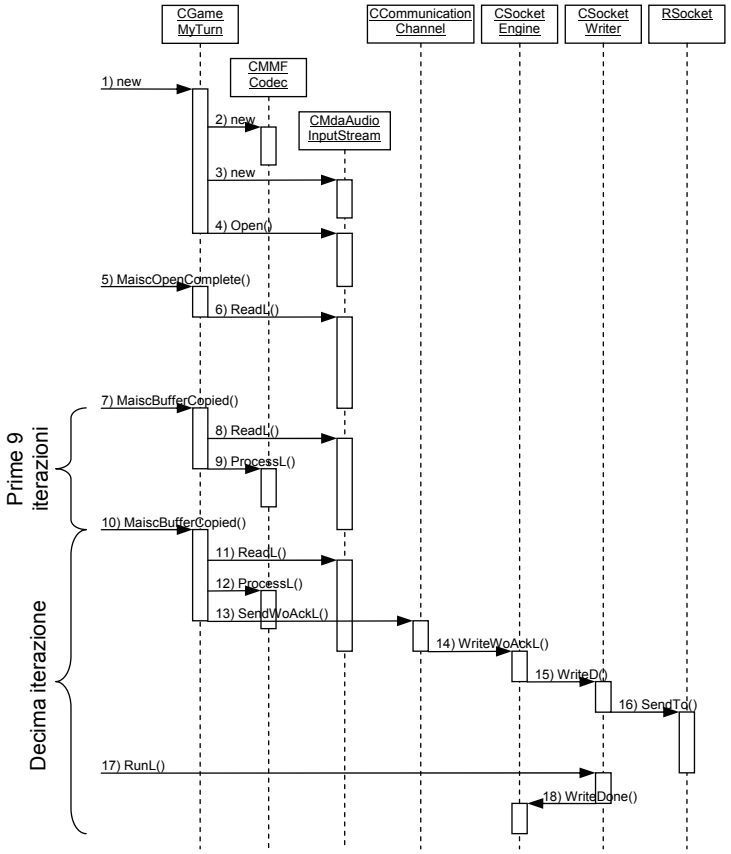

Figura 32. Campionamento e invio dei dati audio

compie alcuna operazione: la creazione degli oggetti contenuti all'interno della classe avviene infatti nella `ConstructL()`. Per questo motivo, è necessario richiamare esplicitamente la `CGameConnected::ConstructL()` dall'interno della `CGameMyTurn::ConstructL()`. Tale operazione, com'è ovvio, deve essere effettuata dallo sviluppatore ogniqualvolta la classe da cui si eredita utilizzi il metodo di costruzione in due fasi.

Tornando al progetto, quindi, durante la creazione dell'oggetto della classe `CGameMyTurn` viene richiamato anche il costruttore della classe `CGameConnected`, nel quale si creano o si inizializzano gli oggetti che serviranno in tutte le classi derivate da `CGameConnected`. In particolare, nella `CGameConnected::ConstructL()` vengono creati i due oggetti della classe `CMMFDescriptorBuffer` (apposita per la gestione di dati multimediali) che serviranno a contenere i blocchi di dati audio PCM e AMR durante le operazioni di codifica e decodifica. Si crea poi [2] l'oggetto che eseguirà l'operazione di codifica AMR. Esso è semplicemente un'istanza della classe `CMMFCodec`, la quale classe comprende però diversi tipi di codec. Per generare il particolare oggetto che esegue la codifica voluta (da PCM su 16 bit a AMR a 4,75 kbit/s), occorre passare alla `CMMFCodec::NewL()` il parametro 0x101FAF68, mentre passando altri parametri vengono generati altri tipi di codec. Il passo successivo consiste nella creazione dell'array di puntatori a buffer che conterranno i dati audio PCM man

mano che vengono campionati. Esso consiste in un oggetto di tipo `RPointerArray` in cui ogni puntatore è associato a un buffer da 320 byte. L'oggetto che conterrà i dati codificati in AMR, costruito subito dopo, è invece un buffer unico, di dimensione pari al prodotto tra il numero di trame AMR inserite in ogni pacchetto RTP (posto pari a 10) e la lunghezza di una di tali trame (13 byte). L'ultimo oggetto che si crea [3], della classe `CMdaAudioInputStream`, è quello che si occuperà del campionamento del segnale proveniente dal microfono. Dopo aver creato tutti gli oggetti che serviranno, si richiede al sistema operativo l'apertura dello stream dal microfono del dispositivo, invocando [4] la funzione di libreria `CMdaAudioInputStream::Open()`.

L'operazione di apertura non è bloccante, ma è previsto che l'esecuzione del programma possa continuare; terminata l'apertura dello stream (operazione che in determinati casi può richiedere un tempo elevato), il sistema operativo richiama [5] la funzione callback denominata `MaiscOpenComplete()`, definita come *pure virtual* nella classe `MMdaAudioInputStreamCallback` e implementata nella classe `CChipflipEngine::CGameMyTurn`, che da essa eredita. Nella `CChipflipEngine::CGameMyTurn::MaiscOpenComplete()` si regolano alcuni parametri relativi al flusso audio che verrà campionato, quali il guadagno di registrazione, la frequenza di campionamento e il numero di canali (il numero di bit per campione è invece fisso e pari a 16), e si avvia l'operazione di lettura dal microfono sul primo dei buffer che compongono l'array, mediante la funzione [6] `CMdaAudioInputStream::ReadL()`.

Anche l'operazione di lettura, come si è detto in precedenza, è gestita in modo asincrono: quando il buffer passato alla `ReadL()` è stato riempito, viene richiamata [7] la callback `MaiscBufferCopied`, anch'essa definita nella classe `MMdaAudioInputStreamCallback` e implementata in `CChipflipEngine::CGameMyTurn`. Per comprendere ciò che accade all'interno di questa funzione, occorre esaminare la soluzione scelta per l'invio dei dati audio. Ogni buffer contenente i dati campionati in PCM, come si è detto, viene codificato in AMR; quando sono stati riempiti 10 buffer (corrispondenti a un tempo di 200 ms), al blocco complessivo viene aggiunto l'header RTP e il pacchetto risultante è trasmesso all'interlocutore. Il numero di buffer da riempire prima dell'invio è stato scelto come compromesso tra il ritardo introdotto e l'overhead causato dall'aggiunta degli header RTP e UDP ad ogni pacchetto. Un ritardo di 200 ms per il solo campionamento sarebbe assolutamente inaccettabile se la comunicazione fosse in tempo reale, ma nel caso del push-to-talk può essere tollerato senza eccessivi problemi.

L'implementazione scelta per la `MaiscBufferCopied()` sfrutta le caratteristiche di multithreading proprie del sistema operativo Symbian: per 9 volte, infatti, si avvia il campionamento del segnale audio dal microfono del dispositivo sul buffer successivo a quello che è stato appena riempito e contemporaneamente si codifica il buffer corrente in AMR; alla decima iterazione, invece, si ricomincia a campionare il segnale sul primo buffer e intanto, oltre a codificare in AMR l'ultimo

buffer, si spedisce all'interlocutore l'intero pacchetto risultante. In dettaglio, nelle prime 9 iterazioni inizialmente si richiama [8] la `CMdaAudioInputStream::ReadL()` passandole il buffer successivo a quello corrente; questa funzione, come si è visto, non è bloccante, quindi il controllo torna immediatamente al programma. Questo prima esegue la conversione da PCM in AMR chiamando [9] la `CMMFCodec::ProcessL()`, poi accoda la trama AMR risultante dalla codifica alle precedenti, e infine incrementa l'indice che rappresenta il buffer su cui si lavora. Alla decima iterazione [10], invece, si passa nuovamente [11] alla `CMdaAudioInputStream::ReadL()` il primo dei buffer contenuti nell'array, si converte l'ultimo in AMR chiamando [12] la `CMMFCodec::ProcessL()`, si accoda la trama AMR risultante alle precedenti e si spedisce il buffer completo passandolo alla funzione [13] `CCommunicationChannel::SendWoAckL()`. Infine si cancella l'intero contenuto di tale buffer, in modo che il ciclo possa ricominciare. Il codice relativo alla funzione `MaiscBufferCopied()`, che realizza le operazioni descritte, è il seguente:

```
void CChipflipEngine::CGameMyTurn::MaiscBufferCopied
   (TInt aError, const TDesC8&)
{
  if (aError == KErrNone)
  {
    if(workBuff < KBuffers-1)
    {
      iMdaAudioInputStream->ReadL(*PCMBufferTx[workBuff
  + 1]);
      PCMTemp->Data().Copy(*PCMBufferTx[workBuff]);
      TCodecProcessResult result = CodecPCM2AMR-
  >ProcessL(*PCMTemp, *AMRTemp);
      completeBuffer->Des().Append(AMRTemp->Data());
      workBuff++;
    }
    else
    {
      iMdaAudioInputStream->ReadL(*PCMBufferTx[0]);
      PCMTemp->Data().Copy(*PCMBufferTx[workBuff]);
      TCodecProcessResult result = CodecPCM2AMR-
  >ProcessL(*PCMTemp, *AMRTemp);
      completeBuffer->Des().Append(AMRTemp->Data());
      iEngine.SendWoAckL(*completeBuffer);
      workBuff = 0;
      completeBuffer->Des().Delete(0,13*KBuffers);
    }
  }
[...]
}
```

La `CCommunicationChannel::SendWoAckL()` è stata inserita nel progetto per effettuare l'invio di un pacchetto di dati all'interlocutore senza attendere il relativo acknowledgement. Analogamente alla `CCommunicationChannel::SendToRemoteL()`, già presente in *Chipflip*, di questa funzione sono state create due versioni: la prima accetta in ingresso dati di tipo `TDesC`, li converte in `TDesC8` e richiama la seconda versione della stessa funzione; quest'ultima, invece, riceve in ingresso dati di tipo `TDesC8` e, dopo aver verificato che lo stato del sistema relativo alla socket sia "connesso", passa i dati ricevuti [14] alla `CSocketEngine::WriteWoAckL()`. Anche di tale funzione, per analogia con la `WriteL()` già presente in Chipflip, esistono due versioni, una delle quali accetta in ingresso dati di tipo `TDesC`, mentre l'altra opera su dati di tipo `TDesC8`. Il comportamento delle due funzioni è però assolutamente identico.

All'interno della funzione `CSocketEngine::WriteWoAckL()`, se, come nel caso in esame, la lunghezza dei dati da trasmettere è maggiore di 5 byte (condizione che, a causa della particolare struttura del progetto, indica che si stanno trasmettendo dati audio), dopo aver verificato che non sia già in corso un'altra operazione di scrittura, i dati in ingresso vengono fatti precedere dall'header RTP e il pacchetto RTP completo viene inviato all'interlocutore.

Più nel dettaglio, per la generazione dell'header RTP, nel costruttore della classe `CSocketEngine` vengono generati i numeri casuali che serviranno come valori iniziali dei campi *Sequence number* e *Timestamp* (che verranno incrementati a ogni invio) e come valore del campo *Synchronization source* (che rimarrà costante per l'intera durata della comunicazione). Il generatore di numeri casuali viene inizializzato chiamando la funzione `Math::Rand()`, a cui si passa un parametro dipendente dall'istante in cui viene eseguita l'applicazione. Poi, ogni volta che si richiama la `CSocketEngine::WriteWoAckL()`, se non è già in corso un'operazione di scrittura sulla socket si aggiunge al blocco di dati audio l'header AMR e si trasmette il pacchetto completo richiamando [15] la `CSocketWriter::WriteD()`, a cui viene passato sia il pacchetto stesso che l'indirizzo a cui questo deve essere spedito; in seguito, indipendentemente dal risultato del tentativo di trasmissione, si incrementa di 1 il sequence number e si aggiunge al timestamp il numero di campioni contenuti nel blocco che si è tentato di spedire. È da notare come l'incremento dei due campi in questione avvenga anche se l'operazione di invio ha dato esito negativo: infatti i valori contenuti in tali campi devono rappresentare (rispettivamente) il numero di pacchetti e di campioni che il mittente ha tentato di spedire, e non quelli che sono effettivamente stati spediti. Il destinatario, in tal modo, ha la possibilità di verificare se un pacchetto è stato perso e, eventualmente, di prendere le opportune contromisure. Se l'operazione di scrittura non è possibile, l'intero blocco di dati viene semplicemente cancellato. Sarebbe stato infatti possibile ritentare la trasmissione in un secondo momento, ad esempio quando il blocco successivo è pronto per la trasmissione, ma operando in questo modo si sarebbero potuti introdurre ritardi eccessivi e in ogni caso non si sarebbe ottenuta la

certezza assoluta che tutti i dati campionati fossero ricevuti correttamente dall'interlocutore (un pacchetto UDP, infatti, può anche essere scartato da un nodo intermedio, senza che si possa far nulla per evitare che ciò accada).

La funzione `CSocketWriter::WriteD()` prima cancella l'oggetto contenente i dati che le sono stati passati all'iterazione precedente, se questi sono ancora presenti, poi inizia un'operazione di scrittura asincrona sulla socket richiamando dapprima [16] la funzione di libreria `RSocket::SendTo()` e quindi la `SetActive()`. Al termine dell'operazione, il sistema operativo richiama [17] la `CSocketWriter::RunL()`, nella quale si cancella l'oggetto contenente i dati appena spediti e, se la scrittura si è conclusa senza errori, si richiama [18] la `CSocketEngine::WriteDone()`, la quale non effettua alcuna operazione. Al termine il ciclo riprende con il campionamento di un altro pacchetto di dati audio, e così via.

➢ Ricezione e riproduzione dei dati audio

Come si è detto, il terminale dell'utente che ha richiesto l'instaurazione della comunicazione, quando tale operazione è completata, apre lo stream di output, si mette in attesa di ricevere dati audio provenienti dall'interlocutore, e man mano che li riceve li manda in riproduzione sul proprio altoparlante. Nel contempo, al fine di aprire e mantenere aperto un eventuale NAT, lo stesso terminale invia all'altro messaggi di keepalive al ritmo di uno ogni 5 secondi.

La figura 33 mostra unicamente la sequenza di funzioni richiamate per la creazione e l'inizializzazione degli oggetti necessari; al termine di queste, l'elaborazione si interrompe finché il sistema operativo segnala il completamento delle operazioni richieste, come verrà descritto in seguito.

Analogamente a quanto avviene sul terminale che ha ricevuto l'INVITE, anche nel caso in questione al termine dell'handshake SIP viene richiamata la funzione `CChipflipEngine::CGameConnecting::HandleSocketEventL()`, passandole il parametro `ESEConnected`. Nel caso in esame, poiché è stato deciso che inizialmente il chiamante si ponga in ascolto, tale funzione modifica lo stato della

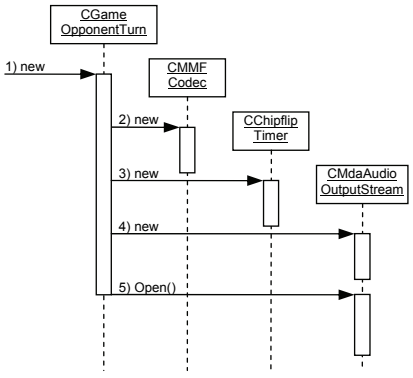

Figura 33. Apertura dello stream in output

comunicazione in "ricezione", creando un nuovo oggetto della classe `CGameOpponentTurn` e assegnandolo a `iCurrentState` attraverso la funzione `CChipflipEngine::SetCurrentState()`.

Viene quindi creata [1] un'istanza della classe `CGameOpponentTurn`, richiamando anche in questo caso, nel corso della costruzione, la `CGameConnected::ConstructL()`, la quale crea i due buffer (oggetti della classe `CMMFDescriptorBuffer`) che saranno utilizzati durante la decodifica dei dati ricevuti. In seguito, nella `ConstructL()` si crea [2] l'oggetto che eseguirà la decodifica delle trame AMR ricevute. Questo è un'istanza della classe `CMMFCodec`, la stessa a cui appartiene l'oggetto (creato all'interno della `CGameMyTurn::ConstructL()`) che effettua l'operazione opposta. La differenza è che ora, per creare il codec desiderato, si passa alla `CMMFCodec::NewL()` un parametro diverso dal precedente, e pari a 0x101FAF67. Quindi, sempre nella `CGameOpponentTurn::ConstructL()`, viene creato l'array (oggetto della classe `CArrayPtrFlat`) che conterrà i puntatori ai pacchetti di dati audio ricevuti e decodificati ma non ancora riprodotti; poi viene generato [3] l'oggetto della classe `CChipflipTimer` (basata a sua volta sulla classe standard `CTimer`) che servirà per inviare i messaggi di keepalive a intervalli di tempo regolari, e infine viene aperto lo stream diretto verso l'altoparlante del dispositivo, creando [4] un oggetto della classe `CMdaAudioOutputStream` e richiamando [5] la funzione `CMdaAudioOutputStream::Open()`. Così come avveniva per l'apertura dello stream in ingresso, tale funzione non è bloccante, ma dopo la sua invocazione il controllo è restituito immediatamente al programma.

Terminata l'operazione di apertura dello stream viene iniziata la spedizione dei keepalive ad intervalli regolari,eseguendo le funzioni mostrate in figura 34. Inizialmente il sistema operativo richiama [1] la funzione callback `MaoscOpenComplete()`, definita come *pure virtual* nella classe `MMdaAudioOutputStreamCallback` e implementata in `CChipflipEngine::CGamOpponentTurn`, che da essa eredita. All'interno di tale funzione per prima cosa viene creato il messaggio di keepalive (costituito dal solo carattere ASCII "k"[16]), che poi viene spedito utilizzando [2] la stessa funzione `CCommunicationChannel::SendWoAckL()` che veniva invocata per l'invio dei pacchetti di dati audio. Tale funzione, anche in questo caso, non fa che richiamare [3] la `CSocketEngine::WriteWoAckL()`, che a sua volta, se non sono già in corso operazioni di invio, passa il messaggio e l'indirizzo del destinatario alla [4] `CSocketWriter::WriteD()`. Questa, però, ora riconosce dalla limitata lunghezza dei dati che le vengono passati che non si tratta di dati audio e quindi non inserisce l'header RTP ma spedisce unicamente il messaggio all'interlocutore richiamando [5] la

[16] Si è scelto di non inviare come keepalive un pacchetto vuoto poiché talvolta tali pacchetti non vengono inoltrati da alcuni router e possono quindi non giungere a destinazione.

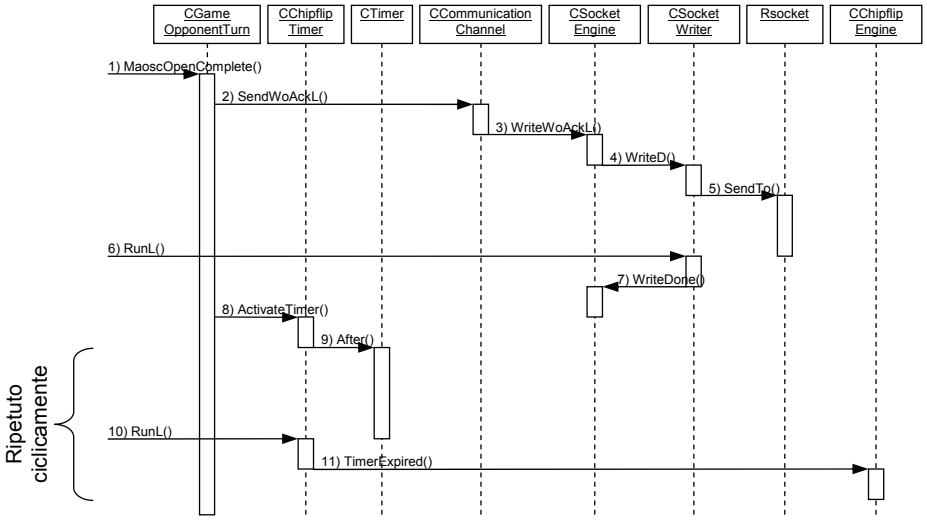

Figura 34. Invio dei keepalive

`RSocket::SendTo()` e la `SetActive()`. Anche in questo caso, se l'invio non è momentaneamente possibile, il messaggio viene scartato: in ogni caso, se ne ritenterà la trasmissione dopo pochi secondi. Se invece l'operazione va a buon fine, il sistema operativo richiama [6] la `CSocketWriter::RunL()`, la quale rimuove dalla memoria i dati spediti e richiama [7] la `CSocketEngine::WriteDone()`, che però non compie alcuna operazione.

Dopo aver inviato il primo keepalive, sempre nella funzione `CChipflipEngine::CGameOpponentTurn::MaoscOpenComplete()` vengono richiamate le funzioni necessarie a spedire i successivi ad intervalli regolari. In particolare viene puntato il timer allo scadere del quale dovrà essere ripetuta l'operazione di invio, richiamando [8] la `CChipflipTimer::ActivateTimer()` e passandole il tempo (misurato in microsecondi) che dovrà passare prima del successivo invio. Questo, nel progetto in esame, è stato fissato pari a 5 secondi, in modo da mantenere aperto il NAT anche se questo è stato configurato in modo da chiudersi dopo poco tempo dall'ultimo pacchetto transitato e anche nel caso in cui un keepalive dovesse andare perduto.

La funzione `CChipflipTimer::ActivateTimer()`, dopo aver verificato che non sia in corso un altro conteggio, chiama la funzione di libreria [9] `CTimer::After()`, che richiede al sistema operativo di ricevere un'opportuna segnalazione allo scadere del periodo di tempo desiderato. Come sempre, la gestione di tale evento asincrono è gestita mediante la funzione `CChipflipTimer::RunL()`, che, invocata dal sistema operativo allo scadere del timer [10], a sua volta richiama [11] la `CChipflipEngine::TimerExpired()`. Questa non fa che richiamare la funzione omonima appartenente all'oggetto che corrisponde allo stato attuale del sistema, vale a dire la `CChipflipEngine::CGameOpponentTurn::TimerExpired()`. All'interno di questa funzione, in maniera assolutamente analoga

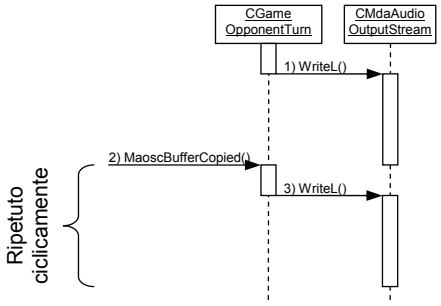

Figura 35. Riproduzione dei dati audio

a quanto accadeva nella `MaoscOpenComplete()`[17], si crea un altro messaggio di keepalive, lo si invia e si imposta nuovamente il timer allo scadere del quale l'operazione dovrà essere ripetuta. In questo modo, sfruttando le caratteristiche di multithreading proprie del sistema operativo Symbian, si ottiene l'invio dei messaggi di keepalive a un ritmo costante, e nel contempo si dà la possibilità al sistema di compiere le operazioni necessarie per la ricezione e la riproduzione dei pacchetti di dati audio in arrivo.

L'operazione di riproduzione (visualizzata in figura 35 e separata da quella di ricezione dei dati, come spiegato più avanti) è iniziata anch'essa nella `CChipflipEngine::CGameOpponentTurn::MaoscOpenComplete()`, subito dopo aver impostato il timer per l'invio dei keepalive. In particolare, vengono impostati alcuni parametri relativi al flusso di dati audio che saranno riprodotti, quali la frequenza di campionamento, il numero di canali[18] e il volume di riproduzione, e, se il buffer che contiene i pacchetti di dati audio ricevuti e decodificati non è vuoto, viene avviata la riproduzione del primo pacchetto di dati PCM tra quelli presenti, passandolo [1] alla `CMdaAudioOutputStream::WriteL()`. In caso contrario, si imposta un apposito flag in modo che non appena si riceve un blocco di dati se ne avvii la riproduzione.

In questa fase sarebbe stato molto semplice creare un buffer di riproduzione in modo da compensare piccole variazioni del ritardo con cui i pacchetti (emessi, come si è visto, a bitrate costante) arrivano a destinazione: sarebbe stato infatti sufficiente dare inizio alla riproduzione dei dati solamente nel momento in cui fosse già stato ricevuto un congruo numero di pacchetti. Si è invece scelto di iniziare subito la riproduzione perché in questo modo, nel caso in cui le variazioni dei ritardi fossero contenute, si eviterebbe di introdurre un ulteriore ritardo; se invece un pacchetto arriva a destinazione con un ritardo eccessivo, si avrà una temporanea degradazione della qualità del segnale riprodotto ma, se il ritardo dei pacchetti successivi non aumenta, da quel momento in

[17] A causa di questa analogia, la sequenza di operazioni necessarie non è mostrata in figura.

[18] Tali parametri, ovviamente, devono coincidere con quelli impostati all'interno della funzione che si occupa del campionamento.

poi si avranno sempre a disposizione dati da riprodurre. In altre parole, se un pacchetto arriva a destinazione con un certo ritardo, da quel momento si ottiene automaticamente un buffer di durata pari a tale ritardo, che sarà sufficiente se i pacchetti successivi non giungeranno a destinazione con un ritardo superiore. D'altronde, anche creando esplicitamente un buffer di una certa lunghezza, non si ha mai la certezza che questo sia sufficiente a compensare totalmente le variazioni del ritardo di trasmissione, specialmente se alcuni dei pacchetti trasmessi vengono persi. Si può notare come, d'altra parte, non sia necessario neppure cautelarsi contro eventuali problemi dovuti a un aumento incontrollato della quantità di dati memorizzati: come si è visto, infatti, i pacchetti sono emessi al tasso costante di uno ogni 20 ms, e ovviamente vengono riprodotti allo stesso ritmo. Quindi, a meno di duplicazioni dei pacchetti in arrivo (eventualità teoricamente possibile ma mai verificatasi durante tutte le prove svolte), non può mai accadere che la velocità con cui vengono riprodotti i dati in arrivo sia inferiore alla velocità con cui essi vengono ricevuti. Tuttavia, è possibile cautelarsi anche da questa eventualità in maniera molto semplice, modificando la funzione che riceve i dati (discussa più avanti) in modo che questi vengano scartati se il numero di pacchetti in memoria supera una determinata soglia.

In ogni caso, quando il blocco di dati audio fornito in ingresso alla `CMdaAudioOutputStream::WriteL()` è passato agli strati inferiori del multimedia framework per essere riprodotto, il sistema operativo richiama [2] la callback `MaoscBufferCopied()`, definita come *pure virtual* nella classe `MMdaAudioOutputStreamCallback` e implementata in `CChipflipEngine::CGameOpponentTurn`, che da essa deriva. All'interno di tale funzione si controlla se nel buffer contenente i pacchetti di dati audio ricevuti (compreso quello appena riprodotto) siano presenti almeno due elementi oppure se ve ne sia soltanto uno. Se si verifica il primo caso significa che sono disponibili nuovi dati da riprodurre, dunque si elimina dalla memoria il pacchetto appena riprodotto, si cancella il corrispondente puntatore dall'array facendo traslare in avanti tutti gli altri e si avvia la riproduzione del blocco di dati ricevuto per primo tra quelli disponibili, passandolo [3] alla `CMdaAudioOutputStream::WriteL()` e facendo così ripartire il ciclo. Nel caso contrario significa che il pacchetto appena riprodotto è l'unico disponibile, ovvero che dopo di esso non sono stati ricevuti altri dati. Questa condizione può essere dovuta a varie cause: l'interlocutore può aver deciso volontariamente di bloccare l'invio dei dati, oppure la comunicazione si può essere interrotta per un problema di rete, ma è anche possibile che l'ultimo pacchetto inviato sia andato perduto o abbia accumulato un ritardo eccessivo. Negli ultimi due casi, trascorso un brevissimo intervallo di tempo, la ricezione dei dati ricomincerebbe senza problemi. Ora, dunque, interrompere la riproduzione non sarebbe la soluzione ottimale: infatti, se trascorre solo una frazione di secondo tra la chiusura dello stream diretto all'altoparlante e la sua riapertura, l'utente percepisce un "clic" che, specie se ripetuto diverse volte, risulta fastidioso. Naturalmente, nel momento in cui si verifica un'interruzione nella ricezione

dei dati, non se ne può conoscere preventivamente la causa e la durata. Per questo motivo si è deciso di fare in modo che, se nell'array è presente un solo elemento, si continui a passare il corrispondente pacchetto alla `CMdaAudioOutputStream::WriteL()` per un certo numero di volte (impostato pari a 10), trascorse le quali si elimina il pacchetto e si interrompe la riproduzione, semplicemente cessando di richiamare la funzione `CMdaAudioOutputStream::WriteL()`. Il codice relativo alla funzione `MaoscBufferCopied()`, che esegue le operazioni descritte, è il seguente:

```
void
   CChipflipEngine::CGameOpponentTurn::MaoscBufferCopied
   (TInt aError, const TDesC8&)
{
  if(aError == KErrNone)
  {
    if (iAudioArrayRx && iAudioArrayRx->Count() > 1)
    {
      iRepetitions = 0;
      delete iAudioArrayRx->At( 0 );
      iAudioArrayRx->Delete( 0 );
      iAudioArrayRx->Compress();
      playPossible = EFalse;
      iMdaAudioOutputStream->WriteL(*iAudioArrayRx-
  >At(0));
    }
    else if (iAudioArrayRx)
    {
      if(iRepetitions <= KMaxRepetitions)
      {
        iMdaAudioOutputStream->WriteL(*iAudioArrayRx-
  >At(0));
        iRepetitions++;
      }
      else
      {
        delete iAudioArrayRx->At( 0 );
        iAudioArrayRx->Delete( 0 );
        iAudioArrayRx->Compress();
        playPossible = ETrue;
      }
    }
  }
[...]
}
```

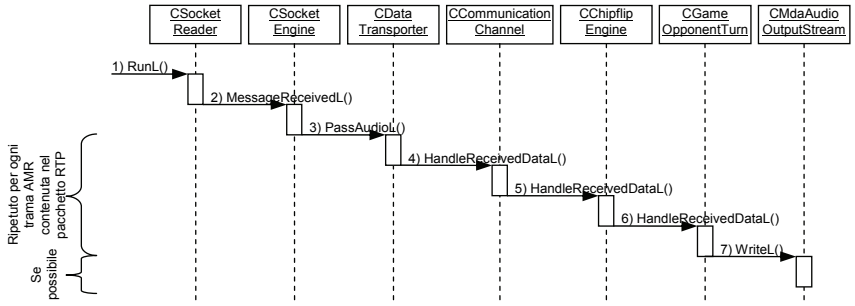

Figura 36. Ricezione dei pacchetti di dati

Se la riproduzione si interrompe a causa della mancanza di dati, il sistema operativo richiama automaticamente la callback `MaoscPlayComplete`, anch'essa definita nella classe `MMdaAudioOutputStreamCallback` e implementata in `CChipflipEngine::CGameOpponentTurn`, passandole come parametro il valore `KErrUnderflow`. Tale funzione, che viene richiamata anche quando l'utente interrompe volontariamente la riproduzione dei dati (come descritto più avanti), in questo caso non compie alcuna operazione.

Come si sarà notato, l'operazione di riproduzione dei pacchetti è quasi del tutto indipendente da quella di ricezione: la riproduzione inizia al momento della ricezione del primo pacchetto, ma poi i nuovi dati da riprodurre vengono prelevati da un buffer nel momento in cui è terminata la riproduzione dei precedenti, senza alcuna relazione col momento in cui sono stati ricevuti. Le operazioni di ricezione, decodifica e inserimento nel buffer dei dati audio provenienti dall'interlocutore, mostrate in figura 36 e descritte nel seguito, vengono cioè eseguite in parallelo sia a quelle relative alla riproduzione sia a quelle necessarie per l'invio dei keepalive. Anche in questo caso, vengono sfruttate le capacità di multithreading del sistema operativo Symbian senza la necessità di creare e gestire esplicitamente più thread separate.

L'implementazione della funzionalità di ricezione dei pacchetti di dati audio provenienti dal terminale dell'interlocutore prevede che per prima cosa, all'arrivo di un qualunque pacchetto sulla socket precedentemente aperta, il sistema operativo richiami [1] la `CSocketReader::RunL()` [19]; all'interno di tale funzione, prima di iniziare una nuova operazione di lettura asincrona, il pacchetto ricevuto viene passato [2] alla `CSocketEngine::MessageReceivedL()`.

[19] Come si ricorderà, al termine dell'handshake SIP i programmi in esecuzione su entrambi i terminali richiamavano la funzione `CSocketReader::Read()`, la quale chiamava la `RSocket::RecvFrom()` seguita dalla `SetActive()`. Il sistema operativo, quindi, al momento della ricezione di un pacchetto, richiama automaticamente la funzione `RunL()` definita nella classe `CSocketReader`, che eredita da `CActive`.

Se il pacchetto ricevuto contiene dati audio, la sua dimensione sarà certamente maggiore di 5 byte: infatti, solo l'header RTP ha una lunghezza pari a 12 byte. La funzione `CSocketEngine::MessageReceivedL()` quindi, esaminando semplicemente la dimensione del pacchetto in arrivo riconosce che si tratta di dati audio e li elabora in maniera opportuna. In particolare, se il sistema era impegnato nell'operazione di commutazione tra invio e ricezione dei dati (descritta più avanti), il pacchetto ricevuto viene semplicemente scartato, in modo da non causare intralci all'operazione stessa. In caso contrario, si elimina l'header RTP del pacchetto ricevuto[20] e se ne suddivide il payload in frammenti di dimensioni pari a 13 byte, ognuno dei quali corrisponde a una trama AMR. Questi vengono passati, nell'ordine con cui erano contenuti all'interno del pacchetto, alla funzione [3] `CDataTransporter::PassAudioL()`. Al termine, il pacchetto ricevuto viene rimosso dalla memoria.

La funzione `CDataTransporter::PassAudioL()`, introdotta per analogia alla `CDataTransporter::StoreDataL()` che viene richiamata quando il pacchetto ricevuto contiene dati relativi al cambio di stato, al contrario di quest'ultima non memorizza i dati in un buffer interno alla classe `CDataTransporter`, ma passa ognuna delle trame AMR ricevute alla [4] `CCommunicationChannel::HandleReceivedDataL()`, che a sua volta passa [5] alla `CChipflipEngine::HandleReceivedDataL()` un puntatore ad un oggetto contenente la trama stessa.

La funzione `CChipflipEngine::HandleReceivedDataL()` richiama la funzione omonima appartenente alla classe che corrisponde allo stato attuale del sistema. Se il sistema non si trova (dal punto di vista dell'applicativo) nello stato `CGameOpponentTurn`, la funzione chiamata non fa che scartare i dati ricevuti. Invece la funzione [6] `CChipflipEngine::CGameOpponentTurn::HandleReceivedDataL()` vede che la lunghezza dei dati che le sono stati passati (13 byte) è maggiore di 5, quindi riconosce che si tratta di una trama AMR, la converte in un pacchetto di 320 byte di dati codificati in PCM e accoda il puntatore relativo a quest'ultimo all'array contenente i pacchetti che sono stati ricevuti ma non sono ancora stati riprodotti, dal quale, come si è visto, i dati vengono prelevati man mano che devono essere riprodotti. Se non è ancora in corso una precedente operazione di riproduzione, infine, la funzione stessa inizia immediatamente a riprodurre il primo dei pacchetti ricevuti, passandolo [7] alla `CMdaAudioOutputStream::WriteL()`.

[20] Tale operazione, benché certamente poco elegante, è compiuta in realtà anche da diversi softphone commerciali; d'altra parte nel caso in esame non sussiste la necessità di distinguere un particolare flusso audio e, non avendo implementato un buffer di ricezione, non è comunque possibile riordinare pacchetti ricevuti fuori ordine (caso peraltro assai raro). L'elaborazione dell'header RTP non avrebbe perciò alcuna utilità pratica.

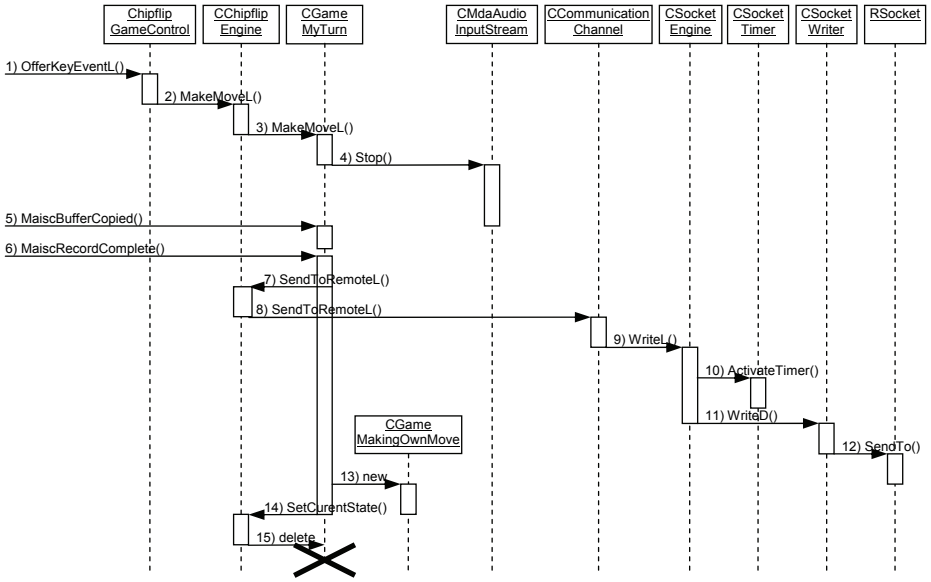

Figura 37. Invio del messaggio e attesa dell'acknowledgement

➢ Commutazione tra invio e ricezione dei dati audio

Nel modello di push-to-talk realizzato, si è previsto che nel corso della comunicazione uno qualsiasi degli utenti possa variare il proprio stato dalla trasmissione alla ricezione o viceversa, dandone comunicazione all'interlocutore e attendendo la ricezione del messaggio di acknowledgement proveniente dal relativo terminale prima di procedere con l'operazione desiderata.

L'implementazione di tale modello prevede che l'utente, per variare il proprio stato, debba unicamente premere il tasto *OK* del terminale. A seguito di tale evento viene eseguita la sequenza di funzioni mostrata in figura 37 e descritta nel seguito.

Il sistema operativo, ogni volta che è stato premuto un tasto, richiama la funzione `OfferKeyEventL()`, che è definita come *virtual* nella classe `CCoeControl` e deve essere implementata in ogni programma che debba gestire eventi generati dalla pressione dei tasti del dispositivo. All'interno del progetto in esame, l'implementazione di tale funzione è fornita all'interno della classe `ChipflipGameControl`, che eredita da `CCoeControl`. Quando l'utente preme *OK*, viene perciò richiamata automaticamente [1] la funzione `ChipflipGameControl::OfferKeyEventL()`, a cui viene passato il parametro `EAknSoftkeyOk`. In questo caso, sia che il sistema fosse in fase di trasmissione sia che fosse in fase di ricezione, tale funzione richiama [2] la `CChipflipEngine::MakeMoveL()`, la quale non fa che chiamare la funzione omonima contenuta nella classe corrispondente allo stato attuale della comunicazione. In questo modo è possibile distinguere se il sistema si trova attualmente in fase di trasmissione o di ricezione e compiere le azioni opportune per il caso in questione.

Se al momento della pressione del tasto *OK* da parte dell'utente il sistema si trovava in fase di trasmissione, quindi, viene chiamata [3] la funzione `CChipflipEngine::CGameMyTurn::MakeMoveL()`. All'interno di questa, per prima cosa, si interrompe il campionamento del segnale audio richiamando [4] la `CMdaAudioInputStream::Stop()`.

Quando viene richiamata questa funzione, il sistema operativo invoca automaticamente la stessa callback chiamata quando viene riempito un blocco di dati audio campionati, ovvero [5] la `MaiscBufferCopied()`, passandole però, oltre ai dati campionati dopo l'ultima volta che tale funzione è stata richiamata, anche il codice di errore `KErrAbort`. In questo caso, la funzione `CChipflipEngine::CGameMyTurn::MaiscBufferCopied()` non fa che scartare i dati ricevuti, in quanto la loro dimensione è inferiore a 320 byte e quindi non è possibile convertirli in AMR, e comunque la perdita di informazioni introdotta è praticamente irrilevante. Inoltre, quando la chiusura dello stream in ingresso dal microfono del dispositivo è completata, viene invocata [6] un'altra callback chiamata `MaiscRecordComplete()`, implementata anch'essa all'interno della classe `CChipflipEngine::CGameMyTurn`, che, se non rileva particolari errori, non compie alcuna operazione.

Sempre nella `CChipflipEngine::CGameMyTurn::MakeMoveL()`, dopo aver richiesto l'interruzione del flusso di dati audio in ingresso, si crea il messaggio che dovrà essere inviato all'altro terminale per informarlo dell'intenzione di iniziare la fase di ricezione (composto dalla stringa di testo "Rx"), quindi si invia il messaggio stesso passandolo [7] alla funzione `CChipflipEngine::SendToRemoteL()`, e infine si varia lo stato della comunicazione in "commutazione in corso" creando [13] un oggetto della classe `CGameMakingOwnMove` e passandolo [14] alla funzione `CChipflipEngine::SetCurrentState()`, la quale lo sostituisce [15] a quello che corrispondeva allo stato precedente.

La funzione `CChipflipEngine::SendToRemoteL()` non fa che richiamare [8] la `CCommunicationChannel::SendToRemoteL()`. Questa è presente in due versioni, una delle quali accetta in ingresso dati di tipo `TDesC`, mentre l'altra opera su dati di tipo `TDesC8`. L'implementazione di tali funzioni prevede che la prima copi i dati ricevuti in un oggetto di tipo `TDesC8` e passi quest'ultimo alla seconda; quest'ultima invece, dopo aver verificato che lo stato della socket sia "connesso" (nel senso già precisato), passa i dati da spedire [9] alla `CSocketEngine::WriteL()`.

Anche di questa funzione sono presenti due versioni, che operano rispettivamente su dati di tipo `TDesC` e `TDesC8`. La loro implementazione, però, è praticamente identica: dopo aver verificato che non sia ancora in corso una precedente operazione di scrittura, si esamina il pacchetto ricevuto come argomento in modo da riconoscere se si trattava del messaggio "Tx" o del messaggio "Rx" e si memorizza questa informazione in un'apposita variabile booleana. Così facendo, sarà possibile ripetere più volte l'invio

dello stesso messaggio in caso di problemi di trasmissione, come sarà spiegato più avanti. Fatto ciò, lo stato della socket viene modificato in "attesa dell'acknowledgement" e, richiamando [10] la `CSocketTimer::ActivateTimer()`, viene fatto partire un timer della durata di 1 secondo scaduto il quale, se non è stato ancora ricevuto l'acknowledgement, si ripeterà l'invio. Quindi viene tentato l'invio del pacchetto all'indirizzo del destinatario mediante la funzione [11] `CSocketWriter::WriteD()`. Infine si memorizza in un'apposita variabile l'informazione relativa al fatto che il tentativo di invio dei dati appena compiuto è il primo: in caso di errori, infatti, si è previsto di ritrasmettere lo stesso messaggio solamente per un certo numero di volte, trascorse le quali si suppone che l'interlocutore sia divenuto irraggiungibile e si effettua la disconnessione.

La funzione `CSocketWriter::WriteD()` prima cancella l'oggetto contenente i dati che le sono stati passati all'iterazione precedente, se questi sono ancora presenti, poi inizia un'operazione di scrittura asincrona sulla socket richiamando dapprima [12] la funzione di libreria `RSocket::SendTo()` e quindi la `SetActive()`. Al termine dell'operazione, il sistema operativo richiama la `CSocketWriter::RunL()`, nella quale si cancella l'oggetto contenente i dati appena spediti e, se la scrittura si è conclusa senza errori, si richiama la `CSocketEngine::WriteDone()`, la quale non effettua alcuna operazione.

Se il timer impostato all'interno della funzione `CSocketEngine::WriteL()` scade prima che venga ricevuto dall'altro terminale il messaggio di acknowledgement, il sistema operativo richiama automaticamente la `CSocketTimer::RunL()`, la quale chiama la `CSocketEngine::TimerExpired()`. Questa, se il numero di tentativi di invio già effettuati è pari al massimo numero di tentativi a disposizione (impostato pari a 10), modifica lo stato della socket in `ESETimedOut` e richiama prima la `Cancel()`, quindi la `SetActive()` e la `User::RequestComplete()`. La chiamata della `Cancel()` annulla eventuali richieste asincrone pendenti e fa sì che il sistema operativo richiami la `CSocketEngine::DoCancel()`, nella quale si interrompe il timer, si annullano eventuali operazioni di lettura e scrittura e si imposta lo stato della socket a "non connesso". Invece la chiamata, in successione, della `SetActive()` e della `User::RequestComplete()` impone al sistema operativo di richiamare immediatamente la `CSocketEngine::RunL()`, nella quale, verificato il particolare stato della socket, si richiama la funzione `CCommunicationChannel::HandleSocketEngineErrorL()` passandole come parametro `ESETimedOut`. Questa passa il valore ricevuto alla `CChipflipEngine::HandleSocketErrorL()`, la quale a sua volta lo passa alle funzioni aventi lo stesso nome e definite all'interno della classe che rappresenta lo stato attuale della comunicazione e all'interno della classe `CChipflipAppUi`.

Quest'ultima mostra all'utente un messaggio in cui lo informa che la socket è stata chiusa a seguito dello scadere di un timer. Per quanto riguarda la prima delle funzioni

chiamate dalla `CChipflipEngine::HandleSocketErrorL()`, invece, non esistendo una funzione `HandleSocketErrorL()` all'interno della classe `CGameMyTurn`, la prima funzione chiamata è in realtà la `CChipflipEngine::CGameBaseState::HandleSocketErrorL()`, definita come *virtual*. Questa chiama prima la `CChipflipEngine::Reset()`, che imposta al valore `EFalse` entrambe le variabili booleane che indicano se l'utente o l'interlocutore stanno trasmettendo dati, quindi chiama la `CCommunicationChannel::ByeL()`, e infine riporta lo stato del sistema dal punto di vista della comunicazione a "registrato", creando un oggetto della classe `CGameRegistered` e passandolo alla `CChipflipEngine::SetCurrentState()`.

A partire da questo momento, la sequenza delle operazioni svolte è la stessa che viene seguita quando l'operazione di disconnessione è iniziata volontariamente da uno dei due utenti. Tale situazione verrà descritta in dettaglio in uno dei successivi paragrafi, al quale si rimanda anche per l'esame del caso in cui la disconnessione è dovuta a problemi di comunicazione[21].

Se, come avviene normalmente, tali problemi non si verificano, il terminale dell'interlocutore riceve il messaggio, aggiorna il proprio display in modo da mostrare lo stato attuale dell'utente e invia regolarmente l'acknowledgement, come verrà descritto in seguito. In questo caso sul terminale da cui è partita la richiesta di variazione dello stato viene eseguita la sequenza di funzioni mostrata in figura 38 e descritta nel seguito.

Il sistema operativo, al momento della ricezione del messaggio di acknowledgement, richiama automaticamente [1] la `CSocketReader::RunL()`; questa passa il contenuto del pacchetto ricevuto [2] alla `CSocketEngine::MessageReceivedL()` e inizia una nuova operazione di lettura asincrona richiamando, attraverso [13] la `CSocketReader::Read()`, la funzione di libreria [14] `RSocket::RecvFrom()` seguita dalla `SetActive()`.

La funzione `CSocketEngine::MessageReceivedL()` esamina prima la lunghezza e poi il contenuto del messaggio ricevuto e, se riconosce che si tratta di un

[21] La mancata ricezione di un acknowledgement può essere dovuta a diversi fattori: il terminale che dovrebbe inviare il messaggio "Tx" può essere ancora impegnato in una precedente operazione di scrittura; il pacchetto (che usa UDP come protocollo di trasporto) può andare perso all'interno della rete; il terminale del destinatario può essere momentaneamente irraggiungibile; il terminale stesso può non riuscire a spedire l'acknowledgement perché impegnato in un'altra operazione di scrittura; il messaggio di acknowledgement si può perdere all'interno della rete.

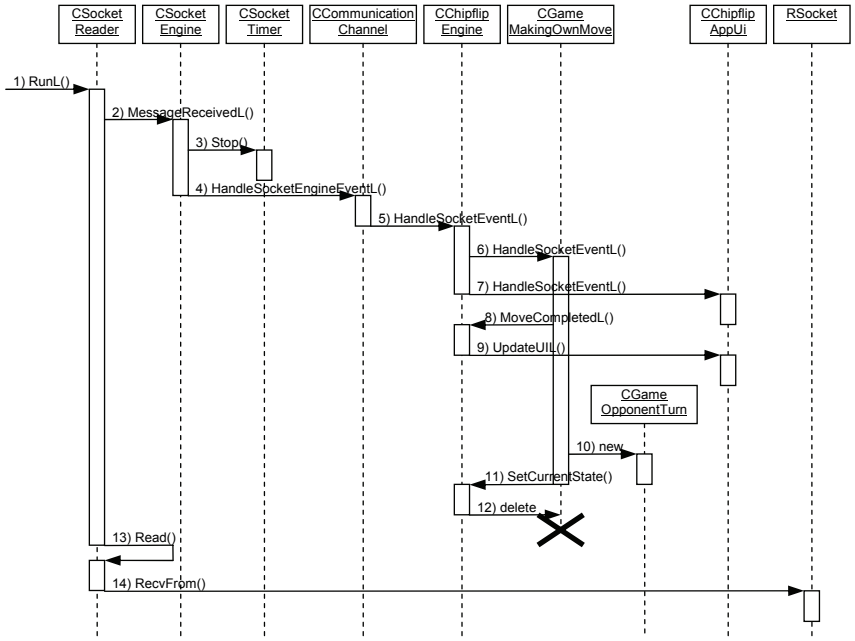

Figura 38. Ricezione dell'acknowledgement e avvio dell'operazione richiesta

acknowledgement e se il sistema era proprio in attesa di tale messaggio[22], interrompe il timer che era stato puntato al momento della trasmissione del "Tx" chiamando [3] la `CSocketTimer::Stop()`, poi riporta lo stato del sistema dal punto di vista della socket a `ESEConnected`, quindi passa il parametro `ESEMessageSent` alla funzione [4] `CCommunicationChannel::HandleSocketEngineEventL()`. Questa inoltra il medesimo parametro alla [5] `CChipflipEngine::HandleSocketEventL()`, che a sua volta lo passa alle funzioni omonime definite nella classe corrispondente allo stato attuale della socket [6] e nella classe `CChipflipAppUi` [7].

La funzione `CChipflipEngine::CGameMakingOwnMove::HandleSocketEventL()`, se riceve come parametro `ESEMessageSent`, richiama [8] la `CChipflipEngine::MoveCompletedL()`, che mediante [9] la `CChipflipAppUi::UpdateUIL()` aggiorna il display del terminale in modo da visualizzare il nuovo stato, e in seguito modifica lo stato della comunicazione in "ricezione" creando [10] un oggetto della classe `CGameOpponentTurn` e passandolo [11] alla funzione `CChipflipEngine::SetCurrentState()`, la quale lo sostituisce [12] a quello che rappresentava lo stato precedente.

[22] Esiste anche la possibilità che un acknowledgement arrivi a destinazione con un ritardo talmente elevato che nel frattempo è stato spedito il successivo messaggio "Tx": al momento dell'arrivo dell'acknowledgement relativo a quest'ultimo, il sistema non ne ha più bisogno e quindi lo scarta.

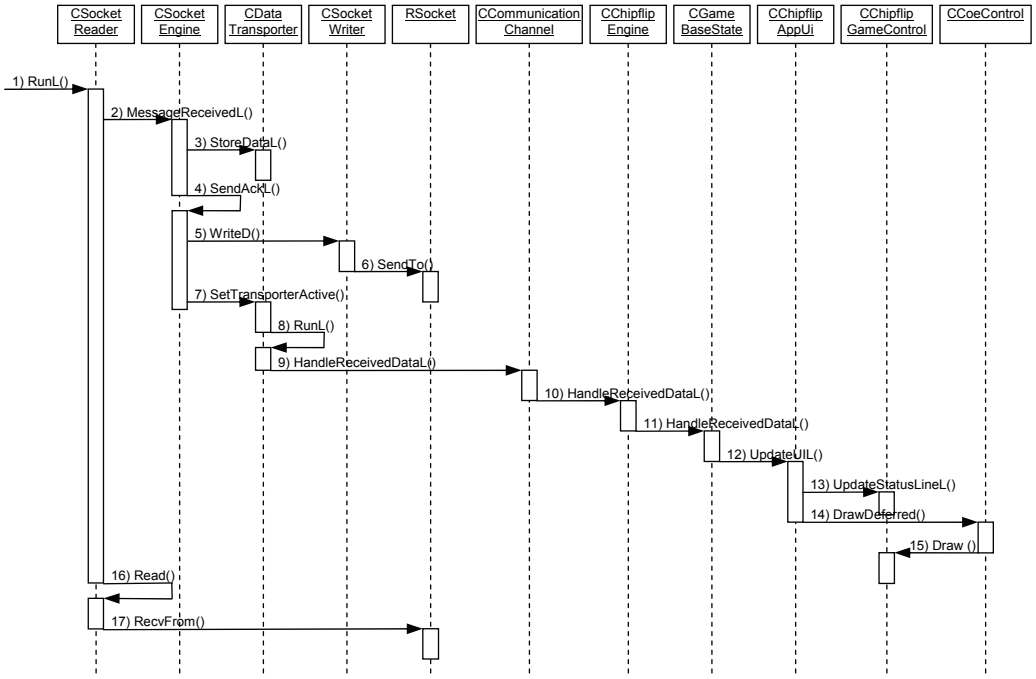

Figura 39. Ricezione del messaggio e invio dell'acknowledgement

A questo punto, come descritto nel paragrafo relativo, viene aperto lo stream diretto verso l'altoparlante del dispositivo e si può iniziare la riproduzione del flusso di dati audio provenienti dall'interlocutore.

Finora è stato preso in esame unicamente il caso in cui l'utente fosse inizialmente in fase di trasmissione e volesse iniziare una fase di ricezione; evidentemente, si può avere anche la situazione opposta. La modalità con cui questa viene gestita è tuttavia assolutamente analoga a quella esaminata, fatta eccezione per il contenuto del messaggio ("Tx" anziché "Rx"), per le classi in gioco (si utilizzerà, ad esempio, `CGameOpponentTurn` al posto di `CGameMyTurn`, ma le funzioni omonime definite all'interno delle due classi avranno la medesima funzione) e per le funzioni relative alla gestione dell'audio (si invocherà la `CMdaAudioOutputStream::Stop()`, e il sistema operativo, una volta chiuso lo stream, richiamerà la `MaoscPlayComplete()`).

Rimane da studiare solamente ciò che avviene quando un terminale riceve un messaggio contenente "Tx" o "Rx" e deve spedire il relativo acknowledgement. La sequenza di operazioni che il sistema svolge in questo caso, mostrata in figura 39, è descritta nel seguito.

Come avviene ogni volta che si riceve un qualunque pacchetto, all'arrivo del messaggio il sistema operativo richiama automaticamente [1] la funzione `CSocketReader::RunL()`, la quale passa il contenuto del pacchetto ricevuto, quale che sia, alla [2] `CSocketEngine::MessageReceivedL()` e inizia una nuova operazione di lettura asincrona richiamando, attraverso [16] la `CSocketreader::`

`Read()`, la funzione di libreria [17] `RSocket::RecvFrom()` seguita dalla `SetActive()`.

La funzione `CSocketEngine::MessageReceivedL()`, se riconosce che il messaggio appena ricevuto è un "Tx" o un "Rx", per prima cosa lo accoda a un apposito array dinamico passando il relativo puntatore [3] alla `CDataTransporter::StoreDataL()`, la quale non fa che richiamare il membro `AppendL()` della classe `CArrayPtr` a cui appartiene l'array di puntatori a buffer.

In seguito la `MessageReceivedL()` invia l'acknowledgement[23] mediante [4] la `CSocketEngine::SendAckL()`, la quale crea il messaggio di acknowledgement (costituito dalla stringa "Ack") e lo spedisce passando il messaggio stesso e l'indirizzo del destinatario [5] alla `CSocketWriter::WriteD()`, che si avvale della funzione di libreria [6] `RSocket::SendTo()` seguita dalla `SetActive()`.

Conclusa la richiesta asincrona di scrittura, sempre all'interno della `MessageReceivedL()` viene richiamata [7] la funzione `CDataTransporter::SetTransporterActive()`. Questa, dopo aver appurato che l'array contiene almeno un puntatore a buffer, richiama la `SetActive()` seguita immediatamente dalla `User::RequestComplete()`. In tal modo il sistema operativo richiama [8[24]] la `CDataTransporter::RunL()`, la quale elabora il contenuto del pacchetto puntato dal primo elemento dell'array passandolo [9] alla `CCommunicationChannel::HandleReceivedDataL()` e richiama nuovamente la `CDataTransporter::SetTransporterActive()`. Operando in questo modo, come già visto per quanto riguardava la ricezione e la riproduzione dei pacchetti di dati audio, diviene possibile scindere l'operazione di ricezione dei dati dalla loro elaborazione: ogni volta che si riceve un pacchetto lo si accoda all'array, dal quale poi il pacchetto stesso viene prelevato solo nel momento in cui è terminata l'elaborazione di quelli ricevuti in precedenza.

L'elaborazione dei dati viene effettuata dalla funzione `CCommunicationChannel::HandleReceivedDataL()`, la quale crea un oggetto contenente i dati in questione e passa un puntatore a tale oggetto [10] alla `CChipflipEngine::HandleReceivedDataL()`. Questa inoltra il puntatore stesso alla funzione omonima contenuta all'interno della classe corrispondente allo stato attuale del sistema [11]. Nel caso in questione, tuttavia, al momento della ricezione del messaggio il sistema può trovarsi in uno stato qualsiasi, e in ogni caso deve compiere le stesse operazioni, ovvero inviare l'acknowledgement all'altro terminale e aggiornare il

[23] Non occorre infatti verificare il tipo di messaggio ricevuto né considerare lo stato attuale del sistema, ma solamente informare l'altro terminale dell'avvenuta ricezione.

[24] A rigore la rappresentazione in figura non è esatta, ma serve per mostrare come sia in realtà l'applicativo a forzare l'invocazione della `RunL()` in un preciso istante, in maniera sincrona.

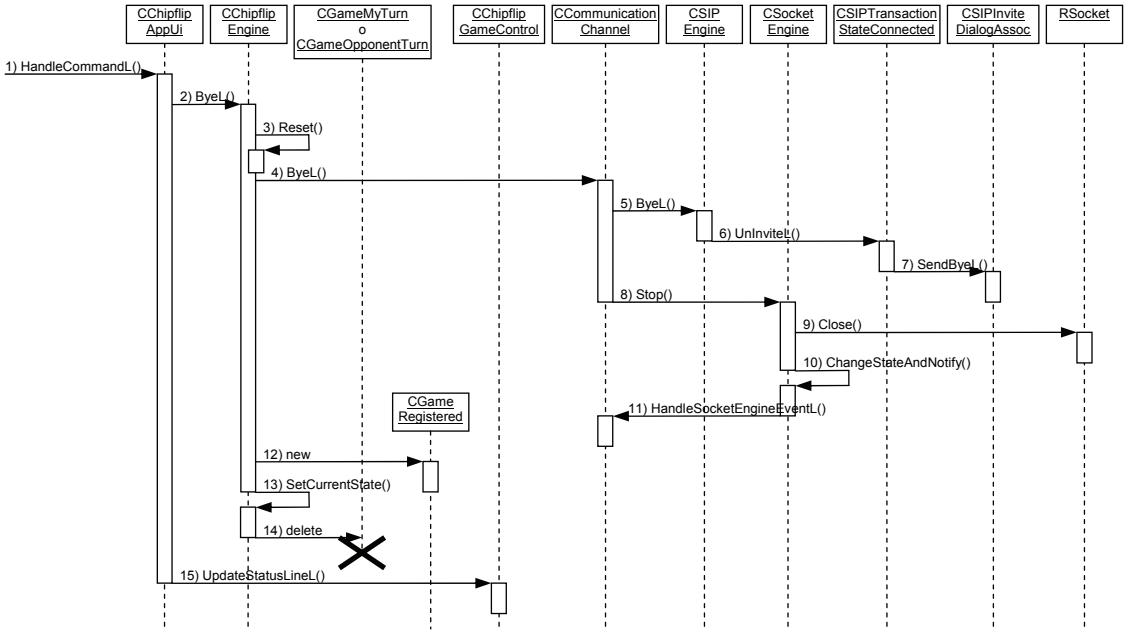

Figura 40. Invio del Bye

display in modo da indicare all'utente lo stato attuale del proprio interlocutore. Per questo motivo, sia nella funzione *virtual* `CChipflipEngine::CGameBaseState::HandleReceivedDataL()` sia nelle funzioni omonime ridefinite nelle classi `CGameMyTurn` e `CGameOpponentTurn` (che ereditano da `CGameBaseState`), se si riceve un "Tx" o un "Rx" vengono effettuate le stesse operazioni[25]. Queste consistono nel memorizzare in un'opportuna variabile booleana l'informazione sull'operazione (invio o ricezione) che l'interlocutore intende iniziare e nell'aggiornare l'interfaccia grafica richiamando, mediante la funzione [12] `CChipflipAppUi::UpdateUIL()`, la `ChipflipGameControl::UpdateStatusLineL()` [13] e la `CCoeControl::DrawDeferred()` [14], che a sua volta richiama [15] la `ChipflipGameControl::Draw()` ma con priorità inferiore rispetto alle operazioni di input.

➢ Termine della chiamata

Durante la chiamata ognuno degli utenti, indipendentemente dall'operazione in corso sul proprio terminale e su quello dell'interlocutore, può decidere in ogni momento di terminare la comunicazione selezionando la voce *Termina* dal menu *Opzioni*. A seguito di tale selezione, viene eseguita la sequenza di operazioni rappresentata in figura 40 e descritta nel seguito.

[25] In figura, perciò, è stato rappresentato unicamente un oggetto della classe `CGameBaseState`.

Per prima cosa, al momento della selezione del comando *Termina*, il sistema operativo richiama [1] la funzione `CChipflipAppUi::HandleCommandL()` con parametro `EChipflipCmdEndGame`. Tale funzione, in questo caso, mostra all'utente una finestra di dialogo in cui lo informa che è in corso l'operazione di disconnessione, inizia l'operazione stessa nonché l'invio del messaggio SIP di BYE (chiamando [2] la `CChipflipEngine::ByeL()`), e aggiorna l'interfaccia utente in modo da tener conto del nuovo stato in cui si trova il sistema, mediante [15] la `ChipflipGameControl::UpdateStatusLineL()`.

La funzione `CChipflipEngine::ByeL()` riporta i parametri della comunicazione alle condizioni iniziali richiamando [3] la `CChipflipEngine::Reset()`, richiede l'invio del messaggio SIP di BYE mediante [4] la `CCommunicationChannel::ByeL()`, e infine modifica lo stato della comunicazione in "registrato" creando [12] un oggetto della classe `CGameRegistered` e passandolo [13] alla `CChipflipEngine::SetCurrentState()`, che lo sostituisce [14] a quello corrispondente allo stato precedente (che può appartenere alla classe `CGameMyTurn` o alla `CGameOpponentTurn`).

La `CCommunicationChannel::ByeL()`, a sua volta, richiama le funzioni `CSIPEngine::ByeL()` [5] e `CSocketEngine::Stop()` [8]. La funzione `CSIPEngine::ByeL()` richiama [6] la `CSIPTransactionStateSessionConnected::UnInviteL()`, che invia il messaggio SIP di BYE mediante la funzione [7] `CSIPInviteDialogAssoc::SendByeL()` definita nella API SIP. Invece la `CSocketEngine::Stop()` interrompe ogni eventuale operazione di lettura e/o scrittura, chiude la socket richiamando [9] la `RSocket::Close()` e modifica lo stato della socket in "non connesso" informandone l'utente, mediante il passaggio del parametro `ESENotConnected` alla `CSocketEngine::ChangeStateAndNotify()` [10].

Quest'ultima memorizza il valore corrispondente al nuovo stato e lo inoltra [11] alla `CCommunicationChannel::HandleSocketEngineEventL()`, la quale a sua volta lo passa alla `CChipflipEngine::HandleSocketEventL()`. Questa, come sempre, richiama le funzioni aventi lo stesso nome definite nella classe corrispondente allo stato attuale della comunicazione e nella classe `CChipflipAppUi`; nessuna di tali funzioni compie però, in questo caso, alcuna operazione[26].

[26] Per tale motivo, le funzioni in questione non sono mostrate in figura.

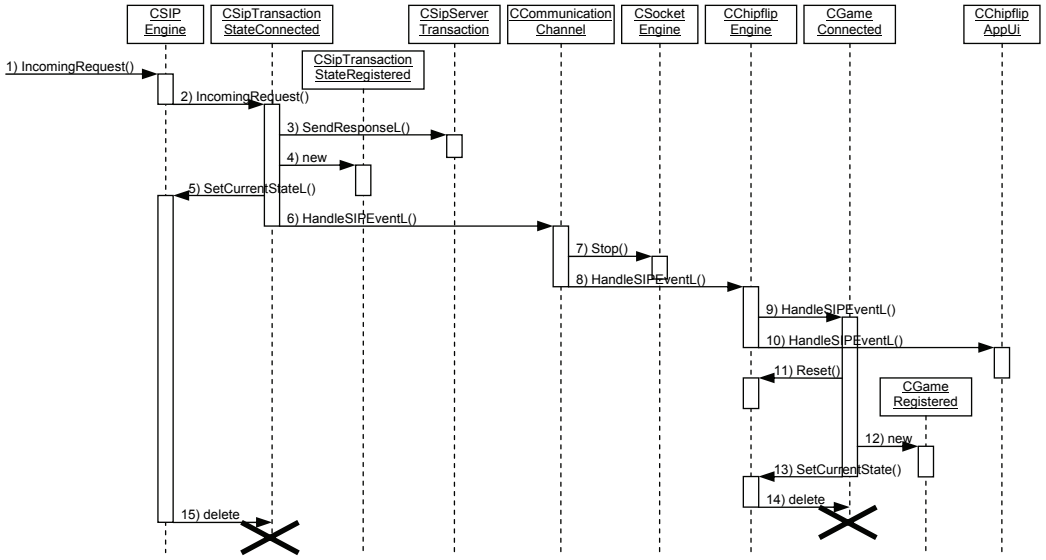

Figura 41. Ricezione del Bye e invio del 200 Ok

Sul terminale che riceve il messaggio SIP di BYE vengono invece eseguite le operazioni mostrate in figura 41 e illustrate nel seguito.

Il terminale che riceve il BYE viene informato dell'evento dal sistema operativo, tramite il plug-in SIP, mediante l'invocazione [1] della funzione `IncomingRequest()`, definita come *pure virtual* nella classe `MSIPConnectionObserver` (appartenente alla API per SIP) e implementata in `CSIPEngine`. Questa non fa che invocare la funzione omonima definita all'interno della classe corrispondente allo stato corrente (considerato dal punto di vista della segnalazione SIP). Se al momento della ricezione del BYE il terminale si trova nello stato "connesso", quindi, viene richiamata [2] la `CSIPTransactionStateSessionConnected::IncomingRequest()`.
All'interno di tale funzione, in questo caso, per prima cosa si genera il messaggio SIP di risposta (200 OK) e lo si invia mediante la funzione [3] `CSIPServerTransaction::SendResponseL()`, che fa parte della API SIP. In seguito si riporta lo stato del sistema (visto nell'ottica della segnalazione SIP) a "registrato", creando [4] un oggetto della classe `CSIPTransactionStateRegistered` e passandolo [5] alla funzione `CSIPEngine::SetCurrentStateL()`, che lo sostituisce [15] a quello che corrispondeva allo stato precedente; infine si passa il parametro `ESByeReceived` alla funzione [6] `CCommunicationChannel::HandleSIPEventL()`. Questa richiama prima [7] la `CSocketEngine::Stop()` e poi inoltra il parametro ricevuto [8] alla `CChipflipEngine::HandleSIPEventL()`.

La prima di queste funzioni blocca eventuali timer attivi, annulla ogni richiesta asincrona, interrompe ogni operazione pendente di lettura o scrittura sulla socket, chiude la socket e modifica lo stato della socket in "disconnesso" mediante la

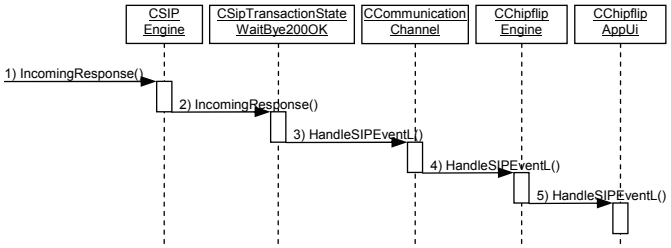

Figura 42. Ricezione del 200 Ok relativo al Bye

`CSocketEngine::ChangeStateAndNotify()`, nello stesso modo già visto esaminando il caso in cui l'operazione di disconnessione era iniziata dal terminale dell'utente in esame anziché da quello dell'interlocutore[27]. La funzione `CChipflipEngine::HandleSIPEventL()`, invece, inoltra il parametro `ESByeReceived` alle funzioni omonime appartenenti alla classe che corrisponde allo stato attuale della comunicazione[28] [9] e alla classe `CChipflipAppUi` [10].

La funzione `CChipflipEngine::CGameConnected::HandleSIPEventL()`, se riceve in ingresso il parametro `ESByeReceived`, riporta i vari parametri allo stato iniziale richiamando [11] la `CChipflipEngine::Reset()` e ripristina lo stato della comunicazione a "registrato" creando [12] un oggetto della classe `CGameRegistered` e passandolo [13] alla `CChipflipEngine::SetCurrentState()`, che lo sostituisce [14] a quello che rappresentava lo stato precedente. La funzione `CChipflipAppUi::HandleSIPEventL()`, invece, nella stessa situazione mostra all'utente una finestra di dialogo in cui lo informa dell'avvenuta disconnessione dall'interlocutore.

A questo punto, salvo errori, il terminale da cui è partita la richiesta SIP di BYE riceve come risposta il 200 OK. Le operazioni eseguite in questo caso, descritte nel seguito, sono mostrate in figura 42.

Come sempre, il programma viene informato dell'avvenuta ricezione da parte del sistema operativo, tramite il plug-in SIP, mediante la chiamata [1] della funzione `IncomingResponse()`, definita come *pure virtual* nella classe `MSIPConnectionObserver` (appartenente alla API per SIP) e implementata in `CSIPEngine`. Questa non fa che invocare la funzione omonima definita all'interno della classe corrispondente allo stato corrente (considerato dal punto di vista della segnalazione SIP). Siccome attualmente il sistema è in attesa del 200 OK relativo al

[27] A causa di tale somiglianza, le funzioni in questione non sono state riportate in figura.

[28] Né la classe `CChipflipEngine::CGameMyTurn` né la `CChipflipEngine::CGameOpponentTurn` implementano la funzione in questione: siccome entrambe queste classi ereditano da `CgameConnected`, viene perciò richiamata la funzione *virtual* `CChipflipEngine::CGameConnected::HandleSIPEventL()`.

BYE[29], viene richiamata [2] la `CSIPTransactionStateWaitBye200OK::IncomingResponse()`, nella quale si distruggono l'array che conteneva i messaggi SIP scambiati durante la sessione, l'oggetto che conteneva l'identificativo della sessione e quello che indicava la tipologia di dialogo in corso, e in seguito si passa il parametro `ESDisconnected` alla funzione [3] `CCommunicationChannel::HandleSIPEventL()`.

Questa, se viene invocata con tale parametro, inoltra [4] il parametro ricevuto alla `CChipflipEngine::HandleSIPEventL()`, che lo passa alle funzioni aventi lo stesso nome contenute nella classe corrispondente allo stato attuale della comunicazione e alla classe `CChipflipAppUi`.

Nessuna delle funzioni `HandleSIPEventL()` contenute in classi che ereditano da `CChipflipEngine::CGameBaseState` compie alcuna operazione quando viene invocata con parametro `ESDisconnected`.[30] Invece, nello stesso caso, la funzione `CChipflipAppUi::HandleSIPEventL()` [5] sostituisce la finestra di dialogo precedentemente aperta con una che informa l'utente dell'avvenuta disconnessione e aggiorna il contenuto dell'interfaccia utente.

➢ Deregistrazione del profilo SIP in uso

La sequenza delle funzioni che vengono richiamate per effettuare l'operazione di deregistrazione del profilo SIP attualmente in uso, ovvero per revocare la registrazione dell'utente sul server SIP precedentemente scelto, è mostrata in figura 43.

Per iniziare l'operazione, l'utente deve selezionare il comando *Deregistra* presente nel menu *Opzioni*. A seguito di tale selezione, il sistema operativo richiama automaticamente la funzione `CChipflipAppUi::HandleCommandL()` [1] con parametro `EChipflipCmdUnRegister`; questa, in tal caso, non fa che richiamare [2] la `CChipflipEngine::DisableProfileL()`, la quale richiama la funzione omonima definita nella classe corrispondente allo stato attuale della comunicazione.

Viene perciò chiamata [3] la funzione `CChipflipEngine::CGameRegistered::DisableProfileL()`, la quale richiama unicamente [4] la `CCommunicationChannel::DisableProfileL()`, che a sua volta chiama [5] la `CSIPEngine::DisableProfileL()`. Questa, se lo stato della segnalazione SIP è "registrato", richiama [6] la funzione `CSIPTransactionStateRegistered::UnRegisterL()`, all'interno della quale viene richiamata [7] la `CSIPProfileRegistry::Disable()`, ovvero la

[29] Si noti come lo stato in questione è differente da quello di attesa di un 200 OK relativo ad altri tipi di richieste, poiché in questo caso il protocollo SIP prevede che la risposta ricevuta non venga confermata da un ACK.

[30] Per questa ragione le funzioni in questione non sono state rappresentate in figura.

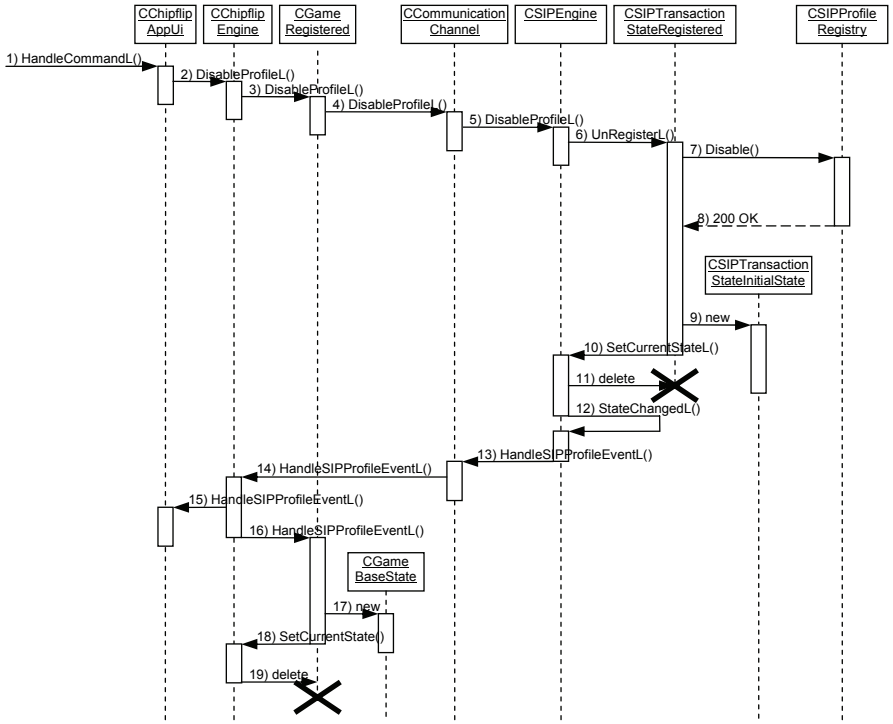

Figura 43. Deregistrazione del profilo SIP in uso

funzione inclusa nella API SIP che disabilita effettivamente il profilo attualmente in uso. In seguito la `DisableProfileL()` riporta lo stato della segnalazione SIP a quello iniziale creando [9] un oggetto della classe `CSIPTransactionStateInitialState` e passandolo [10] alla funzione `CSIPEngine::SetCurrentStateL()`, la quale lo sostituisce [11] a quello precedente, che apparteneva alla classe `CSIPTransactionStateRegistered`.

La funzione `CSIPProfileRegistry::Disable()` invia al server SIP un messaggio di REGISTER il cui campo *expires*, indicante il numero di secondi per cui il profilo che si sta registrando dovrà essere valido, è posto pari a 0. Come nel caso dell'abilitazione del profilo SIP, se l'operazione di deregistrazione si conclude senza problemi il server SIP risponde [8] con un messaggio di 200 OK, la cui elaborazione non deve essere effettuata esplicitamente all'interno del programma ma è gestita automaticamente dalla stessa funzione `CSIPProfileRegistry::Disable()`, la cui esecuzione termina solo al momento della ricezione di tale messaggio.

Invece la funzione `CSIPEngine::SetCurrentStateL()` non solo memorizza nella variabile `iCurrentState` un puntatore allo stato che le viene passato, ma chiama anche [12] la `CSIPEngine::StateChangedL()`, che, nel caso in esame, chiama [13] la `CCommunicationChannel::HandleSIPProfileEventL()` passandole come parametro `ESPOffline`. Questa inoltra il medesimo parametro [14] alla `CChipflipEngine::HandleSIPProfileEventL()`, che a sua volta lo

passa alle due funzioni omonime definite nella classe `CChipflipAppUi` [15] e nella classe corrispondente allo stato attuale della comunicazione [16].

La funzione `CChipflipAppUi::HandleSIPProfileEventL()`, quando le viene passato il parametro `ESPOffline`, informa l'utente dell'avvenuta disconnessione mediante un'opportuna finestra di dialogo; la `CChipflipEngine::CGameRegistered::HandleSIPEventL()` invece riporta la comunicazione allo stato iniziale creando [17] un oggetto della classe `CGameBaseState` e passando quest'ultimo [18] alla `CChipflipEngine::SetCurrentState()`, che lo sostituisce [19] a quello della classe `CGameRegistered`.

4. Test

L'applicativo sviluppato è stato testato in varie configurazioni, eseguendolo sia sul dispositivo mobile sia sul relativo emulatore, in modo da verificarne le prestazioni quando viene utilizzato in situazioni differenti.

4.1. Comunicazione tra due emulatori

La prima e più semplice delle configurazioni esaminate (schematizzata in figura 1) è quella in cui vengono eseguiti sullo stesso computer sia il server SIP Nokia sia due emulatori di terminali, su ognuno dei quali gira il programma *UniPR-Ptt*. In questo caso i due emulatori riescono a registrarsi senza problemi sul server, ma la fase di instaurazione della comunicazione non termina correttamente: il terminale che riceve il messaggio SIP di INVITE, infatti, manda come risposta 482 LOOP DETECTED e interrompe l'handshake. Di conseguenza, in questo caso non è neppure possibile iniziare la chiamata e verificare la qualità del segnale audio trasmesso.

Un'altra configurazione analizzata è quella in cui i due emulatori vengono eseguiti su due computer differenti, mentre il server SIP Nokia viene lanciato su uno dei due computer stessi. All'interno di questo caso occorre però distinguere due sottocasi: nel primo i due computer hanno entrambi un indirizzo IP pubblico o quanto meno sono collegati alla stessa rete privata, mentre nel secondo caso uno di essi ha un indirizzo privato, e l'altro ne ha uno pubblico oppure appartiene ad una rete privata differente dalla precedente.

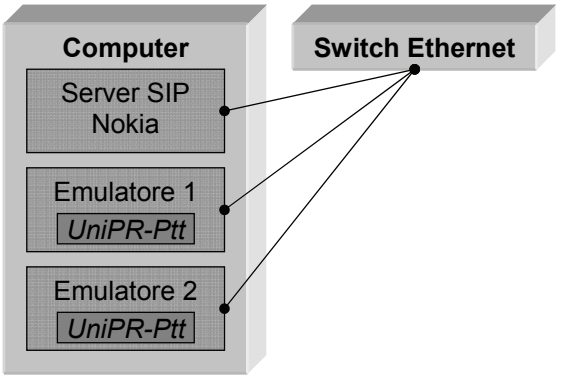

Figura 1. Due emulatori e il server SIP Nokia sullo stesso computer

Nel caso in cui i due emulatori vengano eseguiti su due computer attestati

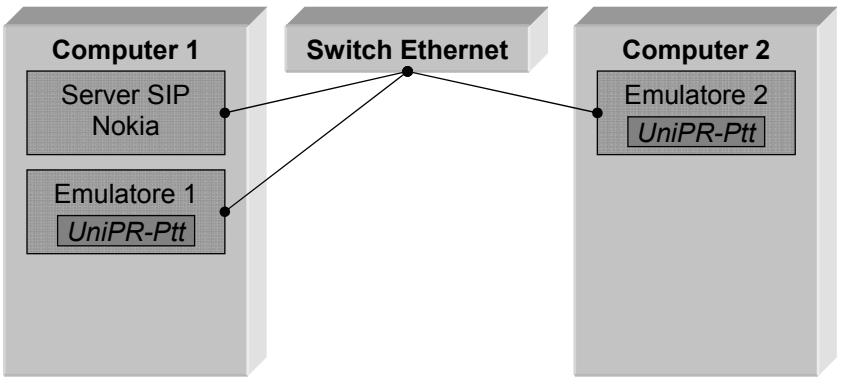

Figura 2. Un emulatore e il server SIP Nokia su un computer, un altro emulatore su un altro, entrambi in rete privata

sulla rete pubblica o sulla stessa rete privata (figura 2), essi possono scambiare dati l'uno con l'altro utilizzando semplicemente gli indirizzi IP (pubblici o privati che siano) assegnati ai terminali. Ognuno degli applicativi *UniPR-Ptt* può conoscere, tramite apposite funzioni, l'indirizzo IP del terminale su cui è in esecuzione e pubblicizzarlo all'interno degli opportuni messaggi SIP. In questo caso, in altre parole, un terminale può indicare all'interno del messaggio SIP di INVITE o di 200 OK il proprio indirizzo IP, e l'altro terminale, che riceve questo messaggio, può mandare i dati direttamente all'indirizzo indicato.

Utilizzando il programma *UniPR-Ptt* in questa configurazione, l'instaurazione della sessione termina senza alcun problema, e si può procedere normalmente con la comunicazione. Ora, tuttavia, la qualità dell'audio risente pesantemente del fatto che si utilizzano degli emulatori anziché dei terminali veri e propri: parlando nel microfono collegato alla scheda audio di uno dei computer e ascoltando il segnale proveniente dagli altoparlanti collegati alla scheda audio dell'altro, si ascolta un suono chiaramente correlato con quello che si sta trasmettendo, ma non intelligibile. In ogni caso, la commutazione tra le fasi di trasmissione e di ricezione non crea alcun problema, e anche il termine della chiamata e la disconnessione avvengono correttamente. Esaminando con un analizzatore di rete i pacchetti RTP che transitano, si può notare come la percentuale di pacchetti persi sia in questo caso molto alta, pari al 23% circa del totale[1]: dall'analisi dei sequence number presenti nell'header RTP dei pacchetti si vede infatti che regolarmente, ogni 3 o 4 pacchetti che si tenta di spedire, uno non viene trasmesso[2]. Da tale osservazione si deduce che le operazioni di campionamento, codifica e trasmissione dei pacchetti sono eccessivamente impegnative per le capacità di elaborazione dell'emulatore. D'altronde, verosimilmente, anche le operazioni opposte di ricezione,

[1] Tale valore è stato riscontrato, con minime variazioni, in tutte le prove effettuate.

[2] L'ipotesi che un pacchetto emesso possa venire perso all'interno della rete appare alquanto improbabile, essendo tutti i terminali collegati ad una stessa rete di capacità molto superiore a quella strettamente necessaria.

decodifica e riproduzione dei dati sembrano troppo gravose per essere eseguite sull'emulatore, e ciò comporta un ulteriore calo della qualità del segnale audio riprodotto. Anche il ritardo misurato tra l'istante in cui il segnale viene registrato e quello in cui viene riprodotto è abbastanza alto, pari a poco più di 1 secondo.

Questi problemi, d'altra parte, non sembrano dovuti a limiti nelle risorse hardware delle macchine su cui venivano eseguiti gli emulatori: uno di questi, infatti, è stato fatto girare su un Athlon XP-M 2800+ con 224 Mb di RAM, mentre l'altro su un Pentium 4 a 3 GHz con 2 Gb di RAM, collegato al primo tramite una rete Ethernet a 100 Mb/s. I risultati dei test sono stati praticamente identici sia che l'emulatore presente sul primo terminale inviasse i dati e l'altro li ricevesse, sia nel caso opposto. D'altronde, la quantità di risorse di sistema utilizzate sulle due macchine durante l'esecuzione del programma non è mai stata tale da poter causare rallentamenti. Si è quindi ipotizzato che i problemi riscontrati fossero dovuti unicamente ai limiti derivanti dall'uso di due emulatori anziché di due smartphone reali.

Nel caso più generale in cui i due emulatori sono eseguiti su due computer collegati a reti private diverse, oppure in cui uno di questi dispone di un indirizzo pubblico mentre l'altro ne ha uno privato, occorre tener conto della presenza dei NAT e agire di conseguenza. In questo caso, infatti, almeno uno dei terminali ha un indirizzo IP privato, e può indicare nel messaggio SIP solamente questo; l'altro terminale non può quindi utilizzare tale indirizzo per inviare i messaggi. Per risolvere il problema è necessario utilizzare un server SIP più avanzato di quello fornito da Nokia, quale ad esempio MjSip. Questo, a differenza del precedente, ha la capacità di intervenire sui messaggi di segnalazione che transitano attraverso di esso, modificandone il contenuto.

Nel caso in questione, almeno uno degli emulatori accede alla rete pubblica attraverso un NAT, che associa all'indirizzo privato assegnato all'emulatore un indirizzo pubblico. Il messaggio SIP di REGISTER proveniente da tale emulatore conterrà quindi nel campo *Via* l'indirizzo privato di quest'ultimo, mentre l'indirizzo di sorgente del pacchetto ricevuto dal server MjSip sarà quello assegnatogli dal NAT. Osservando tale discordanza, il server SIP può rendersi conto che il terminale da cui proviene il messaggio si trova dietro un NAT e agire di conseguenza.

Innanzi tutto, per mantenere attiva l'associazione tra indirizzo privato e indirizzo pubblico anche durante i periodi in cui il terminale non trasmette dati, il server deve inviare a quest'ultimo regolari messaggi di keepalive. Inoltre è necessario sostituire l'indirizzo privato inserito dal terminale all'interno dei messaggi SIP diretti all'interlocutore con un indirizzo pubblico, in modo che i dati inviati da quest'ultimo possano giungere correttamente a destinazione.

Inserendo all'interno dei messaggi SIP l'indirizzo pubblico corrispondente, tramite NAT, a quello privato assegnato all'emulatore, sarebbe possibile far transitare i pacchetti che trasportano dati audio direttamente da un terminale all'altro. Invece si è preferito inserire nei messaggi SIP, al posto del vero indirizzo IP (privato) dell'emulatore, l'indirizzo (pubblico) del server. In particolare, tale modifica deve

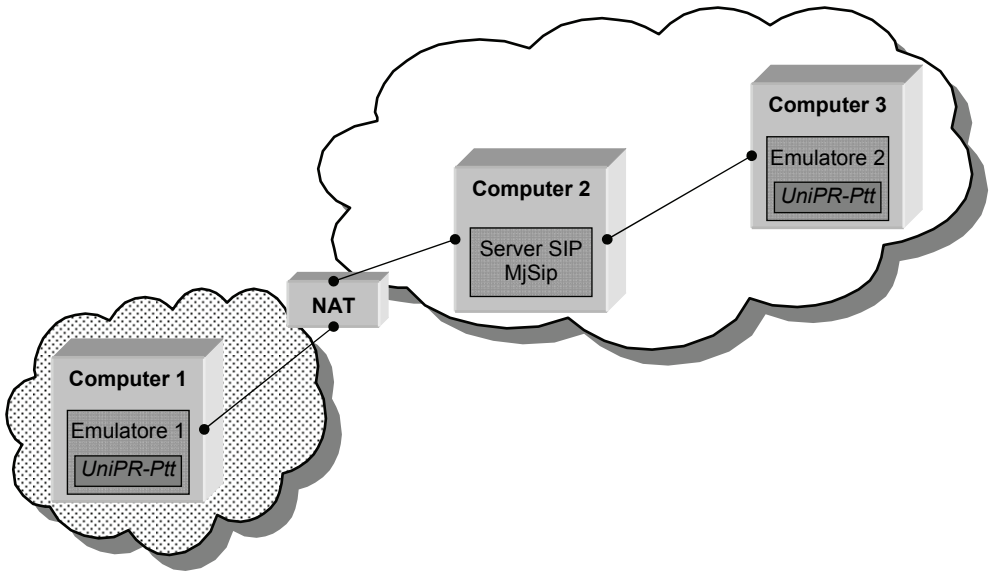

Figura 3. Un emulatore in rete privata, l'altro e il server MjSip in rete pubblica

essere effettuata nei campi *Connection information* e *Media description* dei messaggi SDP contenuti all'interno dei messaggi SIP di INVITE e 200 OK. In tal modo, il terminale che riceve il messaggio crede che l'interlocutore abbia come indirizzo IP quello che in realtà è l'indirizzo del server SIP, dunque invia al server (che quindi funge anche da gateway) tutti i pacchetti diretti all'interlocutore. Il server MjSip dispone della possibilità di salvare su file il contenuto di tutti i pacchetti che transitano attraverso di esso: utilizzando questo anziché il server SIP Nokia è dunque possibile verificare la qualità dell'audio inviato da uno dei terminali prima che questo venga ricevuto e riprodotto dall'altro. Oltre a ciò, ovviamente, il server inoltra i pacchetti al legittimo destinatario, ovvero all'indirizzo IP pubblico che corrisponde, mediante NAT, a quello privato assegnato al terminale dell'interlocutore.

Per testare una configurazione di questo tipo, è stato lanciato il server MjSip su un computer che disponeva di un indirizzo IP pubblico e che era raggiungibile da qualunque altro nodo della rete Internet; i due emulatori invece sono stati eseguiti dapprima su due macchine diverse appartenenti alla stessa rete privata, mentre nella prova successiva (mostrata in figura 3) uno dei due è stato collegato alla rete pubblica. Il caso in cui i due terminali siano collegati a due reti private differenti non è stato testato, ma non dovrebbe differire dall'ultimo di quelli precedentemente menzionati.

In entrambi i casi esaminati, la chiamata viene instaurata senza alcun problema, ma la qualità del segnale audio ricevuto è, come nel caso precedente, decisamente scarsa. La percentuale di pacchetti persi, anche in questo caso, è praticamente costante in tutte le prove effettuate, e pari al 23% circa; il ritardo di trasmissione, simile a quello riscontrato nel caso precedente, è di poco più di un secondo.

Per di più, in questo scenario si verifica un problema che nel precedente non era emerso: il messaggio SIP di BYE, infatti, contrariamente agli altri messaggi di

segnalazione, viene inviato direttamente da un terminale all'altro senza passare attraverso il server. Questo risulta un problema nel caso in cui la richiesta di terminare la comunicazione sia diretta al terminale collegato alla rete privata: in tal caso infatti il messaggio, inviato a un indirizzo privato, non giungerà a destinazione. Il terminale che l'ha inviato, scaduto un opportuno timeout, considererà terminata la comunicazione, ma l'altro non lo potrà ricevere in alcun caso.

La soluzione di tale problema non può passare unicamente attraverso una modifica al progetto *UniPR-Ptt*: infatti l'indirizzo a cui inviare il messaggio di BYE non viene passato esplicitamente alla funzione della API SIP preposta allo svolgimento di tale operazione, ma viene estrapolato dai messaggi scambiati durante l'handshake. Il server MjSip modifica, come si è detto, il campo SDP *Connection information*, ma probabilmente la funzione che invia il Bye ricava l'indirizzo di destinazione da un campo che non viene corretto.

4.2. Comunicazione tra emulatore e terminale

Un altro degli scenari esaminati è stato quello in cui il programma *UniPR-Ptt* viene eseguito da una parte su un terminale reale e dall'altra su un emulatore. Il server SIP utilizzato, in questo caso, è stato *MjSip*, fatto girare su una macchina dotata di indirizzo IP pubblico. Il terminale mobile su cui sono state svolte le prove (mostrato in figura 4) è un Nokia 6630 collegato alla rete UMTS dell'operatore H3G. L'emulatore, analogamente al caso precedente, gira su una macchina collegata ad una rete privata. Anche al terminale mobile, d'altra parte, ad ogni richiesta di connessione viene assegnato dall'operatore un diverso indirizzo IP privato, che poi viene associato tramite NAT a un opportuno indirizzo pubblico. La configurazione in esame è mostrata in figura 5.

In questo caso, se il server SIP veniva messo in ascolto sulla porta 5060 (la porta standard per il servizio SIP), la registrazione dell'utente che utilizzava il terminale mobile non poteva essere effettuata; modificando le impostazioni del server SIP in modo che utilizzasse una porta diversa dalla 5060 (in particolare la 5066) il problema è scomparso. Tale risultato porta a supporre che l'operatore mobile utilizzato tenti di impedire (o quanto meno di intralciare) l'uso di sistemi basati su SIP. In ogni caso, effettuata tale variazione, la registrazione dei due utenti sul server SIP avviene senza problemi.

Figura 4. Lo smartphone Nokia 6630

Quando però l'utente che utilizza l'emulatore tenta di iniziare una chiamata invitando colui che utilizza il terminale vero e proprio, quest'ultimo risponde all'invito

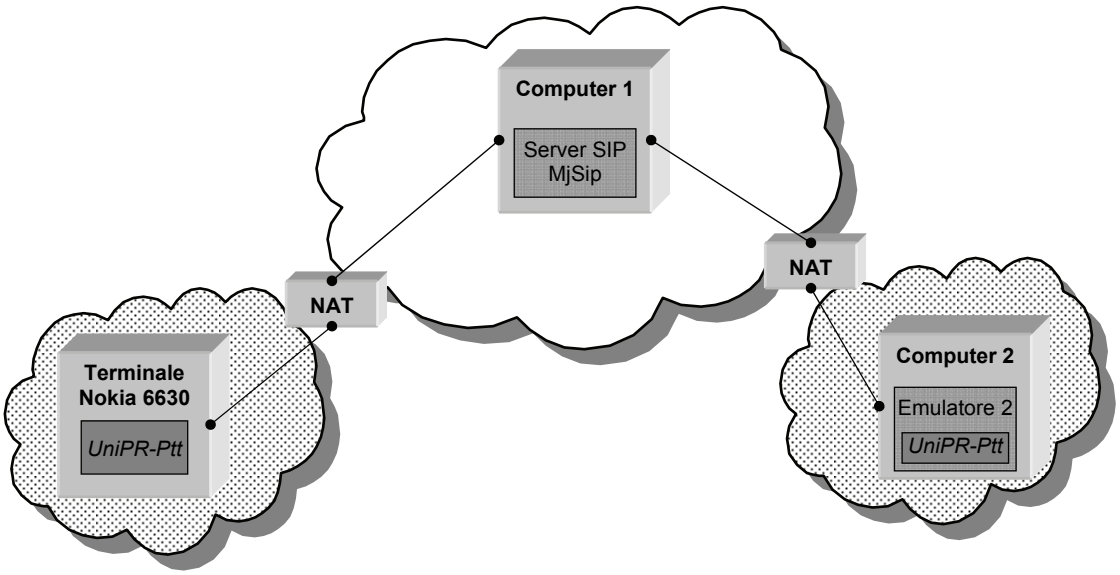

Figura 5. Un emulatore e un terminale su reti private, server MjSip sulla rete pubblica

con un messaggio SIP di tipo 486 UNSUPPORTED MEDIA TYPE, impedendo che la comunicazione possa essere instaurata. Stranamente, invece, se l'invito parte dal terminale ed è diretto verso l'emulatore, non si verifica alcun problema, e la chiamata inizia e procede regolarmente. La qualità del segnale audio, in questo caso, è superiore a quella che si poteva ottenere utilizzando un emulatore sia per campionare il segnale audio che per riprodurlo. In particolare, se l'audio è campionato dall'emulatore e riprodotto dal terminale mobile, la qualità percepita è leggermente migliore rispetto al caso precedente, mentre un netto miglioramento si verifica se il segnale viene campionato dal terminale e riprodotto dall'emulatore. Il ritardo end-to-end che si ottiene è invece abbastanza limitato (pari a circa 0,3 secondi) nella trasmissione da emulatore a terminale, mentre è molto maggiore (circa 2,2 secondi) nel verso opposto. Il motivo di tale squilibrio non è stato chiarito.

Esaminando con un analizzatore di rete il contenuto degli header RTP dei pacchetti scambiati in questa configurazione si può notare come, in effetti, la percentuale di pacchetti persi tra quelli inviati dall'emulatore al terminale sia, analogamente ai casi precedenti, del 23% circa, mentre per quanto riguarda lo stream diretto nel senso opposto, salvo problemi relativi alla copertura di rete, praticamente tutti i pacchetti che si tenta di spedire giungono correttamente a destinazione. Nonostante ciò, la qualità del segnale riprodotto è ancora ben lontana dalla perfezione.

L'asimmetria riscontrata nella qualità dell'audio ricevuto (decisamente migliore quando si utilizza il terminale per la trasmissione e l'emulatore per la ricezione rispetto al caso opposto) porta a supporre che la complessità computazionale delle operazioni di campionamento, codifica e trasmissione sia maggiore rispetto a quella delle operazioni di ricezione, decodifica e riproduzione. D'altra parte, all'aumento della qualità

dell'audio si accompagna un incremento del ritardo di trasmissione, le cui cause non sono ancora state chiarite.

Come si è detto, il server MjSip offre la possibilità di registrare su un file il contenuto di tutti i pacchetti che transitano attraverso di esso. Sfruttando tale possibilità, è dunque possibile verificare la qualità del segnale audio campionato e trasmesso senza che questa possa venire modificata dalle operazioni relative alla ricezione. Riproducendo i file così generati mediante un lettore che supporti il formato AMR, si può notare come la qualità del segnale proveniente dall'emulatore e diretto al terminale mobile sia effettivamente scarsa, mentre quella del segnale che viaggia nel verso opposto si può considerare decisamente buona. La degradazione della qualità che si verifica riproducendo in tempo reale il segnale ricevuto dall'emulatore dovrebbe quindi essere dovuta unicamente alle limitate capacità di elaborazione dell'emulatore stesso.

4.3. Comunicazione tra due terminali

Questo scenario non è stato testato a causa della mancata disponibilità del secondo dispositivo mobile. I risultati delle prove descritte in precedenza portano tuttavia a ritenere che in questo caso, una volta risolti eventuali problemi relativi alle fasi di instaurazione ed abbattimento della comunicazione, la qualità del segnale audio possa risultare più che accettabile.

Infatti, come si è visto, nel passaggio dalla configurazione in cui veniva utilizzato un emulatore sia per la trasmissione che per la ricezione a quella in cui la prima di tali operazioni veniva effettuata da un dispositivo reale, la qualità del segnale ha riportato un netto incremento, e un miglioramento ulteriore si è ottenuto registrando e riproducendo il segnale che transita attraverso il gateway prima che questo venisse ricevuto dall'emulatore. Si può quindi ragionevolmente supporre che il segnale trasmesso tra due dispositivi reali possa essere di buona qualità, specialmente considerando che il formato di codifica scelto genera una compressione molto spinta: il bitrate nominale è infatti di soli 4,75 kbit/s.

4.4. Comunicazione tra terminale e softphone

Nel progetto *Chipflip*, dal quale si è partiti per lo sviluppo di *UniPR-Ptt*, i valori dei vari campi che compongono l'header dei pacchetti SIP/SDP venivano impostati in maniera non standard, in modo che l'instaurazione della connessione potesse avvenire solo tra due terminali che eseguivano il medesimo programma. Questa maniera di procedere, senz'altro lecita nel caso in cui i dati scambiati riguardavano un particolare gioco, avrebbe potuto essere mantenuta senza problemi anche nel progetto sviluppato. In altre parole, si sarebbe potuto impostare il contenuto dei campi che definiscono i parametri

della comunicazione in modo tale che i pacchetti di segnalazione SIP/SDP provenienti da un terminale su cui è in esecuzione il programma *UniPR-Ptt* venissero accettati solamente da un altro terminale sul quale fosse in esecuzione lo stesso programma.

Si è invece scelto di modificare il contenuto dei campi opportuni in modo che questi contenessero valori standard, così da rendere possibile l'utilizzo del programma *UniPR-Ptt* anche per scambiare dati con normali softphone basati su SIP. Questi però, tipicamente, operano in modalità full duplex, e ovviamente non riconoscono i messaggi che, nel progetto realizzato, informano l'interlocutore dell'inizio della fase di trasmissione o di ricezione. Per questo motivo, come si è detto nel capitolo precedente, è stata sviluppata una versione alternativa del progetto in cui l'utente può commutare il proprio stato, dalla trasmissione alla ricezione e viceversa, senza mettere al corrente l'interlocutore di tale variazione. Questa versione potrebbe dunque essere utilizzata per realizzare la comunicazione (pur se in modalità half duplex) con un softphone che supporti il formato audio AMR. Sfortunatamente, nessuno degli applicativi a disposizione ha la possibilità di gestire dati in tale formato, ragion per cui non è possibile affermare con certezza che in tale scenario la comunicazione avverrebbe senza problemi.

Tuttavia, nelle prove effettuate utilizzando come softphone l'applicativo *X-Lite* della Xten Networks, si è verificato che, se il messaggio SIP di INVITE parte dal terminale su cui è in esecuzione *UniPR-Ptt*, *X-Lite* risponde con un messaggio SIP di 200 OK e instaura correttamente la chiamata. Tuttavia, a causa dell'incompatibilità tra i formati di codifica audio utilizzati nei due sistemi, non viene riprodotto alcunché. Se invece si tenta di iniziare la comunicazione da *X-Lite*, si riceve un messaggio di errore SIP di tipo 486 UNSUPPORTED MEDIA TYPE, poiché l'applicativo *UniPR-Ptt* è stato sviluppato in modo da bloccare il tentativo di connessione se il codec utilizzato dall'interlocutore non è supportato.

5. Conclusioni e sviluppi futuri

Il risultato principale del presente lavoro di tesi è consistito nella realizzazione del progetto *UniPR-Ptt*. Questo è un applicativo, scritto in linguaggio C++ e destinato all'impiego su smartphone dotati di sistema operativo Symbian, che permette di scambiare stream di dati audio in modalità push-to-talk tra due terminali, sfruttando i protocolli SIP e SDP per l'instaurazione della sessione e il protocollo RTP per la trasmissione dei dati audio.

Per la realizzazione del progetto in questione è stato necessario dapprima approfondire la conoscenza dei protocolli utilizzati e del metodo di programmazione a oggetti tipico del linguaggio C++, nonché studiarne le peculiarità in ambiente Symbian. In seguito è stato preso in esame il plug-in per SIP fornito da Nokia, ed è stato studiato in particolare l'esempio *Chipflip*. Questo esempio è poi stato modificato per gradi, aggiungendo le varie funzioni necessarie allo scopo prefissato ed eliminando quelle superflue. Il programma risultante risponde sostanzialmente alle specifiche definite, anche se presenta alcuni problemi tuttora non risolti. Una volta trovata la soluzione a tali problemi, è possibile sviluppare ulteriormente il progetto *UniPR-Ptt* introducendo nuove funzionalità, alcune delle quali verranno proposte nel seguito.

Il progetto, così come è stato sviluppato, consente di scambiare flussi di dati audio in modalità push-to-talk tra due emulatori di terminali Nokia Series 60 o tra un emulatore e un terminale vero e proprio. In quest'ultimo caso, tuttavia, si verifica un errore nel caso in cui l'invito sia diretto dall'emulatore al terminale, mentre se lo stesso messaggio è inviato nell'altro verso la comunicazione viene instaurata senza problemi. Il caso in cui la comunicazione avviene tra due terminali non è stato testato, ma è verosimile attendersi che soffrirebbe del problema appena descritto. Un altro problema riguarda il messaggio che permette di terminare la comunicazione: questo, infatti, viene inviato direttamente da un terminale all'altro, senza passare per il server SIP, e quindi può non arrivare a destinazione se i due terminali sono collegati a reti private differenti. Questo inconveniente sembra essere causato dall'implementazione del protocollo SIP contenuta

nel plug-in sviluppato da Nokia, che non consente di forzare il transito attraverso il server dei messaggi SIP inviati dopo la conclusione dell'handshake iniziale.

La soluzione a tale problema, quindi, potrebbe consistere in una modifica nel comportamento del server SIP, il quale dovrebbe sostituire l'indirizzo IP inserito originariamente dal terminale con il proprio non solo nei messaggi SDP ma anche negli header SIP. In questo modo, ognuno degli user agent vedrebbe il server SIP come proprio interlocutore e non potrebbe far altro che inviare ad esso tutti i pacchetti, sia di segnalazione che di traffico, diretti all'altro terminale.

Risolti questi due inconvenienti, dovrebbe essere possibile trasmettere e ricevere flussi di dati audio in modalità push-to-talk tra due terminali. La qualità dell'audio, in questo caso, dovrebbe essere decisamente buona, visto che, come è stato verificato sperimentalmente, nel passaggio dalla situazione in cui si utilizza l'emulatore sia per trasmettere che per ricevere a quella in cui la trasmissione viene effettuata dal dispositivo reale si ottiene un netto miglioramento nella qualità del segnale ricevuto, e che questa cresce ancora se si memorizza il segnale trasmesso e lo si riproduce mediante un apposito lettore.

Il progetto realizzato, a questo punto, potrebbe venire ulteriormente sviluppato in varie direzioni. Una delle estensioni possibili consiste nel supporto di sessioni che coinvolgono più di due interlocutori. In ogni caso, per implementare questa funzione occorre utilizzare la versione del programma *UniPR-Ptt* in cui gli utenti non si scambiano i messaggi che indicano le variazioni di stato e i relativi acknowledgement, nonché definire la politica da adottare nell'eventualità in cui più utenti vogliano trasmettere dati contemporaneamente; il supporto della conferenza potrebbe poi essere realizzato secondo due modalità distinte.

Il primo metodo consiste nell'introdurre la possibilità di invitare un utente a partecipare a una sessione già attiva, facendo poi in modo che gli stessi dati siano inviati a tutti gli interlocutori. Questo modo di procedere non necessiterebbe di alcun server aggiuntivo ma richiederebbe pesanti modifiche al progetto *UniPR-Ptt* e porterebbe facilmente a problemi dovuti all'insufficiente larghezza di banda, quando il numero dei partecipanti alla sessione aumentasse oltre una certa soglia.

In alternativa, sarebbe possibile modificare il server SIP in modo che possa funzionare anche come mixer RTP e definire indirizzi SIP associati a un gruppo di utenti anziché a un utente singolo. Per iniziare una conferenza sarebbe sufficiente inviare l'opportuno messaggio a uno di questi indirizzi, in modo che il server lo inoltri a tutti gli utenti ad esso associati. In seguito, il terminale dell'utente che intende parlare potrebbe inviare un unico flusso, diretto verso il server, il quale poi si dovrebbe occupare di replicarlo a tutti i destinatari. Così facendo si ottimizzerebbe l'uso della banda e non sarebbero necessarie modifiche ai client, ma dovrebbe essere introdotto un server apposito.

Un'altra modifica che potrebbe essere effettuata consiste nello sfruttare effettivamente gli header RTP dei pacchetti, che nella versione corrente vengono

semplicemente scartati, per effettuare il riordino di eventuali pacchetti giunti fuori sequenza. A tale scopo, i pacchetti ricevuti potrebbero essere inseriti in un buffer non nello stesso ordine col quale arrivano a destinazione, ma in ordine crescente di numero di sequenza. Sarebbe inoltre necessario iniziare la riproduzione dei dati solo dopo aver memorizzato un congruo numero di pacchetti, in modo da avere a disposizione dati da riprodurre mentre si attende l'arrivo di un eventuale pacchetto fuori sequenza.

L'introduzione della possibilità di utilizzare codec audio differenti dall'AMR a 4,75 kbit/s potrebbe rappresentare un ulteriore sviluppo del progetto realizzato. Il terminale supporta infatti, in maniera nativa, sia codec AMR a bitrare più elevati (4,75 kbit/s è infatti il minimo bitrate che può essere generato dal codec AMR, il quale d'altronde è realizzato in modo da poter variare la banda utilizzata anche mentre la comunicazione è in corso) che formati di codifica differenti. Se si riuscisse a codificare i dati in diversi formati, si potrebbe passare da uno all'altro di essi anche nel corso di una sessione attiva, in modo da variare il bitrate in funzione del carico attuale della rete.

Affinché il terminale che sta inviando i dati possa conoscere le condizioni della rete e quindi, eventualmente, decidere di cambiare il codec in uso, sarebbe possibile utilizzare RTCP: se in ognuno dei terminali che partecipano alla comunicazione venisse implementato tale protocollo, infatti, chi si trova attualmente in fase di trasmissione potrebbe sapere se il flusso di dati che sta inviando viene ricevuto correttamente o meno, e in base a queste informazioni potrebbe variare il codec utilizzato. Tali operazioni comporterebbero tuttavia un notevole aumento del carico computazionale richiesto al dispositivo, che potrebbe risultare ingiustificato soprattutto se, come è prevedibile, già con un bitrate di soli 4,75 kbit/s si ottenesse un segnale di buona qualità.

L'ultimo (nonché probabilmente il più significativo) degli sviluppi futuri proposti consiste nell'introduzione della possibilità di scambiare i flussi audio in modalità full-duplex. Le modifiche da effettuare al codice per raggiungere tale risultato sarebbero tutto sommato abbastanza limitate: sarebbe infatti sufficiente attivare contemporaneamente sia l'operazione di campionamento e invio dei dati che quella di ricezione e riproduzione. L'esecuzione contemporanea delle operazioni di trasmissione e ricezione dei dati non crea alcun problema (e infatti è stata realizzata anche all'interno del progetto *UniPR-Ptt* per inviare i keepalive mentre è in corso l'operazione di riproduzione). Se però si tenta di effettuare contemporaneamente il campionamento di un segnale e la riproduzione di un altro, richiamando le stesse funzioni utilizzate per eseguire tali operazioni in sequenza, l'esecuzione del programma si interrompe e si ottiene un messaggio di errore.

Stando a quanto dichiarato da Symbian[1], tuttavia, tale restrizione non è dovuta né a limiti intrinseci dell'hardware né al mancato supporto della modalità full-duplex da

[1] FAQ 1199 del 23/12/2004, disponibile sul sito www.symbian.com.

parte del sistema operativo, ma unicamente alla politica scelta per consentire agli sviluppatori di implementare questa modalità. Il supporto delle comunicazioni in full-duplex non rientra infatti nelle funzionalità che devono essere supportate obbligatoriamente dai produttori di terminali affinché questi siano conformi al sistema operativo Symbian, e in quanto tale non è stato incluso nell'implementazione di riferimento, non è documentato ufficialmente e non può essere attivato semplicemente richiamando una dopo l'altra le funzioni che effettuano il campionamento di un segnale e la riproduzione di un altro. Per realizzare applicativi che, su terminali abilitati, possano operare in full-duplex, gli sviluppatori devono richiedere le necessarie informazioni direttamente alla casa produttrice dei terminali stessi.

In particolare, uno degli smartphone che supportano la modalità full-duplex è proprio il modello 6630 della Nokia, utilizzato per i test. Tuttavia, sempre secondo Symbian, per attivare il full-duplex su tale dispositivo è necessario richiamare un'apposita funzione proprietaria, che però non è resa nota né documentata. Per ricevere le informazioni relative, gli sviluppatori devono contattare direttamente Nokia, che, a propria discrezione, può fornire le opportune indicazioni.

Bibliografia

[1] 3GPP, *Architecture for an All IP network*, 3GPP TR 23.922 v1.0.0, 1999.

[2] 3GPP, *IP Multimedia Subsystem (IMS)*, 3GPP TS 23.228 v7.1.0, 2005.

[3] Alcatel, *Session Initiation Protocol – SIP*, Alcatel Internetworking, 2003.

[4] G. Camarillo, M. A. Garcia Martin, *The 3G IP Multimedia Subsystem (IMS): Merging the Internet and the Cellular Worlds*, John Wiley and Sons, 2004.

[5] M. Canali, L. Polentini, *PoC – Push-to-talk over Cellular – Tecnologia e mercato*, Ministero delle Comunicazioni, 2003.

[6] Cisco, *Guide to Cisco Systems' VoIP Infrastructure Solution for SIP*, Cisco Systems, 2000.

[7] D. Craven, *Series 60 Application Development*, University of Western Ontario, 2004.

[8] M. De Gregorio, *Un sistema integrato per la segnalazione ed il controllo di sessioni multimediali*, Università degli Studi di Roma "La Sapienza", 2000.

[9] Digia Inc., *Programming for the Series 60 Platform and Symbian OS*, John Wiley and Sons, 2003.

[10] Forum Nokia, *ChipFlip Application API Documentation*, Version 3.0, Nokia Corporation, 2005.

[11] Forum Nokia, *ChipFlip Application Manual*, Version 3.0, Nokia Corporation, 2005.

[12] Forum Nokia, *Developer Platform 2.0: Known Issues*, Version 2.1, Nokia Corporation, 2005.

[13] Forum Nokia, *Nokia SIP Plug-in 3.0 for Series 60 – Programmer's Guide*, Version 3.0, Nokia Corporation, 2005.

[14] Forum Nokia, *Nokia SIP Plug-in 3.0 for Series 60 – User's Guide*, Version 3.0, Nokia Corporation, 2005.

[15] Forum Nokia, *Series 60 Developer Platform 2.0: Getting Started with C++ Application Development*, Version 1.1, Nokia Corporation, 2004.

[16] Forum Nokia, *SDP Codec API Specification*, Version 1.0, Nokia Corporation, 2004.

[17] Forum Nokia, *SIP Codec API Specification*, Version 1.0, Nokia Corporation, 2004.

[18] Forum Nokia, *SIP Client API Specification*, Version 1.0, Nokia Corporation, 2004.

[19] Forum Nokia, *SIP Frequently Asked Questions*, Version 1.0, Nokia Corporation, 2004.

[20] Forum Nokia, *SIP Profile API Specification*, Version 1.0, Nokia Corporation, 2004.

[21] Forum Nokia, *White Paper: IP Convergence Based On SIP – Enhanced Person-To-Person Communications*, Version 1.0, Nokia Corporation, 2004.

[22] O. Hersent, J.P. Petit, D. Gurle, *IP Telephony : Deploying Voice-over-IP Protocols*, John Wiley & Sons, 2005.

[23] Ixia, *Session Initiation Protocol (SIP) Technology*, Ixia, 2004.

[24] N. Johnson, *Audio Streaming – How to successfully stream audio on Symbian OS v7.0s*, Symbian Ltd, 2004.

[25] S. Leggio, *Session Initiation Protocol and Signaling: State-of-the-Art and QoS Related Open Issues*, University of Finland, 2003.

[26] H. Lindholm-Ventola, J. Aromaa, A. Mustonen, *Session Initiation Protocol*, Satakunta Polytechnic, 2004.

[27] G. Meiklejohn, *Using the sockets API*, Symbian Developer Library, 2005.

[28] Nokia, *Developer Platform 2.0 for Series 60: Designing C++ Applications*, Nokia Corporation, 2003.

[29] Nokia, *Developer Platform 2.0 for Series 60: Introduction to Designing C++ Applications*, Nokia Corporation, 2003.

[30] Nokia, *Push to Talk over Cellular – stay connected*, Nokia Corporation, 2005.

[31] Nokia, *Series 60 Developer Platform 2.0: FAQ*, Version 1.1, Nokia Corporation, 2004.

[32] Nokia, *Series 60 Developer Platform: Introductory White Paper*, Version 1.1, Nokia Corporation, 2004.

[33] Nokia, *Symbian OS: Active Objects And The Active Scheduler*, Version 1.0, Nokia Corporation, 2004.

[34] Nokia, *Symbian OS: Coding Conventions in C++*, Version 1.0, Nokia Corporation, 2004.

[35] U. Pezzano, T. Pratesi, *Utilizzo di reti per la videocomunicazione*, Università degli Studi di Firenze, 1996.

[36] T. Piikivi, *Implementation of Event Subsystem based on Session Initiation Protocol*, Oulu Polytechnic, 2004

[37] J. Rosenberg et al., *SIP: Session Initiation Protocol*, RFC 3261, 2002.

[38] H. Schulzrinne et al., *RTP: A Transport Protocol for Real-Time Applications*, RFC 3550, 2003.

[39] H. Schulzrinne et al., *RTP Profile for Audio and Video Conferences with Minimal Control*, RFC 3551, 2003.

[40] H. Schulzrinne, *The Session Initiation Protocol (SIP)*, Columbia University, New York, 2001.

[41] H. Sinnreich, A. Johnston, *Session Initiation Protocol (SIP) and MCI Advantage*, MCI, 2003.

[42] D. Sisalem, J. Kuthan, *Understanding SIP*, GMD Fokus, 2001.

[43] Sonera MediaLab, *Symbian Application Development White Paper*, Sonera MediaLab, 2003.

[44] Symbian, *Coding Idioms for Symbian OS*, Symbian Ltd, 2002.

[45] Symbian, *Symbian OS C++ Coding Standards*, Symbian Ltd, 2002.

[46] Symbian, *Writing Good Symbian OS Applications*, Symbian Developer Network, 2005.

[47] M. Tasker, *Active Objects*, Symbian Developer Library, 1999.

[48] M. Tasker et al., *Professional Symbian Programming – Mobile Solutions on the EPOC Platform*, Wrox Press Ltd, 2000.

[49] D. Tosi, *Estensioni del protocollo di segnalazione SIP per reti mobili UMTS*, Università degli Studi di Milano Bicocca, 2002.

[50] L. Veltri, *Appunti del corso di Reti di telecomunicazioni C*, Università degli Studi di Parma, 2004.