UNIVERSITÀ DEGLI STUDI DI ROMA
"LA SAPIENZA"

CORSO DI
COMUNICAZIONI ELETTRICHE

Tesi di laurea

"Progetto ed implementazione di un sistema di comunicazione vocale tempo reale mediante l'utilizzo del protocollo RTP"

A.A. 1999/2000

Relatore

Maria Gabriella Di Benedetto

Laureando

Mari Daniele
Matr. 09091663

Indice

Introduzione

Per più di due decenni Internet ha rappresentato per ognuno di noi un veicolo di ricerca, educazione ed informazione, gli ultimi due anni hanno testimoniato la sua enorme crescita e le sue grosse potenzialità nel fornire nuovi ed avanzati servizi. In particolare, l'utilizzo di Internet come mezzo di trasporto per conversazioni telefoniche, noto come "Internet telephony" o "Voice over IP" (VoIP), ha investito l'industria delle telecomunicazioni come una valanga.

La trasmissione di voce in tempo reale su di una rete IP (Internet Protocol) ha attirando l'attenzione di tutte le realtà interessate ad investire nelle nuove tecnologie emergenti di Internet. Il motivo di una tale attrazione verso il fenomeno VoIP è da ricercare soprattutto nella nascita di nuovi e avanzati sistemi di comunicazione multimediali, quali le video conferenze, l'application sharing, il white-boarding e quant'altro. In più la telefonia IP permette una grossa riduzione del costo delle comunicazioni vocali a lunga distanza.

Per quanto riguarda le grosse aziende, l'abbassamento dei costi delle comunicazioni è dovuto ha numerosi aspetti legati a questa nuova tecnologia. Innanzitutto ci si aspetta una grossa riduzione dei costi relativi alle spese dei capitali. Ciò è dovuto alla possibilità di dover gestire e mantenere un'unica infrastruttura, in grado di far convergere dati, video e voce. Per fare un esempio, queste tre diverse categorie di dati, associabili a tre diverse tecnologie, possono essere convogliate in un unico filo che funga da mezzo di trasporto. Inoltre è possibile ridurre notevolmente i costi delle telefonate a lunga distanza "bypassando" la tradizionale rete PSTN. I cambiamenti che, col tempo, sarà necessario apportare ad una struttura di questo tipo saranno certamente meno onerosi di quelli associati alla vecchia tecnologia. Ad esempio, non è un segreto che le tradizionali PBX (Private Branch Excange) sono sia concettualmente che tecnologicamente vecchie, per cui fuori dal normale ciclo di ammortamento di un'azienda.

Ma se per un singolo utente l'abbassamento dei costi ha una grossa rilevanza, per un'azienda esso si presenta come il vantaggio minimo associabile alle reti IP multiservizio. L' aspetto predominante è la convergenza dei vari servizi in un'unica infrastruttura che lancia l'azienda in un mercato competitivo, rivoluzionando tutti i livelli delle applicazioni.

Ciò porta verso una nuova visione del mondo delle comunicazioni multimediali. Il vecchio mondo della telefonia caratterizzato da applicazioni limitate, controlli delle chiamate vincolati ed infrastrutture proprietarie sta per essere scavalcato dalla nuova tecnologia VoIP fondata su un'architettura aperta [1]. La possibilità di costruire delle

infrastrutture aperte, per comunicazioni integrate, permette lo sviluppo di nuovi sistemi comunicativi basati sul Web in grado di conquistare una grossa fetta di mercato. Del resto, servizi come quelli precedentemente elencati non potranno mai essere forniti dall'attuale rete PSTN.

Volendo, si può fare un esempio di come potrebbe essere una telefonata IP integrata. Daniele è seduto davanti al suo computer collegato ad Internet ed, all'improvviso, sente un segnale acustico seguito da un messaggio: "Chiamata audio-video da Gabriele". Egli accetta la chiamata e conversa per un momento. Successivamente ritiene opportuno che anche Katia sia coinvolta nella conversazione. Allora, cerca Katia nella rubrica del suo PC ed invia la chiamata. La chiamata raggiunge il "call-software" di Katia. Tale software era stato predisposto da Katia in modo che dirottasse tutte le chiamate, in parallelo, sul suo telefono cellulare e sul suo PC di casa. Katia sente il proprio cellulare squillare mentre è in macchina e risponde, aggiungendosi alla conversazione in modalità solo-voce. Durante la conversazione Daniele si ricorda di un documento presente in Internet da mostrare, cerca la pagina Web e lo inoltra nella comunicazione. Il display del telefono di Katia ne mostra una versione solo-testo, mentre lei continua a conversare. Gabriele decide di aggiungere Luca alla conversazione che non è raggiungibile, ma il suo "call-software" restituisce una pagina Web contente i suoi appuntamenti ed un link ad un servizio di email vocale. Un servizio del genere pur non essendo ancora disponibile sul mercato, mostra una realtà non lontana. Infatti, il crescente interesse verso l'emergente tecnologia VoIP appare come inarrestabile, visto che coinvolge numerosi settori delle telecomunicazioni.

Uno dei settori in cui la tecnologia VoIP sta avendo notevole influenza è l'E-commerce [2]. Un grande numero di potenziali acquirenti via Web limitano i loro acquisti frustrati dalla procedura delle transazioni e dai sospetti sulla scarsa sicurezza. Per tale motivo sono state sviluppate numerose tecniche in grado di soddisfare le richieste di sicurezza (es., integrità, autenticazione, autorizzazione, moduli elettronici ad hoc e così via). Nonostante questo grosso supporto tecnologico, il contatto umano è ancora un aspetto importante che rende, per molte persone, lo shopping nel mondo fisico più confortevole dello shopping online. La telefonia IP può essere il collegamento mancante nel processo di sviluppo del Web shopping. Ad esempio, si può pensare alla presenza di un tasto "Call Me" che connette l'utente ad un help desk per una conversazione vocale con un operatore dell'azienda. In questo senso le caratteristiche della telefonia IP possono permettere la creazione di un più solido mezzo d'assistenza per gli acquisti online.

E' ormai evidente che l'attuale tendenza di numerose imprese all'avanguardia è quella di sostituire, gradualmente, l'ormai obsoleta struttura PSTN (Public Switched Telephone Network) con reti IP multiservizio. Ma, va tenuto presente che, per una serie di ragioni (economiche, sociali, tecnologiche, qualità dei servizi), la telefonia IP e la tradizionale telefonia PSTN coesisteranno per molto tempo. Per questo motivo è emersa, ultimamente, una nuova classe di prodotti chiamati Internet Telephony Gateways [3] capaci di rendere operativo l'interscambio di dati tra reti telefoniche pubbliche e reti IP. La funzionalità che deve essere supportata da ogni gateway è quella di mappare l'indirizzo IP nel corrispondente numero di telefono PSTN. In questo modo

è possibile rendere disponibili i benefici di VoIP anche da un semplice telefono connesso alla normale rete telefonica. Tramite l'uso di gateway il pubblico, ma anche le grandi compagnie, può beneficiare dei vantaggi finanziari di VoIP senza l'uso diretto di un PC (Personal Computer).

Quando una telefonata è fatta attraverso la modalità VoIP il costo del trasferimento dei dati lungo Internet è nullo. In questo modo il costo delle chiamate a lunga distanza, incluso quello associato alle chiamate internazionali, è ridotto a quello relativo alla sola tratta di copertura PSTN. Di conseguenza una chiamata internazionale verrà a costare quanto una chiamata urbana.

Le modalità di comunicazione vocale attraverso reti IP sono sintetizzabili nelle seguenti quattro categorie:

- Da computer a computer (o PC-toPC).
- Da computer a telefono (o PC-to-phone).
- Da telefono a computer (o phone-to-PC).
- Da telefono a telefono (o phone-to-phone).

La modalità PC-to-PC è un chiaro esempio di come il mondo dell'informatica sia sempre più vicino a quello delle telecomunicazioni. La modalità ibrida tra vecchia e nuova tecnologia (telefono tradizionale-computer) necessita dell'interfaccia che ne permetta la coesistenza. Il concetto di interoperabilità tra vecchia è nuova tecnologia fornisce lo spunto per chiarire quali sono le principali differenze tra le reti IP e le reti PSTN.

La tradizionale rete telefonica è una rete a commutazione di circuito [4], ottimizzata per le comunicazioni vocali in tempo reale con una QoS (Quality of Service) garantita. Quando una sessione di comunicazione viene iniziata si stabilisce un circuito fisico tra il chiamato ed il chiamante. In questo modo i due utenti hanno a disposizione un loro canale per l'intera durata della conversazione. La PSTN dedica alla conversazione un circuito full-duplex ed una larghezza di banda costante pari a 64 kbit/s. Tale larghezza di banda rimane sempre inalterata indipendentemente dal fatto che i due utenti siano in conversazione attiva o in silenzio. Grazie al circuito dedicato ed alla quantità di banda, messi a disposizione di ogni conversazione, la rete PSTN è in grado di fornire un'ottima QoS.

Per contro, la rete IP è a commutazione di pacchetto e non dedica alcun circuito e nessuna larghezza di banda. Si tratta di una rete ottimizzata per la trasmissione dati in modo interattivo. I dati da trasmettere vengono spezzettati in pacchetti ed inviati a destinazione. Durante la trasmissione ogni pacchetto segue un percorso che può essere completamente diverso dagli altri pacchetti relativi allo stesso messaggio. Una volta giunti a destinazione, tutti i pacchetti vengono ricostruiti nella loro forma originale. In questo modo la banda viene gestita in modo dinamico ed efficiente.

La sostanziale differenza tra le due strategie di comunicazione sta proprio in questo. Con la commutazione di circuito, la banda dedicata ad un circuito assegnato, quando non è utilizzata, è sprecata. Mentre, con la commutazione di pacchetto essa può essere utilizzata da altri pacchetti provenienti da qualsiasi sorgente e diretti a qualsiasi destinazione, poiché non vi sono circuiti dedicati (a meno di non considerare circuiti virtuali).

L'efficienza nell'utilizzo delle risorse, tipica delle reti IP, è stata concepita per trasferire dati, la cui natura, non esige il rispetto di vincoli imposti da comunicazioni vocali tempo-reale. Infatti le reti IP non sono in grado di garantire, per il traffico vocale, una QoS al pari di quella fornita dalla PSTN. Come vedremo meglio nel corso della trattazione, è proprio questa la nota dolente della tecnologia VoIP. D'altra parte le grosse opportunità che questo settore può offrire ad ogni utente, e ad ogni azienda, incoraggiano continue ricerche ed investimenti che fanno ben sperare. Nel seguito parleremo di alcuni prodotti già disponibili sul mercato in grado di garantire un'adeguata QoS, soprattutto sulle reti locali.

L'intento del presente lavoro è di descrivere le caratteristiche di un reale sistema di comunicazione, tipo VoIP, implementato nella modalità PC-to-PC. La tesi è stata svolta presso il Laboratorio Voce del dipartimento di INFOCOM (INFOrmazione e COMunicazione). L'intero materiale è disponibile presso la segreteria del dipartimento. La trattazione è organizzata in tre parti fondamentali:

- Nella prima parte sono mostrati gli aspetti generali di un architettura VoIP, i quali, sono presentati in veste introduttiva e successivamente approfonditi.

- La seconda parte consta di una vasta ed attenta analisi dei protocolli di comunicazione impiegati nelle applicazioni VoIP. Particolare riguardo è dedicato al protocollo RTP (Real Time Transport Protocol), standard proposto per comunicazioni tempo reale su reti IP.

- La terza parte descrive in dettaglio le scelte di progetto e le prestazioni del sistema di comunicazione realizzato. Il progetto è implementato attraverso un software in grado di mettere in comunicazione due macchine connesse in rete. E' prevista l'intera catena di ricetrasmissione.

Le tre parti sono distribuite in quattro capitoli, nel seguente modo. Nel cap. I verranno discusse le caratteristiche generali di un'architettura VoIP, esplicitando i problemi associati all'utilizzo di una rete a commutazione di pacchetto per trasmissioni tempo reale. Nel cap. II si tratterà la tecnologia VoIP dal punto di vista protocollare, dando un'approfondita analisi di alcuni standard proposti e di altri standard già consolidati. Nel cap. III si approfondirà l'aspetto della codifica relazionandolo alla gestione delle risorse di rete; in particolare, si darà una breve descrizione di due formati di codifica facenti parte delle raccomandazioni "G.711" e "G.729" dell'ITU (International Communications Union). Nel cap. IV verrà esposto in dettaglio il progetto software di un sistema di comunicazione VoIP del tipo PC-to-PC, dal quale nasce il reale sistema di comunicazione oggetto di queste note: *RTP-Speak*. Il progetto verrà descritto nella sua interezza facendo continui riferimenti alle aspettative teoriche e ad i risultati pratici. Infine, verranno date alcune conclusioni e degli spunti per ulteriori sviluppi del progetto. L'Appendice A contiene il pacchetto software, un manuale d'uso di *RTP-Speak* ed alcune routine di utilità generale. L'Appendice B contiene un

pacchetto software su una versione più completa di RTP-Speak, funzionante ma da ottimizzare.

Ringraziamenti

Desidero ringraziare la prof.ssa Maria Gabriella Di Benedetto per la fiducia dimostratami ed il suo costante sostegno.
Inoltre, desidero ringraziare la mia famiglia che mi è stata sempre vicina durante l'intero corso di studi.

Daniele Mari

Capitolo I

Caratteristiche generali di un’architettura VoIP.

In questo primo capitolo saranno messi in evidenza gli aspetti generali di un’architettura VoIP. L’intento è quello di fornire un’ampia panoramica del fenomeno VoIP, per poi scendere nei dettagli nei successivi capitoli. Per questa ragione, ogni argomento sarà presentato in veste introduttiva. Questa scelta è strettamente legata alla natura stessa della realtà VoIP, la quale, essendo molto vasta, necessita di una carrellata globale di tutte le sue caratteristiche.

Il capitolo è organizzato nel seguente modo. Nel par. I.1 sarà presentata una possibile architettura per comunicazioni VoIP che prende spunto da configurazioni di telefonia IP realmente esistenti. Nel par. I.2 saranno evidenziate le problematiche associate alla QoS; particolare riguardo sarà dedicato al modo in cui si sta agendo per arginarle. Nel par. I.3 si parlerà dei problemi relativi alla QoV (Qualità of Voice) e delle attuali strategie di miglioramento. Infine, nel par. I.4 si farà il punto sulla qualità della comunicazione VoIP, dando alcune prime conclusioni.

I.1 Visione macroscopica di un'architettura VoIP.

Le comunicazioni VoIP sono realizzate su uno scenario molto vasto. La comunicazione vocale può avvenire all'interno di una singola rete, ad esempio una LAN (Local Area Network), o attraverso più reti interconnesse, ad esempio una WAN (Wide Area Network) o, nel più generale dei casi, Internet. A tale proposito in fig. I.1 è mostrato uno schema generale che prende spunto da configurazioni di telefonia IP realmente esistenti.

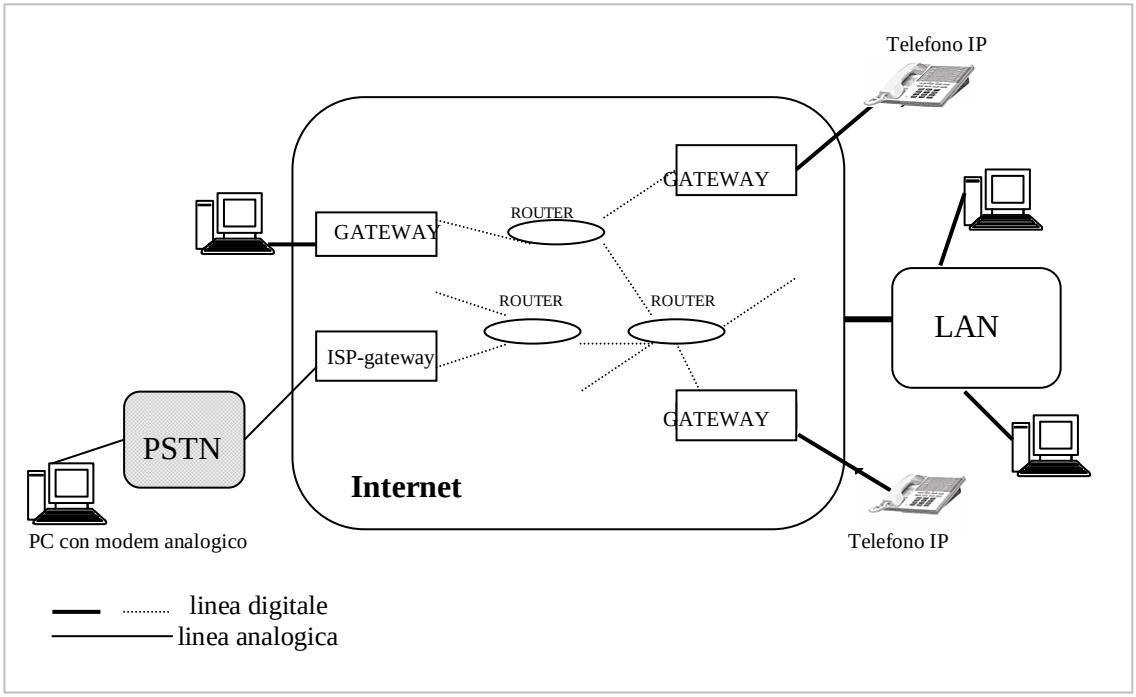


Fig. I.1. Possibile architettura VoIP.

Prima di introdurre i concetti legati alla comunicazione vocale su reti IP, diamo un breve richiamo sulla struttura di una rete a commutazione di pacchetto.

Fondamentalmente la rete consiste di due parti distinte: linee di trasmissione e nodi di commutazione. Le linee di trasmissione sono il mezzo di trasferimento dei bit tra le macchine. I nodi di commutazione, spesso conosciuti come router, sono dei computer utilizzati per connettere più linee di trasmissione. Tutto il traffico da o verso le macchine si svolge attraverso i router. Quando un pacchetto è inviato da una macchina all'altra passa attraverso uno o più router intermedi, i quali lo ricevono nella sua interezza e lo memorizzano finché non si libera la linea d'uscita richiesta, dopodiché lo inoltrano. L'instradamento dei pacchetti dalla macchina di provenienza a quella di destinazione va visto in relazione alla variabilità delle risorse (banda) della rete stessa. L'algoritmo d'instradamento, infatti, ha la responsabilità di decidere su quale linea d'uscita va trasmesso un pacchetto in arrivo, e la decisione va presa nel rispetto di una serie di vincoli volti ad evitare il rischio di "congestione". La congestione si verifica

quando in una parte della rete sono presenti così tanti pacchetti che il router non è in grado di gestirli e comincia a perderne. In condizioni di traffico altissimo le prestazioni possono subire un collasso totale che sfocia nell'impossibilità di trasmettere alcun pacchetto. D'altra parte, a meno di quest'ultima situazione limite, è evidente come la tecnica comunicazione a commutazione di pacchetto utilizza la banda a disposizione in modo estremamente efficiente.

Nel caso di una telefonata IP tra due PC la situazione, in linea di massima, è la seguente. Innanzi tutto è necessario che i due PC siano contemporaneamente collegati in rete. In trasmissione, la voce è acquisita dal microfono ed, attraverso un opportuno software, è convertita in pacchetti IP. Nel caso in cui il chiamato ed il chiamante siano connessi alla rete digitale (LAN, WAN o Internet), i pacchetti saranno direttamente spediti e dirottati verso il destinatario. Diversamente, nel caso in cui i due utenti siano connessi ad Internet da casa tramite un modem, quest'ultimo sarà la porta di collegamento tra il PC e la rete PSTN o ISDN. A valle di tale rete ci sarà un ISP-gateway (ISP, Internet Service Provider) che provvederà alla riuscita della connessione con Internet dirottando i pacchetti vocali verso il destinatario. All'interno della rete IP, i pacchetti saranno dirottati da nodi intermedi (routers) da un punto all'altro fino a destinazione. In ricezione, la voce subirà l'elaborazione contraria. I singoli pacchetti saranno ricombinati dal software di ricezione e la voce sarà udibile dagli speaker.

Spesso le aziende sono connesse ad Internet direttamente attraverso una linea digitale, e ciò garantisce, come vedremo meglio nel seguito, una migliore QoS.

Nel caso che la telefonata avvenga tra due telefoni IP[1] o tra un telefono ed in PC, la situazione è pressoché analoga.

Quanto detto, mostra come, il segnale vocale, durante la comunicazione è soggetto ad una serie d'elaborazioni. Nel caso la comunicazione coinvolga solo canali digitali, si ha:

- *Acquisizione*

  Il segnale vocale è acquisito tenendo conto che, nonostante il suo spettro si estenda fino a circa 20 kHz, la maggior parte dell'energia è concentrata nei primi 4 kHz. Dunque, il segnale è fatto passare in un primo filtro analogico "antialiasing" (ovvero un passa-basso con frequenza di taglio intorno ai 4 kHz), successivamente è campionato con una frequenza di campionamento pari ad 8 kHz (teorema di Nyquist).

- *Digitalizzazione*

  Il segnale campionato è digitalizzato da un opportuno ADC (Analog to Digital Converter), in modo da poter essere trattato da un sistema digitale a precisione finita. Tipicamente si segue la quantizzazione PCM lineare

[1] Attualmente, esistono numerose proposte commerciali di telefoni, dotati di un apposito DSP (Digital Signal Processor), in grado di elaborare il segnale vocale per comunicazioni su reti IP.

(16 bit/campione). La maggior parte dei codificatori prevede in ingresso questo formato.

- *Prefiltraggio digitale*

  Le modalità di prefiltraggio variano a seconda delle implementazioni. In generale si tratta d'elaborazioni rivolte a rimuovere alcuni aspetti di disturbo o di ridondanza presenti nel segnale: echi di linea, silenzi, componente continua e quant'altro. Spesso queste operazioni sono comprese nell'algoritmo di codifica.

- *Codifica*

  Nella realtà di Internet in cui la disponibilità di banda può essere scarsa per lunghi periodi, la compressione vocale gioca un ruolo importante. Di essa si parlerà ampiamente nel cap. III, per il momento diciamo che, la codifica, tende a minimizzare il numero di bit necessari a mantenere l'informazione sul segnale vocale. L'elevata compressione della voce fornita da alcuni codificatori (da 64 kbit/sec a 8, 6.4, 5.3 kbit/sec) permette di risparmiare una notevole quantità di banda. Il risparmio va pagato con una riduzione della qualità della voce, che in ogni caso si mantiene accettabile.

- *Pacchettizzazione*

  Dopo la compressione, il segnale vocale è pronto per essere spedito in rete. La pacchettizzazione [5] [6] è il perno centrale su cui si fonda la trasmissione di dati su reti IP. Di essa si parlerà ampiamente nel seguito, per il momento diciamo che il messaggio vocale è spezzettato ed incapsulato attraverso una serie d'intestazioni. La strategia è analoga al sistema di spedizione postale. Ogni lettera (messaggio) prima di essere spedita è inserita in una busta (pacchetto) su cui si scrive l'indirizzo del destinatario (intestazione). Il segnale vocale viene incapsulato in un pacchetto RTP (Real Time Protocol) con 12 byte[2] d'intestazione, successivamente vengono aggiunti altri 8 byte d'intestazione UDP (User Datagram Protocol) ed infine 20 byte d'intestazione IP. Diciamo subito che i 40 byte d'intestazione RTP/UDP/IP possono essere notevolmente ridotti mediante tecniche di compressione, un esempio è proposto in [7].

[2] Nella documentazione di Internet, spesso, si suole usare il termine "ottetto", per indicare 8 bit, al posto del termine "byte". Questo perché la suite TCP/IP e supportata da alcuni computer che hanno byte di dimensione diversa da 8 bit (es., DEC-10). Chiarito ciò, in questo lavoro, i due termini verranno utilizzati indistintamente per indicare 8 bit.

- *Trasmissione*

  I pacchetti vocali, una volta creati, sono trasmessi all'interno di Internet insieme a tutti gli altri pacchetti di dati. L'indirizzo IP serve ad identificare la destinazione del pacchetto. Il pacchetto IP, va dalla gateway d'ingresso a quella di destinazione passando attraverso nodi intermedi (router) che leggono l'intestazione e lo dirottano di conseguenza. A destinazione i pacchetti vengono ricombinati nella loro forma originale. La rotta seguita dai pacchetti non è fissa ed il tempo che occorre ad attraversare la rete (latenza) non è noto a priori, il ritardo che ne consegue, se eccessivo, può degradare la qualità della conversazione. Ogni pacchetto può proseguire per una via diversa dagli altri ad esso correlati, quindi non è detto che l'ordine di trasmissione coincida con quello d'arrivo. Il fenomeno di variabilità dei ritardi e dell'ordine d'arrivo è noto come "jitter". Un altro fenomeno che può degradare la qualità della conversazione è la perdita di pacchetti. Quando uno o più pacchetti sono persi la conversazione è affetta da vuoti (gap). Ognuno di questi fenomeni sarà ampiamente trattato nel prossimo paragrafo.

Per ciò che riguarda le elaborazioni legate alla trasmissione video, gli aspetti generali presentati nei precedenti sei punti rassuntivi sono concettualmente analoghi. Va da sé che le diversità tra il segnale vocale ed il segnale video implicano delle differenze nelle modalità elaborative. Nel seguito si farà sempre riferimento al solo segnale vocale richiamando, solo ove ritenuto necessario, l'attenzione sugli aspetti inerenti l'integrazione voce-video (o audio/video: A/V).

Standard & Interoperabilità

Va da se che, la creazione di standard per la garanzia dell'interoperabilità delle varie soluzioni VoIP emergenti è un requisito fondamentale. Se viene instradato un pacchetto per il quale è stato adoperato un particolare algoritmo di codifica ed in ricezione il formato non è supportato, il servizio è inutile. Ovviamente, la necessità di standard non coinvolge solo la codifica, ma la gran parte degli algoritmi di ricetrasmissione.

Nei prossimi anni, è previsto un grosso miglioramento della "spina dorsale" di Internet, sia in termini di disponibilità di banda che di controllo della congestione. Ciò spinge verso prospettive immediate di standardizzazione. A tale proposito l'ITU (International telecommunication Union) ha dato vita, con la raccomandazione H.323 [8][9], alla standardizzazione della trasmissione A/V (Audio/Video) su reti IP. Inoltre, la IETF (Internet Engeneering Tasck Force) sta sviluppando protocolli, come il SIP (Session Initializzazion Protocol) [10], in grado di garantire l'interoperabilità.

Gli sforzi per la messa appunto degli standard si dirigono verso tre elementi nodali della telefonia IP: il formato di codifica A/V, il trasporto del flusso A/V ed il trasporto dei segnali di controllo.

I.2 Qualità del Servizio (QoS).

Una rete di comunicazione, per garantire a pieno una buona qualità del servizio [11] [12], deve fornire un trasferimento affidabile dell'informazione e nello stesso tempo garantire l'assolvimento dei vincoli imposti, ad alcuni parametri, dalla particolare applicazione. Nel caso di comunicazioni VoIP le costrizioni sono dettate dalla massima accettabilità dei ritardi, dal jitter e dalla perdita dei pacchetti.

Le reti IP sono state progettate per il trasferimento interattivo di dati e non per applicazioni tempo reale, quali il trasporto di voce e video. I ritardi, il jitter e la perdita di pacchetti sono fenomeni tanto frequenti all'interno di Internet quanto difficili da contenere, perlomeno entro i limiti di accettabilità necessari ad applicazioni VoIP. Per le applicazioni classiche (e-mail, ftp e quant'altro) la variabilità della QoS è un parametro tollerabile. Al contrario, per applicazioni di telefonia su IP la garanzia di una buona QoS si presenta come una grossa sfida. D'altra parte le affascinanti potenzialità di un servizio integrato da voce, dati e video sono motivo di forte incoraggiamento.

Mentre in all'interno di Internet la congestione e la scarsità di risorse sono sempre in agguato, all'interno di una LAN si possono raggiungere prestazioni eccellenti visto che, spesso, la banda a disposizione e tanta.

Probabilmente la strada più efficiente, per raggiungere le prestazioni desiderate, è quella di intervenire nella struttura della rete stessa piuttosto che cercare di adeguare l'esistente struttura alle nuove esigenze di VoIP. Invece, attualmente la tendenza è quella dell'adeguamento; il motivo di ciò, probabilmente, va ricercato nei costi relativi al rifacimento d'alcune parti della struttura delle reti IP, che potrebbero essere improponibili, per non parlare dei tempi; alcuni sostengono che la continua proliferazione di grandi quantità di banda a basso costo, col tempo, estinguerà il problema sulla QoS.

I.2.a Ritardi.

I ritardi, che inficiano il trasporto di voce su reti IP, sono di varia natura. Tali ritardi, se eccessivi, possono degradare molto la qualità della conversazione. Infatti, c'è la concreta possibilità che le parole dei due interlocutori si sovrappongano, rendendo la conversazione fastidiosa o addirittura incomprensibile. Affinché un utente non si accorga del ritardo, quest'ultimo deve essere sotto la soglia di circa 300ms. Studi fatti indicano che ritardi superiori ai 300ms sono avvertiti come se si conversasse in half-duplex piuttosto che in ful-duplex. Ad ogni modo, la tolleranza del ritardo varia significativamente da utente ad utente e da applicazione ad applicazione. Attualmente, all'interno di Internet, non è raro che i ritardi varino nell'intervallo di 200-450ms, perciò la comunicazione non è sempre gradevole.

Va tenuto presente che la natura dei ritardi è duplice: possono essere sia "fissi" sia "variabili".

I ritardi fissi sono da attribuire ai tempi necessari per l'elaborazione del segnale ed in particolare agli algoritmi di codifica e di pacchettizzazione. Si tratta di ritardi predicibili la cui entità può essere facilmente messa in conto a priori, ed in parte può essere gestita mediante modifiche alla complessità dei vari algoritmi. I maggiori ritardi imputabili alla codifica sono propri di quei compressori vocali che lavorano su trame[3] del segnale vocale; ciò da luogo ad una sorta di soglia di ritardo che è implicita nell'algoritmo ed al disotto della quale è impossibile scendere, a meno di un ripensamento del principio di codifica. D'altra parte è possibile ottimizzare l'implementazione dell'algoritmo in modo da minimizzare il più possibile il ritardo endogenerato. Di fatto, la complessità dell'algoritmo deriva dalla necessità di comprimere tanto, e non è raro che la riduzione della complessità sia pagata in termini di qualità (a tale proposito nel capitolo III vedremo un esempio concreto: il passaggio dal G.729 al G.729A).

I ritardi variabili sono meno docili e sono dovuti alla natura stessa della rete. Si è gia detto in precedenza che le modalità di trasmissione delle reti a commutazione di pacchetto non garantiscono a priori la necessaria quantità di banda tanto meno un percorso dedicato. Per tale ragione, i ritardi introdotti dalla rete (network latency) variano in dipendenza della disponibilità di risorse. Quest'ultima, a sua volta, è legata al carico istantaneo della rete, il quale non è per niente predicibile. Un router che deve dirottare un pacchetto su una linea è costretto ad attendere finché, su quella linea, c'è banda sufficiente.

Uno dei requisiti che la rete dovrebbe avere per garantire la voluta QoS è un meccanismo di controllo della quantità di traffico generabile dagli utenti (es., admission control and policing). Purtroppo la cosa non è semplice in quanto, ad esempio, spesso un utente non fa nulla per molto tempo e poi all'improvviso ha bisogno di una grossa quantità di banda per trasferire un grosso file; dunque, la rete, può passare da situazioni di poca attività a situazioni di grosso carico con tempi del tutto impredicibili. Il fatto è che, se per certi tipi di dati qualche millisecondo d'attesa in più non è un problema, per un pacchetto vocale ciò può essere decisivo. Infatti, com'è facilmente intuibile, se un pacchetto vocale arriva con eccessivo ritardo conviene scartarlo, poiché se inserito nel processo di riconbinazione dei pacchetti inficia di più sulla degradazione della comunicazione. In sostanza, è meglio accontentarsi di qualche "gap" sparso qua e là piuttosto attendere qualche secondo affinché arrivi il pacchetto mancante.

Un'interessante soluzione, proposta per arginare il problema dei ritardi, è quella legata al protocollo RSVP (Resource ReSerVation Protocol) concepito di recente. Si tratta di un protocollo progettato per riservare una certa porzione di banda proprio ad applicazioni che richiedono risposte in tempo reale, quali quelle di VoIP. Ciascun router che supporta tale protocollo ha la capacita, difatti, di riconoscere una priorità nei pacchetti che instrada, distinguendo tra quelli urgenti (real-time packets) e quelli meno, ed agendo di conseguenza. Una buona descrizione introduttiva su tale protocollo è disponibile in [13], ed una descrizione dettagliata è reperibile in [14], in ogni caso, l'argomento sarà ripreso nel cap. II.

[3] Per trama si intende un tratto di voce, tipicamente 10-30ms, che può essere considerato stazionario, dal quale si estraggono una serie di parametri in grado di modellizzare l'intero tratto.

E’ evidente che l’implementazione di un sistema di comunicazione vocale su reti IP necessita di un costante monitoraggio dei ritardi associati ai pacchetti. Questa è una delle esigenze che ha dato vita al protocollo di trasporto RTP, nel seguito vedremo come, attraverso il campo “sequence number” presente nella sua intestazione, riesca a tenere traccia dei ritardi relativi ad ogni pacchetto.

In tab. I.2.a è riportato un campionario riassuntivo dei ritardi. Il “jitter buffer” di cui fin’ora non si è fatto menzione sarà approfondito pel prossimo paragrafo. Il “queuing” è il periodo d’attesa dei pacchetti, nei buffer di ricetrasmissione delle macchine, prima del loro processamento.

Menzione a parte merita il ritardo introdotto dal modem, nel caso sia presente. Poiché molti utenti accedono ad Internet via modem e continueranno a farlo per molto tempo, una strategia che riduca il ritardo introdotto dal modem è necessaria per lanciare con successo le applicazioni VoIP basate sul PC. In [17] è disponibile un’ampia trattazione dell’argomento, in particolare è messo in evidenza quanto l’algoritmo di codifica incida sul ritardo imputabile al modem. Il ritardo di processamento varia in relazione al modello ed alla configurazione.

Elemento Causa	Tipo di ritardo	Dipendenza
CODIFICA	fisso	algoritmo
PACCHETTIZZAZIONE	fisso	algoritmo
QUEUING	variabile	uplink
RETE	variabile	traffico
JITTER BUFFER	fisso/variabile	algoritmo/traffico
MODEM	fisso	più fattori

Tab. I.2.a. Campionario dei ritardi.

I.2.b Jitter.

Il peggior effetto della variabilità dei ritardi è il fenomeno del jitter. In sostanza, mentre in trasmissione i pacchetti sono generati ad un tasso costante, in ricezione arrivano secondo intervalli temporali casuali (vedi fig I.2.b). Come vedremo nel seguito, questo fenomeno necessita di essere compensato attraverso l’implementazione di un “jitter-buffer”.

E’ ovvio che nell’implementare un sistema di voce su IP non si può prescindere da questo fenomeno. Infatti, per la comprensione del parlato è necessario che l’ordine dei pacchetti in arrivo combaci con quello in trasmissione, mentre i dati che non sono real-time non necessitano di quest’accortezza.

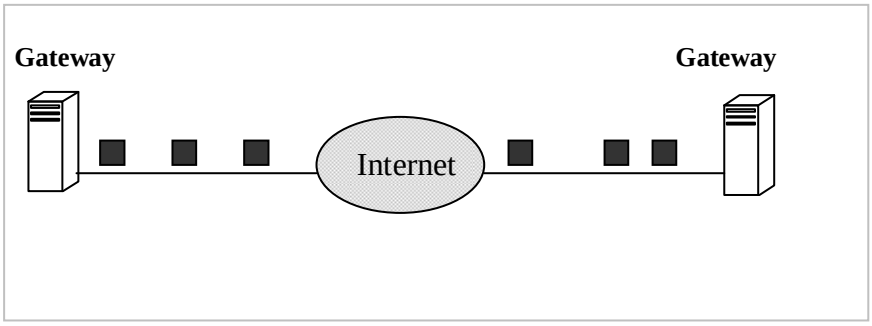


Fig. I.2.b. Jitter dei pacchetti.

L'algoritmo di decompressione vocale è fatto supponendo che le trame vocali arrivino ad intervalli costanti. Mentre il ritardo dovuto alla rete è aleatorio e la sua entità non è per niente predicibile. Ciò fa sì che la sequenza temporale, in trasmissione, non sia perfettamente mappata in ricezione. Dunque, non è possibile in alcun modo adattare l'algoritmo di decompressione al jitter dei pacchetti. La soluzione comunemente adottata è di realizzare un buffer all'interno del quale immagazzinare temporaneamente i pacchetti per poi presentarli al decodificatore secondo la giusta sequenza temporale e ad un tasso costante. Il tempo d'attesa, per ogni pacchetto, è solitamente rapportato al caso peggiore (che nel budget dei ritardi va vincolato ad un massimo di accettabilità). Questa tecnica, in letteratura, è nota come "equalizzazione dei pacchetti ricevuti". Un corretto dimensionamento del jitter-buffer garantisce un ascolto dei pacchetti secondo la giusta sequenza ma introduce ulteriore ritardo. Questo ritardo introdotto è strettamente legato alla disponibilità di risorse della rete, infatti, in una LAN è consigliabile un buffer che equalizi poco mentre in Internet il buffer deve equalizare molto. Un approccio interessante è quello di implementare un buffer la cui dimensione vari secondo il carico della rete, o in modo automatico, o da parte dell'utente mediante un apposito pulsante presente nell'interfaccia grafica. A tale proposito vedremo nel seguito come sia il campo "time stamp", presente nell'intestazione del protocollo RTP, che il protocollo RTCP (Real Time Control Protocol) diano delle indicazioni ad hoc per una scelta di questo tipo.

Per il calcolo del jitter solitamente si fa riferimento a due valori: la media degli intervalli temporali tra i pacchetti in arrivo (avarage of interarrival time) ($m_{\Delta t}$) e la deviazione standard ($\sigma_{\Delta t}$). La situazione ideale è rappresentata da $m_{\Delta t}$ pari al tasso d'emissione dei pacchetti e $\sigma_{\Delta t}$ pari a zero. Una corretta misura di tali valori deve tenere conto di quei fenomeni che possono alterare il significato delle informazioni. Ad esempio, la perdita di pacchetti potrebbe indurre a considerare l'$m_{\Delta t}$ tra due pacchetti ricevuti superiore al suo valore reale (uno sguardo al sequence number dell'intestazione RTP è sufficiente ad evitare l'errore). Lo stesso fenomeno del jitter, se eccessivo, può indurre ad errori. Infatti, supponendo che vengano spediti i pacchetti i, i+1 ed i+2 e che l'ordine d'arrivo sia i, i+2 ed i+1 con un $m_{\Delta t}$ pari a quello in trasmissione si potrebbe essere indotti a dedurre l'assenza di jitter (ancora una volta il sequence number associato ai vari pacchetti è sufficiente ad evitare l'errore). Questi concetti saranno ampiamente trattati nel cap. II e nel cap. IV.

I.2.c Perdita di pacchetti.

La perdita di pacchetti è tipica delle reti IP e le cause sono attribuibili a numerosi fattori. Gli errori di linea, l'eccessivo ritardo e la congestione della rete sono le cause più frequenti.

Gli errori di linea sono dovuti al fatto che la rete in se è non affidabile. Per questo, un mancato riscontro sul checksum[4] fa scartare un pacchetto.

L'eccessivo ritardo è legato alla congestione della rete; ad esempio, un pacchetto non può essere immediatamente instradato su una linea perché occupata. Il protocollo IP prevede nella sua intestazione un campo (*time-to-leave*) che viene decrementato ogni secondo, quando diventa zero il pacchetto viene distrutto. L'IP prevede 255 secondi di vita, ciò è impensabile per un'applicazione VoIP, infatti, vedremo che, il tempo di vita di un pacchetto vocale è associato ai tempi del jitter-buffer (es., 50ms). In sostanza, il pacchetto vocale è scartato dall'applicativo VoIP anche se l'IP lo tiene ancora in vita. Ciò va inquadrato nel principio secondo il quale per un pacchetto di voce la celerità è la cosa più importante.

La congestione della rete è l'aspetto più pericoloso perché spesso è causa di perdita di più pacchetti consecutivi. Quando, un router, trova tutte le linee d'uscita occupate, non può instradare pacchetti, quindi è costretto a mantenerli in memoria; siccome la capienza del buffer di ricezione dei router è limitata, quest'ultimo, può essere costretto a non accettare più pacchetti in ingresso perché non ha più posto dove metterli, quindi li perde.

La perdita di un pacchetto è percepita dall'ascoltatore come in "vuoto" e può essere causa di forte degradazione della voce. Tipicamente ogni pacchetto è composto da 20-30ms di segnale vocale, quindi la lunghezza di ogni pacchetto è simile a quella di un "fonema" vocale. La perdita di qualche pacchetto sparso incide poco sulla comprensione del parlato ed è avvertita, dall'ascoltatore, come un fenomeno di fastidio. Orientativamente, nell'arco di un'intera conversazione, se la perdita di pacchetti è superiore al 5% la comprensione del parlato comincia a diventare un problema. Nel caso in cui si perdano due o più pacchetti consecutivi (*burst of loss*), c'è il rischio concreto che il significato di una parola detta non sia percepito. Per contrastare questo fenomeno sono state proposte numerose soluzioni, in particolare in [16] s'illustra l'effetto del "processo di interleaving". In sintesi, si propone l'utilizzo di un sistema che distribuisca in maniera casuale gli errori dovuti alla perdita di più pacchetti. Ciò può essere ottenuto trasmettendo i pacchetti in modo non ordinato, così che la perdita di più pacchetti consecutivi si trasformi nella perdita di pacchetti isolati (vedi fig. I.2.c.1). Il problema legato a questa tecnica è l'incremento del ritardo fisso dovuto all'aggiunta dell'algoritmo di interleaving, soprattutto nel caso di compressione delle intestazioni RTP/UDP/IP.

[4] Il "checksum" è un numero, ottenuto con un certo algoritmo, dai dati trasmessi. Lo stesso numero deve essere riscontrato dai dati ricevuti, altrimenti il pacchetto è affetto da errori.

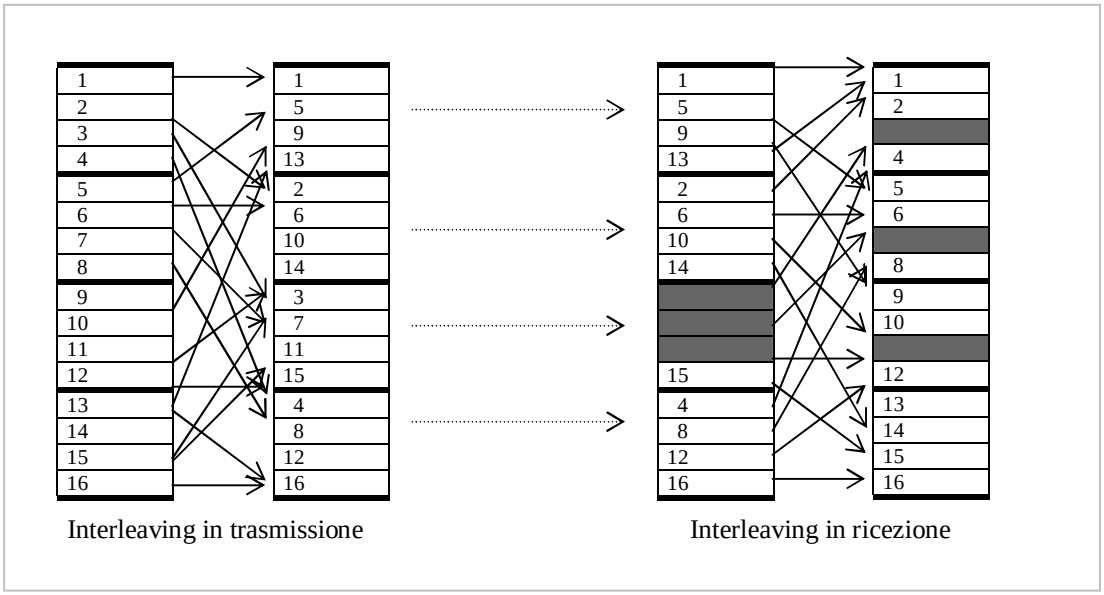


Fig. I.2.c.1 Processo di interleaving. Esempio di trasmissione di 16 pacchetti. I pacchetti in grigio sono quelli persi; lo stream ricostruito in ricezione non è affetto da "burst of loss".

La voce è molto più sensibile alla perdita di pacchetti che non agli errori di bit, quindi è conveniente continuare ad instradare un pacchetto con qualche bit corrotto piuttosto che scartarlo. Assolutamente da escludere è l'ipotesi di recuperare i bit originali ritrasmettendo il pacchetto, in quanto ciò darebbe luogo ad un ritardo inaccettabile e ad un jitter eccessivo; per questo motivo, come vedremo meglio nel seguito, si preferisce utilizzare il protocollo di trasporto UDP anziché il TCP (Trasmission Control Protocol). In [6] è messo in evidenza come sia opportuno affrontare il problema degli errori di bit puntando sulla robustezza del protocollo e dell'algoritmo di codifica. Per robustezza del protocollo nei confronti degli errori sui bit di voce s'intende il voler evitare che qualsiasi errore di bit possa causare lo scarto dell'intero pacchetto. Per questo motivo è auspicabile che il checksum sia dedotto solo da una parte del pacchetto piuttosto che dalla sua interezza.

E' opportuno che, soprattutto, l'algoritmo di codifica sia robusto nei confronti dei pacchetti persi. Ad esempio, è possibile che ogni trama sintetica[5], oltre a contenere le informazioni della trama da cui è stata ottenuta, contenga in sé anche qualche informazione sulle trame adiacenti. In questo modo il codificatore è in grado di far fronte ad un pacchetto perso attraverso l'informazione contenuta nel pacchetto adiacente. Questo metodo, noto come tecnica d'interpolazione, va pagato sia in termini di ritardo sia di banda. Per cui, ancora una volta, bisogna fare i conti con numerosi vincoli e decidere il giusto compromesso.

[5] Con il termine "trama sintetica" si intende un tratto di voce codificato.

I.3 Qualità della Voce (QoV).

Gli utenti spesso usano la qualità della voce come banco di prova per valutare l'intera qualità della rete, di conseguenza, un'adeguata QoV, è un'obbiettivo chiave d'ogni sistema di comunicazione. In queste note sarà preso in considerazione il problema del degrado della QoV in relazione a due fenomeni di grande interesse: *l'alto tasso di compressione* e *l'eco*.

Mantenere una buona qualità della voce e contenere il più possibile i ritardi e la quantità di banda necessaria è un problema che qualsiasi codificatore, progettato per applicazioni VoIP, deve affrontare. Solitamente un'alta compressione va sempre pagata in termini di QoV. I codificatori d'ultima generazione tendono a svincolarsi dal problema distinguendo la codifica dei periodi di silenzio dalla codifica della voce vera e propria. Il principio è molto semplice; mediante l'implementazione di un VAD (Voice Activity Detector), i periodi di silenzio vengono riconosciuti dal codificatore e compressi con pochi parametri fissi, i quali saranno riconosciuti dal decodificatore che li utilizzerà per generare un confortevole rumore di fondo (comfort noise). In alcuni casi si ha la possibilità di risparmiare fino al 50% di banda.

In tab. I.3.1 sono elencate le caratteristiche degli attuali standard di codifica presenti nella raccomandazione ITU (International Telecommunications Union).

Standard	**Bit rate**	**Lunghezza della trama**	**Qualità**	**Anno di finalizzazione**
G.711 PCM	64 kbit/sec	0.125 ms	Eccellente	1972
G.726, G.727 ADPCM	16, 40 kbit/sec	0.125 ms	Buona	1990
G.728 LD-CELP	16 kbit/sec	0.625 ms	Buona	1992, 1994
G.729 CS-ACELP	8 k bit/sec	15 ms	Buona	1995
G.723.1 MPC-MLQ	5.3 & 6.4 kbit/sec	37.5 ms	Discreta, Buona	1995
G.729 CS-ACELP Annex A	8 kbit/sec	15 ms	Buona	1996

Tab. I.2.1. Codificatori vocali standard dell'ITU.

Dalla tab. I.2.1 è evidente come il ritardo endogenerato (pari alla lunghezza della trama), il bit rate e la qualità della voce siano strettamente legati. Il G.711, il G.729 ed il G.729A saranno approfonditi nel cap. III.

Va da se che la scelta di un'alta compressione a scapito della qualità della voce non può prescindere dal tipo di rete. Ad esempio, il traffico presente nelle LAN, solitamente, non è tale da richiedere sistemi estremamente dedicati ad un uso efficiente della rete a dispetto della QoV (quasi sempre il G.711 va benissimo). In Internet il discorso è completamente diverso visto che la congestione della rete è sempre in agguato.

Un altro fenomeno che incide sulla qualità della voce è l'eco. Fondamentalmente si ha a che fare con due tipi di eco. Il primo, noto come *eco ibrido*, è dovuto alla conversione ibrida tra la connessione bifilare del telefono e la connessione quadrufilare in uscita dalla porta IP verso la rete d'Internet. In tal caso è comunemente adottata una cancellazione adattiva all'interno della porta IP [17]. Va da se che questo tipo di eco non è presente in un'architettura PC-to-PC. Il secondo, noto come *eco acustico*, nasce dal sistema, tipo viva voce, speaker microfono ed è tipico dell'architettura PC-to-PC (vedi fig. I.3).

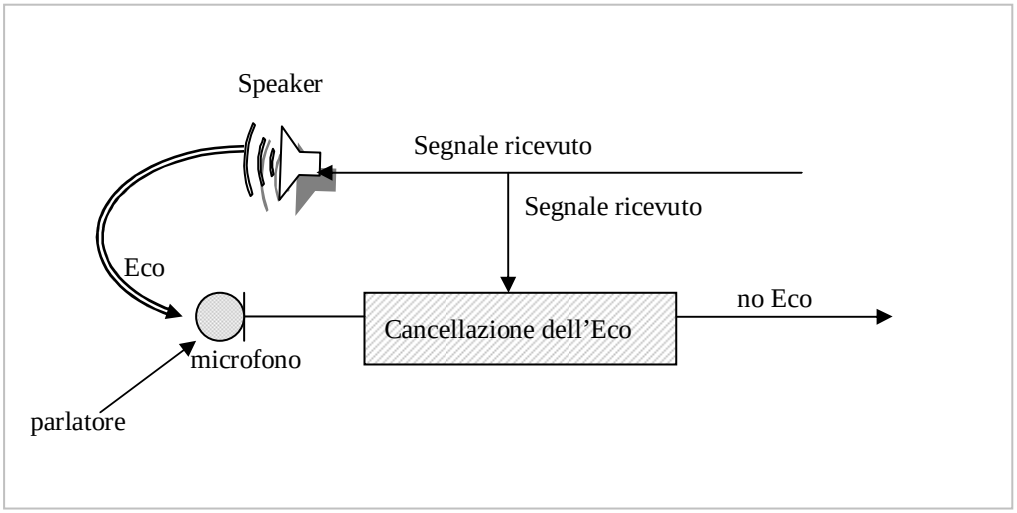


Fig. I.3. Processo di cancellazione dell'Eco acustico.

La voce che fuoriesce dall'altoparlante è catturata dal microfono e trasmessa in dietro (*acustic feedback*). In una situazione di questo tipo il microfono può ricevere voce da molte vie di riflessione (*multipath echo*). Quello che, solitamente, accade è un ritorno della propria voce.

La cancellazione di questo tipo di eco può essere implementata nel software applicativo, a parte o nel codificatore, ed in ogni caso, aggiunge ulteriori ritardi alla trasmissione. In fig. I.3 è schematizzato un metodo di cancellazione adattiva. Il cancellatore d'eco si adatta, di volta in volta, al tipo di eco previsto in seguito al segnale acustico che ha in ingresso. Va da se che questo tipo di eco coinvolge soltanto i sistemi full-duplex.

I.4. Qualità della comunicazione: dove siamo.

Gli utenti di VoIP richiedono una buona qualità del servizio, la quale è principalmente dipendente dalle condizioni istantanee di carico della rete IP. Infatti, la sfida da affrontare è soprattutto quella di arginare i problemi legati alla variabilità di risorse all'interno di Internet. Potrebbe sembrare paradossale, ma è proprio la crescente popolarità di Internet a giocare a sfavore della QoS. Infatti, il sempre più crescente utilizzo di Internet limita la disponibilità di banda aumentando il rischio di congestione della rete. A sua volta, la congestione da luogo a ritardi nell'instradamento dei pacchetti. A causa di questi ritardi i pacchetti possono essere persi o scartati. Quelli che non vengono persi enfatizzano, inevitabilmente, il jitter di rete.

Nei precedenti paragrafi, spesso, si è messo in evidenza la molteplicità dei vincoli da soddisfare per garantire una buona qualità nella comunicazione (QoS & QoV). Fondamentalmente, i vincoli a cui si fa riferimento sono i seguenti: *banda*, *ritardo* e *complessità computazionale*. Ognuno di questi vincoli contrasta col soddisfacimento di un altro, tanto che spesso, in letteratura, ricorre la suggestiva fig. I.4.1.

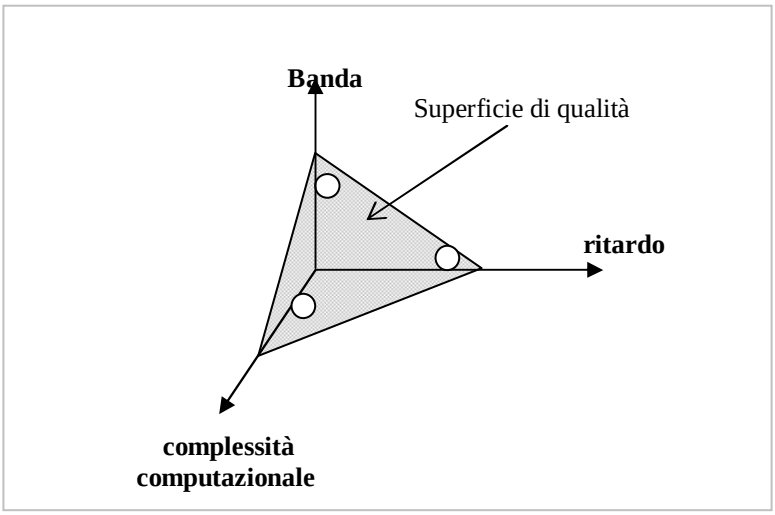


Fig. I.4.1. Qualità della comunicazione.

Ogni soluzione di comunicazione vocale può essere mappata sulla superficie di qualità mostrata in figura. Il punto più alto, sull'asse della banda, corrisponde alla telefonia tradizionale, la quale richiede una grossa quantità di banda (64 kbit/sec), ma una bassissima complessità computazionale, e garantisce un ritardo impercettibile. Il punto vicino all'asse del ritardo corrisponde alle attuali proposte VoIP rivolte ai singoli utenti, i quali, dispongono di una bassa quantità di banda e di una potenza di calcolo modesta; di conseguenza, le soluzioni VoIP per gli utenti, dovendo adeguarsi a tale situazione, devono necessariamente essere a bassa complessità computazionale, il ché, sommato alla scarsità di banda, risulta in alti ritardi.

Le soluzioni VoIP per le grandi compagnie sono tutt'altra cosa. Numerose aziende, ad esempio la Cisco System, la Lucent, ed altre, continuano ad investire nel settore ed a proporre reti IP multiservizio complete ed affidabili. Queste applicazioni VoIP vanno messe nel tratto ad alta complessità computazionale (DSP dedicati), ritardo moderato e bassa richiesta di banda (circa 5.3 - 8 kbit/sec, per la voce). Ciò comporta un'accettabile qualità del servizio ma a costi elevati, per questo, le proposte sono, per ora, rivolte alle grosse aziende.

Osservando la fig. I.4.1 potrebbe sembrare impossibile svincolarsi dal piano sul quale il raggiungimento di una specifica va pagato in termini dell'altra.

Ma in realtà i problemi presenti nella tecnologia VoIP, sostanzialmente legati alla qualità della comunicazione, non appaiono affatto insuperabili, non a caso la situazione va sempre migliorando.

Capitolo II
VoIP dal punto di vista protocollare.

La trasmissione VoIP avviene attraverso reti organizzate secondo una serie di strati o livelli, ciascuno progettato sopra il suo predecessore. Le modalità di comunicazione tra macchine connesse ad una rete IP multiservizio sono il cuore delle trasmissioni VoIP. In questa sede saranno trattati i protocolli attualmente utilizzati nelle applicazioni A/V (Audio/Video). Alcuni di essi sono già in uso mentre altri stanno per essere dispiegati all'interno di Internet [18][19].

Il capitolo è organizzato seguendo grossomodo il principio "a livelli" delle reti IP in modo da analizzare puntualmente le caratteristiche d'ogni strato. Nel par. II.1 verrà presa in considerazione l'architettura a strati delle reti IP inquadrando, dalla cosiddetta "suite TCP/IP", la pila protocollare che interessa le applicazioni VoIP: la "suite UDP/IP". Nel par II.2 si faranno alcuni richiami sui protocolli facenti parte dello *strato di rete* (l'IPv4 e l'IPv6). Nel par II.3 lo *strato di trasporto* sarà discusso confrontando le proprietà dei suoi due protocolli caratteristici: l'UDP (User Datagram Protocol) ed il TCP (Trasmission Control Protocol). Il par. II.4 verranno presi in considerazione gli aspetti relativi al multicasting in ambiente IP. Il par. II.5 verrà dedicato allo *strato di applicazione*, ove verranno discusse le caratteristiche di due protocolli emergenti: l'RSVP e l'RTP/RTCP. Infine, nel par. II.6, il protocollo RTP/RTCP sarà commentato in relazione al suo indiscusso utilizzo nelle applicazioni VoIP.

II.1 Organizzazione “a livelli” delle reti IP.

Volendo descrivere la comunicazione di voce su IP dal punto di vista protocollare è inevitabile fare riferimento al modello OSI (Open System Interconnection). Nonostante quasi nessuna rete ne rispetta a pieno le funzionalità, i fondamenti teorici del modello OSI sono, comunque, presenti in ogni rete a strati.

Il modello OSI fu proposto dall’ISO (International Standard Organizzation) ed in letteratura esistono numerose fonti che lo descrivono, in particolare in [4] è disponibile una buona introduzione ed un confronto costante con alcune reali architetture di rete. In fig. II.1.1 sono schematicamente mostrati i sette strati dell’OSI.

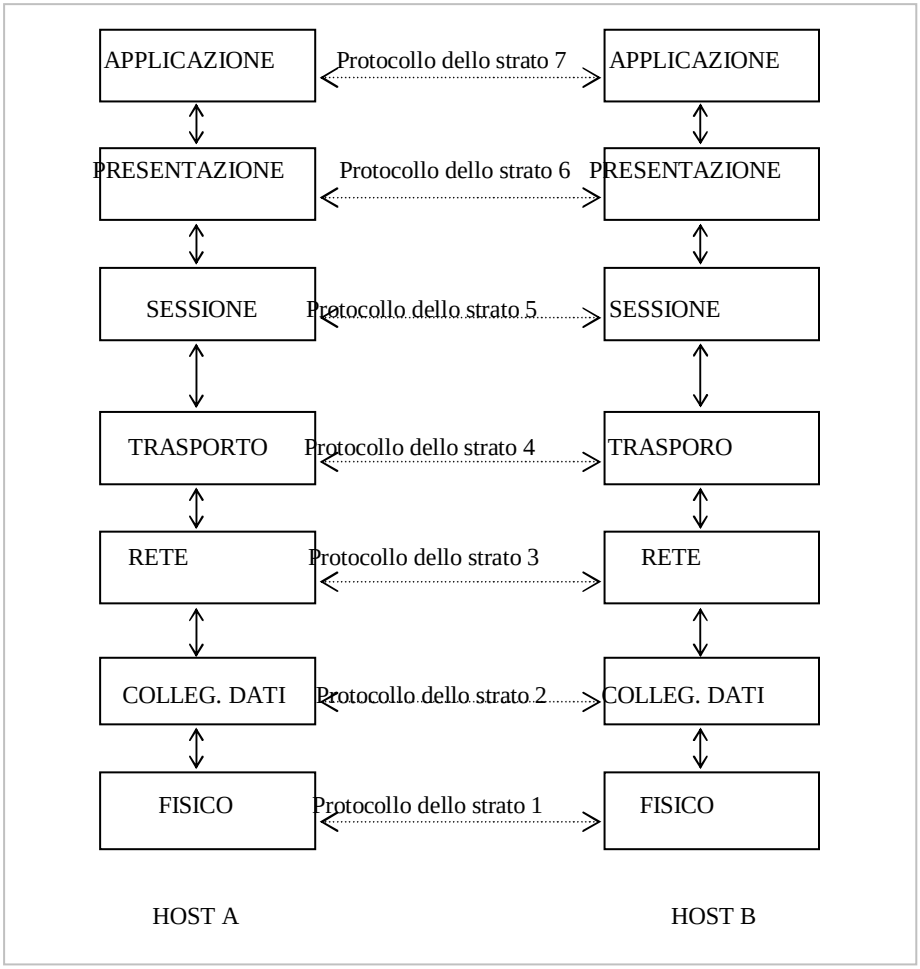


Fig. II.1.1 Comunicazione tra due host (due calcolatori in remoto) secondo le modalità OSI. Le frecce tratteggiate indicano la comunicazione virtuale mentre, le frecce continue indicano il reale percorso dei dati.

La comunicazione avviene attraverso i vari livelli, ognuno dei quali è progettato sopra il suo predecessore. Ogni livello comunica con il suo “paritario” attraverso una serie di regole (*protocolli*) progettate ad hoc. I dati, in realtà, fluiscono da uno strato all’altro all’interno dello stesso host. Il principio su cui si fonda il modello è quello dei

servizi, un insieme d'operazioni che ogni strato fornisce al soprastante. Ogni strato è preparato a svolgere tali operazioni per conto di quelli superiori, che ne richiedono il servizio. Il modo in cui sono implementate tali operazioni non è noto agli strati superiori, così che ogni strato sia svincolato dal dipendere dalla tecnologia dello strato sottostante. In questo modo, l'evolversi della tecnologia relativa ad uno strato non incide sull'organizzazione del sovrastante. Questo è il principio su cui sono progettate tette le reti di computer realmente esistenti.

L'interconnessione di più reti diverse, ad esempio Internet, necessita d'interfacce che ne garantiscano l'interoperabilità.

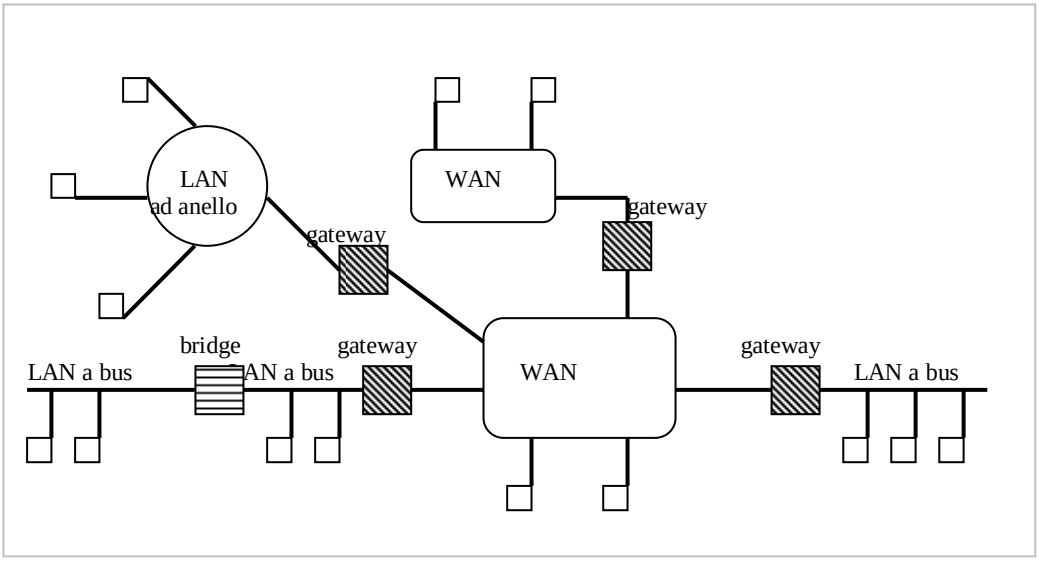


Fig. II.1.2 Interconnessione di più reti.

In fig. II.1.2 sono schematizzate più reti interconnesse. Ogni tipo di rete è connessa ad un'altra attraverso una macchina che supporta alcuni dei sette strati visti in precedenza. In particolare si ha:

- I *ripetitori* (non presenti in figura) supportano il solo strato 1. Si tratta d'elementi d'amplificazione dei segnali elettrici.

- I *bridge* (ponti) supportano gli strati 1 e 2. Il loro compito è quello di inoltrare l'intero pacchetto tra LAN che si differenziano fino allo strato di collegamento dati.

- I *gateway* (porte) supportano gli strati 1, 2 e 3. Il loro compito è di far operare reti che si differenziano fino al terzo strato. Talvolta in letteratura si usa una convenzione diversa usando il termine "router" al posto di gateway.

- I *convertitori di protocollo* (non presenti in figura) provvedono all'interfacciamento degli strati superiori al terzo.

E’ necessario precisare che in letteratura si possono trovare definizioni diverse, perciò, la precedente terminologia, va vista come semplice elemento di chiarezza sulle funzionalità supportate dai nodi di collegamento tra reti di calcolatori.

Spesso in letteratura s’incontra l’acronimo TCP/IP (Trasmission Control Protocol / Internet Protocol), con il quale si sott’intende un set di protocolli sviluppati per permettere la cooperazione e la condivisione di risorse tra computer connessi ad Internet. In realtà, TCP ed IP sono solo due dei protocolli appartenenti a tale set, ma, poichè il TCP e l’IP sono i protocolli di Internet più conosciuti, è uso comune indicare con la sigla TCP/IP l’intera famiglia di protocolli (**Internet Protocols Suite** [20]). Inoltre, in letteratura, non è raro leggere TCP/IP anche se ci si riferisce ad applicazioni VoIP, per le quali, in realtà, si fa uso del protocollo di trasporto UDP. Detto ciò, nel seguito, con l’acronimo **UDP/IP** ci si riferirà alla “suite” protocollare di Internet rivolta ad applicazioni di voce su IP, come schematicamente mostrato in fig II.1.3[6]. La figura è altamente strutturata e da l’idea di quanto oneroso e complesso sia il lavoro delle reti IP multiservizio.

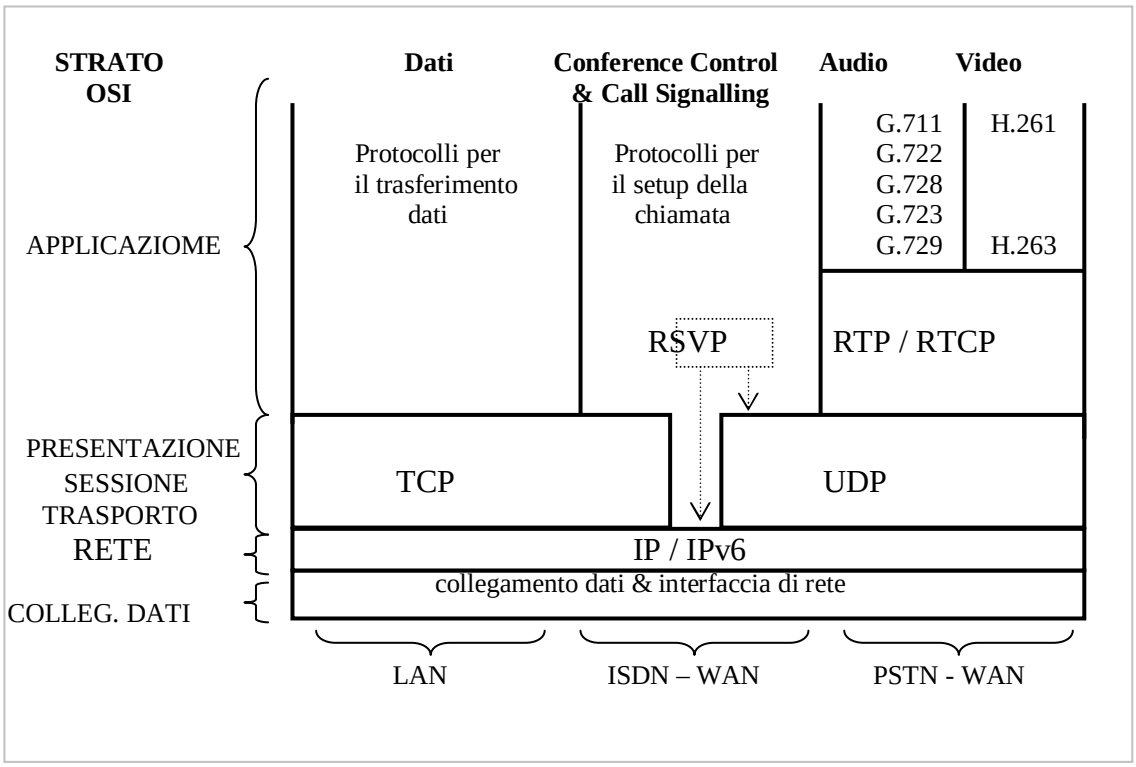


Fig. II.1.3. Suite UDP/IP in relazione al modello OSI.

Il protocollo di rete IP svolge più o meno le stesse funzioni del protocollo di rete dell’OSI, anche se l’IP è stato progettato per gestire l’interconnessione dell’enorme numero di reti WAN e LAN che compongono Internet. Al contrario, l’OSI quasi non prende in considerazione l’interconnessione di reti.

[6] Le sigle riportate in figura, di cui non si è fatta ancora menzione, saranno chiarite in seguito.

Lo strato di collegamento dati e quello fisico dipendono dal particolare tipo di rete verso cui ci s'interfaccia.

E' interessante notare come, lo strato d'applicazione sia di per sé esplicativo delle proprietà d'integrazione dei flussi eterogenei (dati, voce e video) che riguardano le comunicazioni VoIP. Le varie applicazioni si proiettano tutte verso un unico protocollo, l'IP, il quale ha l'arduo compito di fornire agli strati superiori una molteplicità di servizi.

II.2 I protocolli dello strato di rete nelle comunicazioni VoIP.

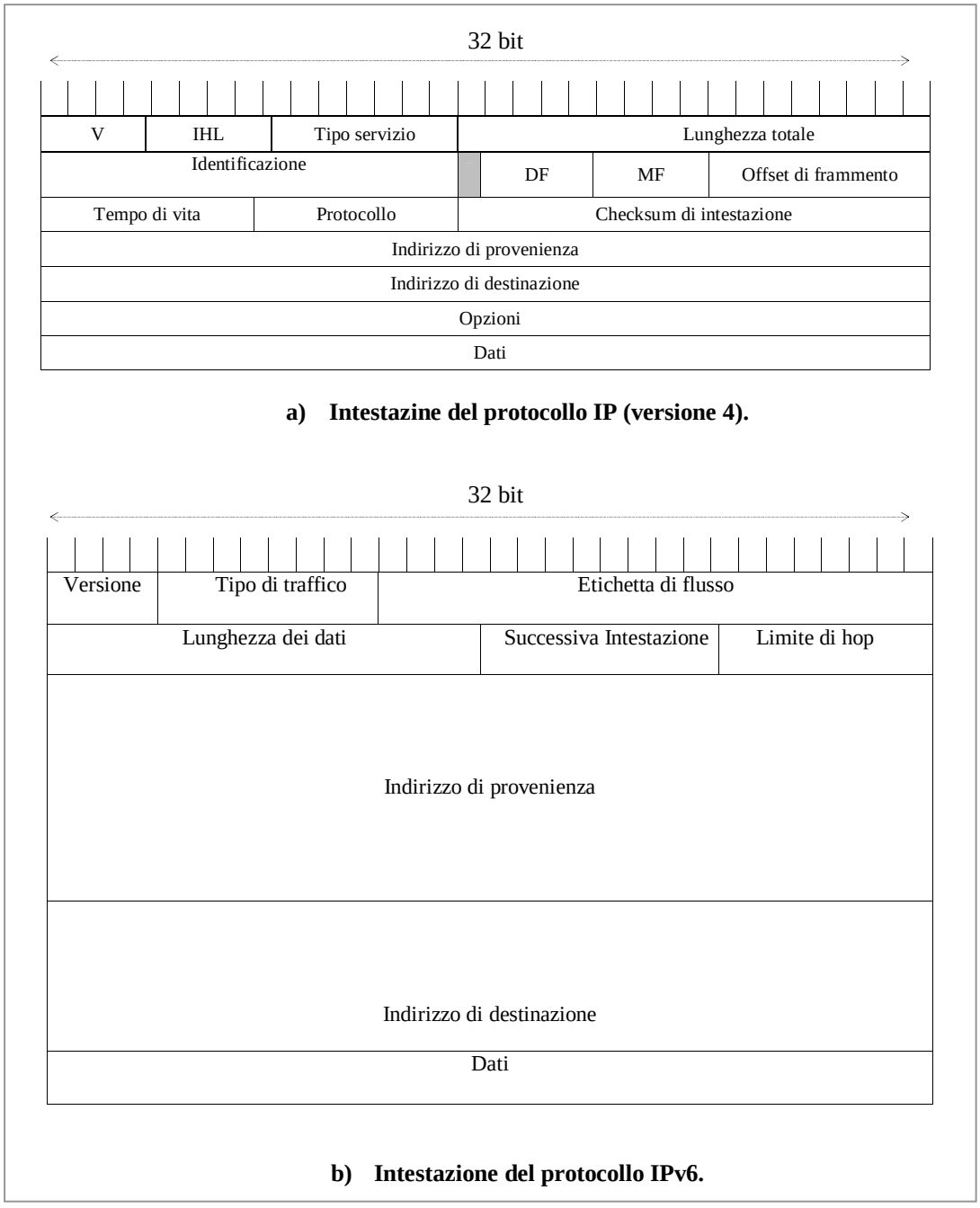
Lo strato di rete di Internet è gestito dal protocollo IP, il quale si occupa del dirottamento dei datagrammi[7] per tutto il cammino, dalla provenienza alla destinazione. Questo compito potrebbe apparire banale ma è veramente laborioso. Nel cap. I, si è già avuto modo di notare come l'instradamento dei pacchetti lungo la rete deve sempre fare i conti con la congestione di quest'ultima. Inoltre, l'IP nel gestire il controllo della congestione deve tenere conto dell'interconnessione di più reti. Il servizio che l'IP deve fornire allo strato di trasporto deve essere indipendente dalla tecnologia della particolare sottorete; inoltre, deve essere tale da far sì che lo strato di trasporto sia mantenuto in uno stato d'inconsapevolezza del numero, del tipo e della tecnologia delle sottoreti presenti.

Il protocollo IP dalla sua nascita ha subito numerose revisioni e modifiche rivolte ad adattarlo alla crescente struttura di Internet. Una caratteristica che si è mantenuta sempre costante è la sua natura *connectionless* (senza connessione), il che lo rende inaffidabile. La caratteristica connectionless è stata voluta dalla comunità dell'ARPA (Advanced Research Projects Agency) Internet in quanto la rete è intrinsecamente inaffidabile, pertanto il controllo degli errori e del flusso di dati deve essere a carico dell'host. A proposito di ciò, durante la standardizzazione dell'OSI, si ebbe una storica diatriba, sui servizi che lo strato di rete doveva offrire, tra i sostenitori del servizio rivolto alla connessione e quelli del servizio rivolto alla non connessione; la questione si risolse accettando entrambe le possibilità. Quest'argomento merita particolare attenzione giacché permette di precisare una fondamentale differenza tra le reti IP e la rete telefonica pubblica.

La rete pubblica offre un servizio orientato alla connessione; infatti, l'utente, facendo il numero, stabilisce una connessione con il numero chiamato poi parla ed infine chiude la connessione agganciando la cornetta. Nelle reti IP la cosa è completamente diversa, poiché è l'utente (che in questo caso è lo strato di trasporto) a doversi preoccupare di verificare che tutti i pacchetti spediti siano giunti a destinazione ed, eventualmente, nell'ordine desiderato. Il lavoro di verifica è esplicato dallo strato di trasporto attraverso il protocollo TCP, ma ciò, come vedremo nel par. II.3, richiede un prolungamento dei tempi di trasmissione. Per molte applicazioni tempo reale la trasmissione veloce è più importante che quella fedele, per questo lo strato di trasporto non deve preoccuparsi dell'affidabilità in modo da non rallentare, inutilmente, le operazioni di consegna (in questo caso si utilizza l'UDP). Questi concetti saranno ulteriormente approfonditi nel prossimo paragrafo. Per il momento concludiamo affermando che l'IP non si preoccupa dell'affidabilità, che è a discrezione e carico dello strato di trasporto.

[7] Ai termini "datagramma" e "pacchetto" è spesso dato lo stesso significato ma, in effetti, tra loro c'è una certa differenza. Tecnicamente, datagrammi è la giusta parola da usare quando si parla di protocolli UDP ed IP; poiché un datagrammi è un insieme di dati rappresentanti il dialogo di questi protocolli, mentre un pacchetto è un qualcosa di fisico e comincia ad avere significato a partire dal livello di collegamento dati. In molti casi un pacchetto contiene semplicemente un datagrammi quindi la loro differenza è semplicemente cosa da puristi.

Nella fig. II.2.1 sono riportate le intestazioni dei protocolli IP ed IPv6. L'IPv6 è la versione 6 ed il suo stato attuale è di standard proposto, mentre con l'acronimo IP s'intende la versione 4 (standard attuale). La decisione di sviluppare una nuova versione dell'IP è stata dettata dal sempre più crescente numero di nuove applicazioni multimediali, tra cui VoIP. L'IPv6 offre agli strati superiori nuovi servizi tendenti a migliorare la QoS e la sicurezza. Il nuovo formato degli indirizzi non solo risolve il problema della disponibilità di spazio (128-bit anziché 32-bit) ma supporta, anche, una configurazione automatica del sistema. Molte delle opzioni incorporate nell'IPv4 e non integrate in tutte le implementazioni sono, invece, pienamente integrate nell'IPv6.



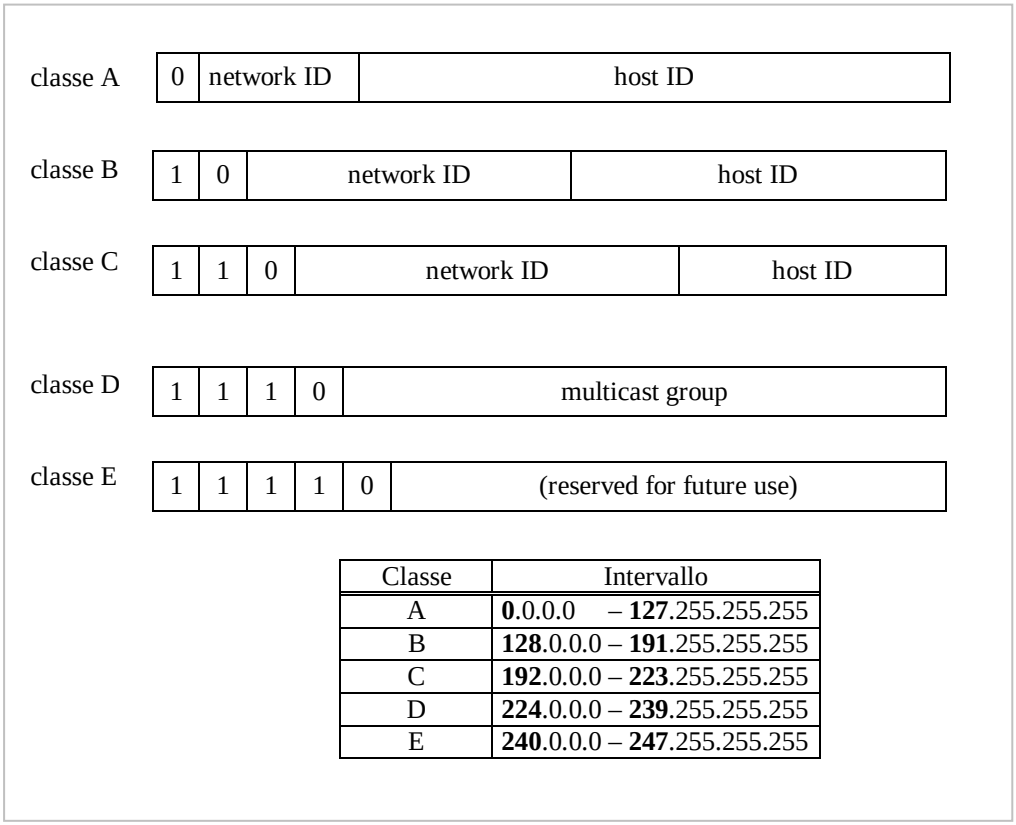
a) Intestazine del protocollo IP (versione 4).

b) Intestazione del protocollo IPv6.

Fig. II.2.1. Intestazione del protocollo IP, **a)** versione 4 (standard) **b)** versione 6 (standard proposto).

Per non perdere di vista il fine di queste note, si è voluto evitare di dare una carrellata descrittiva dei vari campi. La scelta è quella di descrivere le funzionalità dell'IP attraverso riferimenti pratici integrati con l'applicazione VoIP, per questo nel corso della trattazione non mancheranno numerosi richiami alla fig. II.2. Per approfondimenti sulle scelte ed il significato dei vari campi si rimanda alle classiche fonti bibliografiche [21][22].

Prima di concludere questo paragrafo è opportuno considerare il formato d'indirizzamento dell'IPv4 poiché ritornerà utile nel seguito. Ogni indirizzo IPv4 (32-bit) è inquadrabile in uno dei formati rappresentati in fig. II.2.2.



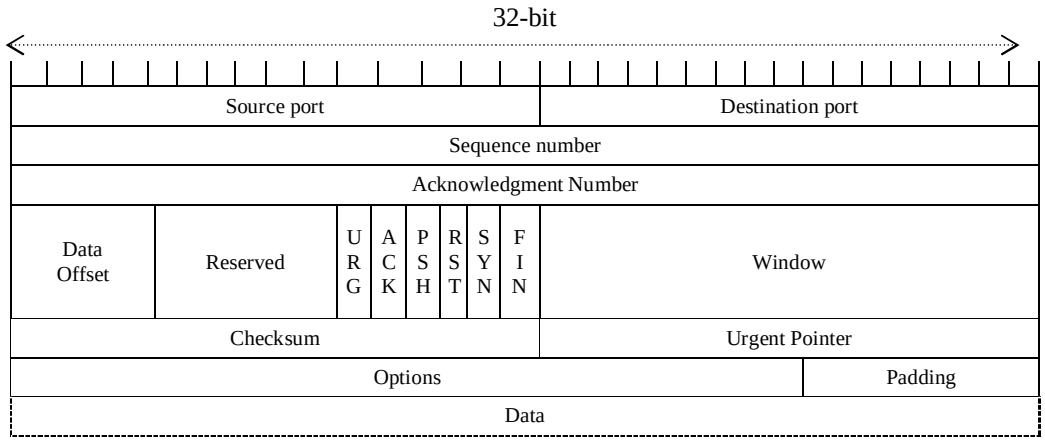
Classe	Intervallo
A	**0**.0.0.0 – **127**.255.255.255
B	**128**.0.0.0 – **191**.255.255.255
C	**192**.0.0.0 – **223**.255.255.255
D	**224**.0.0.0 – **239**.255.255.255
E	**240**.0.0.0 – **247**.255.255.255

Fig. II.2.2. Formati dell'indirizzo IPv4.

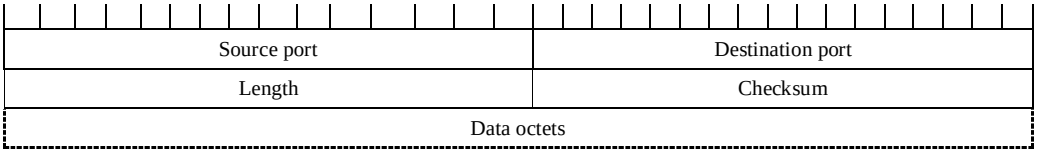
Solitamente gli indirizzi IP sono rappresentati da quattro numeri separati da punti (*dotted-decimal notation*), ognuno di essi rappresenta uno dei quattro byte dell'indirizzo a 32-bit. Un'altra notazione comunemente adottata è la *host-name notation* (pcvoce2.ing.uniroma1.it), simile alla precedente ma al posto dei numeri vi sono dei termini, oppure *simple-hostname* (pcvoce2).

II.3 I protocolli dello strato di trasporto nelle comunicazioni VoIP.

I protocolli dello strato di trasporto nell'ARPANET sono il TCP (Trasmission Control Protocol) e l'UDP (User Datagram Protocol). Entrambi permettono la comunicazione tra due host (*end system*) remoti, ma secondo modalità estremamente differenti. In fig. II.3.1 sono mostrati i formati di entrambi i protocolli. Per avere piena consapevolezza nelle scelte di progetto, relative ad un'applicazione VoIP, è necessario comprendere quali sono i servizi offerti dai due protocolli agli strati d'applicazione.



a) Formato dell'intestazione TCP.



b) Formato dell'intestazione UDP.

Fig. II.3.1. Formati dell'intestazione dei protocolli: a) TCP, b) UDP.

L'UDP è definito in [23], la sua struttura è estremamente sottile (aggiunge appena 8 byte d'incapsulamento). Attraverso l'UDP è possibile inviare datagrammi senza stabilire alcuna connessione (*connectionless*) e senza nessuna garanzia di consegna o di rispetto della sequenzialità. L'UDP si appoggia sia sull'IPv4 sia sull'IPv6.

Al contrario il TCP, definito in [24], è orientato alla connessione (*connection-oriented*) e fornisce un servizio affidabile. Il TCP si prende cura di numerosi dettagli

quali l'acknowledgments, il time-out, la ritrasmissione e quant'altro. Anche il TCP, come l'UDP, gira sia sull'IPv4 sia sull'IPv6.

Quello che, in linea di massima, avviene utilizzando il TCP è la seguente cosa. Un host (*client*) stabilisce una **connessione** con un altro host (*server*), scambia i dati con esso attraverso la connessione, e successivamente termina la connessione. Il TCP garantisce anche l'**affidabilità** della comunicazione poiché esegue una serie di controlli sui dati trasmessi attraverso la connessione. Quando i dati vengono mandati attraverso il TCP, esso richiede un'approvazione (*acknowledgement*) dall'host ricevente. Se non riceve l'acknowledgement, il TCP automaticamente, senza alcun intervento dallo strato d'applicazione, ritrasmette i dati ed aspetta per più tempo. Dopo un certo numero di ritrasmissioni, il TCP rinuncia. Le ritrasmissioni avvengono finché non si supera un certo limite di tempo (*timeout*) che va dai 4 ai 10 minuti a seconda delle implementazioni. In ogni caso il TCP è supportato da algoritmi che stimano dinamicamente l'RTT[8] (Round Trip-Time) tra i due host remoti, in questo modo può sapere quanto tempo deve aspettare per l'acknowledgement. Solitamente l'RTT in una LAN è dell'ordine dei millisecondi, mentre in una WAN può essere di qualche secondo. Inoltre, non è da escludere che, a causa delle improvvise variazioni del carico della rete, il TCP possa misurare sulla stessa connessione un RTT di 1 secondo, poi di 20 secondi e poi di 4 secondi.

Il TCP, inoltre, attraverso il campo *sequence number* (vedi fig. II.3.1a), è in grado di garantire la sequenzialità dei dati trasmessi. Ad esempio, se l'applicazione manda 4096 byte ad un host remoto, il TCP manda 4 segmenti[9]: il primo contenete i sequence number 1÷1024 (uno per ogni byte spedito), il secondo con i sequence number 1025÷2048, il terzo con i sequence number 2045÷3069, ed infine il quarto con i sequence number 3070÷4096. In ricezione un'eventuale jitter è equalization in base ai sequence number, ed i dati sono forniti all'applicazione nel loro giusto ordine. Può anche accadere che un pacchetto venga duplicato (es., l'host trasmittente non ricevendo l'acknowledgement entro l'RTT, ritrasmette il pacchetto pensando che sia perso, mentre è solo eccessivamente ritardato a causa di un overload della rete), in tal caso il TCP ricevente se n'accorge attraverso il sequence number e lo scarta.

Questo complesso meccanismo di controllo non è fornito dall'UDP. L'UDP non fornisce nessuna funzionalità di acknowledgement, di sequenziazione dei byte, di stima dell'RTT, di timeout o di ritrasmissione. Inoltre, l'UDP non si preoccupa per nulla d'eventuali datagrammi duplicati. Tutto ciò, se serve, è a carico dell'applicazione.

La connessione stabilita dal TCP è anche *full-duplex*, ovvero un'applicazione può contemporaneamente spedire e ricevere dati su una stessa connessione; anche l'UDP può essere full-duplex.

Una caratteristica molto importante del TCP, che lo ha reso meritevole dell'aggettivo affidabile, è il controllo di flusso (*flow control*). Il TCP in ricezione dice sempre al suo paritario quanti byte è in grado di accettare. Ciò è noto come finestra (*window*) d'avvertimento. Di volta in volta, la finestra (vedi fig. II.3.1.a) è pari

[8] Il Round Trip-Time è il tempo che un dato impiega durante il tragitto nella rete, dalla trasmissione alla ricezione.
[9] Per segmento s'intende l'unità di dato che il TCP passa all'IP.

all’ammontare delle stanze attualmente disponibili nel buffer di ricezione del SO (Sistema Operativo). In questo modo è garantito che l’host trasmittente non potrà mai mandare in overflow il buffer dell’host ricevente. La dimensione della finestra varia dinamicamente: all’aumentare dei dati ricevuti dal paritario trasmittente la finestra decresce, ma, nello stesso tempo, all’aumentare dei dati letti dall’applicazione la finestra cresce. Nel caso in cui la dimensione della finestra è zero il buffer di ricezione è pieno, dunque è necessario che il trasmittente attenda finché l’applicazione del ricevente legga dati dal buffer in modo che si liberino posti.

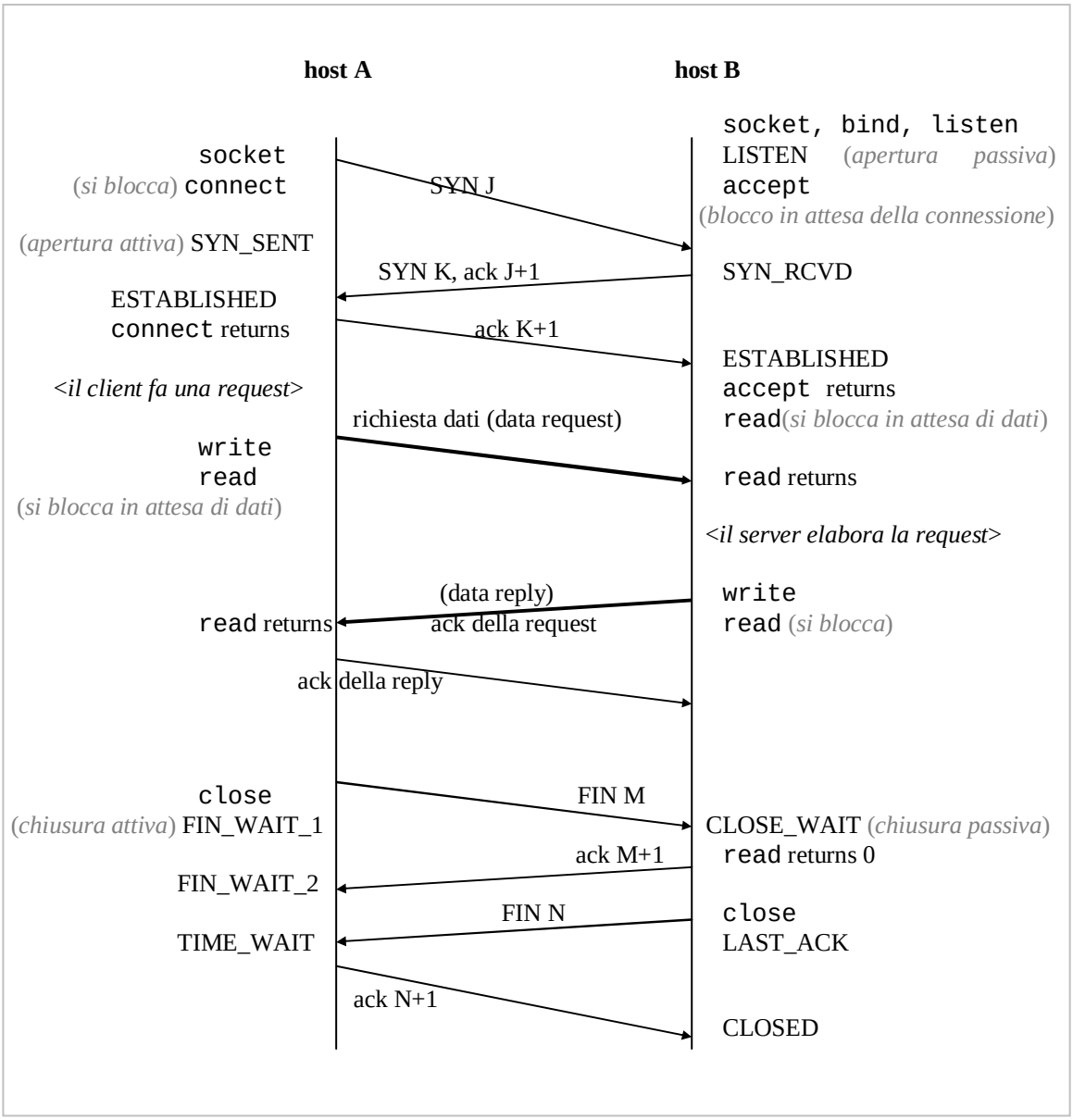


Fig. II.3.2. Scambio di pacchetti in una connessione TCP.

Il controllo sul flusso fornito dal TCP garantisce che mai nessun pacchetto sarà perso per problemi di overflow del buffer in ricezione.

L’UDP non fornisce alcun servizio di flow control, per questo non è raro che un host trasmetta datagrammi ad un tasso superiore a quello con il quale l’host ricevente

riesce a smaltirli. Ciò, spesso, è causa di *burst of loss*, il ché inficia notevolmente la comunicazione vocale. Infatti, è assolutamente sconsigliato mandare un flusso continuo di dati (ad esempio un trasferimento di file) attraverso l'UDP.

In fig. II.3.2 è mostrato un tipico scambio di messaggi tra due processi paritari in una connessione TCP completa: set-up della connessione, trasferimento dati e chiusura della connessione.

In fig. II.3.2 sono presenti le varie chiamate di sistema utilizzate per lo scambio di pacchetti tra due host che comunicano attraverso il TCP. Il linguaggio di programmazione cui si fa riferimento è l'ANSI C. Numerosi dettagli in merito sono disponibili in [41]. Per il momento ci riferiremo ad esse intendendole come uno strumento attraverso il quale è possibile approfondire le modalità di comunicazione tra due paritari TCP.

Per stabilire la connessione con il TCP sono necessari almeno tre pacchetti, noti come *three-way handshake* del TCP, identificabili in fig II.3.2 nei tre segmenti: SYN J, SYN K & ack J+1, ack K+1. Inizialmente l'host B si prepara ad accettare connessioni in arrivo (socket, bind, listen, accept) e si blocca nello stato di "apertura passiva". L'host A lancia una "connessione attiva" (connect) attraverso l'invio di un segmento di sincronizzazione (SYN) attraverso il quale informa l'host B sul sequence number iniziale (J) dei dati che intende mandare sulla connessione. L'host B si preoccupa di inviare l'acknowledgement (ack J+1) sul pacchetto ricevuto ed il proprio segmento di sincronizzazione (SYN K). L'host A invia l'acknowledgement (ack K+1) sulla sincronia inviatagli dall'host B. A questo punto, entrambi gli host sono pronti a scambiarsi i dati: request e reply.

Questa modalità di connessione è molto simile a quella del tradizionale sistema telefonico.

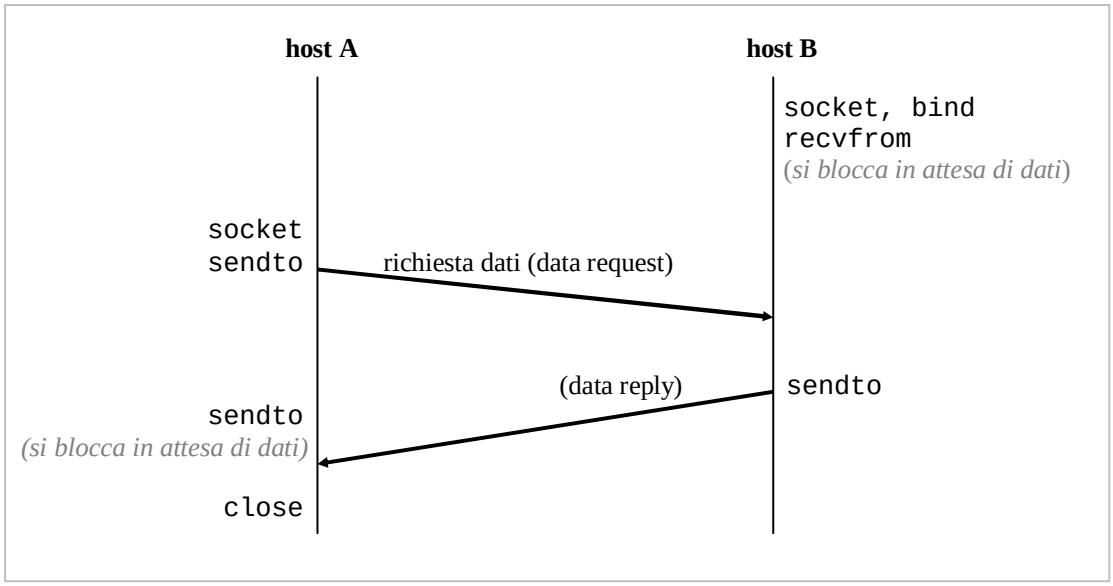


Fig. II.3.3. Trasmissione di pacchetti attraverso l'UDP.

I segmenti necessari per terminare la connessione sono quattro: FIN M, ack M+1, FIN N, ack N+1. L'host A attraverso l'invio del segmento FIN M comunica all'host B di aver terminato (close) l'invio di dati. L'host A non trasmetterà più dati quindi si dice che è nello stato di "chiusura attiva". La ricezione del segmento FIN pone l'host B nello stato di "chiusura passiva", in pratica sa che non riceverà dati ma può ancora trasmetterne. Dopodiché l'host B manda l'acnowledgement (ack M+1) del FIN ricevuto. Lo stato di chiusura passiva informa l'applicazione con una sorta di end-of-file. L'applicazione, non avendo più nulla da elaborare, chiude la connessione (close), quindi l'host B invia un proprio FIN (N) attende l'acnowledgement (ack N+1) su tale FIN e, dopo averlo ricevuto termina la connessione (CLOSE).

E' evidente come, stabilire una connessione tra due host remoti richieda vari messaggi di controllo ai quali vanno aggiunti tutti quelli necessari alla garanzia dell'affidabilità. Nel caso dell'UDP, invece, la trasmissione è priva di ogni controllo e non viene fatto alcun set-up. In effetti, l'UDP è una semplice interfaccia d'utente all'IP. Nella fig. II.3.3 sono riassunte le modalità comunicative tra due macchine attraverso il protocollo UDP.

II.3.a Trasporto dei dati in un'applicazione VoIP.

Dopo aver discusso le principali differenze tra i servizi offerti dal TCP e quelli offerti dall'UDP, è opportuno chiedersi in quali casi convenga utilizzare l'UDP, nonostante non sia affidabile, ed in quali altri il TCP. In parte la risposta è stata già data, ed in questo paragrafo si vogliono puntualizzare alcuni concetti che torneranno utili in seguito.

Si è già detto che il trasferimento di dati audio e video necessita di una certa celerità, quindi il servizio di controllo estremamente meticoloso e complesso offerto dal protocollo TCP non è adatto al trasporto di questi dati. Infatti, tutte le procedure di controllo (quali la ritrasmissione dei pacchetti persi, l'attesa per l'overload e quant'altro) rendono il servizio troppo lento per le applicazioni VoIP. Invece, il protocollo di trasporto UDP offre un servizio tanto semplice quanto veloce. Per cui, tra i due, il protocollo di trasporto che meglio si adatta alle necessità di velocità imposte da VoIP è proprio il protocollo UDP. Diciamo subito che quanto detto vale per il trasporto dei dati, tutt'altra cosa è il trasporto dei segnali di controllo (di questo ci occuperemo nel seguito). Naturalmente l'UDP è inaffidabile, ma in una comunicazione tempo-reale è preferibile tollerare qualche pacchetto perso piuttosto che lunghi ritardi. Il discorso è che un pacchetto in eccessivo ritardo va in ogni caso considerato perso; infatti, non ha senso aspettare qualche secondo per completare un fonema con i suoi ultimi 20ms, giacché sarebbe lo stesso incomprensibile. Confrontando la fig. II.3.2 con la fig. II.3.3 è evidente come, mentre l'UDP necessita di due soli pacchetti per gli scambi di request e reply, il TCP richiede all'incirca 10 pacchetti (assumendo che sia necessario stabilire una nuova connessione per ogni scambio di request-reply). Oltre a ciò vanno considerati i tempi di RTT. E' possibile osservare che il minimo tempo di transizione necessaria ad uno scambio request-reply con l'UDP è pari a RTT+STP, ove con l'acronimo STP

s'intende il "server processing time" per la request. Con il TCP, invece, il valore diventa 2xRTT+STP. Del resto va notato che, nel caso si utilizzi il TCP, anche un singolo pacchetto perso potrebbe rallentare drasticamente la trasmissione.

Un ulteriore piccolo svantaggio del TCP sta nella lunghezza dell'intestazione: 40 byte contro gli 8 byte dell'UDP. L'utilizzo del solo TCP senza l'ausilio dell'RTP non basterebbe, a meno di una lunga bufferizzazione dei dati, in quanto privo di alcune informazioni necessarie (timestamp ed informazioni sulla codifica, di cui si parlerà nel par. II.5.b.

L'aspetto della velocità nel trasferimento dati e della sovraintestazione sono quelli che in letteratura vengono sempre messi in evidenza, ma ci sono altre caratteristiche di notevole importanza, che spingono i ricercatori a puntare sull'UDP, e che in letteratura non vengono mai citate se non in occasioni sporadiche.

Il protocollo UDP, nella sua semplicità, non ha solo il merito di offrire un servizio veloce, ma anche quello di supportare servizi di *broadcasting* e *multicasting*. L'UDP è il protocollo di trasporto utilizzato in tutte le applicazioni bradcasting e multicasting. Quest'aspetto è di fondamentale importanza per le applicazioni VoIP. Infatti, si è più volte messo in evidenza le potenzialità offerte da quest'emergente tecnologia si esprimono fondamentalmente nella possibilità di un multiservizio; in altre parole, non solo un servizio integrato da voce dati e video, ma anche servizi di video conferenza, white-boarding e quant'altro.

Ricordiamo che:

- *Sistema unicast*: offre la possibilità di indirizzare un pacchetto ad una singola destinazione.

- *Sistema broadcast*: è possibile indirizzare un pacchetto a tutti gli host della sottorete, mediante un codice speciale presente nel campo d'indirizzo.

- *Sistema multicast*: è possibile indirizzare un pacchetto ad un sott'insieme scelto di macchine presenti nella rete.

Type	IPv4 ?	IPv6 ?	TCP ?	UDP ?
unicast	•	•	•	•
multicast	opz.	•		•
broadcast	•			•

Fig. II.3.a. Differenti forme d'indirizzamento.
Legenda: • = tipo supportato; opz. = opzionale.

In tab. II.3.a è mostrato un breve riepilogo delle modalità comunicative supportate dai protocolli di rete e di trasporto. Il multicasting è opzionale nell'IPv4 ma

raccomandato nell'IPv6. Il broadcasting non è supportato dall'IPv6. Sia il bradcasting sia il multicasting richiedono il protocollo UDP; queste modalità non sono supportate dal TCP.

Nel par. II.5 vedremo come l'RTP/RTCP, che si appoggia sull'UDP, esprime il massimo delle sue potenzialità quando lavora in ambiente multicast. A tale proposito, nel par. II.4 vedremo alcuni concetti base relativi al multicasting.

II.3.b Signalling in ambiente VoIP.

Un importante punto che caratterizza le applicazioni VoIP è legato al trasporto dei segnali di controllo della comunicazione (*signalling transport)*. Attualmente, non è completamente chiaro a quale protocollo di trasporto affidarsi per il VoIP signalling. La maggiore difficoltà è di riuscire ad ottenere in ambiente IP un'alta affidabilità e ritardi contenuti, compatibilmente alle necessità dei segnali di controllo (che certamente sono meno stringenti di quelle del flusso di dati audio e video). In mancanza di un protocollo ad hoc, la tendenza è quella di alternare l'utilizzo del protocollo UDP e del protocollo TCP.

A tale proposito in [25] è proposto un protocollo (RSTP: Reliable Signalling Transport Protocol) leggero, affidabile ed a basso ritardo in grado di raggiungere parte degli obbiettivi proposti.

In ogni caso la situazione è la seguente. I segnali di controllo e i dati non vocali necessitano di un trasporto affidabile, poiché devono essere trasmessi e ricevuti senza che siano persi. I dati vocali, invece, necessitano di un trasporto che sia veloce, anche a costo dell'inaffidabilità. Di conseguenza, un sistema di comunicazione real-time su reti IP necessita di entrambi i servizi offerti dagli attuali protocolli dello strato di trasporto.La situazione è ben rappresentata in fig. II.1.3.

Le funzionalità di controllo della chiamata (*call signalling & conference control)* sono il cuore d'ogni sistema di comunicazione VoIP. Tali funzionalità riguardano:

- Il set-up della chiamata.
- Gli scambi d'informazioni sulla disponibilità di risorse della rete.

  Questo concetto riguarda sia l'utilizzo del protocollo RSVP che del protocollo di controllo RTCP. L'RTCP si appoggia sull'UDP, per cui in alcuni casi anche i segnali di controllo si appoggiano al servizio inaffidabile dell'UDP. Questa particolarità sarà chiarita nel paragrafo dedicato all'RTP.

- I segnali di controllo sui comandi.

  Ad esempio, nel caso in cui l'utente interagisce attraverso un'applicazione grafica (GUI: Grafical User Interface),

saranno i segnali di tale applicazione (chiama, termina, rifiuta, volume+/-, etc.) ad affidarsi al TCP.

- I messaggi informativi sulla sessione in corso.

  Ad esempio, informazioni dinamiche sui partecipanti ad una sessione multicast, o sulla qualità del servizio.

Tutti questi segnali di controllo necessitano dei servizi di framming, sequenziamento, rilevazione degli errori, correzione degli errori e quant'altro che solo il TCP può offrire. In ogni modo, come vedremo nel par. II.5, per alcuni tipi di controllo è sufficiente il servizio offerto dall'UDP.

II.4 Multicasting nelle reti IP.

Diciamo subito che un datagramma multicast può essere ricevuto solo dalle interfacce (strato di collegamento dati) iscritte alla sessione multicast. Di solito un'applicazione multicast può interessare sia una singola LAN sia una WAN. In fig. II.4.1 è schematizzato cosa accade in un'applicazione multicast, nel caso si utilizzi l'IPv4.

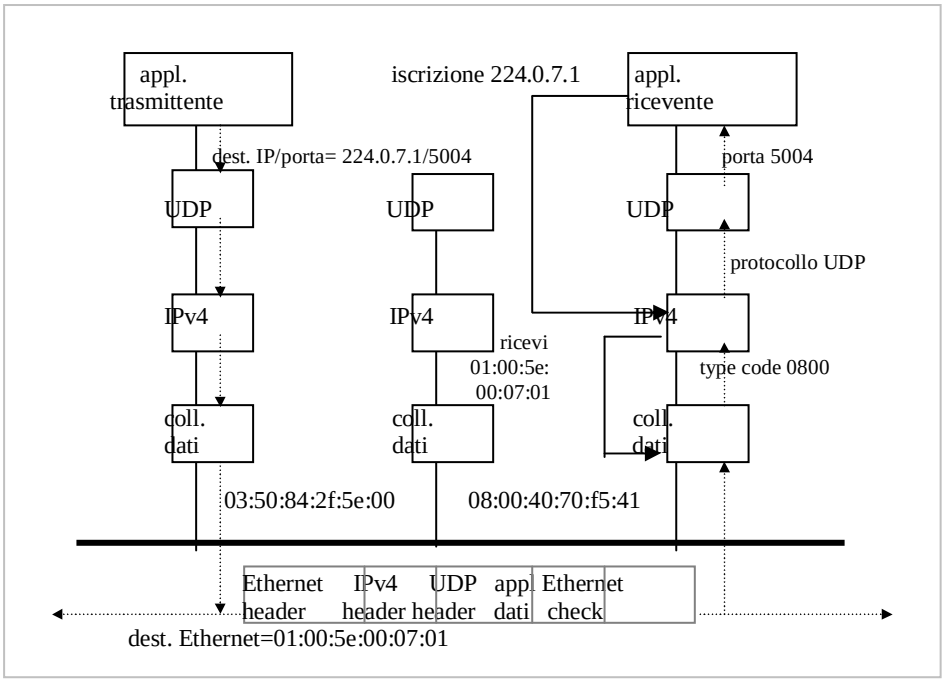


Fig. II.4.1. Esempio di un datagramma UDP multicast.

In fig. II.4.1 sono mostrati tre host su una Ethernet. L'applicazione trasmette un datagramma multicast attraverso il formato D dell'indirizzamento IP (vedi fig. II.2.2), lo strato di trasporto incapsula i dati attraverso l'intestazione UDP e li passa allo strato IP. I dati vengono ulteriormente incapsulati attraverso l'intestazione IPv4 ed infine attraverso l'intestazione Ethernet (18 byte). L'interfaccia Ethernet dell'host al centro della figura confronta il proprio indirizzo Ethernet (03:50:84:2f:5e:00) con quello di destinazione presente nel datagramma (08:00:40:70:f5:41); poiché non sono uguali, l'interfaccia ignora il pacchetto. L'interfaccia dell'host di destra, pur non riscontrando l'uguaglianza tra gli indirizzi Ethernet riceve comunque il datagramma, in quanto l'applicazione ha provveduto ad iscrivere la macchina a qualsiasi gruppo d'indirizzi Ethernet, per cui legge l'intero frame. Il frame viene messo nel buffer d'attesa dell'IPv4[10]. A livello dello strato di rete si compara l'indirizzo IP (224.0.7.1) con quelli multicast a cui si è iscritto l'host; il riscontro ha successo e si guarda al campo

[10] Il campo "type code" (2 byte) presente nell'intestazione Ethernet indica che il tipo di frame è IPv4 (0800, formato esadecimale).

“protocollo” che indica che il datagramma va passato all’UDP. Lo strato di trasporto riscontra con successo il numero di porta ed i dati vengono opportunamente elaborati dall’applicazione. Dunque, un datagramma, prima di essere processato, deve superare tre controlli di corrispondenza (indirizzo Ethernet, indirizzo IP e numero di porta), e ciò vale sia per applicazioni unicast sia per applicazioni multicast.

Quanto detto rende evidente come, l’aspetto multicast di un’applicazione si costruisce in conformità ad un particolare formato d’indirizzamento che riguarda il formato dell’indirizzo IP (formato D). In ogni caso, va precisato che anche l’indirizzamento Ethernet è coinvolto attraverso un’opportuna mappatura dell’indirizzo IP. In fig. II.4.2 è sintetizzata la modalità di tale corrispondenza.

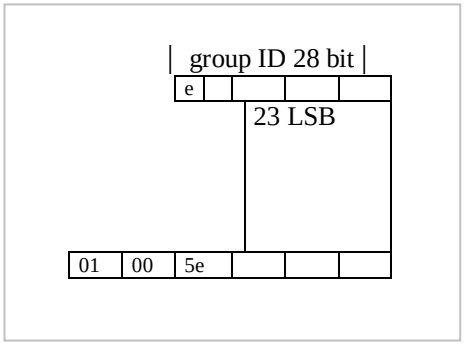


Fig. II.4.2 Mappatura dell’indirizzo IPv4 multicast nell’indirizzo Ethernet.

I primi 4 bit più significativi (MSB) dell’indirizzo IPv4 sono pari ad `e` mentre i successivi 28 bit sono d’identificazione del gruppo. I 3 byte più significativi dell’indirizzo Ethernet sono “sempre” `01:00:5e`; ed il bit successivo è “sempre” `0`. I restanti 23 bit dell’indirizzo Ethernet corrispondono ai 23 bit meno significativi (LSB) dell’indirizzo IPv4. Quindi gli ultimi 4 bit del byte più significativo dell’indirizzo IPv4 ed il successivo bit sono ignorati.

Il multicasting su una singola LAN è piuttosto semplice. Un’host manda pacchetti multicast e tutti gli host interessati lo ricevono.

In fig. II.4.3 è schematizzata una WAN costituita da quattro LAN connesse attraverso quattro router multicast. Per descrivere cosa accade in una sessione audio multicast supponiamo che su quattro host giri un programma che iscrive gli stessi alla sessione audio. Nel momento in cui un host s’iscrive alla sessione multicast manda un messaggio IGMP (Internet Group Managment Protocol) a tutti i routers collegati, in questo modo li informa della sua partecipazione. In questo modo ogni router sa cosa fare nel caso gli giunga un pacchetto con un indirizzo multicast. I router si scambiano le informazioni attraverso un *protocollo di routing* (MRP: Multicast Routing Protocol).

Osserviamo la fig. II.4.3. L’host trasmittente (*hs*) manda pacchetti audio multicast nella rete ed accade che:

- L'host *h1* riceve il pacchetto perché è iscritto alla sessione ed anche il router *r1* lo riceve, poiché ogni router multicast "deve" ricevere tutti i pacchetti multicast.

- Il router *r1* manda il pacchetto al router *r2* poiché, il protocollo di routing, ha informato il router *r1* che *r2* necessita di ricevere pacchetti multicast. Il router *r2* manda il pacchetto all'interno della LAN in modo che *h2* ed *h3* lo ricevano. Inoltre ne invia una copia al router *r3*.

- Il router *r3* invia il pacchetto all'interno della LAN in modo che *h4* lo possa ricevere. Il router *r4* non riceve una copia del pacchetto, in quanto, attraverso il routing protocol, ha informato *r3* che non ha bisogno di pacchetti multicast visto che nessuno degli host connessi alla LAN è iscritto alla sessione audio multicast.

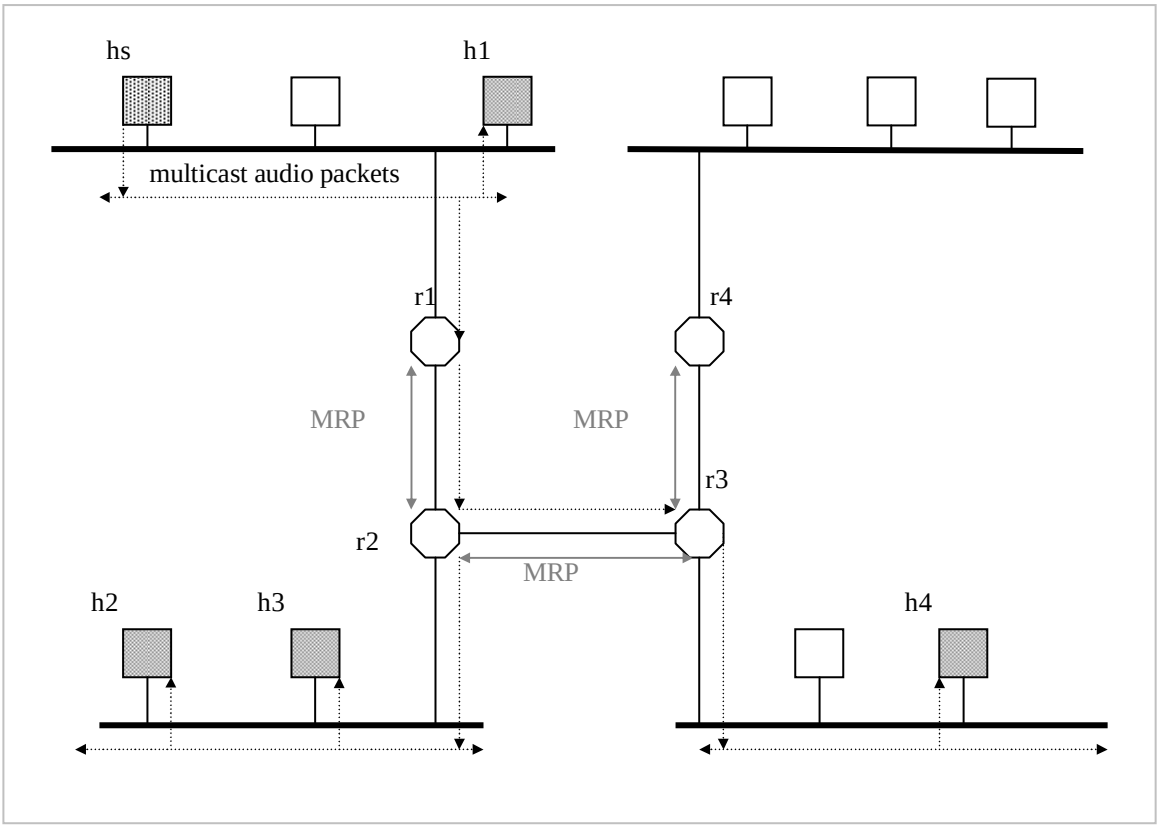


Fig. II.4.3. Multicasting su una WAN.

Il broadcasting è concettualmente analogo al multicasting con l'unica differenza che i pacchetti broadcasting vengono ricevuti da tutti gli host appartenenti alla rete. Comunque, solitamente, il broadcasting è applicato all'interno di una singola LAN.

II.5 Lo strato di applicazione per comunicazioni VoIP.

Attualmente Internet fornisce un servizio che è frutto del suo meglio (*best-effort*) senza garantire una QoS che soddisfi le esigenze delle applicazioni real-time. Per questo motivo, da un po d'anni, sono nate numerose proposte di nuovi protocolli che intendono integrarsi con gli standard acquisiti della suite TCP/IP. Ognuno di essi è progettato per fare logicamente parte dello strato cui è rivolto, ma dal punto di vista implementativo va inquadrato nello strato di applicazione. Per questa ragione si è scelto di descrivere le caratteristiche del protocollo RTP e del protocollo RSVP in questo paragrafo dedicato allo strato applicativo. I servizi offerti dal protocollo RSVP sono caratteristici dello strato di rete; nonostante ciò, in fig II.1.1, è rappresentato nello strato di applicazione, in quanto, non essendo ancora uno standard acquisito, non tutti i sistemi lo supportano. La stessa cosa vale per il protocollo RTP, le cui caratteristiche sono tipiche dello strato di trasporto. Chiarito ciò, nel seguito della trattazione, si farà riferimento ad essi come facenti parte, indifferentemente, del loro strato logico o di quello applicativo.

II.5.a RSVP: una base per l'architettura di Internet a servizi integrati.

Le applicazioni tempo reale in Internet necessitano la garanzia di una QoS attualmente raggiungibile solo attraverso una preallocazione di risorse. Ciò riguarda la potenza computazionale e la capacità di memoria degli host (*end-systems*) e dei sistemi intermedi (router, bridge, gateway e quant'altro). Inoltre, i servizi integrati di voce e video (servizi A/V) richiedono un'elevata quantità di banda che non sempre è disponibile per tutto il periodo di una conversazione, perciò sarebbe opportuno preallocarla. Prima di riservare le risorse richieste dall'applicazione è necessario che assicurarsi che ne esistano a sufficienza (**admission control**). Ciò implica la necessità di un manager delle risorse presente negli host ed in ogni nodo. Tale manager deve verificare che i dati da spedire non eccedano le risorse riservabili, nel qual caso andranno scartati o elaborati con minore priorità di quelli conformi alle specifiche del traffico (**policing control**) [26]. Inoltre, i dati provenienti da flussi diversi devono essere dirottati su linee differenti (**scheduling**) in modo da garantire, il più possibile, ad ogni flusso le risorse richieste. In fig II.5.a.1 è mostrato uno schema implementativo di un manager di risorse. Una tipica situazione può essere la seguente: in un router, i pacchetti ricevuti devono essere immagazzinati in dei buffer ed elaborati dalla CPU (Control Unit Processing); per evitare che il pacchetto sia perso per mancanza di buffer disponibili o eccessivamente ritardato per indisponibilità di risorse di CPU, è necessario che venga classificato in base alle sue esigenze di QoS e posto nella giusta scala di priorità rispetto agli altri. Il manager di risorse può scambiare le informazioni necessarie[11] (vedi frecce tratteggiate in fig. II.5.a.1 ed in fig. II.1.3) attraverso il protocollo RSVP. L'*admission control* in fig. II.5.a.1 informa sulla sufficienza delle

[11] Si parla di segnali di controllo (Signaling Informations).

risorse richieste, mentre il *policing control* da l'assenzo o meno alle richieste in base a costrizioni amministrative. Se entrambi i controlli danno esito positivo le risorse vengono riservate e la voluta QoS è garantita.

L'RSVP permette di riservare risorse per un flusso di pacchetti IP senza stabilire una connessione esplicita e supporta, anche, la comunicazione IP multicast, determinando le risorse da riservare in relazione al numero di macchine in ascolto.

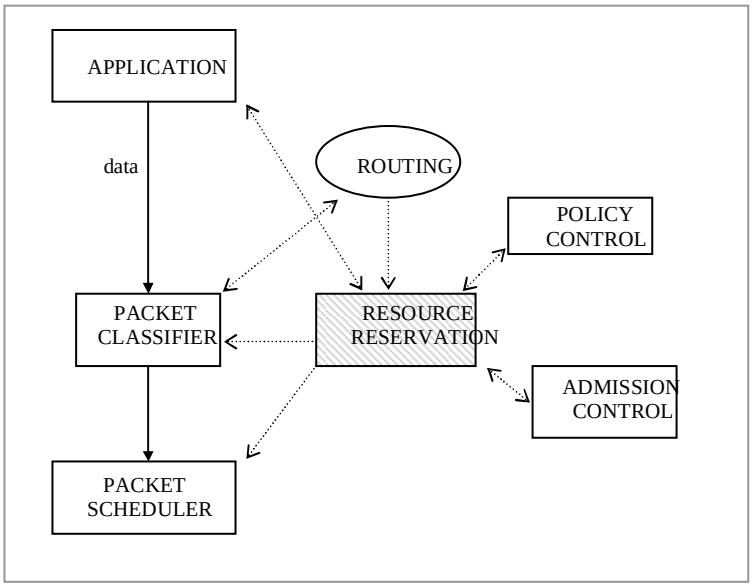


Fig. II.5.a.1 Modello a blocchi di un manager di risorse.

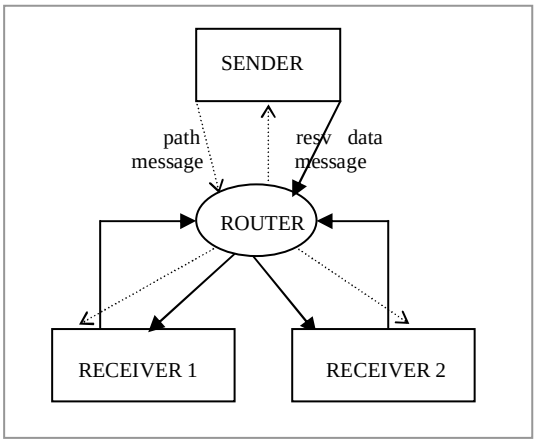


Fig. II.5.a.2. Scambio di controlli con l'RSVP.

L'RSVP è nato per integrarsi nella suite UDP/IP, infatti, esso non intende sostituire il protocollo di rete IP, bensì completarlo. Difatti, l'RSVP è utilizzato per scambiare le sole informazioni di controllo tra i nodi della rete, mentre il flusso di dati è affidato all'IP. Le informazioni di controllo descrivono il traffico in modo da valutare le

risorse allocabili. I sistemi RSVP-compatibili valutano le richieste di spazio nei buffer, di cicli della CPU, di banda e cosi via attraverso lo scambio di *Signaling Information* (vedi fig. II.5.a.2, ove sono evidenziate alcune primitive di servizio).

Il protocollo RSVP è progettato per applicazioni multicast, ma può essere usato anche in unicast, difatti si appoggia sia all'IPv6 sia all'IPv4.

Mediante l'RSVP una richiesta di QoS inizia dall'host trasmittente ed, attraverso ogni nodo dirottatore, arriva all'host ricevente. Di conseguenza, l'uso del protocollo RSVP, per implementare garanzie sul servizio, ha senso solo se tutte le macchine della rete, all'interno della quale s'intende trasmettere, sono RSVP-compatibili. Inoltre, va notato che non tutte le reti facenti parte di Internet sono adatte ha supportare bene un servizio che riservi le risorse. Per cui, non è detto che l'introduzione del protocollo RSVP assicuri la QoS voluta all'interno di Internet, mentre la sua introduzione in una rete locale può dare notevoli risultati.

II.5.b RTP: Real-Time Transport Protocol.

L'RTP [27] definisce il formato di datagramma in grado di supportare una comunicazione tempo-reale su reti IP multiservizio. Va subito precisato che, nonostante il nome, esso non da nessuna garanzia sulle prestazioni tempo-reale. Tuttavia, è in grado di fornire tutti i servizi necessari agli strati delle applicazioni A/V integrate, tra cui:

- La possibilità di ricostruire il flusso vocale in ricezione ogni volta che i pacchetti arrivano ad intervalli irregolari (*jitter equalization*).

- La possibilità di individuare quali e quanti pacchetti siano andati persi (*lost detection*).

- La sincronizzazione degli eventi suono, dati e video (*cros-media syncronization*).

- La garanzia d'interoperabilità con diversi formati di codifica.

- La gestione delle sorgenti multicast.

Tutto ciò lo ha reso meritevole dell'aggettivo real-time.

Nel seguito si discuterà la struttura dell'intestazione RTP facendo esplicito riferimento alle problematiche relative all'interfaccia *codificatore vocale/protocollo RTP*. Per quanto riguarda le applicazioni video, si prenderanno in considerazione i soli aspetti teorici.

Il formato dell'intestazione RTP è riportato in fig II.5.b; pur rischiando di appesantire troppo il discorso si è scelto di commentare ogni campo dell'intestazione in modo da dare maggior enfasi allo spirito con il quale è stato progettato il sistema di comunicazione vocale, RTP-Speak, che verrà descritto nel cap. IV.

I primi dodici “ottetti” costituiscono l’intestazione fissa del formato RTP, mentre la lista degli identificatori CSRC (da first a last) compare solo in caso di presenza di mixer[12]. Di conseguenza il numero di byte dovuti all’intestazione RTP è pari:

$$l = 12 + q \qquad \text{(II.5.b.1)}$$

ove, con q si indica il numero di mixer presenti tra due host che comunicano.

Ogni campo nel formato d’intestazione ha un particolare significato che è interpretato dal processo paritario durante la comunicazione. L’informazione portata da ogni campo è strettamente legata alla particolare applicazione: solo audio, solo video, audio e video. Nel seguito si riporta la descrizione ed il significato dei vari campi.

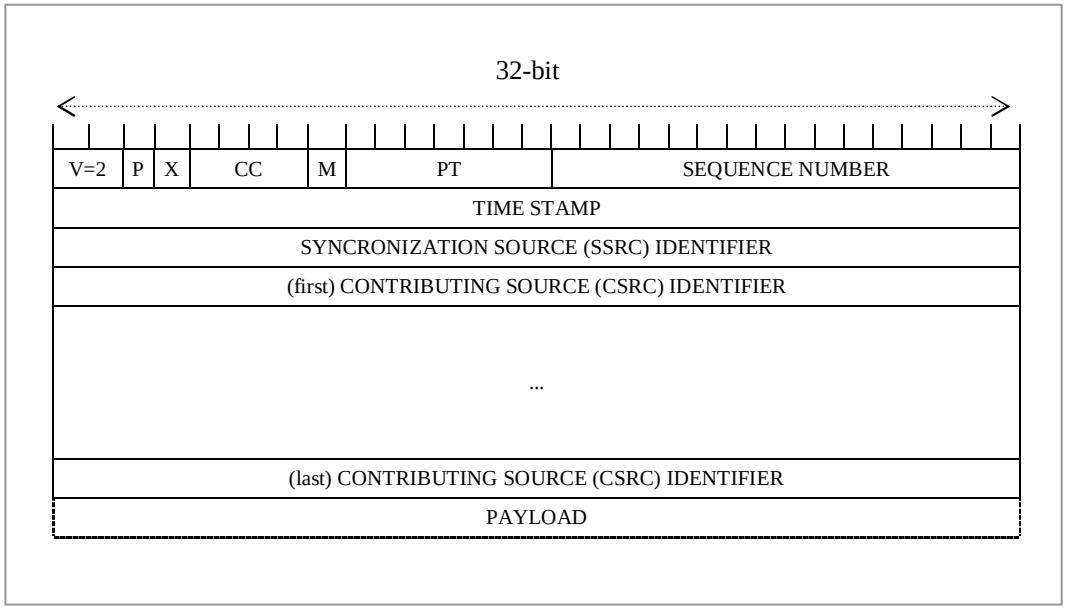


Fig. II.5.b. Intestazione del protocollo RTP.

- *Version* (**V**): 2 bit.
  Questo campo serve ad identificare la versione dell'RTP in uso. Nel presente lavoro ci occuperemo dell’ultima versione, la 2, identificata dai bit: 10. La versione 1 non viene presa in considerazione in quanto ha un interesse esclusivamente storico, inoltre la versione 2 non è compatibile con la 1.

- *Padding* (**P**): 1 bit.
  Questo campo serve ad identificare uno o più ottetti aggiuntivi di riempimento (padding) posizionati alla fine del payload. Se il bit P è posto a 1 allora sono presenti ottetti di riempimento. Si tratta di bit che non fanno parte del payload e nell’operazione di decodifica vanno ignorati. Il numero di ottetti di riempimento presenti nel payload è dato dall’ultimo ottetto. I bit di riempimento possono

[12] I mixer sono dei gateway che operano nello strato d’applicazione, al di sopra dell’RTP. Per approfondimenti si rimanda a[15].

essere necessari sia ad alcuni algoritmi di cripzione sia ad alcuni algoritmi di codifica. Il bit-rate di tali algoritmi non permette, sempre, un perfetto allineamento ad ottetti.
La discussione dell'interfaccia G.711/RTP e G.729/RTP, presentata nel cap. III, servirà a chiarire meglio questo concetto.

- *Extension* (**X**): 4 bit
  Il campo extension, se posto a 1 (0001), indica la presenza di un'estensione dell'intestazione fissa. Per un gran numero d'applicazioni il formato dell'intestazione fissa appare più che completo, tuttavia alcuni particolari profili necessitano d'alcuni campi su misura. L'RTP, dunque, prevede questa possibilità attraverso l'incremento dell'intestazione.
  In [27] è definito il formato dell'estensione. E' ovvio che, nel caso sia presente un'estensione, indicando "e" la lunghezza in byte di quest'ultima, la II.5.b.1 diventa:

$$l = 12 + q + e \qquad \text{(II.5.b.2)}$$

- *CSRC Counter* (**CC**): 4bit.
  Questo campo indica il numero di identificatori CSRC che seguono l'intestazione fissa. Con quattro bit a disposizione è previsto che tale numero vari da 0 a 15.

- *Marker* (**M**): 1 bit.
  Il "marker bit" è il primo esempio di campo che contiene un'informazione specifica di un certo profilo (es., profilo RTP per segnali audio). Infatti, è un particolare profilo[13] che definisce quale deve essere la sua interpretazione da parte del processo paritario. Alla luce di ciò il marker bit dovrebbe far parte dell'estensione dell'intestazione, tuttavia è presente nell'intestazione fissa. Il fatto è che le applicazioni che necessitano di tale campo sono talmente tante da averlo reso titolare dell'intestazione fissa. In generale, lo si utilizza per far sì che eventi significativi, ad esempio i confini di una trama, vengano "marchiati" (marker) nel flusso di pacchetti. Ad esempio, per applicazioni per le quali, durante i periodi di silenzio, vengono mandati pacchetti con *comfort noise* si può avere la seguente situazione: il primo pacchetto di un getto del parlato (*talkspurt*), cioè il primo pacchetto dopo un periodo di silenzio, è distinto dagli altri settando il marker bit, mentre i marker bit dei pacchetti successivi saranno posti a zero.
  L'informazione sul talkspurt può essere utilizzata per sincronizzazioni di vario tipo. Nelle applicazioni A/V tale informazione è spesso utile per sincronizzare l'audio con un particolare evento video.
  Va comunque notato che per le operazioni di sincronizzazione, la presenza del marker bit non è necessaria, ma è fortemente consigliata nel caso di codifiche in

[13] Nel presente lavoro si fa riferimento al profilo specificato in [16].

cui è prevista la soppressione del silenzio; in tutti gli altri casi il marker bit va posto a zero.

- *Payload Type* (**PT**): 7 bit.
  Come il marker bit, anche il payload type, è un campo contenete informazioni interpretabili attraverso uno specifico profilo; ed anche in questo caso, la sua indubbia necessità in numerose applicazioni lo ha reso titolare dell'intestazione fissa. Solitamente un profilo rivolto alla specifica applicazione definisce una mappatura di defoult dei bit all'interno del campo, al quale permette di interpretare l'informazione portata dal payload type. Per maggiore chiarezza si può consultare [28, tab.1], la cui mappatura è stata seguita nel progetto del presente lavoro, vedi par. II.3. La mappatura proposta permette di scrivere un software che sia interoperabile con i codificatori standard ed allo stesso tempo, mediante i valori non assegnati (unassigned), aperto ad altri tipi di codifica. In sostanza, la mappatura proposta in [30] permette di implementare il proprio codificatore e di interoperare con quelli standard. Ciò significa scrivere un codice che accetti sia i formati di "payload RTP" specifici degli standard che quelli non standard che si vogliono inserire.

- *Sequence Number*: 16 bit.
  Il sequence number è incrementato di uno per ogni pacchetto RTP spedito. L'utilizzo di questo campo è semplice e fondamentale. Nelle applicazioni A/V è utilizzato in ricezione per rilevare uno o più pacchetti persi. Nelle applicazioni solo audio può essere utilizzato anche per il recupero della giusta sequenza temporale dei pacchetti (jitter equalizzation).
  Il sequence number può essere usato anche per determinare l'esatta posizione di un pacchetto durante la decodifica video, in questo modo è possibile decodificare i pacchetti senza preoccuparsi dell'ordine sequenziale.
  Solitamente, il valore iniziale del sequence number è scelto in modo random ed impredicibile, in modo da difendersi da attacchi ad eventuali cripzioni del massaggio.

- *Time Stamp*: 32 bit.
  Il time stamp riflette l'istante di campionamento del primo campione presente nel datagramma RTP. Il valore iniziale del time stamp è consigliabile che sia random ed impredicibile per le stesse ragioni del sequence number. Le modalità d'incremento time stamp dipendono fortemente dal tipo di codifica.
  Per applicazioni audio a tasso-fisso[14] il time stamp va incrementato di "uno" per ogni periodo di campionamento (T_s); così se f_s=8 kHz, quindi, T_s=0.125 ms, e l'applicazione audio legge trame da 20ms (es., GSM), che corrispondono a 160 periodi di campionamento, il time stamp va incrementato di 160 per ogni

[14] Codificatori audio che prevedono il campionamento del segnale vocale ad una frequenza di campionamento (f_s) fissa.

datagrammi spedito. Nelle applicazioni A/V o solo audio il time stamp è utilizzato per compensare il jitter dei pacchetti.
Va notato che alcuni pacchetti RTP consecutivi possono avere il medesimo time stamp giacché (logicamente) generati nello stesso momento; è il caso di pacchetti appartenenti alla stessa trama video.

Il "sequence number ed il "time stamp" sono spesso utilizzati per sincronizzare l'audio con il video (*cross-media syncronizzation)*. L'assenza di uno dei due renderebbe il compito molto difficile, sempre che non si usino codificatori audio che non prevedono la soppressione del silenzio.

- *Syncronization Source (**SSRC**) Identifier*: 32 bit.
  Per "syncronizzation source" s'intende una sorgente di un flusso di datagrammi RTP. Per sorgente s'intende un'host trasmittente, perciò si hanno più SSRC solo nel caso d'applicazioni **multicast**. Nel caso di una singola sorgente mandi diversi flussi di dati (es., audio da microfono e video da telecamera) entrambi i flussi devono avere lo stesso identificatore SSRC. Tutti i datagrammi provenienti dalla stessa SSRC vengono raggruppati dall'host ricevente per essere emessi (playback).
  Anche in questo caso è consigliabile che l'SSRC identifier sia random ed impredicibile. Inoltre ogni SSRC identifier deve essere globalmente unico in per ogni sessione RTP. Maggiori dettagli in proposito sono disponibili in [27].

- *Contributing Source (**CSRC**) Identifier*: 32 bit.
  Il campo "contributing source identifier" è opzionale ed è legato ai concetti di *mixer* e *traslator*. L'utilizzo di questo campo è legato all'interconnessione di reti. Per approfondimenti si rimanda a [27].

Le precedenti considerazioni sui vari campi dell'intestazione RTP danno l'idea di come l'interpretazione del formato RTP sia strettamente legata alla particolare applicazione. La sua struttura, a prima vista, può sembrare eccessivamente articolata, ma in realtà lo è per esigenze di flessibilità. Infatti, i servizi che uno strato dovrebbe offrire agli strati superiori vanno inquadrati in relazione alle esigenze di quest'ultimi. Le applicazioni VoIP richiedono servizi ad alta complessità perciò, necessariamente, il protocollo che li gestisce deve essere fortemente articolato, soprattutto se gli strati inferiori non sono stati progettati per richieste di questo tipo. Del resto, lo stesso *payload*, che non fa parte dell'intestazione RTP è mutevole a seconda del tipo di codifica. A tale proposito, seguono alcuni criteri implementativi riguardanti il formato del payload per codificatori audio. Per tali considerazioni si è preso spunto dai suggerimenti disponibili in [28].

Per introdurre i concetti relativi al formato audio del payload RTP è necessario distinguere tra codificatori basati sul campione (*sample-based encoder*) e codificatori basati sulla trama (*frame-based encoder*).

SIGLA ITU-T	PRINCIPIO DI CODIFICA	TIPO DI CODIFICA	RAPPR. DELL'INF.	VELOCITA'	TASSO DI COMPR[15].
G.711	PCMA e PCMμ	basata sul campione	8 bit/campione	64 kbit/sec.	1:2
G.729	CS-ACELP	basata su trame da 10ms	10 byte/trama	8 kbit/sec.	1:16

Tab. II.5.b.

Nelle codifiche basate sul campione, ogni campione è rappresentato da un numero fisso di bit. Per cui le indicazioni sul formato del payload sono molto semplici. Un datagrammi può contenere un qualsiasi numero di campioni audio (ovviamente, nei limiti ammessi dai protocolli sottostanti) con il "vincolo" che il numero di bit contenuti nel payload sia multiplo di otto. Il vincolo si riferisce a problemi d'interoperabilità d'implementazioni diverse: è necessario sapere che i byte presenti nel payload rappresentino l'informazione di un numero finito di campioni.

I codificatori basati sulla trama comprimono una trama audio, di lunghezza fissa, in una trama sintetica, tipicamente anch'essa di lunghezza fissa. Il principio generale è quello secondo il quale da una certa trama è possibile estrarre informazioni, rappresentabili con un numero di bit inferiore a quello originale, che ne permettano la ricostruzione in fase di decodifica. Dunque, analogamente a quanto visto prima, ogni pacchetto deve contenere un numero intero di trame. Se tutte le trame hanno la stessa lunghezza l'host ricevente può capire quante trame sono contenute in un datagrammi RTP semplicemente dividendo la lunghezza del payload per la lunghezza delle trame sintetiche, la quale è definita dalla codifica. Altrimenti, le trame stesse dovranno indicare la loro estensione. Le trame vanno inserite nel pacchetto in base alla loro età, così che la più vecchia, che deve essere emessa per prima, stia immediatamente dopo l'intestazione RTP.

Il sistema di comunicazione che sarà presentato nel cap. IV supporta entrambi i tipi di codifica (vedi tab. II.5.b).

Nonostante non ci siano vincoli alla lunghezza del payload (tranne i limiti imposti dagli strati sottostanti), vanno presi alcuni accorgimenti in relazione alla perdita di pacchetti. Infatti, più sono lunghi i pacchetti e meno sovraintestazione[16] introducono ma anche un più alto ritardo. Inoltre, più lungo è il pacchetto perso maggiore è la degradazione della voce. Dunque è necessaria una scelta ragionevole.

Per i segnali audio, l'intervallo di pacchettizzazione di defoult è di 20 ms, sia nel caso in cui la trama elaborata dal codificatore è minore o uguale a 20 ms, che nel caso di codificatori basati sul campione; altrimenti la lunghezza del payload è pari a quella della

[15] Il tasso di compressione è riferito al PCM lineare: 16 bit/campione, 128 kbit/s.

[16] I 40 byte d'intestazione RTP/UDP/IP, nel caso di codificatori ad alta compressione (es., G.729 o G.723), superano la lunghezza del payload (es., 20 byte nel caso del G.729 e 30 byte nel caso del G.723). Questo fenomeno è conosciuto con il nome di "sovraintestazione".

trama elaborata (es., G.723: 30 ms). Queste ultime indicazioni rappresentano un consiglio, quindi ogni applicazione deve essere in grado di determinare da sé la lunghezza del payload e l'eventuale numero di trame presenti.

II.5.c RTCP: Real-Time Transport Control Protocol

L'RTCP è il protocollo di controllo dell'RTP e fa parte della sua specifica [27]. Spesso in queste note si fa riferimento ad entrambi i protocolli indicandoli con l'acronimo "RTP/RTCP" in modo da evidenziare il loro stretto legame ed allo stesso tempo la loro indipendenza. Infatti, i servizi offerti dall'RTCP sono logicamente legati a quelli offerti dall'RTP, ma i due protocolli sono implementati separatamente (nel cap. IV vedremo che ognuno ha la propria socket, la propria porta UDP, e così via). L'RTCP non trasferisce i dati, ma alcune informazioni di controllo correlate allo stream di dati RTP. I messaggi RTCP sono trasmessi allo stesso indirizzo IP (unicast o multicast) dei corrispondenti pacchetti RTP, ma con un numero di porta differente (solitamente pari al numero di porta associato all'RTP più uno).

Un pacchetto RTCP è fatto di un'intestazione fissa seguita da una serie d'elementi strutturati che dipendono dal particolare tipo di pacchetto RTCP. Infatti, le funzionalità dell'RTCP si esprimono secondo una serie di formati di pacchetto ad hoc per ogni tipo di messaggio. Ciò lo rende estremamente versatile, o come si suole leggere in letteratura: *scalabile*. La scalabilità è una proprietà voluta per far sì che si possa scegliere una o più delle sue funzionalità, a seconda delle esigenze legate alla particolare applicazione. I formati associati ai pacchetti RTCP sono cinque:

- SR: Sender Report
- RR: Receiver Report
- SDES: Source description items
- BYE: Bye message
- APP: Application Specific functions

Ogni formato inizia con una parte fissa d'intestazione, simile a quella dei pacchetti di dati RTP, seguita da elementi strutturati di lunghezza variabile. Solitamente, più pacchetti RTCP sono spediti insieme in una singola incapsulazione UDP (pacchetto RTCP composto).

In questa sede si eviterà di descrivere uno ad uno i vari formati RTCP, per i quali si rimanda alla fonte classica: [27]. Lo spirito di questo paragrafo è quello di descrivere le funzionalità legate all'RTCP, a prescindere dal particolare formato.

I più importanti elementi di controllo sono i cosiddetti *sender report* (SR) e *receiver report* (RR). Entrambi sono una sorta di resoconto sulla QoS. I sender report

sono generati da ogni host trasmittente insieme ai pacchetti di dati RTP, mentre i receiver report sono generati dagli host che non trasmettono i pacchetti di dati RTP.

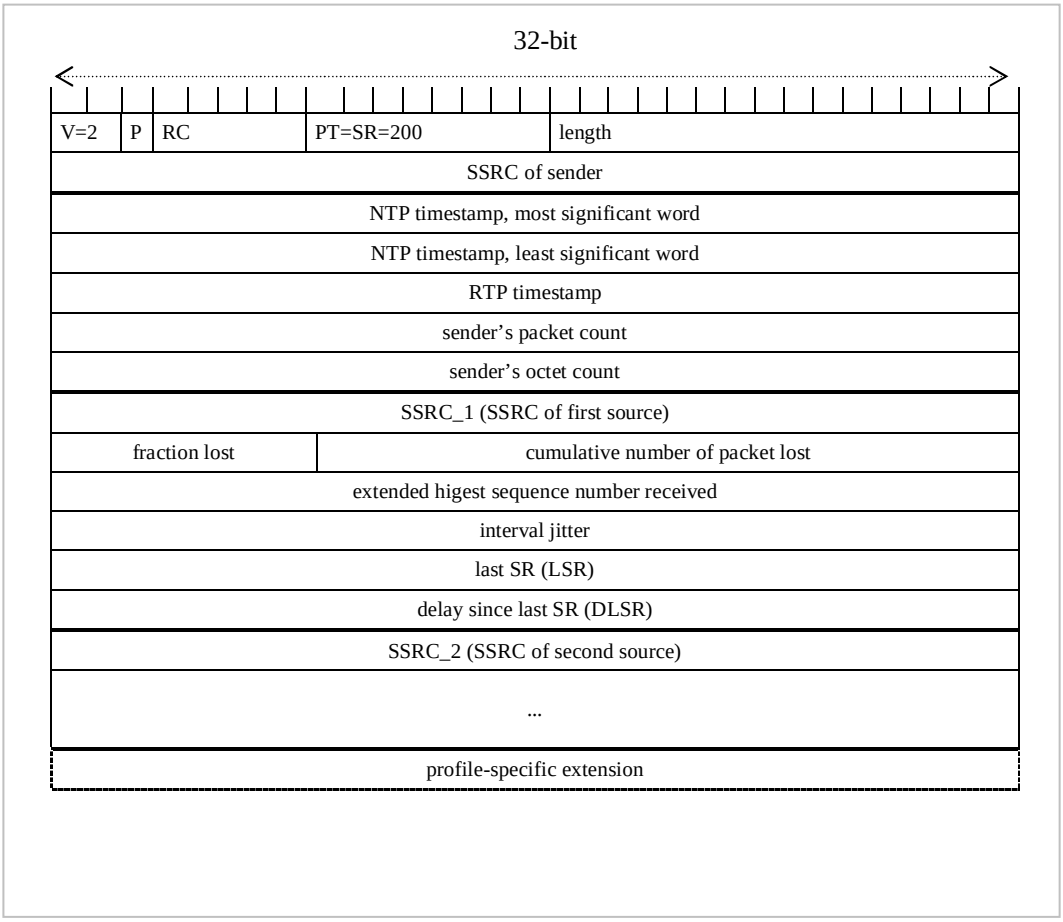


Fig. II.5.c.1. Pacchetto RTCP SR: Sender Report.

Entrambi i report contengono informazioni sull'attuale QoS oltre che alcuni *report block*. Il formato SR è mostrato in fig. II.5.c.1. Il formato RR si differenzia da quello dell'SR per la sola mancanza dei 20-byte di *sender info*, in quanto ne fanno uso i soli trasmettitori attivi. L'intestazione è composta da sei campi, per un totale di 8 byte. I primi due campi, version e padding, hanno significato analogo a quelli dell'RTP; il terzo campo, reception report count, dice quanti report block sono presenti nel pacchetto; il quarto campo, packet type, serve ad identificare il tipo di pacchetto (SR o RR); il quinto campo, length, contiene la lunghezza, in byte, del pacchetto; infine, nel sesto campo è contenuto l'identificatore SSRC, syncronization source identifier, di chi ha originato il pacchetto.E' evidente come ogni report block contiene una serie di informazioni statistiche sul flusso di dati: la frazione di pacchetti persi fino all'ultimo report nonché il numero totale, il più alto sequence number ricevuto ed il jitter. Ogni report block è associato ad un certo partecipante multicast attraverso il suo SSRC. Nel blocco relativo al sender info vengono incluse delle informazioni sull'host trasmittente: NTP (Network Time Protocol) [29], l'RTP timestamp, il numero di pacchetti ed il numero di byte trasmessi. E' prevista, anche, un'estensione del pacchetto (profile-

specific extensions) nel caso l'applicazione necessiti d'ulteriori informazioni addizionali.

Il formato SDES (*source descriptions*) descrive la sorgente con maggiori dettagli; possono essere incluse informazioni del tipo: user name, indirizzi e-mail, numeri di telefono, informazioni geografiche e così via. Il formato BYE indica che un'utente sta lasciando il multicast. Queste informazioni possono essere mostrate sul display di ogni utente. Il formato APP, in scia con la proprietà di scalabilità, è rivolto ad esprimere funzionalità di una specifica applicazione.

Le caratteristiche del protocollo RTCP si prestano bene ad una comunicazione adattiva in base alle attuali condizioni di carico della rete. Ad esempio, il tasso di pacchetti persi, se eccessivo, può indicare che la rete è in fase di saturazione, dunque è probabile che diminuendo il tasso di trasmissione, la qualità della comunicazione migliora. Adottare un codificatore (video o audio) ad un più alto tasso di compressione significa richiedere meno banda alla rete, quindi diminuire il tasso di pacchetti persi migliorando la QoS. Il beneficio va pagato in termini di cicli della CPU e in una minore qualità nelle immagini e della voce. Va da se che la perdita di pacchetti merita questo sacrificio giacché è l'effetto più disastroso che inficia la comprensione del parlato. Uno schema a blocchi della trasmissione di controllo adattiva è mostrato in fig. II.5.c.2.

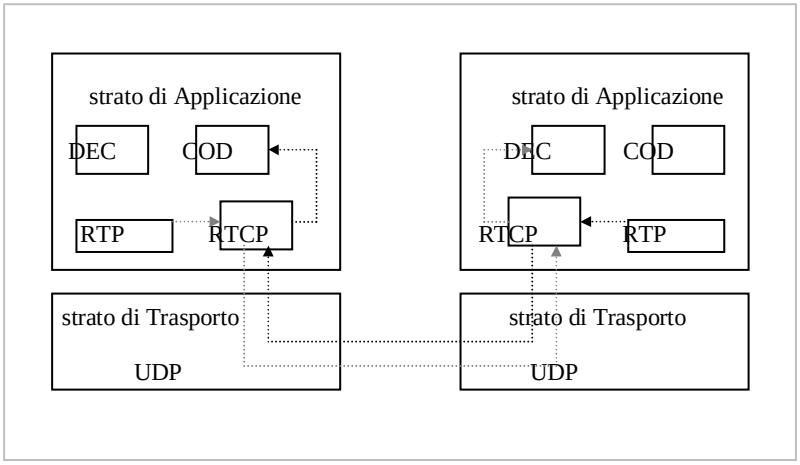


Fig. II.5.c.2. Comunicazioni (virtuali) di controllo tra processi paritari.

La fondamentale funzione dell'RTCP è quella di monitorare costantemente la qualità del servizio e di fornire informazioni sui partecipanti in una sessione multicast. Va comunque precisato che l'RTCP non è esaustivo di tutti i possibili controlli necessari ad una sessione multicast. Infatti, esso si limita a monitorare alcuni aspetti ed ha comunicare i risultati a chi di dovere (es., RSVP, codificatore). Di conseguenza, il controllo vero e proprio su una sessione multicast è solitamente affidato ad un protocollo ad hoc (Session Control Protocol) che si appoggia sul TCP (per esigenze di affidabilità della trasmissione dei dati di controllo).

In ultima analisi va considerato l'aspetto legato al tasso d'emissione dei pacchetti RTCP in relazione alla banda disponibile. In altre parole, tenendo conto del fatto che l'RTP trasporta i dati mentre l'RTCP monitorizza la trasmissione, qual è il

tasso “ottimo” di trasmissione dei pacchetti RTP? Visto che l’RTCP offre diversi servizi di diversa importanza è opportuno considerare la cosa nel seguente modo. Si stabilisce una ragionevole quota di banda da affidare all’RTCP (tipicamente il 5%) ed all’interno di essa si stabiliscono delle sottoquote (tipicamente il 20%) secondo una scala di priorità relazionata al tipo di pacchetto RTCP. In sostanza, data la banda disponibile in una sessione (*session bandwidth*[17]) una parte di essa andrà affidata al flusso di monitoraggio. La parte da affidare al monitoraggio deve essere sufficientemente grande da far si che l’RTCP assolva il proprio compito ad abbastanza piccola da non togliere eccessiva banda all’RTP. In [27] si suggerisce di dare all’RTCP, di tutti i partecipanti al multicast, il 5% dell’intera banda. Tale valore va comunicato all’RSVP, se i sistemi lo supportano. La precedente quota va divisa equamente tra tutti i partecipanti è quindi va aggiornata dinamicamente al variare del loro numero. L’aggiornamento dinamico è indispensabile in una sessione multicast poiché, a differenza del flusso vocale, il flusso di controllo non è auto-limitantesi; se fosse a tasso costante, crescerebbe linearmente con il numero di partecipanti, e richiederebbe sempre più banda a discapito del flusso di dati trasportati con l’RTP.

[17] Il calcolo della banda disponibile in una sessione può essere dedotto sia dai limiti imposti dalla rete che da un’assunzione ragionevole. In ogni caso, il calcolo coinvolge anche i protocolli degli strati sottostanti (UDP ed IP). La session bandwidth è indipendente dal tipo di codifica, ma è chiaro che la particolare codifica scelta può avere bisogno di più o meno banda.

II.6 Considerazioni sui protocolli RTP ed RTCP

Fornire un servizio telefonico integrato in Internet è una sfida complessa ed affascinante. La maggiore difficoltà sta nel riuscire ad ottenere una QoS al pari di quella offerta dalla tradizionale rete PSTN. Per riuscire nell'impresa sono necessari interventi che coinvolgono l'intera struttura delle reti IP. I servizi richiesti dal trasporto integrato (dati, voce e video) necessitano di prestazioni ottenibili solo con il coinvolgimento di tutti gli strati della rete. In particolare si è visto come l'RSVP e lIPv6 vadano ad integrare le funzionalità dello strato di rete, mentre l'RTP/RTCP sta a cavallo tra lo strato di trasporto e quello applicativo.In questa sede ci occuperemo soprattutto del protocollo RTP/RTCP perché forma le basi del sistema di comunicazione tempo-reale riguardante il presente lavoro.

Spesso, in letteratura si legge che il protocollo RTP/RTCP è volutamente non completo e che, pur facendo logicamente parte dello strato di trasporto, non è un vero e proprio protocollo di trasporto. L'RTP ha delle importanti proprietà che sono caratteristiche dello strato di trasporto: è presente negli host (end-systems), permette il demultiplexing dei pacchetti. Tuttavia demanda molte funzionalità tipiche dello strato di trasporto all'UDP: checksum, numero di porta, framming. Infatti, l'RTP non contiene un campo di lunghezza; si assume che il framming sia un servizio fornito dai protocolli sottostanti e che ogni pacchetto RTP venga incapsulato con un'unica intestazione del protocollo di trasporto (tipicamente l'UDP). Inoltre, siccome molte applicazioni non necessitano del framming, sarebbe uno spreco di cicli della CPU e di banda aggiungerlo come parte dell'RTP. Nel caso l'RTP sia usato con protocolli di trasporto basati sui messaggi (es., three-way handshake del TCP) basterà semplicemente definire un profilo RTP che comprenda un campo di lunghezza (16 o 32 bit, a seconda delle esigenze). E' evidente che l'RTP/RTCP non può essere identificato come uno strato di trasporto a se. In effetti, a ben guardare la sua è una natura integrativa e va visto come supporto alle applicazioni tempo-reale. La stessa cosa vale per gli strati superiori. Il legame con la particolare applicazione è forte, ma la sua flessibilità lo svincola da qualsiasi forma di dipendenza (proprietà di scalabilità). Ad esempio, ogni tipo di codifica richiede un'implementazione a parte che può essere completamente diversa dalle altre, ma la sua struttura fa sì che esso possa interfacciarsi bene con qualsiasi tipo di codifica.

Apparentemente alcuni campi presenti nel formato RTP potrebbero sembrare ridondanti, tanto da suggerire una versione più "sottile" in modo da limitare il problema della sovraintestazione. In realtà ogni campo ha una caratteristica che per certe applicazioni appare come superflua mentre per altre è fondamentale. Ad esempio, in un'applicazione unicast il campo CSRC non è necessario, ma in una conferenza tale campo è di grosso aiuto nella gestione della multiutenza. La versione RTP sottile potrebbe essere fatta dai soli campi: V, P, PT e Sequence Number. Una scelta di questo tipo porterebbe ad una serie di svantaggi, ad esempio:

- L'assenza dell'identificatore SSRC non permetterebbe l'uso di mixer e traslator.

- Poiché l'uso dell'SSRC è necessario nelle applicazioni multicast, la sua assenza romperebbe la compatibilità con molti standard (es., suite H.323 [8] [9]).

- A meno che ogni implementazione supporti la versione sottile, saranno necessarie una serie di gateway che permettano l'interoperabilità.

- Senza il campo "time-stamp" le sincronizzazione tra flussi eterogenei (cross-media syncronization) può risultare estremamente difficile, a meno che non si usino codificatori senza la soppressione del silenzio (ma in questo caso la cosa risulterebbe alquanto inefficiente visto che con questo tipo di compressione si riesce a ridurre la necessità di banda fino al 50%, mentre il guadagno sulla sovraintestazione sarebbe di pochi byte per pacchetto).

- Gran parte delle proprietà dell'RTCP verrebbero meno.

E' ormai evidente che la massima espressione delle potenzialità dell'RTP/RTCP si manifesta in sessioni multicast con supporto A/V, ma le sue caratteristiche sono utili anche ad applicazioni meno articolate (es., unicast e solo audio). Per queste ultime si consiglia, per garanzia di compatibilità, comunque un'implementazione completa, vale a dire che preveda tutti i campi dei formati.

Capitolo III

Generalità sulla codifica vocale in ambiente VoIP.

La codifica del segnale vocale [30] è un argomento estremamente vasto e non ancora completamente esplorato. Nei precedenti capitoli si è messo in evidenza come la codifica del segnale vocale ha una grossa rilevanza nelle applicazioni VoIP. Infatti, poiché le reti IP sono intrinsecamente inaffidabili è necessario far fronte ad una serie d'artefatti. Il problema più grosso è la perdita di pacchetti, soprattutto se si perdono più pacchetti in sequenza (*burst of loss*). Inoltre, la disponibilità di banda non è sempre sufficiente. Attualmente, molti utenti si connettono ad Internet da casa tramite modem analogici ed hanno un limite di banda che non va oltre i 56 Kbps, mentre la banda richiesta per avere una qualità telefonica della voce è di 64 Kbps. A farsi carico di questi problemi è spesso il codificatore. Infatti, numerose ricerche hanno portato alla nascita di codificatori ad alto tasso di compressione (in modo da ridurre le richieste di banda) e ad alta robustezza nei confronti dei pacchetti persi (per contenere i "gap" nella conversazione).

Per queste ragioni una comunicazione vocale robusta su un canale inaffidabile è un settore di ricerca chiave nello sviluppo della tecnologia VoIP. Tuttavia, il massimo che ci si può aspettare da questo settore è un'accettabile qualità della voce nonostante la perdita di pacchetti e la scarsa disponibilità di banda. Infatti, risultati "ottimi" a riguardo si potranno ottenere solo attraverso interventi strutturali sulle reti IP.

Solitamente, si distingue tra due fondamentali strategie di codifica: la codifica basata sul campione e la codifica basata sulla trama. Alla prima categoria appartengono quei compressori che tendono a minimizzare la quantità di bit necessaria a rappresentare ogni singolo campione del segnale vocale. Della seconda categoria fanno parte quelle tecniche di compressione che tendono a minimizzare il numero di bit necessari alla rappresentazione di un certo tratto di segnale vocale (trama o frame). Esistono anche delle tecniche "ibride" che guardano sia al singolo campione sia alla trama.

Tipicamente le codifiche basate sulla trama sono quelle che permettono il più elevato tasso di compressione, ma all'aumentare di esso tendono a produrre una voce con un suono metallico.

In questo capitolo sarà trattato l'argomento relativo all'interfaccia: *protocollo RTP/codificatore vocale*. In particolare si farà riferimento a due tecniche di codifica del segnale vocale: la PCM (*Pulse Code Modulation*) e la CS-ACELP (*Coniugate Structure – Algebraic Code Excited Linear Prediction*). Si tratta delle codifiche implementate nel sistema di comunicazione progettato, il quale sarà descritto nel cap.IV.

Il segnale vocale può essere rappresentato da una serie di bit che costituiscono la quantizzazione dei suoi campioni. Data una rappresentazione digitale del segnale

vocale, la codifica (o compressione) è una tecnica che permette di poter rappresentare il segnale con un numero di bit inferiore alla rappresentazione originale.Senza voler entrare nel merito delle metodologie d'acquisizione e digitalizzazione del segnale vocale, si procederà, nel seguito, con la descrizione dei principi che governano le due tecniche di codifica sopra citate.

In particolare, nel par. III.1 sarà presa in considerazione la tecnica di pacchettizzazione del segnale vocale che fuoriesce dalla codifica PCMA (Pulse Code Modulation A-law) e PCMμ (Pulse Code Modulation μ-law). Nel par. III.2 si considererà la pacchettizzazione delle trame sintetiche del codificatore G.729.

III.1 Cenni al formato di codifica G.711

Il PCM (Pulse Code Modulation) è un metodo di conversione in digitale di un segnale analogico strettamente limitato in banda. Infatti, l'acronimo "PCM" si riferisce al campionamento nel tempo ed alla quantizzazione in ampiezza di un'onda PAM (Pulse Amplitude Modulation). Questa conversione è una forma di codifica ed, infatti, in letteratura, il PCM è noto come un formato di codifica.

Il codificatore PCM è il più semplice rappresentante dei codificatori basati sul campione. Il segnale vocale, solitamente, è campionato ad 8000 Hz ed ogni singolo campione viene quantizzato. Numerosi studi concordano sul fatto che, nel caso si usi la classica quantizzazione uniforme, per mantenere un'alta qualità nella rappresentazione del segnale vocale sono necessari almeno 11-16 bit/campione. Il formato di rappresentazione del segnale vocale a quantizzazione uniforme è conosciuto con l'acronimo "PCM-lineare". Questo formato da luogo ad un *bit-rate* di 88 – 128 kb/sec.

Per mantenere l'alta qualità della rappresentazione ed avere un bit-rate minore è necessario affidarsi ad un formato di quantizzazione non-uniforme [30]. In particolare, la distribuzione degli intervalli di quantizzazione deve essere tale da seguire le caratteristiche del segnale vocale. Il segnale vocale ha un andamento la cui ampiezza varia di poco ed improvvisamente di tanto. Per cui è ovvio che, per garantire una buona qualità nella rappresentazione con un numero di livelli di quantizzazione inferiore a 11 –16, è necessario distribuire l'ampiezza dei livelli di quantizzazione in modo tale che segua proprio quella del segnale vocale. Un andamento non-lineare in grado di assolvere questo compito e, ad esempio, la legge logaritmica. Alternativamente, è possibile quantizzare il logaritmo del segnale in ingresso piuttosto che il segnale stesso, infatti, in questo modo il segnale viene mappato opportunamente. Questa tecnica è nota con il termine *companding* (*comp*-ressing & *ex*-panding). Si tratta di un processo di distorsione controllata del segnale analogico prima della sua quantizzazione, e consiste nel comprimere i valori alti del segnale in fase di codifica per poi riespanderli in fase di decodifica. Mediante questa tecnica è possibile rappresentare il singolo campione con solo 8 bit ottenendo una qualità di rappresentazione praticamente pari a quella ottenibile con 11 – 16 bit.

La rappresentazione PCM non-lineare, nel 1960, diede vita a due famosi standard con un bit-rate di 64 kbit/sec: il PCMA (Pulse Code Modulation A-law) ed il PCMμ (Pulse Code Modulation μ-law). La legge A è lo standard adoperato nell'America del nord, in Giappone e nella Corea del sud, mentre la legge μ è lo standard adoperato nel resto del mondo. L'espressione matematica della legge A e della legge μ è data, rispettivamente, dalla (III.1.1) e dalla (III.1.2):

$$\begin{aligned} \bar{s}(n) &= s_{\max}\frac{1+\log(As(n))}{1+\log(A)} \quad per \quad \frac{1}{A} < s(n) \le 1 \\ \bar{s}(n) &= s_{\max}\frac{As(n)}{1+\log(A)} \quad per \quad 0 \le s(n) < \frac{1}{A} \end{aligned} \tag{III.1.1}$$

ove, *A=87.6* (CCITT: International Telegraph and Telephone Commitee)

$$\bar{s}(n) = s_{\max} \frac{\log(1+\mu \mid s(n) \mid)}{\log(1+\mu)} sign(s(n)) \qquad \text{(III.1.2)}$$

$$\text{ove,} \qquad sign(x) = \begin{cases} 1 & x > 0 \\ 1/2 & x = 0 \\ -1 & x < 0 \end{cases}$$

μ *=1000 (*USA, Giappone, sud Corea)

Il significato dei simboli è il seguente: $\bar{s}(n)$ rappresenta il generico campione del segnale vocale distorto secondo la legge A (o μ), $s(n)$ rappresenta il generico campione del segnale vocale da elaborare, s_{max} è il valore di picco del segnale vocale.

Dal 1960 ad oggi i progressi nell'elaborazione numerica dei segnali hanno dato vita a nuove forme di codifica vocale ad un più alto tasso di compressione mantenendo un'ottima qualità. Ma, grazie alla loro semplicità, all'eccellente qualità della rappresentazione ed al bassissimo ritardo, entrambi i formati sono, ancora, largamente in uso. In ogni caso, va tenuto presente quanto detto nei precedenti capitoli a proposito della disponibilità di banda nelle reti IP. In particolare, un bit-rate di 64 kb/s non è certo l'ideale per le applicazioni VoIP all'interno di Internet mentre è addirittura consigliato all'interno di una rete locale. Il problema della disponibilità di risorse all'interno di Internet non è solamente legato alla garanzia dei 64 kb/s per la voce, ma va tenuto presente che nel caso si comunichi in modalità A/V la richiesta di banda cresce esponenzialmente.

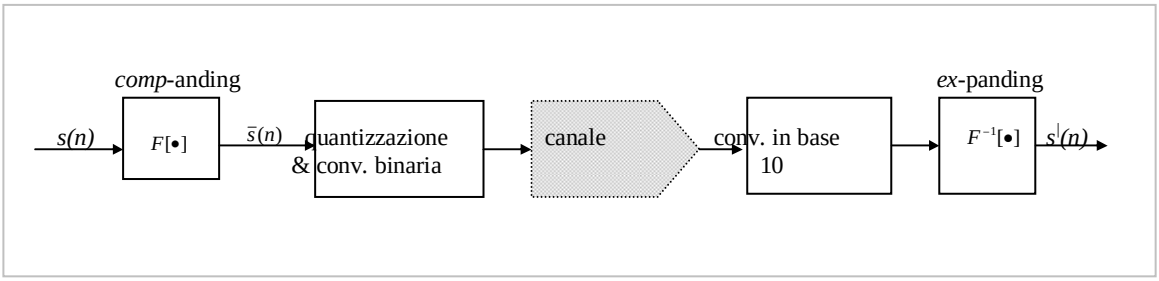


Fig. III.2.1. Diagramma a blocchi della codifica PCMA e PCMμ a 64 kb/s. Il simbolo F[•] si riferisce alla conversione a legge A (o a legge μ) del segnale vocale.

L'acronimo G.711 identifica l'omonima raccomandazione dell'ITU-T ove sono specificati due formati di codifica: il PCMA ed il PCMμ. Comunque, il codice (ANSI

C) adoperato, per implementare la codifica a legge μ e la codifica a legge A, è liberamente distribuito dalla Sun Microsystem Inc. Quest'ultima è la versione adoperata per implementazione il sistema di comunicazione vocale discusso nel cap. IV. Il codice è disponibile in Appendice A.1.

In fig. III.1 si riporta un possibile diagramma a blocchi della codifica G.711. I valori caratteristici sono i seguenti: frequenza di campionamento (f_s) pari a 8 KHz, quantizzazione uniforme a 256 livelli, 8 bit per campione.

III.1.a Interfaccia RTP/G.711

In questa sezione è presentato il formato di pacchettizzazione RTP per il codificatore G.711. Il formato del payload in questione è quello implementato nel sistema di comunicazione vocale presentato nel cap. IV, e segue i suggerimenti disponibili in [28]; per comodità, nel par. III.3 si riporta la definizione dei PT.

Il formato del payload RTP è lo stesso per entrambi gli standard (PCMA e PCMμ) della raccomandazione. Ogni byte G.711 deve essere allineato ad ottetti nel payload RTP. Non ci sono limiti sul numero di byte per pacchetto, se non quelli imposti dalla rete. In ogni caso, secondo quanto osservato nel par. II.5.b, si consiglia di pacchettizzare 20 ms di voce, in altre parole 160 byte per pacchetto.

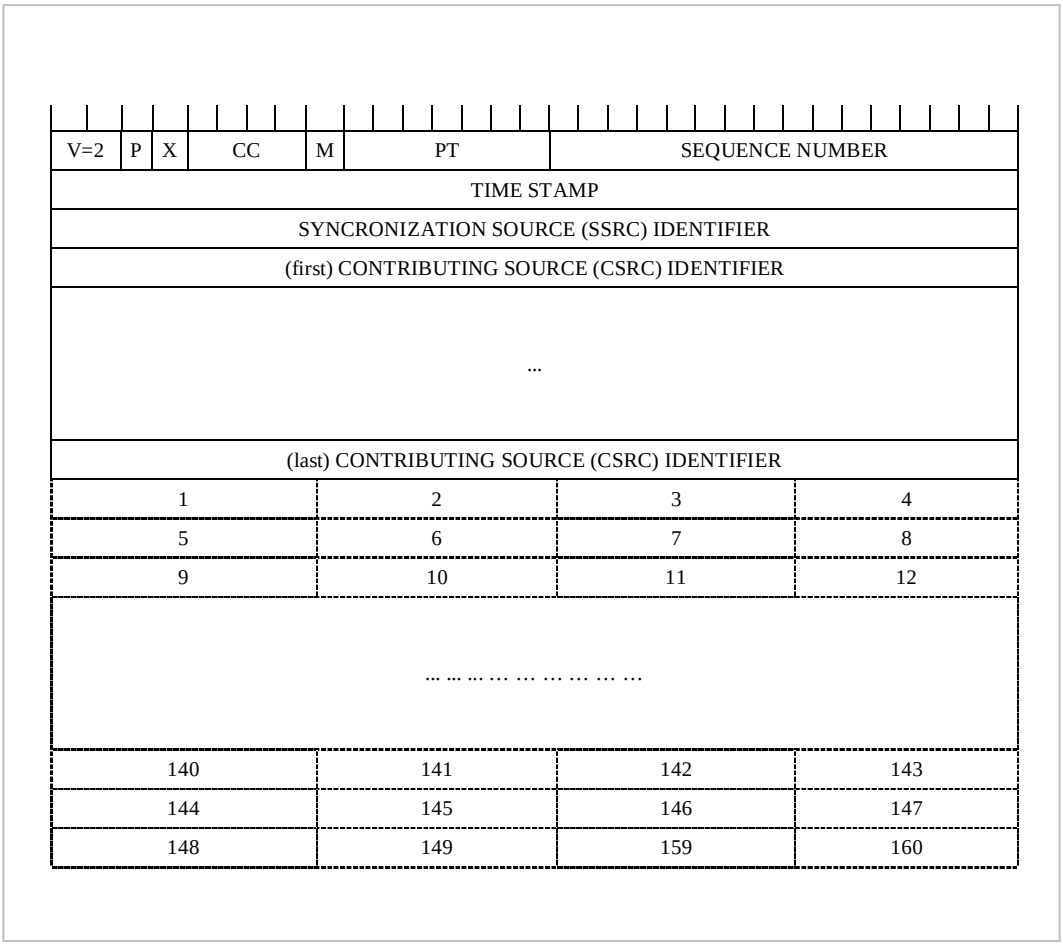


Fig. III.1.a. Intestazione RTP seguita dal formato di *payload* per il G.711 (PCMA, PCMμ).

La mappatura è riportata in fig. III.1.a. I parametri sono incapsulati nell'ordine in cui il codice ANSI C della raccomandazione li genera. La rappresentazione si riferisce a word di 32-bit ed ogni byte della word è trasmesso in ordine a partire dal più significativo. La convenzione usata nella rappresentazione è conforme all'ordinamento dei byte della rete IP: *big-endian*. Di questa convenzione e delle sue implicazioni si discuterà ampiamente nel cap. IV.

La convenzione adoperata per rappresentare i simboli in fig. III.1.a è molto semplice: ogni ottetto è numerato da 1 a 160 e rappresenta il generico campione della voce trasmessa.

Il formato del payload è sempre quello rappresentato in figura a meno di due eventualità:

- Il campo di *padding* (P) è pari ad uno.

  Ciò significa che non tutti i byte del payload sono campioni di voce. In tal caso, il numero di byte di riempimento è riportato nell'ultimo ottetto. Il controllo di quest'eventualità spetta all'applicazione.

- La lunghezza del payload è inferiore a 160 byte.

  Ciò significa che, per la particolare applicazione, si è scelto di incapsulare meno di 20 ms di voce. Anche in questo caso, il controllo di quest'eventualità spetta all'applicazione.

Prima di concludere, è opportuno precisare che il PCMA ed il PCMμ devono essere sempre trasmessi nella modalità ad 8 bit per campione, per cui le modalità a 56 Kb/s ed a 48 kb/s supportate dal G.711 non sono applicabili all'RTP.

III.2 Descrizione del G.729

L'acronimo G.729 si riferisce all'algoritmo di codifica vocale specificato nell'omonima raccomandazione ITU-T. Si tratta di un codificatore CS-ACELP (Coniugate Structure – Algebraic Code Excited Linear Prediction) ad 8 kbit/sec. Una versione a complessità ridotta è specificata nell'Annex A della raccomandazione. Entrambi gli algoritmi sono pienamente interoperabili, per questo non c'è nessuna ragione per distinguerli nel formato di pacchettizzazione, del quale si parlerà nella prossima sezione.

In fig. III.2.2 è mostrato un diagramma a blocchi dell'algoritmo di codifica, mentre in fig. III..2.3 è rappresentato l'algoritmo di decodifica. Osservando le figure è evidente come l'algoritmo sia molto articolato, di conseguenza, per non perdere di vista il fine cui sono rivolte queste, si eviterà di darne una descrizione dettagliata. Tuttavia, per comprendere a pieno le scelte adottate per il formato del payload RTP, è necessario dare una descrizione concettuale delle fondamenta che governano questo tipo di codifica.

Il principio generale di codifica è il CELP, che sfrutta il legame, ben noto in letteratura, tra la *predizione lineare* (LP) ed i *filtri autoregressivi* (AR). Il filtro predittivo e quello autoregressivo sono concettualmente distinti, tuttavia i rispettivi coefficienti sono legati dalla stessa equazione: l'equazione di Youle-Walker [31]. Combinando opportunamente i due modelli è possibile ottenere dei codificatori ad elevata compressione. Il principio è quello mostrato in fig. III.2.1, ove è rappresentata una versione semplificata del processo di codifica e di decodifica.

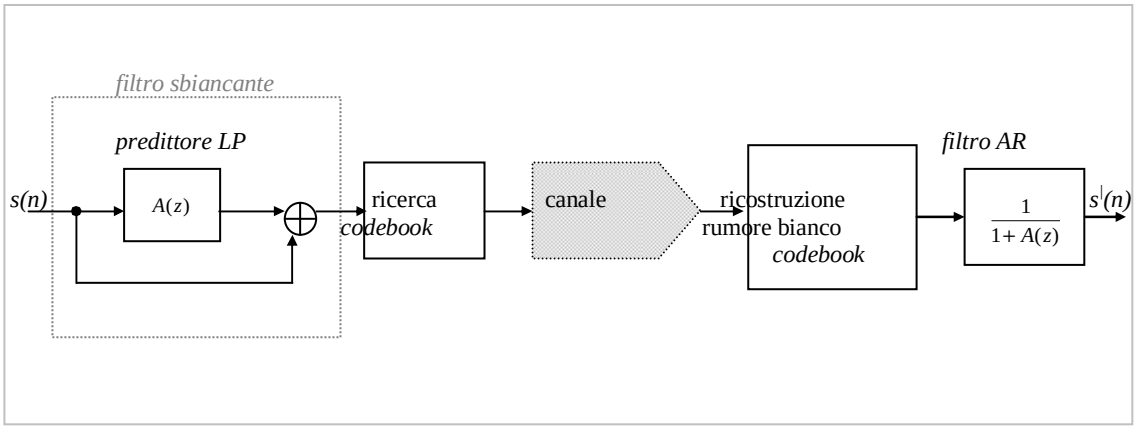


Fig. III.2.1. Diagramma a blocchi (**versione semplificata**) degli algoritmi di codifica e di decodifica del CS-ACELP G.729.

Per descrivere l'algoritmo conviene partire dal decodificatore. Gli elementi essenziali sono: il filtro autoregressivo ed il codebook.

La caratteristica peculiare del modello autoregressivo[18] è quella che esso generi il processo aleatorio in esame a partire da un *rumore bianco*[19] in ingresso. I parametri del modello sono stimati a partire dalle conoscenze che si hanno sul processo. Più precisamente si ammette che esista un filtro IIR (Infinite Impulse Response) che avendo in ingresso un rumore bianco dia in uscita i campioni misurati [*s(0), s(1),…, s(N-1)*]. I coefficienti di tale filtro sono stimati usando o direttamente i dati misurati (es. metodo di Burg [31]) o i campioni stimati della funzione di autocorrelazione (FAC) (es., metodo dell'autocorrelazione). L'ipotesi fondamentale su cui si basa tale modello è che il processo sotto esame sia stazionario. Alla luce di ciò bisogna tenere conto che il segnale vocale è stazionario solo per brevi tratti (circa 10 - 30 msec.). Per questo motivo, dell'intero segnale a disposizione, bisogna considerare tratti di lunghezza opportuna (trame o frame). La modellizzazione AR va eseguita su ogni singola trama ed in questo senso va condotta una sorta d'analisi del tipo"tempo-frequenza", in modo da evitare di violare la necessaria ipotesi di stazionarietà del processo in esame.

In sostanza, avendo a disposizione un opportuno rumore bianco in ingresso al filtro AR, teoricamente è possibile ottenere in uscita il segnale vocale originale. Il fatto è che, per ottenere in uscita al filtro AR il segnale desiderato, non va bene un qualsiasi rumore bianco, bensì quello che è in grado di eccitare il filtro opportunamente, poiché contiene in sé le caratteristiche peculiari del segnale vocale che si vuole ricostruire. Dunque, è il segnale vocale stesso a dover dire quale rumore è in grado di ricostruirlo. A questo punto entra in gioco la predizione lineare come parte principale del processo di codifica.

L'ipotesi di stazionarietà è necessaria anche nel modello predittivo. La predizione lineare fa riferimento ad una serie aleatoria e si pone il problema di predire il generico campione in funzione di N campioni precedenti. Si può dimostrare che il segnale che fuoriesce dal filtro predittivo sommato al segnale stesso in ingresso da luogo ad un rumore bianco (in letteratura esistono numerosi test di bianchezza, a titolo d'esempio si segnala il "test di Anderson" [32]). Tale rumore contiene in se alcune caratteristiche spettrali del segnale vocale che lo ha generato per questo è il candidato ideale ad eccitare il filtro AR in ricezione. Inoltre, i coefficienti di predizione possono essere stimati in modo da minimizzare, in senso statistico, il quadrato della differenza tra il valore predetto e quello vero. Le equazioni che si ottengono sono dello stesso tipo di quelle del modello AR. Di conseguenza è possibile dimostrare che, attribuendo al filtro AR gli stessi parametri del filtro predittivo, l'elaborazione che il rumore bianco subisce in fase di decodifica è esattamente l'opposto di quella che lo ha generato.

Se la quantità di bit richiesta per rappresentare i parametri del filtro LP ed il rumore bianco è minore di quella necessaria a rappresentare il segnale vocale originale si è realizzato un codificatore. I codebook, presenti sia in fase di codifica sia di decodifica, servono proprio a questo.

[18] Per una descrizione più approfondita si rimanda a [23].

[19] Per "rumore bianco" s'intende un segnale il cui spettro di densità di potenza è costante, in altre parole l'autocorrelazione dei campioni del segnale è un impulso. Solitamente, i campioni di un segnale siffatto hanno distribuzione Gaussiana a media nulla.

Il codebook contiene una serie di possibili sequence di rumore bianco. In questa sede non entreremo nel merito dell'implementazione del codebook nel G.729, poiché, concettualmente non aggiunge nulla di nuovo. Per quanto riguarda il fine delle presenti note basterà sapere che, in qualche modo, in fase di codifica vengono scelti degli indici che corrispondono al miglior vettore che il decodificatore userà per eccitare il filtro AR. Questa scelta deriva dal fatto che per rappresentare gli indici piuttosto che il rumore stesso sono necessari meno bit. Tuttavia questa convenienza va pagata in termini di qualità della voce generata. Infatti il numero di sequenze eccitatorie estraibili dal codebook è limitato per cui non è possibile ottenere la sequenza ottima per ogni segnale vocale da codificare.

Questo tipo di codifica è conosciuto in letteratura con l'acronimo AbS (*Analisys by Sintesys*). In effetti, a ben guardare, la codifica è realizzata attraverso un processo d'analisi (la predizione lineare) ed un successivo processo di sintesi (il modello autoregessivo). L'analisi LP permette di disporre di un numero limitato di parametri in grado di caratterizzare le proprietà statistiche del segnale vocale. I parametri in questione, una volta quantizzati e trasmessi, sono in grado di eccitare opportunamente il filtro AR che rigenera il segnale vocale originale, o meglio una sua stima.

Detto ciò, nel seguito si riportano le principali caratteristiche tecniche del G.729. Informazioni aggiuntive sono disponibili in [33]-[36].

Il codificatore CS-ACELP, rappresentato in fig. III.2.2, è progettato per operare con un segnale strettamente limitato in banda e campionato a 8000 Hz. In natura nessun segnale e limitato in banda, quindi il segnale vocale previsto in ingresso è il risultato di un prefiltraggio analogico anti-aliasing.

Sia i campioni in ingresso che quelli in uscita sono rappresentati attraverso la quantizzazione lineare PCM a 16 bit/campione. Il codificatore opera su trame da 10 ms.

I campioni PCM in ingresso vengono filtrati con un filtro passa-alto con frequenza di taglio (*cutoff frequency*) pari a 140 Hz. Questa pre-elaborazione serve ad eliminare la continua e le componenti a bassa frequenza. I campioni in uscita sono indicati con *s(n)*.

L'analisi basata sui principi della predizione lineare è fatta ogni volta per ogni trama vocale. Il metodo adoperato è quello conosciuto come metodo dell'autocorrelazione [31], utilizzando una finestra temporale [37] asimmetrica composta da due parti: la prima parte è mezza funzione di *Hamming*, la seconda è in quarto di funzione *coseno*. La finestra è lunga 30 ms (240 campioni) ed è data da:

$$w_{lp} = \begin{cases} 0.54 - 0.46\cos\left(\dfrac{2\pi n}{399}\right) & n = 0,...,199, \\ \cos\left(\dfrac{2\pi(n-200)}{159}\right) & n = 200,...,239. \end{cases}$$

Le "stime" dei coefficienti (a_i con i = 1,…, 10) del filtro LP, il cui ordine è 10, sono ottenute attraverso l'equazione (III.2.1) di Youle-Walker:

$$\hat{R}_{tr}(k) = -\sum_{i=1}^{p} \hat{a}_i \hat{R}_{tr}(k-i) + \sigma_u^2 \delta(k) \qquad \begin{matrix} k = 0,1,\ldots, p \\ p = 10 \end{matrix} \qquad \text{(III.2.1)}$$

Il significato dei simboli della (III.2.1) è il seguente: il pedice"tr" indica la generica "trama", il simbolo "^" sta per stima, mentre il pedice "u" si riferisce al rumore bianco eccitatorio. I parametri (a_i e σ^2) presenti nell'equazione (III.2.1) sono ottenuti attraverso l'algoritmo di Levinson-Durbin, il quale permette di abbassare il costo computazionale da $o(p^2)$ a $o(p^3)$, ove "p" rappresenta l'ordine del modello. Con il simbolo $\hat{R}_{tr}(k)$ s'intende la "stima" del generico campione della funzione di autocorrelazione del segnale finestrato $w_{lp}(n)s(n)$.

I campioni necessari all'analisi LP sono ripartiti nel seguente modo (vedi fig. III.2.4): 120 campioni del segnale passato, 80 campioni del segnale attuale, 40 campioni del segnale futuro. I 40 campioni del segnale futuro aggiungono 5 ms extra di ritardo all'algoritmo di codifica.

Successivamente, i coefficienti del filtro LP vengono convertiti nei coefficienti LSF (Line Spectral Frequencies) e quantizzati vettorialmente (LSP: Line Spectrum Pairs) con 18 bit [35].

La trama d'ingresso è suddivisa in due *sottotrame* da 5ms che sono elaborate separatamente. Per la seconda sottotrama sono utilizzati sia i coefficienti LP quantizzati sia quelli non quantizzati, mentre per la prima si utilizzeranno i coefficienti LP interpolati.

Le questioni successive riguardano il codebook. Per ogni sottotrama l'eccitazione è rappresentata dal contributo proveniente da un codebook-adattivo e da un codebook-fisso. I parametri relativi ai codebook sono trasmessi separatamente per ogni sottotrama.

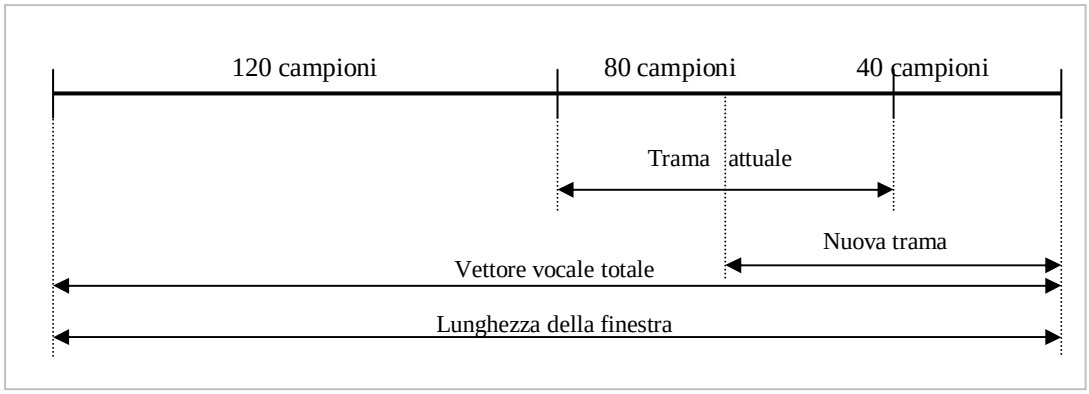


Fig. III.2.4. Suddivisione delle trame vocali nell'analisi LP.

Gli indici del codebook-adattivo si ottengono attraverso una doppia procedura:

- *Ricerca a ciclo aperto*. Riguarda un'analisi sull'intera trama. Il pitch si stima in base deduzioni sul segnale vocale sbiancato.

- *Ricerca a ciclo chiuso*. Riguarda un'analisi su ogni sottotrama. Gli indici (P_1, P_2) del codebook-adattivo ed i guadagni (F_1, G_1, F_2, G_2).

Gli indici P_1, P_2 sono quantizzati rispettivamente con 8 bit (prima sottotrama) e con 5 bit (la seconda sottotrama); mentre i guadagni F_1, G_1, F_2, G_2 rispettivamente con 3, 4, 3, 4 bit. Il codebook-fisso è un "algebraic-codebook" a 17 bit (13 per C_1 e C_2, 4 per S_1 ed S_2).

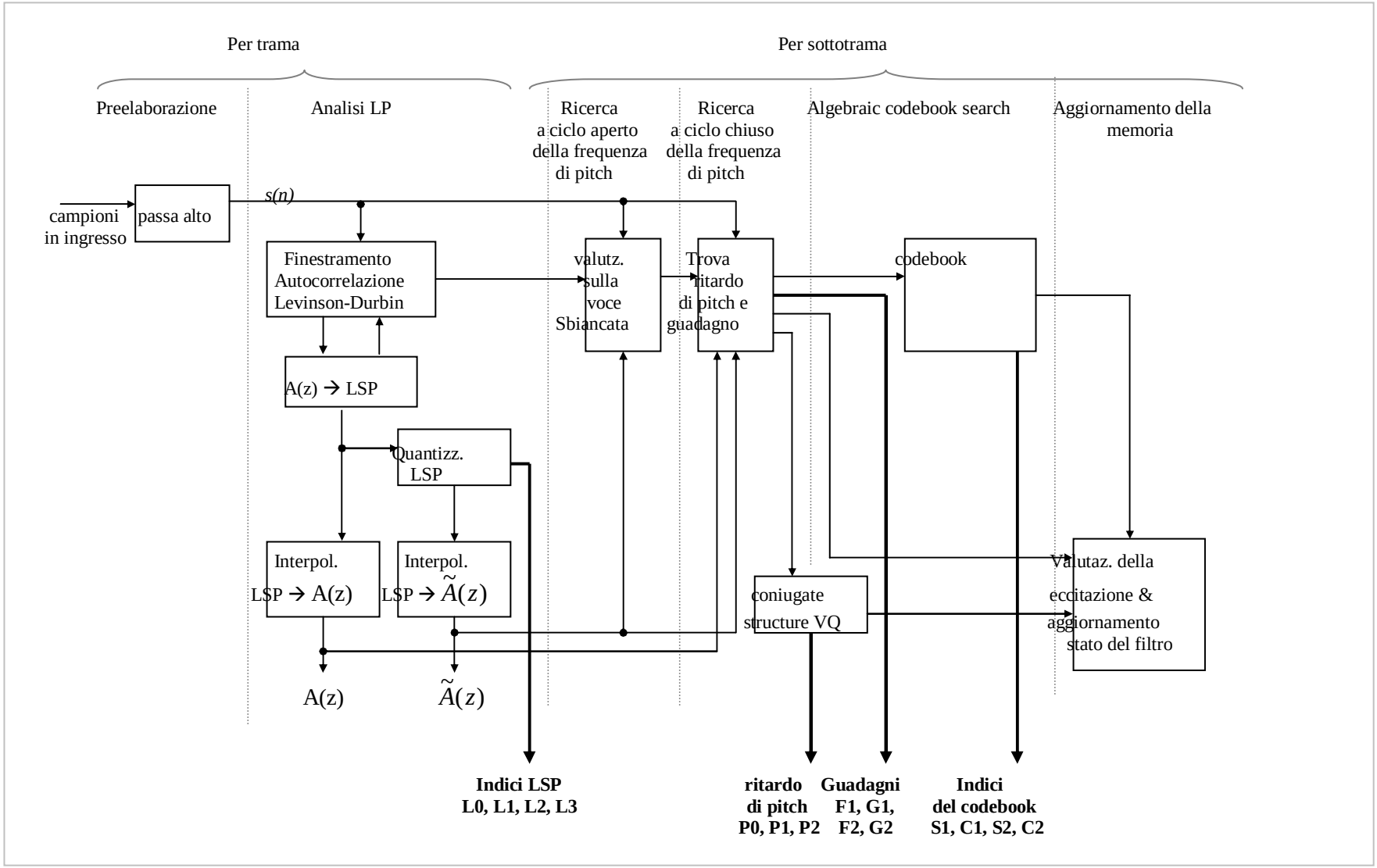


Fig. III.2.2 Diagramma a blocchi del codificatore CS-ACELP G.729.

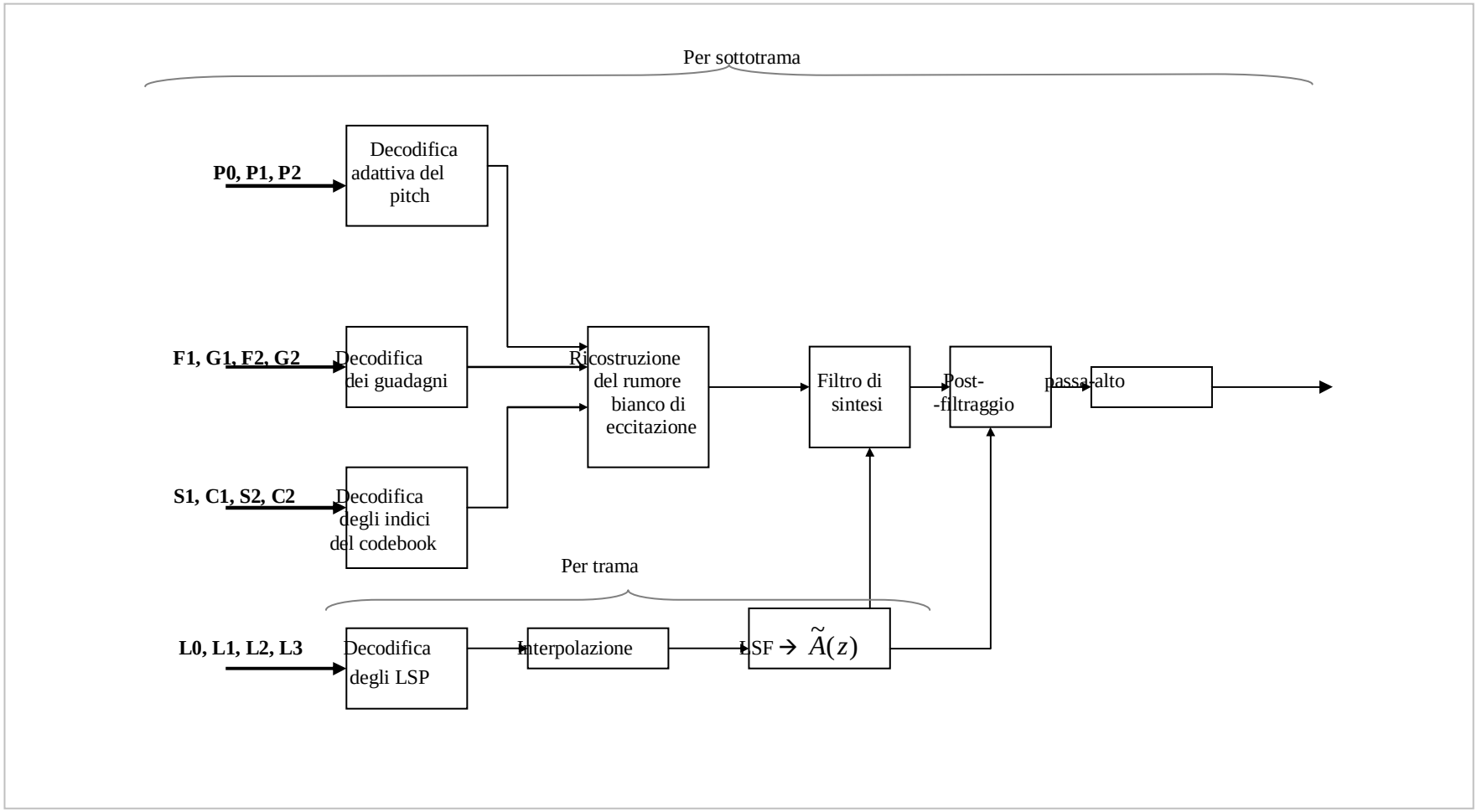


Fig. III.2.3. Diagramma a blocchi del decodificatore CS- ACELP G.729.

L’allocazione dei bit relativa ad ogni sottotrama è mostrata in tab. III.2.

Parametri	*Codeword*	*sottotrama 1*	*sottotrama 2*	*Totale Bit*
Coeff. LSP	L_0, L_1, L_2, L_3			18
Ritardo del codebook-adattivo	P_1, P_2	8	5	13
Bit di parità del ritardo (P_1)	P_0	1		1
Indice del codebook-fisso	C_1, C_2	13	13	26
Segno del codebook fisso	S_1, S_2	4	4	8
Guadagno del codebook (passo 1)	F_1, F_2	3	3	6
Guadagno del codebook (passo 2)	G_1, G_2	4	4	8
TOTALE				**80**

Tab. III.2. Allocazione dei bit per 10 ms di trama vocale elaborata dal G.729 CS-ACELP.

La funzione del decodificatore, vedi fig. III.2.3, è quella di decodificare i parametri trasmessi secondo la seguente modalità:

- Gli indici del codebook-adattivo, gli indici del codebook-fisso ed i guadagni servono per ricostruire il segnale eccitatorio.

- I parametri LP servono per mettere in piedi il filtro di sintesi.

Le elaborazioni successive, post-filtraggio adattivo e passa-alto servono a migliorare la qualità della voce rigenerata.

III.2.a Interfaccia RTP/G.729

In questa sezione è discusso il formato di payload RTP consigliato in [28] per il codificatore G.729; per comodità, nel par. III.3 si riporta la definizione dei PT. Il formato di pacchettizzazione in questione è quello implementato nel sistema di comunicazione vocale presentato nel cap. IV.

Come accennato in precedenza, il G.729 Annex A [38] è una versione la cui complessità è ridotta del 40%, circa, rispetto al G.729. Ciò è dovuto ad una semplificazione delle ricerche nel codebook-fisso ed adattivo. La riduzione della complessità è pagata in termini di qualità della voce che si riduce leggermente. Gli algoritmi presenti nelle raccomandazioni G.729 e G.729 Annex A sono pienamente interoperabili, perciò, nel seguito, non si farà alcuna distinzione.

Si è già detto che il G.729/G.729A lavora su trame da 10 ms, ed in alcune applicazioni real-time (ad esempio, nel caso sia presente un VAD ed un CNG) si invia proprio una trama per volta. Tuttavia l’intervallo di pacchettizzazione consigliato per le applicazioni VoIP è di 20 ms (vale a dire due trame per pacchetto RTP).

In ogni caso, i parametri trasmessi sono quelli elencati nella tab. III.2 e la loro mappatura, riferita a 20 ms, è riportata in fig. III.2.a. I parametri sono incapsulati nell'ordine in cui il codice ANSI C della raccomandazione li genera. La rappresentazione si riferisce a word di 32-bit ed ogni byte della word è trasmesso in ordine a partire dal più significativo. La convenzione usata nella rappresentazione è conforme all'ordinamento dei byte della rete IP: *big-endian*. Di questa convenzione e delle sue implicazioni si discuterà ampiamente nel cap. IV.

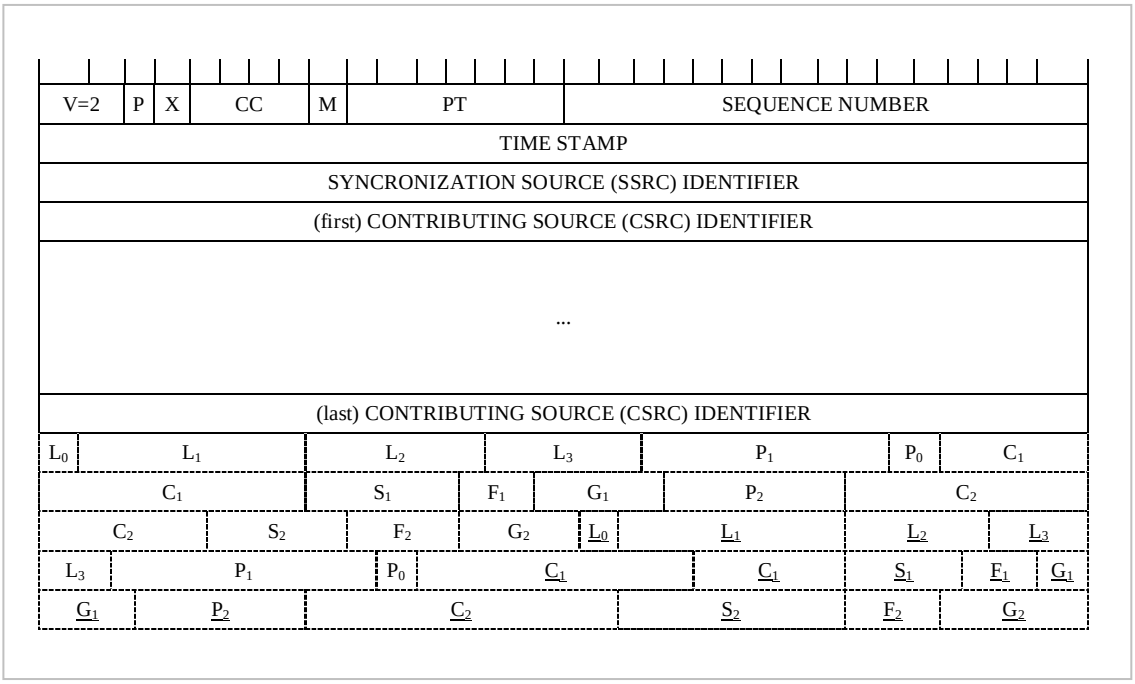


Fig. III.2.a. Intestazione RTP seguita dal formato di *payload* per il G.729/G.729A.

La convenzione adoperata per rappresentare i simboli in fig. III.2.a è la seguente. I parametri non sottolineati si riferiscono alla codifica di 10 ms di trama vocale, mentre quelli sottolineati si riferiscono ai 10 ms successivi. La suddivisione verticale tratteggiata indica il contenuto informativo dei bit secondo la tab. III.2, ma il payload è diviso in word da 4 byte ciascuna. In totale la lunghezza del payload è di 20 byte.

Confrontando la fig. III.2.a con la fig. III.1.a è evidente il guadagno offerto dal G.729/G.729A in termini di banda occupata (solo 20 byte contro i 160 del G.711).

Una questione di fondamentale importanza è quella relativa al *padding*. Infatti, va tenuto presente che il G.729/G.729A lavora su trame da 10 ms, quindi la risoluzione sulla cattura dell'informazione è proprio di 10 ms, quindi, c'è la possibilità di perdere una minima parte dell'informazione. Ad esempio, nel caso si debbano comprimere 3,658 sec. di voce si dovrebbero analizzare 365,8 trame da 10 ms; ciò significa che gli ultimi 8 ms non vengono presi in considerazione, in quanto di dimensione insufficiente all'analisi. Ciò significa che nel payload non potrà mai esserci zero-padding, visto che la sua lunghezza (in byte) è sempre un multiplo di 10.

Prima di concludere, vale la pena anticipare che il codice sorgente (in ANSI C) fornito nella raccomandazione G.729/G.729A non è nella forma adatta al tipo di pacchettizzazione proposto per questo necessita numerose modifiche. Di ciò si parlerà ampiamente nell'Appendice A.3.a.

III.3 Mappa dei Payload Type per l'RTP.

Nella tab. III.3 è riportato il mapping di riconoscimento dei codificatori.

PT	codificatore	dati	f_s (Hz)	canali
0	PCMU	A	8000	1
1	1016	A	8000	1
2	G726-32	A	8000	1
3	GSM	A	8000	1
4	G723	A	8000	1
5	DVI4	A	8000	1
6	DVI4	A	16000	1
7	LPC	A	8000	1
8	PCMA	A	8000	1
9	G722	A	8000	1
10	L16	A	44100	2
11	L16	A	44100	1
12	QCELP	A	8000	1
13	reserved	A		
14	MPA	A	90000	
15	G728	A	8000	1
16	DVI4	A	11025	1
17	DVI4	A	22025	1
18	G729	A	8000	1
19	reserved	A		
20	unassigned	A		
21	unassigned	A		
22	unassigned	A		
23	unassigned	A		
dyn	G729D	A	8000	1
dyn	G729E	A	8000	1
dyn	GSM-HR	A	8000	1
dyn	GSM-EFR	A	8000	1
dyn	L8	A	var.	var.
dyn	RED	A		
dyn	VDVI	A	var.	1
24	unassigned	V		X
25	CelB	V	90000	X
26	JPEG	V	90000	X
27	unassigned	V		X
28	nv	V	90000	X
29	unassigned	V		X
30	unassigned	V		X
31	H261	V	90000	X
32	MPV	V	90000	X
33	MP2T	A/V	90000	X
34	H263	V	90000	X
35-71	unassigned	?		X
72-76	reserved	N/A	N/A	X
77-95	unassigned	?		X
96-127	dynamic	?		X
dyn	BT656	V	90000	X
dyn	H263-1998	V	90000	X
dyn	MP1S	V	90000	X
dyn	MP2P	V	90000	X
dyn	BMPEG	V	90000	X

Tab. III.3. Payload Type per codificatori audio e video.

Le considerazioni fatte sul PT dei formati di codifica G.711 e G.729 sono inquadrabili all'interno del profilo A/V suggerito in [28], a cui si riferisce la tab. III.3.

I valori presenti nella colonna PT sono i valori del campo *payload type* dell'intestazione RTP. Attraverso di essi è possibile riconoscere il particolare formato di codifica del payload. Buona parte di tali valori è assegnata. L'intervallo 96-127 può essere sfruttato per una definizione dinamica durante una sessione RTP, ad esempio, attraverso un conference control protocol. Ciò può essere utile nel caso si usino formati di codifica che non rientrano nel campo fisso (es., PCMU a 8000 Hz stereo, quindi 2 canali). Il payload type 13 ed il 19 sono riservati per il formato di comfort noise (da specificare in un RFC). I payload type "dyn" indicano i PT dinamici.

La colonna "tipo di dati" si riferisce al flusso di dati trasportabili, per cui, a parte l'MP2T, i flussi differenti devono avere PT differenti anche all'interno della stessa sessione.

Le indicazioni del profilo A/V suggerite in [28] sanciscono ancora una volta la proprietà di scalabilità dell'RTP.

Capitolo IV

Aspetti progettuali ed implementativi di un reale sistema di comunicazione vocale VoIP.

Il progetto di un sistema di comunicazione VoIP impone la messa a punto di una serie di elaborazioni su segnali eterogenei: dati, voce e video. Nei precedenti capitoli si è visto quali sono gli aspetti teorici cui fare riferimento per implementare un sistema VoIP. Nel presente capitolo, facendo frutto delle precedenti considerazioni, sarà presentato un reale sistema di comunicazione vocale tempo reale nella modalità PC-to-PC. Il sistema in questione sarà introdotto gradualmente, partendo da un progetto teorico di più ampio respiro. L'intento è di affrontare in modo critico i vari argomenti, esposti nei precedenti capitoli, selezionando le questioni strettamente legate ai problemi implementativi. Per questa ragione, non mancheranno numerosi riferimenti alle più note tecniche di *networking*.

Il capitolo è organizzato nel seguente modo.

Nel par. IV.1 sarà discusso un sistema VoIP multiservizio. Nel par. IV.2 saranno discusse le questioni di comunicazione tempo reale in ambiente integrato. Particolare riguardo è dedicato agli aspetti di multiplexing dell'Input/Output, in particolare saranno esposte in dettaglio le questioni relative alla sincronizzazione della molteplicità degli ingressi. Nel par. IV.3 si considereranno gli aspetti strettamente legati all'interfaccia di rete, dando particolare enfasi al legame "applicazione – protocollo UDP". Inoltre si considererà l'applicabilità del modello client/server alle comunicazioni VoIP. Infine, nel par. IV.4 sarà descritto il sistema di comunicazione realizzato. Il progetto è implementato attraverso un software in grado di mettere in comunicazione più macchine connesse in rete. Le caratteristiche di tale sistema verranno approfondite e motivate, dando numerosi spunti per aumentarne le potenzialità.

IV.1 Progetto in modalità PC-to-PC.

Le considerazioni fatte nei precedenti capitoli hanno evidenziato quanto possa essere complesso il progetto di un sistema di voce su IP (come servizio integrato da voce, dati e video). La principale difficoltà è di garantire un trasporto A/V, con un'adeguata QoS, su una rete intrinsecamente inaffidabile. Inoltre, riuscire a far convivere segnali eterogenei che si appoggiano su differenti architetture protocollari non è cosa da poco. Un ulteriore problema è quello legato al sincronismo dei vari segnali, in un canale, la rete IP, ove la variabilità di risorse è del tutto aleatoria. Di conseguenza, il progetto di un sistema VoIP richiede lo sviluppo di un software che preveda funzionalità d'elaborazione di segnali di varia natura, funzionalità di controllo, un'interfaccia utente e quant'altro.

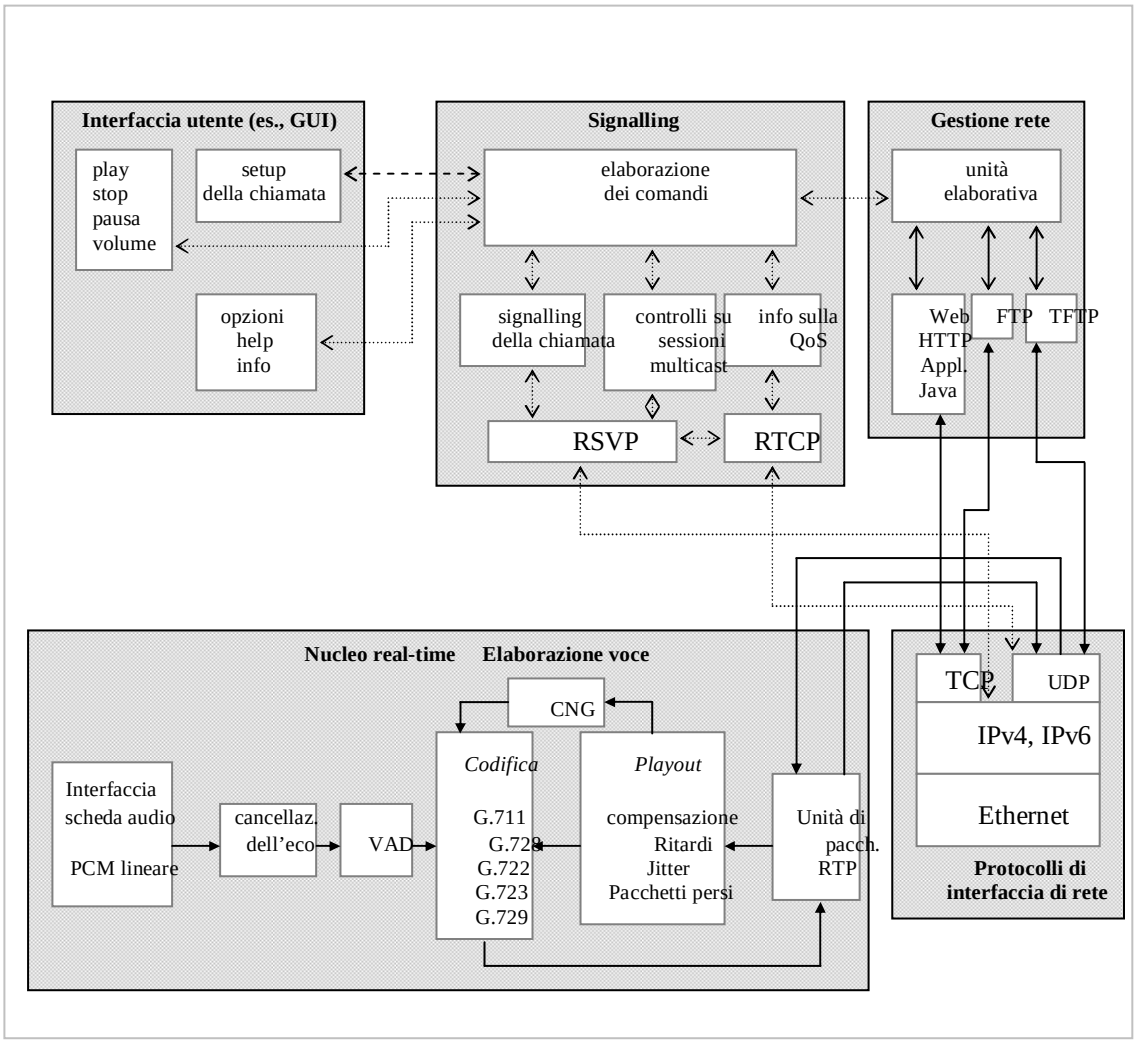


Fig. IV.1. Architettura software di un sistema di comunicazione VoIP. Modalità PC-to-PC.

Tutte le considerazioni fatte nei precedenti capitoli sono sintetizzate nella rappresentazione di fig. IV.1, ove è mostrata una possibile architettura software relativa

al trasporto di dati e voce. Per esigenza di sintesi, non sono state prese in considerazione le unità elaborative rivolte alla compressione ed alla trasmissione delle immagini. La convenzione adottata per la rappresentazione è la solita. Le frecce tratteggiate indicano i flussi di controllo, mentre quelle continue i flussi di dati. La rappresentazione protocollare è in sintonia con la *suite UDP/IP* mostrata in fig. II.1.3, ed a tale proposito è opportuno porre l'accento su alcune questioni:

- L'unità di pacchettizzazione RTP è rappresentata nel blocco di elaborazione della voce.

  Si è già detto che la pacchettizzazione RTP è strettamente legata al particolare codificatore in uso. In particolare nel cap. III si è visto un esempio relativo due formati di compressione: il PCM e il CS-ACELP. Per questa ragione, è preferibile generare il payload RTP direttamente nell'algoritmo di codifica. L'interfaccia RTP/codificatore varia a seconda del tipo di codifica, mentre l'interfaccia RTP/UDP è sempre la stessa, per cui si è ritenuto opportuno enfatizzare quest'aspetto.

- Il protocollo RTCP è rappresentato nel blocco di signalling.

  Nonostante l'RTP e l'RTCP facciano parte della stessa specifica, la loro implementazione e nettamente indipendente. L'RTCP svolge un compito di monitoraggio della QoS e d informazione sui partecipanti ad una sessione VoIP, mentre l'RTP è dedicato al trasporto dei dati real-time. Il loro legame è relativo ad alcune informazioni comuni presenti nelle rispettive intestazioni (es., SSRC, time stamp) ed alla loro cooperazione nel l real-time.

- Il TFTP (Trivial File Transfer Protocol) si affida all'UDP.

  Nei precedenti capitoli, si è posto l'accento sul fatto che l'UDP non dà alcuna garanzia sul trasporto dei dati, affermando che per tale ragione non è adatto al trasferimento di grosse quantità di dati, come un file. Il TFTP è un'eccezione alla regola. L'UDP è stato scelto poiché più semplice da implementare rispetto al TCP (800 righe di codice C "bootstrap" anziché 4500), e poiché il TFTP è utilizzato solo in modalità bootstrap nelle LAN, mai nelle WAN. In ogni caso, il TFTP include di per sé funzionalità di sequenziamento,di acnowledgements, di timeout e di ritrasmissione.

E' chiaro che un buon progetto non può non tenere conto dei problemi di interoperabilità. A proposito di ciò, vale la pena precisare che, seguendo le linee guida degli standard dominanti nel settore VoIP (H.323 e SIP), alcuni elementi rappresentati in fig. IV.1 sono necessari, mentre altri sono opzionali. In particolare, per garantire un'interoperabilità minima tra le varie soluzioni architetturali proposte, sono necessari almeno i seguenti elementi: l'RTP, l'RTCP ed il G.711.

Tutto ciò evidenzia come l'implementazione di un sistema VoIP richieda un software altamente sofisticato ed articolato.

All'interno dello scenario rappresentato in fig. IV.1 è inquadrabile il sistema di comunicazione vocale cui sono rivolte le presenti note. Tale sistema, che d'ora in poi, per comodità, indicheremo con l'acronimo *RTP-Speak*, sarà presentato nel par. IV.4. In ogni caso, in questo capitolo, non mancheranno riferimenti espliciti ad *RTP-Speak*. Tali riferimenti, serviranno a dare maggior enfasi alle varie considerazioni teoriche e, contemporaneamente, a motivare le scelte adottate per l'implementazione di *RTP-Speak*.

Nel seguito, saranno esposte in dettaglio le fondamenta su cui si basa un sistema di comunicazione vocale tramite reti di calcolatori, qual è *RTP-Speak*. In particolare, si cercherà di chiarire come, attraverso un progetto software, si possa instaurare una comunicazione tra due o più calcolatori connessi in rete. Nel corso del capitolo, saranno presi in considerazione i vari aspetti schematizzati in fig. IV.1.

IV.2 Comunicazione real-time in ambiente integrato. Problematiche di progetto.

Com'è noto, una rete di calcolatori è un insieme di computer autonomi in grado di scambiarsi informazioni (*ambiente integrato*). Le possibili topologie di rete sono tante, in ogni caso, per esigenze di generalità si prescinderà dalla particolare topologia di rete. In particolare, si farà riferimento a due sole macchine interconnesse, esplicitando la particolare topologia di rete solo ove sia necessario.

In fig.IV.2.1 si considerano due host appartenenti alla stessa LAN (Local Area Network). Questa figura sintetizza numerosi concetti che saranno la base delle successive considerazioni. Innanzi tutto, le due macchine sono rappresentate "a strati" secondo la convenzione progettuale d'ogni rete di calcolatori. Come messo in evidenza nel par. II.1, le due macchine comunicano "virtualmente" attraverso processi paritari (linea tratteggiata) ma la reale modalità di comunicazione è un'altra (linea continua).

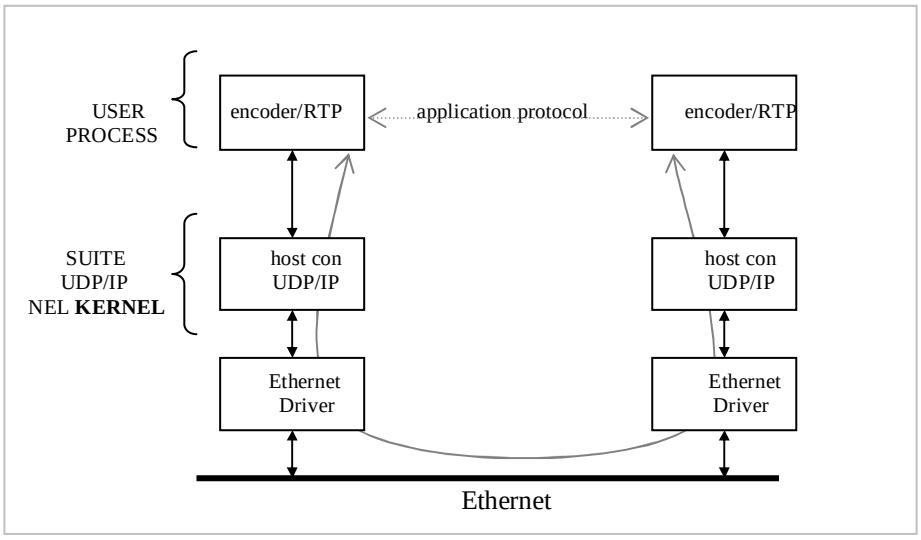


Fig.IV.2.1. Due macchine sulla stessa Ethernet comunicanti attraverso l'UDP.

In fig.IV.2.1 è messo in evidenza come l'applicazione sia un processo utente (User Process) mentre la suite UDP/IP faccia parte del *kernel* della macchina.

Il kernel è il nucleo [39] del Sistema Operativo, ed ha la caratteristica di essere sempre presente in memoria centrale poiché è fatto di programmi che una volta invocati vanno eseguiti immediatamente. La priorità necessaria alle operazioni di I/O (Input/Output) fa sì che, normalmente, i protocolli IP e UDP (anche il TCP) facciano parte del kernel. Nel par. IV.3 vedremo proprio come il progettista può interfacciarsi con il kernel dalla macchina attraverso opportune chiamate di sistema.

La precedente figura può essere facilmente generalizzata considerando più host connessi alla stessa LAN, più LAN connesse alla stessa WAN e così via fino all'intera rete d'Internet. In fig.IV.2.2 si riporta una possibile topologia di rete su cui instaurare la comunicazione. D'ogni macchina è opportuno conoscere almeno, il tipo di sistema

operativo e l'indirizzo IP. La sottorete 1 ha una topologia a stella, mentre le sottoreti 2 e 3 a bus.

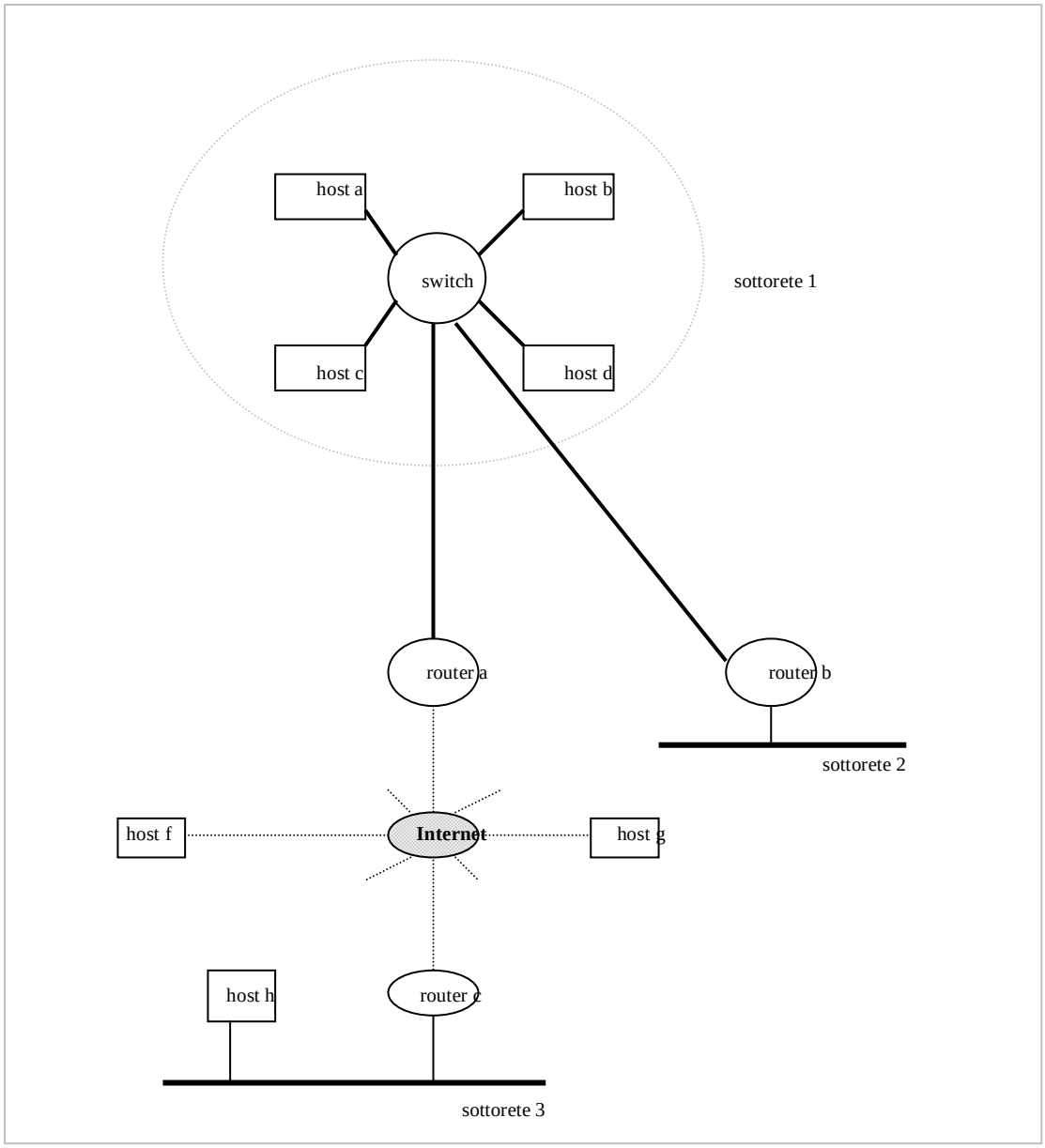


Fig. IV.2.2. Esempio di più reti interconnesse

Gli aspetti di cui tenere conto nel realizzare un'applicazione che vada oltre la propria LAN, sono molti, consideriamone alcuni. Si può subito dire che, al progettista basta fare riferimento a due o più macchine appartenenti alla propria LAN per rendere il progetto funzionante anche all'interno dell'intera Internet. Infatti, una volta scelto il protocollo standard di trasporto, sarà poi preoccupazione dei router o dei bridge colmare i vuoti tra reti eterogenee. Ciò fa pensare alla possibilità di un progetto estremamente versatile. In ogni caso, è importante non trascurare la compatibilità tra il software e i diversi tipi di SO. Infatti, come vedremo nel par. IV.3, il sistema di comunicazione che si vuole implementare necessita d'opportune chiamate al SO. Per rendere il progetto

universalmente compatibile è necessario scrivere un software che si adatti ai vari SO esistenti, oppure più software interoperabili. Per il momento ci limiteremo a considerare la stesura di un software che sia idoneo ad ogni piattaforma POSIX-compatibile. L'acronimo POSIX [40] sta per Portable Operating System Interface, si tratta di un progetto che definisce l'interfaccia standard del C basata sul SO UNIX. Inoltre, va tenuto presente che quando si fa un progetto sfruttando "standard proposti", non ancora universalmente utilizzati (es., l'RSVP), ci si deve preoccupare di implementarli in tutti i nodi della rete coinvolti nella comunicazione. Nonostante ciò, non è detto che le cose vadano sempre bene. Ad esempio, un'applicazione multicast che funziona benissimo in una WAN può non andare bene in un'altra, poiché non tutti i router supportano il multicast.

IV.2.a Modello client/server ed I/O multiplexing in ambiente VoIP.

L'intento di questo paragrafo è di introdurre le fondamenta di un'applicazione client/server in grado di trasportare voce su reti IP, in particolare verranno presi in considerazione gli aspetti legati all'Input/Output.

La programmazione di rete coinvolge la scrittura di programmi che dialogano con altri programmi lungo una rete di computer, uno di essi e solitamente chiamato *client* e gli altri *server*. Gran parte dei SO supportano una serie di programmi precompilati che comunicano lungo una rete seguendo la modalità *client/server* (es., l'FTP, il TELNET, e quant'altro). Un client è un programma che gira sull'host che fa una richiesta (*request*), mentre un server è un programma che gira sull'host remoto che assolve la richiesta (*response*). Solitamente un client comunica con un server alla volta mentre un server, in ogni momento, comunica con più client. Va tenuto presente che non è necessario che il client ed il server girino su host differenti, infatti, talvolta, identificano due diversi processi appartenenti allo stesso host.

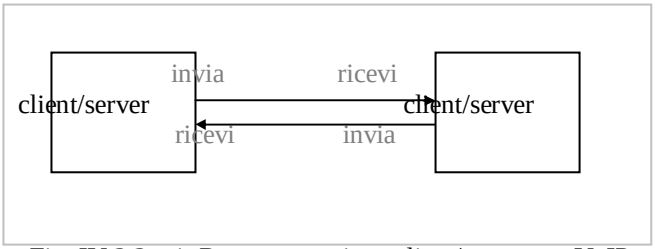


Fig. IV.2.2.a.1. Rappresentazione client/server per VoIP.

Alla luce di ciò, è possibile inquadrare nell'ottica client/server un'applicazione di voce su IP. Si può facilmente immaginare che, in un'applicazione tempo-reale in modalità *full-duplex*, ogni host coinvolto nella comunicazione sia contemporaneamente client e server. Ciò è schematicamente illustrato in fig. IV.2.2.a.1.

In fig. IV.2.2.a.1 sono rappresentati due host in comunicazione point-to-point, ma la cosa è banalmente generalizzabile a più host in comunicazione broadcast o multicast.

Tenendo presente quanto detto nei precedenti paragrafi, d'ora in poi, si supporrà che il flusso di dati real-time sia trasmesso attraverso l'incapsulazione RTP/UDP/IP, mentre i segnali di controllo attraverso la catena protocollo-di-signalling/TCP/IP e RTCP/UDP/IP.

Detto ciò, consideriamo l'aspetto full-duplex di un'applicazione VoIP. Com'è noto, in una comunicazione full-duplex si è in grado di trasmettere e, contemporaneamente, ricevere attraverso lo stesso canale. La tradizionale comunicazione telefonica attraverso la rete PSTN ne è un esempio. Inoltre, va tenuto presente che le sorgenti di I/O sono almeno due e vanno gestite entrambe contemporaneamente.

Nel caso di un'applicazione VoIP che gira su un'host connesso in rete, si può banalmente osservare che essa debba gestire almeno i seguenti ingressi: l'input da microfono e l'input dall'interfaccia di rete. Ovviamente dovrà gestire altrettante uscite: l'output verso gli speaker e l'output verso rete. Naturalmente, la cosa non finisce qua, infatti, i segnali di I/O in genere sono molti, ad esempio si ha:

- I segnali di controllo gestibili dall'utente.

  Nel caso si permetta all'utente di interagire col programma mediante un'interfaccia grafica, i segnali in questione possono essere quelli provenienti dal mouse; altrimenti, quelli provenienti dallo standard input (la tastiera) e così via. Sono quei segnali che in fig. IV.1 sono rappresentati nel blocco d'interfaccia utente.

- I segnali di controllo provenienti dalla rete.

  Il flusso di dati relativi all'RTCP è separato da quello relativo all'RTP. Entrambi si affacciano verso l'UDP ma su porte differenti. Solitamente, la porta riservata all'RTP è scelta in un range opportuno ed è di valore pari (es., 5004), mentre quella riservata all'RTCP è la successiva (5005). Oltre al flusso di dati trasportati dall'UDP, bisogna poter gestire quelli provenienti dalle porte TCP.

- I flussi di dati di varia natura.

  Ovviamente i segnali audio, quelli video ed dati tradizionali si affacciano su porte differenti. Spesso questi segnali, in particolare quelli audio e quelli video, vanno trattati in modo che possano essere opportunamente sincronizzati.

Nel seguito, per semplicità, faremo riferimento ai soli I/O di rete ed I/O della scheda audio riferiti al flusso vocale (vedi fig. IV.2.2.a.2). Osservando la figura è evidente che il software deve essere in grado di gestire più eventi contemporaneamente. Consideriamo, ad esempio, gli eventi legati agli input. Nel caso in cui siano pronti dati in ingresso dal microfono, diremo che l'input audio è leggibile, ed allo stesso modo, diremo che l'input di rete è leggibile quando nel buffer del SO ci sono pacchetti da elaborare, che provengono dalla rete. Supponiamo che il processo sia bloccato in attesa che l'input di rete sia leggibile, e che, nello stesso tempo, vi siano degl'input audio pronti per essere elaborati e spediti. Se il processo è in grado di accorgersi di ciò e di sbloccarsi sull'audio leggibile elaborarlo e successivamente riconsiderare l'input di rete, allora, il processo in questione è in grado di fornire un servizio full-duplex.

Il principio è piuttosto semplice. Innanzi tutto, si stabilisce l'insieme degli eventi d'interesse, che nel nostro caso potrebbe essere: l'input audio è leggibile, l'input di rete è leggibile, l'output audio è scrivibile[20], l'output di rete è scrivibile, il tempo d'attesa è trascorso. Successivamente, il processo informa[21] il kernel che vuole essere sbloccato da un'eventuale situazione d'attesa se accade uno degli eventi previsti. Questi eventi accadono in modo asincrono, per tale ragione la loro gestione, all'interno del SO è molto complessa. Ovviamente, i vari sistemi di I/O hanno dei buffer in grado di immagazzinare una quantità ragionevole d'informazione in attesa che venga elaborata, ma l'applicazione deve, comunque, prevedere un proprio buffer ad hoc.

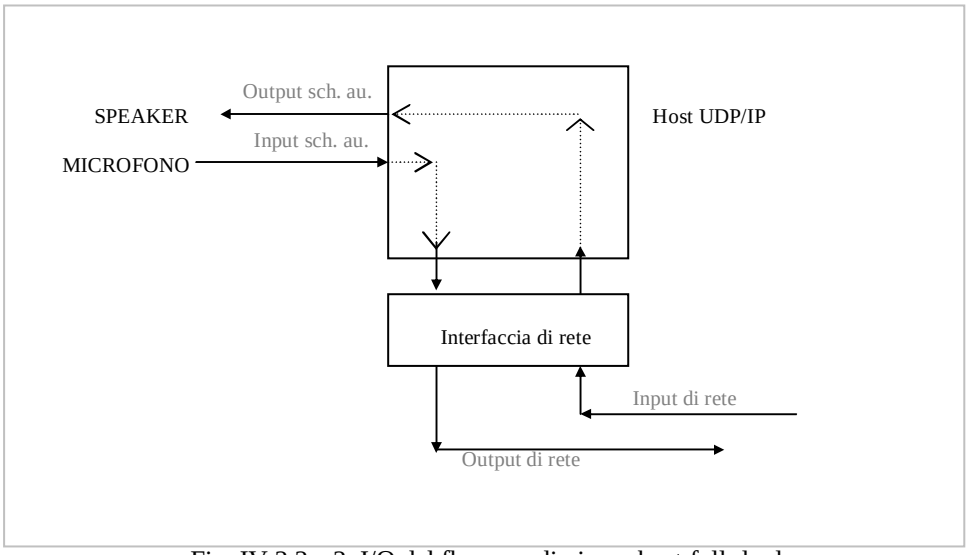


Fig. IV.2.2.a.2. I/O del flusso audio in un host full-duplex.

La situazione è ben rappresentata dal modello di fig. IV.2.2.a.3, che in letteratura è conosciuto con il nome di *I/O multiplexing* [41].

[20] Per scrivibile s'intende pronto per essere elaborato e/o trasmesso.
[21] Ad esempio, una tipica "chiamata di sistema" in grado di gestire più eventi, conforme allo standard POSIX, è la *select()*.

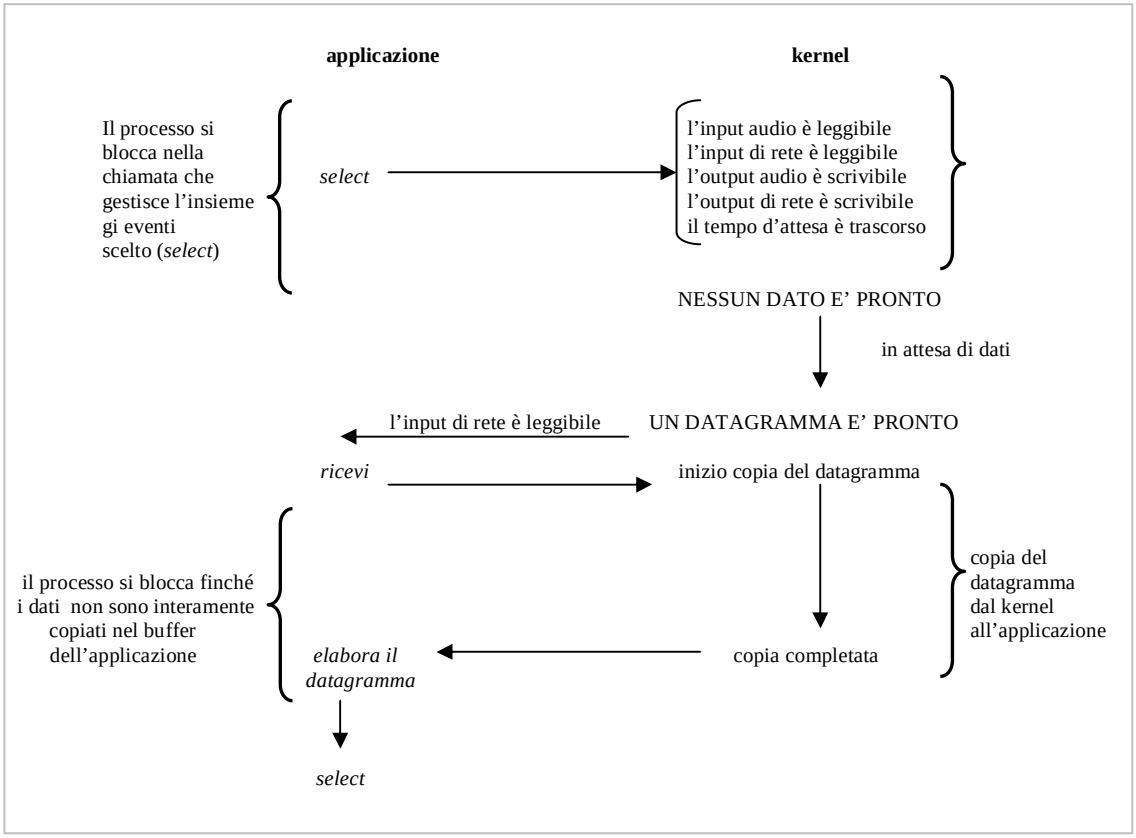


Fig. IV.2.2.a.3. Rappresentazione del modello I/O multiplexing.

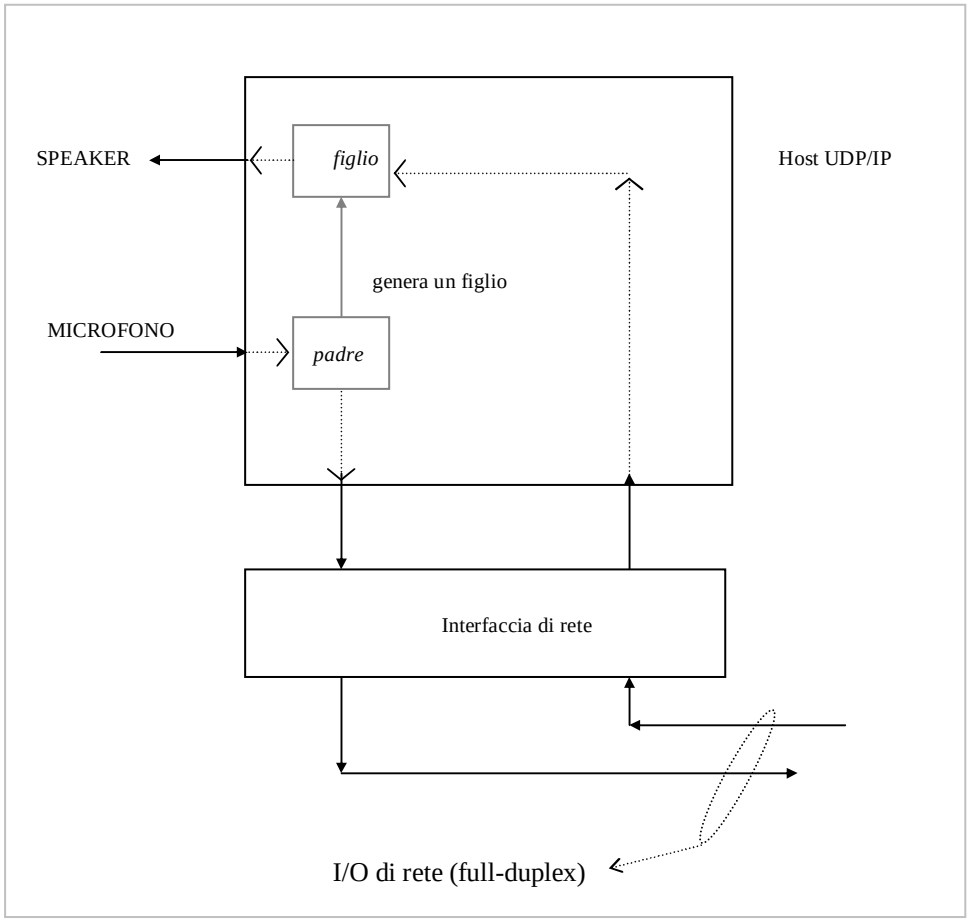


Fig. IV.2.2.a.4. Modello I/O multiplexing (nonblocking).

Per poter realizzare un'efficiente comunicazione full-duplex è necessario tenere presente un'ulteriore aspetto. Affinché un processo offra un servizio full-duplex, è necessario che riesca a non bloccarsi su un evento, nel caso in cui ne debba gestire più di uno. In parole povere, facendo riferimento al precedente esempio, non è auspicabile che il processo si blocchi nella gestione del datagramma rinviando la gestione dei dati audio ad un secondo momento. Se questo accade, il processo è inefficiente e può introdurre inutili ritardi, che aumentano linearmente con il numero d'eventi da gestire. Per evitare ciò è necessario che il processo sia in grado di frammentizzarsi e gestire in parallelo i vari eventi. La situazione è schematicamente rappresentata in fig. IV.2.2.a.4.

La terminologia di fig. IV.2.2.a.4 si riferisce a quella dei sistemi UNIX, i quali offrono la possibilità di lavorare in parallelo attraverso la, cosiddetta, generazione di figli (la tipica chiamata di sistema è la *fork()*). Vale la pena ribadire che nel corso della trattazione i riferimenti, sia alle chiamate di sistema[22] UNIX ed sia alle funzioni del linguaggio ANSI C, saranno numerosi, poiché *RTP-Speak* è realizzato in C.

Nel seguito saranno prese in considerazione alcune questioni legate al dimensionamento dei fondamentali buffer d'applicazione.

IV.2.b I/O e bufferizzazione.

In questa sezione prenderemo in considerazione le questioni legate ai buffer dell'applicazione. Per quanto riguarda il solo flusso vocale, fondamentalmente, si hanno due tipi di buffer: il *buffer-audio* ed il *jitter-buffer*.

Del *jitter-buffer*, si è gia parlato in precedenza. Ricordiamo che la sua funzione fondamentale è quella di compensare, in ricezione, il più possibile le condizioni fluttuanti della rete. Le modalità di compensazione possono variare da applicazione ad applicazione, lo stesso dicasi per la sua dimensione. Ad esempio, nel caso di una comunicazione "simplex" la dimensione del buffer è conveniente che sia ragionevolmente estesa (es., 0,5 sec.); infatti, in tal caso, non ci si deve preoccupare più di tanto del ritardo introdotto, poiché è percepibile solo all'inizio della comunicazione, quindi risulta più conveniente equalizare al meglio il jitter. Diversamente, nel caso di una comunicazione "full-duplex", ma anche in una comunicazione "half-duplex", è opportuno dimensionare il jitter-buffer attraverso un giusto compromesso tra rete e ritardi da essa introdotta; quindi, la dimensione del buffer in una LAN deve essere certamente minore di quella all'interno di Internet (es., 50 ms in un LAN e 150 ms in Internet). Inoltre è opportuno che la dimensione del jitter-buffer sia gestibile anche dall'utente in modo che possa adattarsi al carico istantaneo della rete; infatti, è ovvio che la stessa rete può essere affetta da ritardi più o meno lunghi a seconda del carico a cui è sottoposta in quel momento. La gestibilità, può essere messa a disposizione

[22] In queste note, talvolta si parla di *chiamate di sistema* altre volte di *funzioni di libreria*; la differenza sta solo nella loro implementazione. Per quanto ci riguarda la questione è irrilevante.

dell'utente attraverso la scelta in un range opportuno di valori (software d'interfaccia utente di fig. IV.1).

Le scelte di progetto e le modalità implementative del jitter-buffer di RTP-Speck saranno discusse nel par. IV.4.

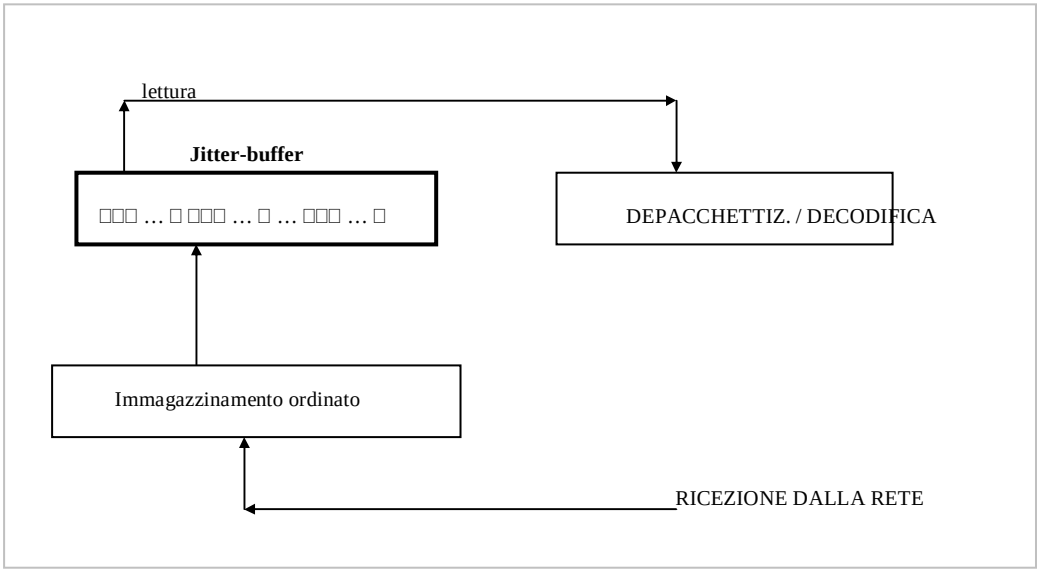


Fig. IV.2.2.b. Rappresentazione a blocchi dell'accesso al jitter-buffer.

Il buffer-audio (*kernel-audio-buffer*), solitamente, è meno progettabile. Normalmente, il driver valuta automaticamente la dimensione ottima dei parametri del buffer in base alle caratteristiche di campionamento (frequenza di campionamento, numero di bit per campione e numero di canali). Solitamente lo spazio disponibile nel buffer-audio è diviso in alcuni blocchi di uguale dimensione, noti come frammenti (*fragment*). La dimensione del frammento è importante, poiché l'applicazione può leggere i dati presenti in ogni frammento solo se l'intero frammento è pronto (letto o scritto), indipendentemente dal fatto che l'applicazione voglia leggere l'intero frammento o un unico byte. Nelle applicazioni A/V è importante che la dimensione del frammento sia relativamente piccola. Altrimenti, i ritardi tra gli eventi video ed i suoni ad essi associati risultano troppo lunghi. Alcuni dispositivi professionali non permettono la dimensionabilità di tale parametro; tuttavia, nei casi in cui è possibile, quest'aspetto non va assolutamente trascurato. Nei casi in cui non è possibile intervenire può essere utile conoscere tale dimensione per adeguare ad essa gli altri parametri correlati.

E' importante notare che l'applicazione audio, normalmente, non accede direttamente all'hardware. La tecnica è quella nota come *metodo a doppia bufferizzazione* [42] ed è schematicamente mostrata in fig. IV.2.2.c. Fondamentalmente, nel kernel esistono due buffer (*kernel-audio-buffer*) accessibili via DMA[23] (Direct

[23] I dati possono essere trasferiti dirattamente dall'hardware audio alla memoria centrale senza l'intervento della CPU.

Memory Access) dal dispositivo audio; un'ulteriore buffer (*audio-buffer*) va implementato nell'applicazione. Durante le operazioni di registrazione o playback, uno dei due kernel-audio-buffer è utilizzato dal dispositivo audio mentre l'altro è letto o scritto dall'applicazione. Quando il dispositivo ha terminato l'elaborazione sull'attuale kernel-audio-buffer in uso, passa all'altro. Questo si ripete fin tanto che il dispositivo è in uso. Mediante la doppia bufferizzazione, il dispositivo e l'applicazione possono lavorare in parallelo. Quindi, l'applicazione ha il tempo per fare qualche elaborazione, nonostante il dispositivo sia in uso; di conseguenza, è possibile registrare o ascoltare l'audio senza pause. L'ammontare del tempo a disposizione dell'applicazione è relativo alla dimensione del buffer ed ai parametri di campionamento.

> *Ad esempio, nel caso di RTP-Speak la situazione è la seguente. L'applicazione registra la voce con i seguenti parametri di campionamento: 8 kHz, 16-bit/campione, mono. Per cui, la velocità di registrazione è 16 kbyte/s, a questa velocità vengono immagazzinati i dati nel buffer audio. La dimensione del buffer audio è intorno ad 8129 byte. L'applicazione legge ed elabora 320 byte in ogni loop di ricetrasmissione (vedi fig. IV.4), quindi ogni loop non può durare più di 20 ms, altrimenti l'audio-buffer va in overruns.*

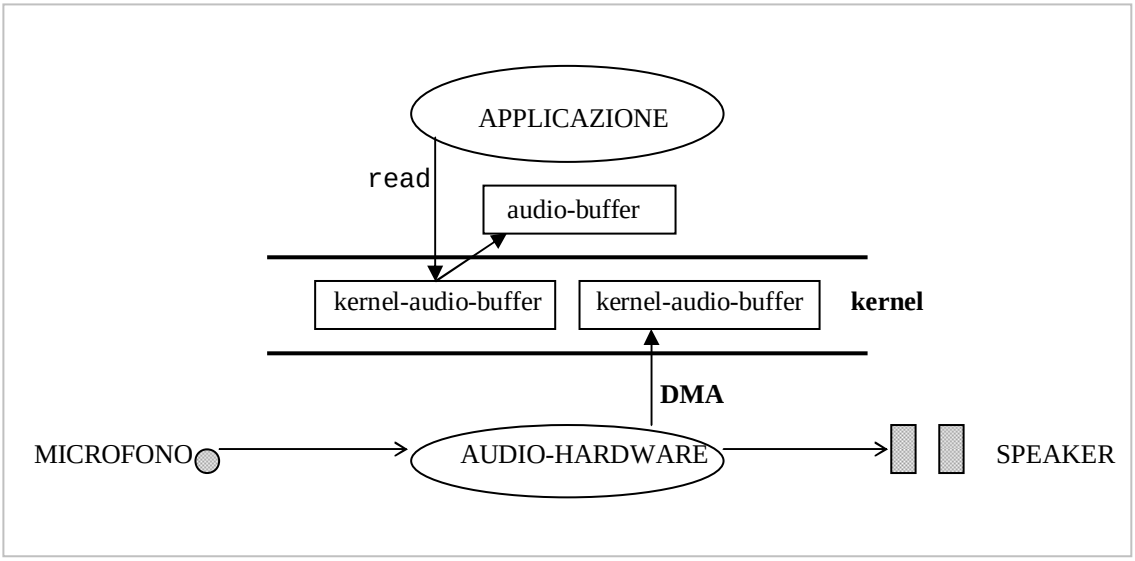


Fig. IV.2.2.c. Audio I/O: doppia bufferizzazione.

Se l'applicazione ha bisogno di più tempo per processare i dati, il buffer va in *overruns*, o *underruns* a seconda che si legga o si scriva, ed il dispositivo audio è costretto a scartare alcuni dati o, peggio a generare rumore. In tal caso, sia la registrazione sia il playback, saranno pieni di gap e rumori, quindi inaccettabili. Ciò significa, che per quanto è possibile è necessario dimensionare il buffer in relazione al tempo necessario ad elaborare i dati letti.

In questo senso rientra in gioco il discorso relativo al codificatore. Tanto più è complesso l'algoritmo di codifica tanto più tempo ci vuole per processare i dati, per cui la dimensione del buffer-audio che va bene per un tipo di codifica può non andar bene per un'altra. E' il caso della codifica G.771 e G.729; dalla tab. I.2.1 è evidente la grossa

differenza del ritardo endogenerato (0.125 ms per il G.711 contro 15 ms del G.729). Dai precedenti valori si evince che un buffer-audio di 8192 byte va benissimo par il G.711, ma è improponibile per il G.729. Infatti, il solo ritardo dovuto all'elaborazione G.729 quasi eguaglia il tempo a disposizione tra una lettura e l'altra dalla scheda audio, in più vanno considerate le altre elaborazioni di routine.

> *Numerose prove fatte con RTP-Speak mostrano come, la qualità della trasmissione con un buffer-audio da 8192 byte, ed il G.729/G.729A, è pessima. In particolare, la voce emessa risulta spezzettata ed incomprensibile. Mentre utilizzando il G.711 la qualità è ottima.*

Ritornando alla questione del full-duplex, è doveroso precisare un aspetto importante. Non tutti i dispositivi audio in commercio sono full-duplex. Esistono molti dispositivi che sono semplicemente half-duplex. Quindi, il software andrebbe scritto in modo tale che, si accorga delle eventuali limitazioni hardware, e possa informare l'utente. Di conseguenza, per garantire la comunicazione largamente interoperabile, è auspicabile la possibilità di mettere in piedi una comunicazione half-duplex. In questo modo, se due utenti vogliono comunicare, ma uno di essi ha un dispositivo audio che non supporta il full-duplex, la comunicazione può avvenire comunque in modalità half-duplex.

IV.2.2.c Comunicazione in rete e "socket pair".

Un flusso di dati, per essere trasmesso attraverso una rete a commutazione di pacchetto, deve essere spezzettato in opportuni datagrammi (*streaming*). Nel par. II.5.b e nel cap. III, si è ampiamente discusso su come incapsulare le trame vocali sintetiche in pacchetti RTP. In questa sezione, si supporrà di aver già realizzato le routine di pacchettizzazione/depacchettizzazione RTP e si considereranno le sole problematiche relative alla trasmissione dei pacchetti vocali attraverso l'utilizzo dei protocolli di trasporto. In pratica si, si entrerà nel merito dell'interfaccia "nucleo real-time"/"protocolli di rete" (vedi fig. IV.1).

Diciamo subito che, nelle intestazioni dei protocolli dello strato di trasporto (UDP ed TCP) e del protocollo di rete (IPv4 e IPv6), sono sempre presenti due coppie di campi, poiché necessari ad instaurare la comunicazione. La prima coppia, relativa al protocollo di trasporto, è costituita dai campi: *source port* e *destination port*. La seconda coppia, relativa al protocollo di rete, è costituita dai campi: *source address* e *destination address*.

Vediamo di approfondirne il significato. Quando due macchine comunicano, devono necessariamente conoscere i loro indirizzi IP, in questo modo ognuna di esse può identificare l'altra in un insieme di più host. Quest'informazione è contenuta all'interno dell'intestazione IP nei campi: *source address* e *destination address*. Ciò non basta, poiché non è raro che, una o entrambe, le macchine trasferiscano contemporaneamente file di varia natura con altre macchine. Di conseguenza è

necessario specificare il particolare I/O attraverso il quale s'intende comunicare. Ad esempio, si può pensare ad un host che trasmette voce in real-time e contemporaneamente richiede un file attraverso un ftp, sempre allo stesso host. Ovviamente, gli indirizzi IP delle due macchine non dipendono dal tipo di dati trasmesso ma è indispensabile che i due diversi flussi di dati viaggino attraverso I/O differenti. A specificare quali sono gli I/O coinvolti ci pensa il protocollo di trasporto mediante la coppia di campi: *source port* e *destination port*.

Relativamente all'indirizzo IP si è già parlato nel par. II.2, il concetto di "porta" è nettamente diverso. La porta è un'astrazione che i protocolli di trasporto adoperano per distinguere tra destinazioni multiple all'interno della stessa macchina. I protocolli TCP/IP identificano una porta attraverso un numero intero positivo.

Una volta noti gli indirizzi IP e le particolari porte attraverso cui s'instaura la connessione, sono state masse in piedi le prime condizioni essenziali per la comunicazione tra le macchine.

La coppia di campi di cui si è parlato è nota in letteratura come **sockets pair** o semplicemente **sockets**.

E' chiaro che la stessa necessità delle sockets obbliga a far sì che due connessioni distinte non abbiano le stesse sockets. A limite due connessioni possono avere sockets che si distinguono per almeno un campo. In fig.IV.2.2.c è riportato un esempio. Poiché in entrambe le connessioni, sono coinvolte le stesse macchine, gli indirizzi IP sono gli stessi. In entrambi i casi si suppone che si faccia un FTP quindi l'host a ha sempre lo stesso numero di porta, quello assegnato al FTP [43]. L'unica differenza sta nel numero di porta associato all'host b. Solitamente, il numero di porta assegnato al client dal software di rete è tale da essere unico.

	HOST a		HOST b	
	Indirizzo IP	porta	Indirizzo IP	porta
Connessione 1	151.100.9.29	1234	151.100.9.29	21
Connessione 2	151.100.9.29	1235	151.100.9.29	21

Tab. IV.2.2.c. Soket pair in due connessioni distinte tra gli stessi host.

Il concetto di sockets pair appare ovvio quando si usa il TCP, poiché è un protocollo orientato alla connessione. Per quanto riguarda l'UDP, non appena saranno mostrate le tipiche chiamate di sistema che sanciscono la comunicazione (par. IV.3), il precedente concetto apparirà altrettanto ovvio, nonostante esso non sia orientato alla connessione.

In ogni caso, in un'applicazione VoIP, la distinzione tra dati RTP e dati RTCP va fatta a livello dello strato di trasporto, infatti, l'RTP/RTCP non ha un campo dedicato al numero di porta. A proposito di ciò, nel prossimo paragrafo verrà discusso l'interfacciamento dell'RTP/RTCP con lo strato di trasporto.

IV.2.2.d Interfaccia: RTP/RTCP - strato di trasporto.

In questa sezione si parlerà delle questioni relative al trasporto dei pacchetti RTP/RTCP su reti IP. Si è già detto che l'RTP/RTCP si affida allo strato di trasporto per il (de)multiplexing dei dati RTP e dei flussi di controllo RTCP. Normalmente, l'RTP/RTCP si appoggia sull'UDP, per cui le considerazioni che seguono fanno riferimento alla seguente situazione.

In [27] si suggerisce di trasportare i dati RTP attraverso una porta pari ed i corrispondenti pacchetti RTCP attraverso la successiva (dispari). Le applicazioni che intendono seguire questo suggerimento possono adoperare qualsiasi coppia di porte UDP, purché sia assegnabile. Ad esempio, si può pensare che l'applicazione assegni, in ogni sessione, una coppia di defoult (5004 per l'RTP e 5005 per l'RTCP) e dia all'utente la possibilità di sceglierne una diversa. In ogni caso, non è auspicabile che la coppia di porte sia fissa, poiché in tal caso si possono avere problemi d'interoperabilità con altre implementazioni.

Sia l'UDP che il TCP hanno un campo di 16-bit per l'assegnazione del numero di porta ai differenti processi. I valori interi positivi a disposizione sono vanno da 1 a 65535, ma va notato che non tutte le coppie di porte sono assegnabili, per cui l'applicazione lo deve prevedere.

> *Seguendo le indicazioni del profilo A/V [28], le porte di defoult scelte per RTP-Speak sono: 5004 per l'RTP e 5005 per l'RTCP. Va, comunque, notato che non è necessario avere delle porte di defoult.*

In [43] è contenuta la lista dei numeri di porta già assegnati dall'Internet Assigned Numbers Authority (IANA). I numeri di porta sono divisi in tre intervalli (vedi fig. IV.2.2.d):

- Le *well-known ports*: da 0 a 1023. Quest'intervallo è controllato ed assegnato dalla IANA. Solitamente, lo stesso numero di porta è assegnato per un dato servizio (es., Web server) sia al TCP che all'UDP. Infatti, non esiste alcuna conflittualità tra le porte relative ai due protocolli, poiché concettualmente distinte.

- Le *registered ports*: da 1024 a 49151. Quest'intervallo non è controllato dalla IANA, tuttavia tende ad inserire l'uso di queste porte all'interno di una lista di convenienza per la comunità di Internet. Anche in questo caso, se possibile si tende ad assegnare, per un dato servizio, lo stesso numero di porta per entrambi i protocolli di trasporto.

- Le *dynamic ports*: da 49152 a 65 535. La IANA non dice nulla sull'uso di queste porte. Si tratta di un intervallo di assegnazione dinamica. In ambiente UNIX, il SO tende ad assegnare queste porte per quelle applicazioni che non costituiscono un particolare servizio, la loro assegnazione dura quanto il ciclo di

vita dell'applicazione. Solitamente ci si riferisce a queste porte indicandole come porte effimere (*ephemeral ports*).

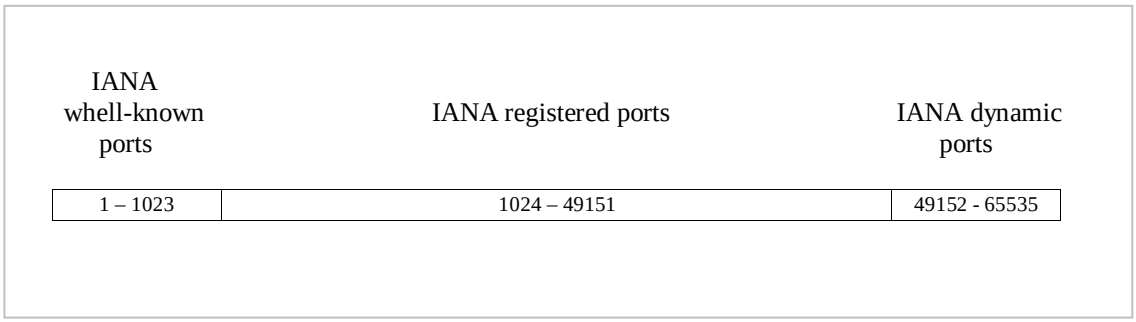


Fig. IV.2.2.d. Distribuzione dei numeri di porta.

La scelta di defoult suggerita in [27] è fatta in modo da rimanere sopra il valore 5000. Questo perché alcuni sistemi operativi UNIX (es., le implementazioni BSD: Berkeley Software Distribution) utilizzano l'intervallo tra 1 e 1023 per processi privilegiati (es., ftp, telnet, echo, daytime e quant'altro) e l'intervallo tra 1024 e 5000 come porte effimere.

E' ovvio che, in una comunicazione RTP bidirezionale, i numeri di porta possono essere assegnati in modo indipendente. In altre parole, se l'host a invia voce all'host b attraverso la porta 5004, non è necessario che l'host a riceva l'audio dalla porta 5004. Questa semplice considerazione serve a chiarire una questione importante. Un'applicazione VoIP non deve imporre alcuna restrizione sul numero di porta da adoperare poiché ciò renderebbe difficile, se non impossibile, l'interoperabilità con implementazioni diverse. Il numero di porta deve essere, per così dire, trasparente. Ad esempio, in una sessione unicast un'host può indicare all'altro host dove desidera ricevere i pacchetti vocali (un modo per farlo è quello di utilizzare l'SDP: Session Description Protocol).

Prima di concludere sulle questioni relative all'interfaccia RTP/strato-di-trasporto, è opportuno ribadire alcune questioni. L'RTP/RTCP non ha alcun campo di lunghezza per cui si affida al protocollo di trasporto per tutto ciò che riguarda la lunghezza dei dati. In ogni caso, la lunghezza massima ammissibile è quella supportata dai protocolli sottostanti (es., la lunghezza massima di un datagramma IPv4 è 65535 byte). Comunque, è consigliato non superare i 200 byte di lunghezza per ogni pacchetto vocale RTP. Solitamente si incapsula un unico pacchetto RTP in ogni pacchetto UDP, ma ciò non è necessario. Nel caso l'applicazione preveda più pacchetti RTP in unico pacchetto UDP[24], è essa stessa a doversi preoccupare del *framming*. Infine, ribadiamo che il protocollo RTP/RTCP non è limitato ad appoggiarsi all'UDP, la scelta è di convenienza e secondo le motivazioni discusse in precedenza.

[24] Un motivo di tale scelta potrebbe essere quello di diminuire la sovraintestazione.

IV.3 Interfaccia: applicazione – suite UDP/IP.

In questo paragrafo saranno prese in considerazione le fondamentali chiamate di sistema, relative ai SO UNIX, utilizzabili per realizzare una trasmissione dati vocali, non solo, attraverso il protocollo di trasporto UDP. Pur rischiando di appesantire troppo la trattazione, si è scelto di fare riferimento esplicito alle funzioni in linguaggio C, poiché altamente espressive delle possibilità di programmazione in rete. Questo paragrafo non intende essere un manuale di programmazione in rete, bensì un esempio di come, attraverso un progetto software, è possibile stabilire una connessione tra due o più macchine connesse in rete. Il linguaggio C in ambiente UNIX non è l'unico mezzo per conseguire tale obbiettivo[25], ma è quello al quale si farà riferimenti in quanto con esso si è scelto di implementare *RTP-Speak*. Per questo, le considerazioni che seguono serviranno a descrivere meglio le scelte implementative di *RTP-Speak*.

L'intento è di descrivere in modo operativo l'interfaccia RTP/RTCP-UDP, supponendo di avere già pronti i pacchetti vocali RTP. L'interfaccia sarà discussa facendo riferimento alle "API sockets"[41], attraverso le quali, è possibile legare il mondo delle applicazioni con quello della comunicazione (vedi fig. IV.3).

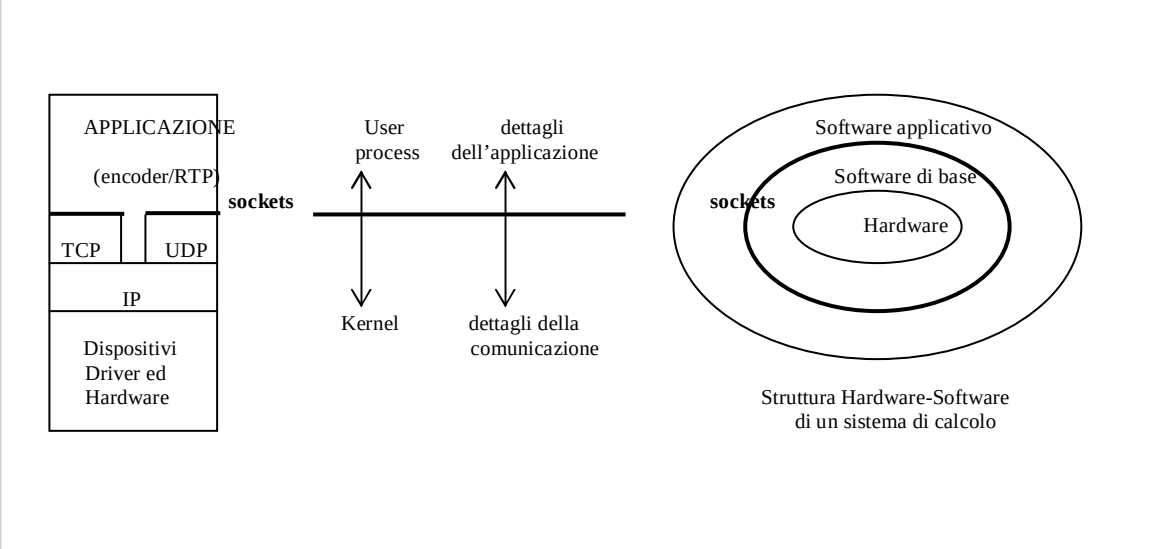


Fig. IV.3. Suite del protocollo d'Internet, sockets API e struttura Hardware/Software di un host.

L'acronimo API sta per Application Program Interface. Non esistono API standard per l'RTP/RTCP.

La fig. IV.3 evidenzia, ancora una volta, che l'RTP, nonostante il nome (real-time *transport* protocol) non fa parte dello strato di trasporto. Di ciò si è discusso ampiamente nel par. II.6, ad ogni modo, la fig. IV.3 fornisce lo spunto per puntualizzare un concetto molto importante. Si potrebbe pensare che, se il protocollo RTP/RTCP fosse uno standard, probabilmente, farebbe parte del SO ed, infatti, teoricamente esso fa

[25] Ad esempio, il linguaggio Java può supportare l'interfaccia RTP/RTCP-UDP.

parte dello strato di trasporto della suite UDP/IP. Visto che in realtà, attualmente, l'RTP/RTCP è ancora uno standard proposto, in pratica, dal punto di vista dell'implementatore, fa ancora parte dello strato applicativo. Tutto ciò porterebbe a concludere che, prima o poi, l'RTP/RTCP sarà implementato nel kernel di una macchina. In realtà, queste considerazioni vanno ridimensionate. L'RTP è saldamente accoppiato all'applicazione, basti pensare alla varietà di codificatori, per cui un'implementazione nel kernel avrebbe poco senso.

Si potrebbe affermare che l'RTP/RTCP è un pacchetto di regole da aggiungere all'UDP per creare un protocollo di trasporto "candidato" alla trasmissione di voce su reti IP.

IV.3.a UDP client/server.

In questa sezione descriveremo, attraverso un esempio teorico client/server, le API-sockets che legano l'applicazione all'UDP. In fig. IV.3.a sono mostrate le funzioni invocate in un tipico caso generale.

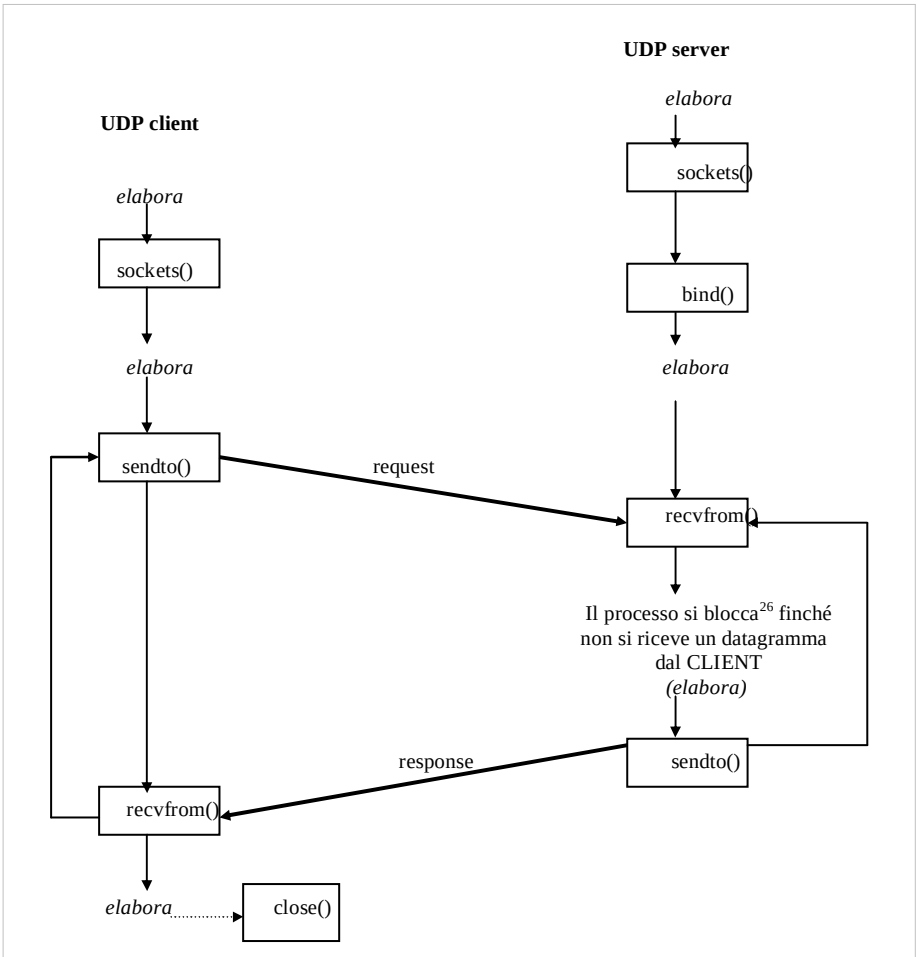


Fig. IV.3.a.1. Funzioni socket invocate in un tipico UDP client/server.

[26] Per semplicità nella rappresentazione si suppone avere un processo che si blocca sul singolo evento. Tuttavia, le considerazioni possono essere agevolmente estese al modello "I/O multiplexing (nonblocking)" introdotto nel par. IV.2.a.

Va precisato che le funzioni presenti in figura non esauriscono l'intera applicazione, ma sono quelle scelte, secondo un'esigenza di sintesi, come caratteristiche dei passi salienti da effettuare nell'interfacciamento RTP/UDP. Nel seguito, tali passi saranno brevemente descritti in modo da poter concretizzare molte considerazioni fatte sull'UDP.

D'importanza fondamentale è il concetto di "socket". Una possibile definizione è quella data in [4], che riportiamo testualmente:

> *I socket sono dei punti terminali a cui le connessioni possono essere collegate dal fondo (il lato del sistema operativo) ed a cui i processi possono essere collegati dalla cima (il lato dell'utente).*

Volendo realizzare un *server* che riceva e trasmetta dati, attraverso il protocollo UDP, la cosa più semplice da fare è di creare un processo che faccia le seguenti cose:

- Dia origine ad un socket, tramite la chiamata socket.
- Assegni un indirizzo al socket, tramite la chiamata bind.
- Si metta in attesa di dati in arrivo sul quel socket, tramite la chiamata recvfrom.
- Dopo aver fatto le opportune elaborazioni sui dati ricevuti, trasmetta dati attraverso la chiamata sendto.

Per il *client* la situazione è analoga tenendo conto che solitamente per esso, come vedremo nel seguito, non è necessaria la chiamata bind.

Vediamo di chiarire il significato delle precedenti considerazioni.

Ogni processo che voglia comunicare nell'ambito di un sistema distribuito deve creare per prima cosa un socket. Invocando la funzione socket si ha la possibilità di specificare il tipo di protocollo di trasporto desiderato. Successivamente va messo in piedi il dialogo USER PROCESS-KERNEL, uno schema a blocchi è mostrato in fig. IV.3.a.2.

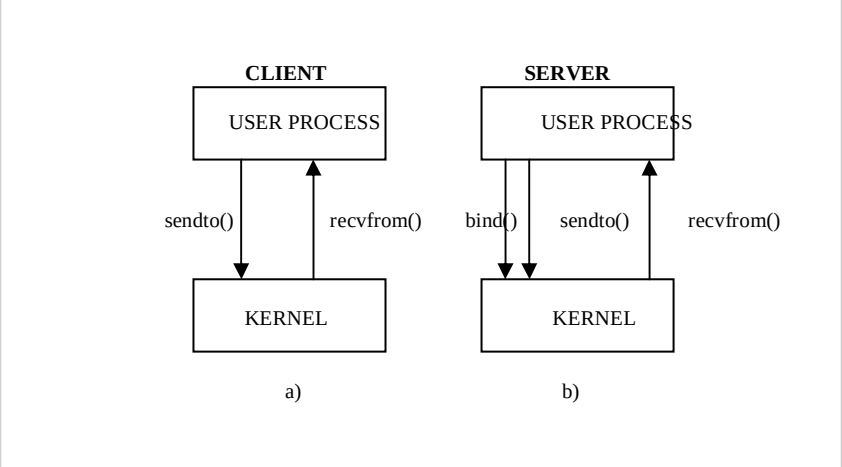


Fig. IV.3.a.2. Trasferimento dati tra processo e SO, nel *client* a) e nel *server* b).

La chiamata bind assegna un indirizzo al socket precedentemente creato. Si tratta di una funzione che passa al kernel del SO la struttura di dati (sockaddr_in), preventivamente inizializzata con il numero di porta UDP, attraverso la quale s'intende ricevere i dati, e gli indirizzi IP delle macchine che possono contattare il *server* su detto.

A questo punto il *server* è in grado di ricevere dati in ingresso attraverso la chiamata recvfrom. I dati ricevuti seguono la direzione mostrata in fig.IV.2.1 (linea continua) ed, infatti, la recvfrom passa i dati dal kernel al processo, il quale si sblocca e li elabora. All'interno del kernel rimangono una serie d'informazioni importanti tra cui l'indirizzo IP e il numero di porta UDP del mittente.

La chiamata sendto, trasferendo i dati da trasmettere dal processo al kernel, invia la risposta del *server* alla richiesta del *client*. I dati saranno trasmessi al socket del *client* avente uno specifico indirizzo. Il *client* deve trovarsi in attesa di dati attraverso la chiamata recvfrom.

Successivamente, il *server* si pone in attesa di nuove "request" attraverso una nuova recvfrom ed il tutto si ripete finché il *client* non chiude la connessione, attraverso la chiamata close.

Osservando la fig. IV.3.a.2 si può notare come la chiamata bind presente nel *server* non compare nel *client*. Ciò significa che il *server* ha legato (bind) una specifica porta alla particolare applicazione, mentre il *client* comunica attraverso una porta effimera (ephemeral port) scelta dal kernel appena è invocata la chiamata sendto. Questa condizione è tipica di molte applicazioni. Alcune specifiche applicazioni richiedono che anche il *client* faccia la bind.

Un'autorevole trattazione delle chiamate di sistema di cui si è parlato, e non solo, è disponibile in [41] [44].

Ritornando alla fig. IV.3.a.1, va notato che la rappresentazione è abbastanza generale. Le diciture "elabora" possono riferirsi al processamento di dati di qualsiasi natura. Nel caso di trasmissione di voce real-time le elaborazioni si riferiscono soprattutto a quelle relative al segnale vocale. Inoltre, in figura il client ed il server fanno parte di host diversi, tuttavia, in una comunicazione vocale bidirezionale, essi vanno visti come facenti parte dello stesso host.

IV.3.b Problemi relativi all'uso dell'UDP.

Nel par. II.3 si è ampiamente discusso sul fatto che l'UDP, contrariamente al TCP, non effettua alcun controllo sui dati trasmessi o ricevuti. Per questa ragione il trasporto dei dati con l'UDP necessita di una serie d'accortezze. Per avere un'idea concreta, in questa sezione, saranno presi in considerazione, e commentati, una serie d'esempi pratici.

Buffer di ricezione della socket UDP & velocità del processore

Attraverso un semplice esempio, mostreremo quali effetti può avere la mancanza del controllo di flusso nell'UDP.

Supponiamo di avere un client che spedisca un numero fisso di datagrammi, diciamo 3000 pacchetti da 172 byte ciascuno, ad un server. Se il server viene fatto girare su una macchina lenta (es., un 386) mentre il client viene fatto girare su una macchina più veloce, è quasi inevitabile che il client mandi in overflow il server. Quello che accade è la seguente cosa. Il client spedisce i 3000 datagrammi e tutti arrivano all'interfaccia di rete del server; ciò nonostante, è facile che una parte di essi venga persa dall'applicazione in quanto non riesce ad elaborarli con sufficiente celerità. La macchina ricevente è lenta, e non è detto che riesca ad elaborare i datagrammi ricevuti prima che il *socket-buffer* di ricezione vada in overflow. La facilità con cui si può mandare in overflow la macchina ricevente è dovuta alla mancanza di controllo sul flusso dei dati. Ad esempio, se i dati fossero trasmessi attraverso il TCP non ci sarebbe alcun problema.

Ogni socket ha un buffer di trasmissione ed uno di ricezione. I buffer di ricezione, sia nel caso del TCP che nel caso dell'UDP, servono a mantenere temporaneamente i dati finché non vengono letti dall'applicazione. Attraverso il TCP, lo spazio disponibile nel buffer di ricezione è notificato attraverso il campo *window* presente nella sua intestazione. In questo modo, il buffer di ricezione del TCP non può mai andare in overflow poiché il paritario non è abilitato a spedire più datagrammi di quelli immagazzinabili. Il TCP offre un controllo dinamico sul flusso da trasmettere in base alla disponibilità in ricezione. Al contrario, con l'UDP i datagrammi vengono spediti comunque nella loro interezza e nel caso il buffer di ricezione sia pieno essi vengono scartati. Per questa ragione un trasmittente veloce mandi in overflow un ricevitore più lento. Di conseguenza questa situazione deve essere presa in considerazione dall'applicazione.

Se il server è fatto girare sulla macchina più veloce ed il client su quella più lenta, nessun datagramma è perso per un overflow del buffer di ricezione.

> *Numerose prove fatte dimostrano la concreta perdita d'informazione. Le prove sono state fatte mediante una versione modificata di RTP-Speak (una sorta di FTP attraverso l'uso dell'UDP). Il client prende i dati da un file vocale che viene pacchettizato e spedito. Il server ricompone il file originale mediante i pacchetti ricevuti. Inoltre, il server è dotato di un contatore interno che indica il numero totale di pacchetti elaborati dall'applicazione. Attraverso il comando* "netstat –s | - tail" *ha disposizione nei sistemi UNIX, è possibile confrontare il numero totale di pacchetti ricevuti dall'interfaccia di rete con quello fornito dal contatore dell'applicazione. Una serie di prove ha reso evidente che il numero pacchetti ricevuti dall'interfaccia di rete, quasi sempre, coincide con quello dei pacchetti trasmessi dall'host trasmittente, e comunque, non si discosta di molto, mentre quello dei pacchetti elaborati è nettamente inferiore. Inoltre, se il file viene pacchettizato ad intervalli di 40 ms, anziché di 20 ms, la perdita di pacchetti si riduce notevolmente. Ciò conferma che la perdita è dovuta soprattutto all'overflow in ricezione.*

La possibilità che l'utente ha di intervenire sulla dimensione del socket-buffer in ricezione è molto limitata e varia a seconda del particolare SO. E' ovvio che, pensare di far girare il server sulla macchina più veloce ed il client su quella più lenta è una qual cosa di improponibile per applicazioni come VoIP. La soluzione migliore è quella di implementare un processo di ricezione che sia più snello possibile. Di solito ciò è abbastanza naturale. Ad esempio, il processo di decodifica è di per sé più veloce di quello di codifica.

Ricezione di datagrammi indesiderati

La possibilità di ricevere datagrammi indesiderati è un problema meno grave di quello dall'overflow, ma che comunque, non va trascurato. Il problema risiede nel fatto che l'UDP, al contrario del TCP, non realizza una connessione esplicita tra le macchine comunicanti. Consideriamo una comunicazione VoIP tra due host, in particolare il flusso di dati vocali trasportato attraverso il protocollo UDP. Esiste il rischio concreto, anche se minimo, che qualche processo su un qualsiasi altro host remoto mandi dei datagrammi alla stessa socket-pair di uno dei due host. Questa situazione può essere disastrosa, infatti, gli eventuali dati non vocali ricevuti ed emessi dagli speaker possono dar luogo a rumori molto intensi, talvolta anche dannosi all'orecchio dell'ascoltatore.

Questo problema può essere risolto agevolmente. Una possibile soluzione potrebbe essere quella di stabilire una comunicazione attraverso la cosiddetta *connected UDP socket*. Va subito precisato che una socket UDP connessa non ha nulla a che vedere con la connessione stabilita dal protocollo TCP, per approfondimenti in merito si rimanda a [41]. Quando l'applicazione mette in piedi una comunicazione UDP connessa specifica l'indirizzo IP ed il numero di porta del suo paritario. Tutti i datagrammi provenienti da altre socket-pair vengono automaticamente scartati dal kernel. Questa situazione ha un'efficacia limitata, visto che è applicabile solo a comunicazioni unicast. Una soluzione più generale, che vale anche per applicazioni multicast e broadcast, è quella di implementare direttamente nell'applicazione un controllo sulla provenienza dei dati e decidere di conseguenza.

Datagrammi persi e signalling

In precedenza, si è più volte messo in evidenza l'inaffidabilità del protocollo UDP. Nonostante ciò esistono situazioni in cui anche i dati di controllo vengono trasportati attraverso l'UDP. Si è anche affermato che, non tutti i dati di controllo sono affidabili all'UDP. Infatti, quei controlli che sono vitali per la vita del processo che gestisce la comunicazione non vanno mai affidati all'UDP. Per fare un esempio, si può considerare la seguente eventualità. Supponiamo che un processo sia in attesa di un segnale di controllo sul quale deve prendere una decisione. Se tale segnale viene trasmesso dal paritario attraverso l'UDP ed un router lo scarta per qualche ragione, il processo sull'host ricevente rimarrà bloccato per sempre in attesa di dati che non arriverà mai. Volendo limitare i danni pur utilizzando l'UDP, l'unica strada percorribile è quella di implementare un time-out sul processo in ricezione. Quando il tempo

d'attesa è trascorso s'invia una nuova richiesta al paritario. Ma questa soluzione non è del tutto efficace poiché è possibile che anche la notifica del time-out stesso sia persa, perciò si rimane in un nulla di fatto. Se poi si pensa di aggiungere più affidabilità all'UDP, tanto vale utilizzare il TCP.

IV.3.c Questioni relative all'ordinamento dei byte.

Per ordinamento dei byte s'intende la convenzione adoperata da una macchina nel memorizzare valori che richiedono più di un byte. Le convenzioni più comuni sono due: *big-endian* e *little-endian*. I termini indicano quale fine (end), di un multibyte è immagazzinata nell'indirizzo di partenza del valore (vedi fig. IV.3.c.1). Nel primo caso si parla di sistema a finale grande, nel secondo a finale piccolo.

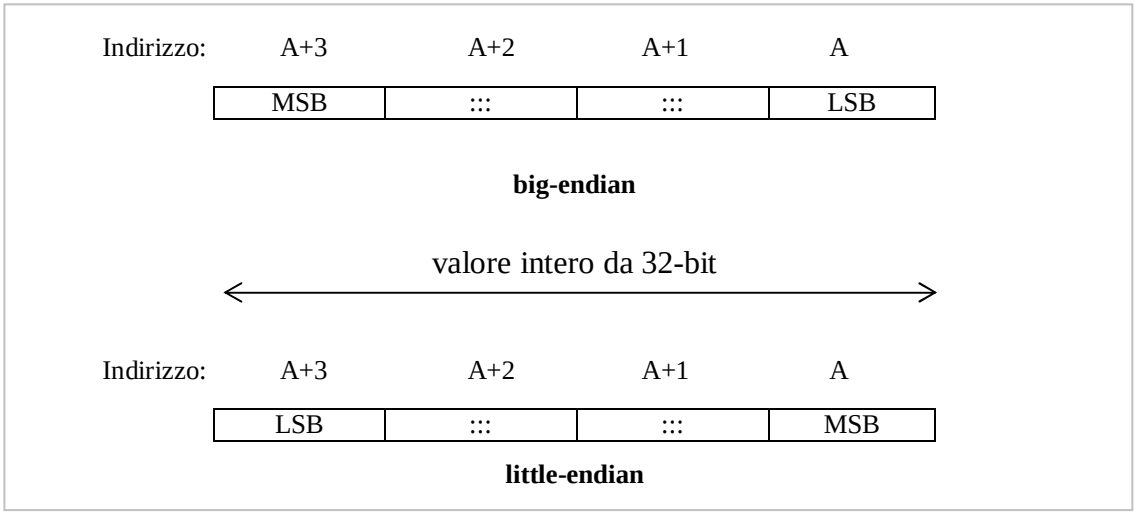


Fig. IV.3.c.1. Rappresentazione dei formati big-endian e little-endian, per un intero a 32-bit.

Nell'ambito della stessa macchina l'ordinamento dei byte [45] è una cosa irrilevante. Diversamente, nel caso di più macchine interconnesse, la cosa, pur essendo banale, non va sottovalutata, poiché ha delle forti implicazioni sulla riuscita dei programmi.

Le due differenti implementazioni mostrate in fig. IV.3.c.1 non sono scelte in base a particolari convenienze, si tratta solo di un fatto storico; purtroppo, non esiste uno standard di riferimento[27]. I protocolli di Internet seguono sempre la convenzione big-endian. Nel seguito ci riferiremo all'ordinamento di una certa macchina attraverso l'acronimo *hbo* (host byte order), e ci riferiremo all'ordinamento dei protocolli TCP/IP con l'acronimo *nbo* (network byte order).

I processi che girano sullo stesso host non devono preoccuparsi del particolare hbo, ma la comunicazione tra processi in un sistema distribuito (networking) deve considerare quest'aspetto come fondamentale.

[27] In [41] è disponibile una routine in grado di determinare la modalità di ordinamento della macchina sulla quale è innescata.

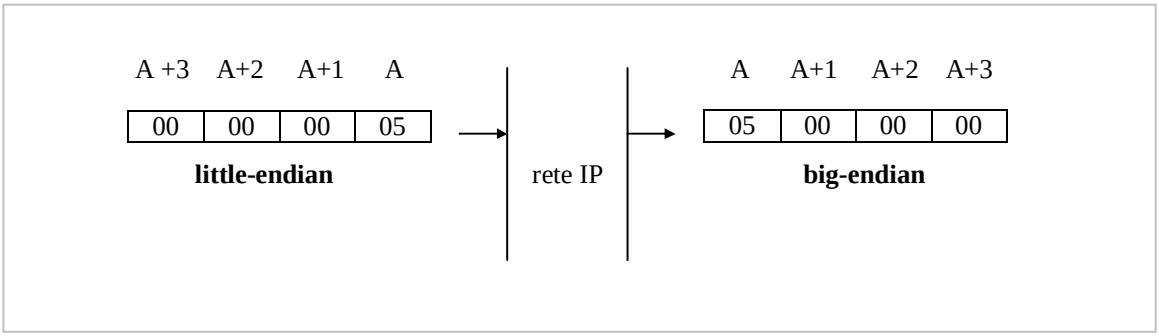


Fig. IV.3.c.2. Esempio di trasferimento dati tre macchine con hbo differenti. I valori rappresentati in figura sono in formato esadecimale.

Se quest'aspetto non viene adeguatamente preso in considerazione i problemi possono essere notevoli. Vediamo un esempio pratico. Supponiamo che due host con architetture a 32-bit si scambino dati, in particolare l'host a invia il valore 5 all'host b. Inoltre, l'host a è un sistema a finale piccolo (little-endian) mentre l'host b è un sistema a finale grande (big-endian), la situazione è rappresentata in fig. IV.3.c.2. I quattro byte sono trasmessi lungo la socket dell'host a (nell'ordine A, A+1, A+2, A+3) ove vengono memorizzati (nell'ordine A, A+1, A+2, A+3). Osservando la figura, il problema è evidente. Il valore trasmesso è 5 (in esadecimale, 0x00000005) mentre quello ricevuto è 83,886,080 (in esadecimale, 0x05000000), quindi la comunicazione non ha avuto successo. Prendendo spunto da questo semplice esempio è possibile mettere in evidenza alcune questioni importanti:

- La convenzione relativa all'ordinamento dei byte è riguarda solo i valori che richiedono più di un byte di memorizzazione (valori multibyte). Ad esempio, se si trasmettono solo ed esclusivamente, caratteri si può fare a meno di considerare il problema.

- Se le macchine coinvolte nella comunicazione seguono lo stesso ordinamento, la convenzione adoperata dalla rete non ha importanza. Ad esempio, osservando la fig. IV.3.c.1 si può affermare che se l'hbo fosse stato little-endian per entrambe le macchine, nonostante l'nbo è big-endian, non ci sarebbero stati problemi. Tuttavia, affinché l'implementazione non abbia problemi di compatibilità è fortemente consigliato prendere, comunque, gli opportuni accorgimenti.

- Nel caso di comunicazioni vocali l'errore visto in fig. IV.3.c.2 può portare a fastidiosi rumori, ed in alcuni casi, anche dannosi all'orecchio.

Un'applicazione VoIP deve tenere conto di questi problemi soprattutto nel caso dell'implementazione dei protocolli non-standard. Ad esempio, consideriamo il caso dell'RTP. Osservando l'intestazione RTP, mostrata in fig. II.5.b, è evidente la presenza di alcuni campi multibyte, sequence number, time stamp, SSRC, CSRC, e così via, i quali vanno opportunamente elaborati sia in trasmissione che in ricezione. Per fare ciò è

necessario rendere questi dati immuni dal particolare ordinamento di byte. I sistemi POSIX-compatibili mettono a disposizione dell'implementatore un gruppo di chiamate di sistema adatte allo scopo (`htons, htonl, ntohs, ntohl`) [41].

> *RTP-Speak è un esempio di uso di queste funzioni. Va comunque precisato che, il solo uso di tali funzioni non è sufficiente ad evitare tutti i possibili problemi. Vediamo perché. L'intestazione RTP è rappresentabile, in C, attraverso una struttura a campi di bit [RFC 1889], ma la pacchettizzazione dei dati delle strutture varia a seconda della particolare implementazione. Per cui non è consigliato trasmettere strutture binarie lungo una socket. La soluzione adottata in RTP-Speak è di trasmettere tutti i dati numerici come stringhe di testo. Inoltre, il solo uso delle funzioni sopra citate fallirebbe nel caso di architetture a 64-bit.*

Senza queste accortezze si possono generare degli errori madornali. Ad esempio, il *jitter-buffer*, per equalizare il jitter si affida all'informazione contenuta nel "sequence number" (16-bit), oppure a quella nel "time stamp" (32-bit). Se tali campi sono affetti da errore di ordinamento, il jitter non può essere correttamente equalizato.

IV.4 Descrizione del sistema di comunicazione RTP-Speak.

Questo paragrafo, è dedicato alla descrizione progettuale ed implementativa di RTP-speak. Una rappresentazione a blocchi dell'algoritmo che lo governa è mostrata in fig. IV.4.1. In figura sono mostrate solo le principali caratteristiche, ulteriori dettagli sono disponibili nell'Appendice A. Confrontando questa figura con la fig. IV.1 è evidente che RTP-Speak è una versione "sottile" di un progetto di più vasta portata. RTP-speak è stato realizzato seguendo l'intento di ottenere una struttura versatile, ovvero una base per un futuro sviluppo che non sia limitato verso una sola via. In futuro RTP-Speak potrebbe essere un'applicazione in grado di coprire tutte le funzionalità rappresentate in fig. IV.1 e non solo, oppure un'applicazione di e-mail vocale, o ancora un sistema di testing per codificatori adatti a canali inaffidabili. Per permettere ciò RTP-Speak è stato reso sufficientemente versatile ed aperto a nuove aggiunte, oltre a quelle previste in fig. IV.1.

Le attuali caratteristiche di RTP-Speak sono le seguenti:

- Flusso di dati: *solo audio*
- Comunicazione: *tempo reale*
- Modalità di comunicazione: *simplex*
- Opzioni di comunicazione: *broadcast*
- Interfaccia utente: *riga di comando*
- Acquisizione voce: *da microfono*
- Codifiche supportate: *G.711, G.729*
- Compatibilità di sistema: *sistemi POSIX-compatibili*

L'algoritmo verrà descritto considerando dapprima il processo trasmittente e poi il processo ricevente (rispettivamente host a ed host b di fig. IV.4). Il processo trasmittente sarà descritto considerando i quattro blocchi fondamentali: interfaccia utente, acquisizione, codifica e pacchettizzazione, interfaccia di rete. Analogamente, il processo ricevente verrà descritto nelle funzionalità di: inizializzioni, interfaccia di rete, equalizzazione del jitter, decodifica e depacchettizzazione, playout. Durante la descrizione saranno considerate le scelte progettuali fondamentali, molte delle quali sono state già ampiamente discusse nel corso del presente capitolo. Iniziamo, dunque, con il processo trasmittente.

PROCESSO TRASMITTENTE

Interfaccia utente

L'utente può lanciare il processo e settarne le opzioni, direttamente da riga di comando. Per comodità, nel seguito, ci riferiremo alla riga di comando attraverso il simbolo $. Il comando che lancia il processo trasmittente è il seguente: `vmailto`. Successivamente va specificato l'indirizzo IP del destinatario.

L'indirizzo IP può essere sia in formato dotted-decimal che in formato hostname:

```
$ vmailto 151.100.9.29
```

oppure,

```
$ vmailto pcvoce2.ing.uniroma1.it
```

Attraverso i precedenti comandi il processo è lanciato con le sue caratteristiche di default. Le opzioni che modificano il default del programma riguardano il nome del dispositivo il formato di codifica ed un help, ma è facile poter inserire ulteriori opzioni (ad esempio, sui parametri d'acquisizione e campionamento, sul tasso di pacchettizzazione, sulla porta UDP e quant'altro). Il default di dispositivo audio è *dev/dsp* mentre quello di codifica è il *PCMA*. Un esempio di setup delle opzioni è il seguente:

```
$ vmailto –d dev/audio –c G729 151.100.9.29
```

Per trasmettere in *broadcast* è sufficiente digitare uno degli indirizzi chiave previsti per il broadcast (ovvero, uno dei seguenti: 151.100.9.255, 151.100.255.255, 151.255.255.255, 255.255.255.255).

Per lanciare un breve messaggio d'aiuto è sufficiente scegliere l'opzione "–h":

```
$ vmailto –h
```

in questo modo comparirà il seguente messaggio:

```
$ Usage: vmailto [-d devicename] [-c codername] destination

  -d              devicename
  -c              type of encoder
                  Available coders are: PCMA, PCMU, G729
  destination     IP address or host-name to contact

  escape character is ^C
```

E’ prevista un rilevamento automatico degli errori di scrittura sulla riga di comando. Ad esempio, se l’indirizzo IP digitato non corrisponde a nessun host presente nella rete, perché c’è un errore di scrittura (es., 151100.9.29, manca un punto tra “151” e “100”) oppure perché l’indirizzo non esiste affatto, il processo verrà interrotto è comparirà il seguente messaggio d’errore:

```
$ Error: destination (IP address or hostname) missing
```

Nel caso si digiti un opzione non prevista (es., “-f”) si ha:

```
$ getopt: -f unknoun option
$ Usage: vmailto [-d devicename] [-c codername] destination

  -d             devicename
  -c             type of encoder
                 Available coders are: PCMA, PCMU, G729
  destination    IP address or host-name to contact

  escape character is ^C
```

Nel caso si digiti un codificatore non disponibile (es., “-c G728”) si ha:

```
$ connection impossibile G728 unavailable.
 Available coders are: PCMA, PCMU, G729.
```

Gli errori propri del programma sono gestiti invocando i messaggi del SO, ed a seconda della gravità il processo può essere interrotto. Ad esempio, un errore di scrittura sulla socket non comporterà la terminazione del programma ma un semplice messaggio d’avvertimento:

```
$ sendto: connection refused
```

La terminazione del processo è innescabile in modo asincrono con effetto istantaneo, digitando “^C”, cui seguirà un breve messaggio di notifica:

```
$ Trasmission interrupted.
 54769 packets sent.
```

Acquisizione audio

Dopo il setup di alcune funzionalità generali, il processo lancia, una volta per tutte, il setup della scheda audio. Per rendere chiaro e modulare il progetto, il setup

dell'audio è implementato in un file indipendente. In questa fase si settano le impostazioni di acquisizione attraverso i seguenti passi:

1. Si apre il dispositivo audio (es., *dev/dsp*) per la registrazione.
2. Si setta il numero di bit per campione (16 bit/campione).
3. Si setta il numero di canali (es., 1 mono, 2 stereo).
4. Si imposta la frequenza di campionamento (8000 Hz).
5. Si imposta la dimensione dell'audio-buffer (vedi par. IV.2.b).

Fatto ciò si avvia il principale *loop di trasmissione* (vedi fig. IV.4.1). La prima operazione che viene fatta è quella di leggere, attraverso una semplice `read`, dal dispositivo audio, 20 ms (defoult) di voce registrata e porli in un secondo buffer, dal quale vengono trasferiti verso il codificatore. Dopo aver fatto la codifica, la pacchettizzazione RTP e l'incapsulazione UDP/IP/Ethernet, si riprende, leggendo i successivi 20 ms; e così via fino alla terminazione del programma.

Codifica e pacchettizzazione RTP

Siamo nel vivo del principale loop di trasmissione. I comandi da prompt servono ad indicare il particolare codificatore da innescare (PCMA (defoult), PCMU oppure G729). Si è già detto più volte che la codifica e l'RTP sono strettamente legati, infatti, ancora una volta, si è voluto marcare ciò rappresentandoli in un unico blocco (fig. IV.4.1). In questa fase si inizia a costruire il pacchetto di voce da spedire.Il formato di codifica viene selezionato in base alle informazioni da prompt.

In questa fase si hanno tutte le informazioni necessarie all'intestazione RTP, quindi si inizia dall'intestazione RTP. In particolare, il padding è valutato in base alla lettura dalla scheda audio ed il payload type dalle informazioni deducibili dalla riga di comando. Successivamente si innesca la codifica. L'ingresso del codificatore è rappresentato dal buffer che contiene i 20 ms di voce, l'uscita è un'array che ha il formato di payload RTP previsto (secondo quanto discusso nel cap. III). In pratica, la costruzione dell'intestazione RTP è separata dalla codifica, mentre il payload va costruito direttamente nel codificatore. Le routine d'intestazione RTP fanno parte di un unico file, mentre quelle di codifica sono suddivise in file separati. In questo modo si ha una grossa flessibilità implementativa.

Incapsulazione UDP/IP/Ethernet

La pacchettizzazione UDP/IP/Ethernet è effettuata direttamente nel kernel del SO. L'interfacciamento con il kernel è messo in piedi in due fasi.

Una prima fase, precedente al principale loop di trasmissione, riguarda l'apertura delle socket e le inizializzazioni necessarie: indirizzo IP dell'host da contattare, porta da cui trasmettere (defoult, 5004) e quant'altro. Per avere la possibilità di applicazioni multicast e broadcast si è scelto di non connettere la socket UDP. L'apertura di una socket UDP connessa avrebbe velocizzato le operazioni di trasmissione [41] ma, nello stesso tempo, avrebbe limitato l'applicazione a due soli host, il che è contrario al principio di *multiservizio* su cui si fonda VoIP. Data la finalità dell'applicazione e la sua

snellezza è sembrato sovrabbondante implementare l'RTCP. Tuttavia, per non perdere in compatibilità con eventuali altre applicazioni si è scelto di aprire una socket anche per il flusso RTCP (porta di defoult, 5005). Va comunque precisato che, in un futuro sviluppo di RTP-Speak che preveda una modalità full-duplex è suggerito implementare anche l'RTCP.

La seconda fase rappresenta l'ultima operazione del loop di trasmissione. Si tratta dell'invio vero e proprio del pacchetto vocale. Per quanto detto nel par. IV.3.c e per aumentare il più possibile le probabilità di compatibilità con altre applicazioni, il pacchetto viene trasmesso come un array di caratteri (1 byte a carattere). Successivamente il loop riprende dalla lettura dalla scheda audio.

PROCESSO RICEVENTE

Inizializzazioni

Il processo ricevente viene lanciato da riga di comando semplicemente digitando "`rcvmsg`" che sta per "**r**ece**i**v**e** **m**es**s**a**g**e":

```
$ rcvmsg
```

Il processo ricevente, una volta lanciato, innesca le inizializzazioni della scheda audio. Il setup dell'audio è del tutto analogo a quello relativo al processo trasmittente, quindi si eviterà di ridescriverlo. Essendo, RTP-Speak, in fase sperimentale, non si è ritenuto necessario inserire opzioni da riga di comando. Inoltre, è previsto uno sviluppo full-duplex, per cui il processo trasmittente e quello ricevente verranno opportunamente fusi in un unico processo, dunque è sembrato inefficiente considerare opzioni che potrebbero rivelarsi inutili.

Dopo queste inizializzazioni il processo si rivolge all'interfaccia di rete, iniziando il loop principale di ricezione (vedi fig. IV.4).

Interfaccia di rete

Valgono le considerazioni fatte per il processo trasmittente, tranne che per i seguenti aspetti.

Visto che non c'è una well-known port per VoIP, non è da escludere che la porta di defoult (5004 per l'RTP e 5005 per l'RTCP) scelta sia gia in uso da un'altra applicazione[28]. Per evitare conflittualità, si informa, tramite un'opportuna chiamata di sistema (setsockopt [41]), il sistema operativo la volontà di utilizzare la porta scelta. Le socket aperte vengono connesse attraverso la chiamata di sistema "bind", in modo da legare la porta alla particolare applicazione. Inoltre, siccome la capienza del socket-buffer di ricezione non è nota a priori, si innesca una standardizzazione (setsockopt

[28] Per limitare quest'eventualità si è scelto un defoult che è al di fuori delle *emphemeral port*.

[41]) della capienza di tale buffer. Ciò non garantisce affatto l'assenza di overflow ma, comunque, tende a limitare i danni.

Dopo le precedenti inizializzazioni si innesca il principale *loop di ricezione*. Immediatamente, il processo si blocca in attesa di pacchetti in arrivo sulla socket preventivamente aperta. Ogni pacchetto ricevuto viene passato alle routine di immagazzinamento ed equalizzazione del jitter.

Jitter-buffer

Il jitter-buffer implementato ha il duplice compito di buffer e di equalizzazione della variabilità dei ritardi. L'implementazione è strettamente legata alla particolare applicazione. Nel caso di RTP-Speak, il jitter-buffer è una routine che riceve il pacchetto in ingresso e lo memorizza in una lista temporanea. La lista è ordinata secondo il *sequence number* presente nell'intestazione RTP d'ogni pacchetto ricevuto. Come è noto, i ritardi introdotti dai programmi per l'immagazzinamento dei dati cresce in modo esponenziale [46] [39] con il numero di quest'ultimi. Per evitare questa crescita esponenziale, il numero massimo di pacchetti presenti nella lista è fisso (defoult: 20 pacchetti). I pacchetti presenti nella lista sono solo quelli che non sono ancora stati decodificati. Ogni volta che un pacchetto è decodificato ed ascoltato, viene cancellato dalla lista. Quest'accortezza permette, non solo di evitare una crescita esponenziale dei ritardi dovuti all'algoritmo, ma anche che una conversazione venga interrotta per mancanza di memoria disponibile nello stack [39]. L'assenza di quest'accortezza non permetterebbe conversazioni senza limiti di tempo.

La routine di immagazzinamento fa la seguente cosa: all'inizio della sessione si ha un primo loop interno a quello principale che serve ad immagazzinare i primi 20 pacchetti ricevuti; successivamente, il loop interno non viene più eseguito ed ogni pacchetto ricevuto viene immagazzinato in modo ordinato, decodificato ed ascoltato. La routine d'immagazzinamento prevede anche che, se un pacchetto ha un ritardo eccessivo (arriva dopo del suo precedente che è stato già ascoltato) non va immagazzinato, poiché conviene considerarlo perso.

E' chiaro che la dimensione di defoult del buffer (20 pacchetti da 20 ms) è eccessiva per un'applicazione bidirezionale, poiché introduce 0,4 sec. di ritardo; quindi, da sola, va oltre il limite d'accettabilità dei ritardi (vedi par. I.2.a). Va da se che la scelta adottata è dovuta al fatto che la comunicazione è simplex, quindi questo ritardo si manifesta solo all'inizio ed al termine della sessione, di conseguenza è tollerabile.

Futuri sviluppi di RTP-Speak, che porteranno ad una comunicazione bidirezionale, potranno tenere conto dei criteri di progetto di cui si è parlato nel par. IV.2.b. In ogni caso, questo parametro è facilmente modificabile ed una limitata manovrabilità può essere messo a disposizione dell'utente.

Depacchettizazione RTP

In questa sede, per depacchettizazione RTP s'intende sia l'usufruire dell'informazione presente in un certo campo che la divisione tra header RTP e payload.

La lettura delle informazioni presenti nell'intestazione avviene in modalità sparsa a partire da quando un pacchetto è ricevuto fino all'innesco del decodificatore, mentre la separazione tra header e payload avviene nel momento in cui il decodificatore viene lanciato. Non esiste una particolare routine di separazione, semplicemente al decodificatore viene fornito ciò di cui ha bisogno.

Decodifica

Il decodificatore ha il compito di fornire al buffer-audio il tratto di voce ricostruito. L'ingresso del decodificatore è rappresentato dal payload RTP che contiene i 20 ms di voce codificati, l'uscita è un'array che ha il formato ricostruito (PCM-lineare a 16 bit per campione). L'ingresso va preso dal jitter-buffer attraverso un puntatore che contiene l'indirizzo dell'attuale pacchetto da elaborare. Anche le routine di decodifica, come quelle di codifica, sono suddivise in file separati. In questo modo si ha una grossa flessibilità implementativa.

Playout

Il playout è al termine del principale *loop di ricezione* (vedi fig. IV.4.1). La prima operazione che viene fatta è quella di scrivere, attraverso una semplice `write`, nel dispositivo audio, 20 ms (defoult) di voce decodificata così che possano essere ascoltati. Dopo che un pacchetto è stato ascoltato viene eliminato dalla lista del jitter-buffer, aggiornando i puntatori e liberando la memoria dello stack. Successivamente si riprende, elaborando il prossimo pacchetto; e così via fino alla terminazione del programma.

Anche il processo di ricezione, può essere interrotto dall'utente in modo asincrono, semplicemente digitando "^C". Il segnale di interrupt innesca la chiusura del processo, seguita da un breve messaggio di notifica:

```
$ Trasmission interrupted.
 54760 packets received.
```

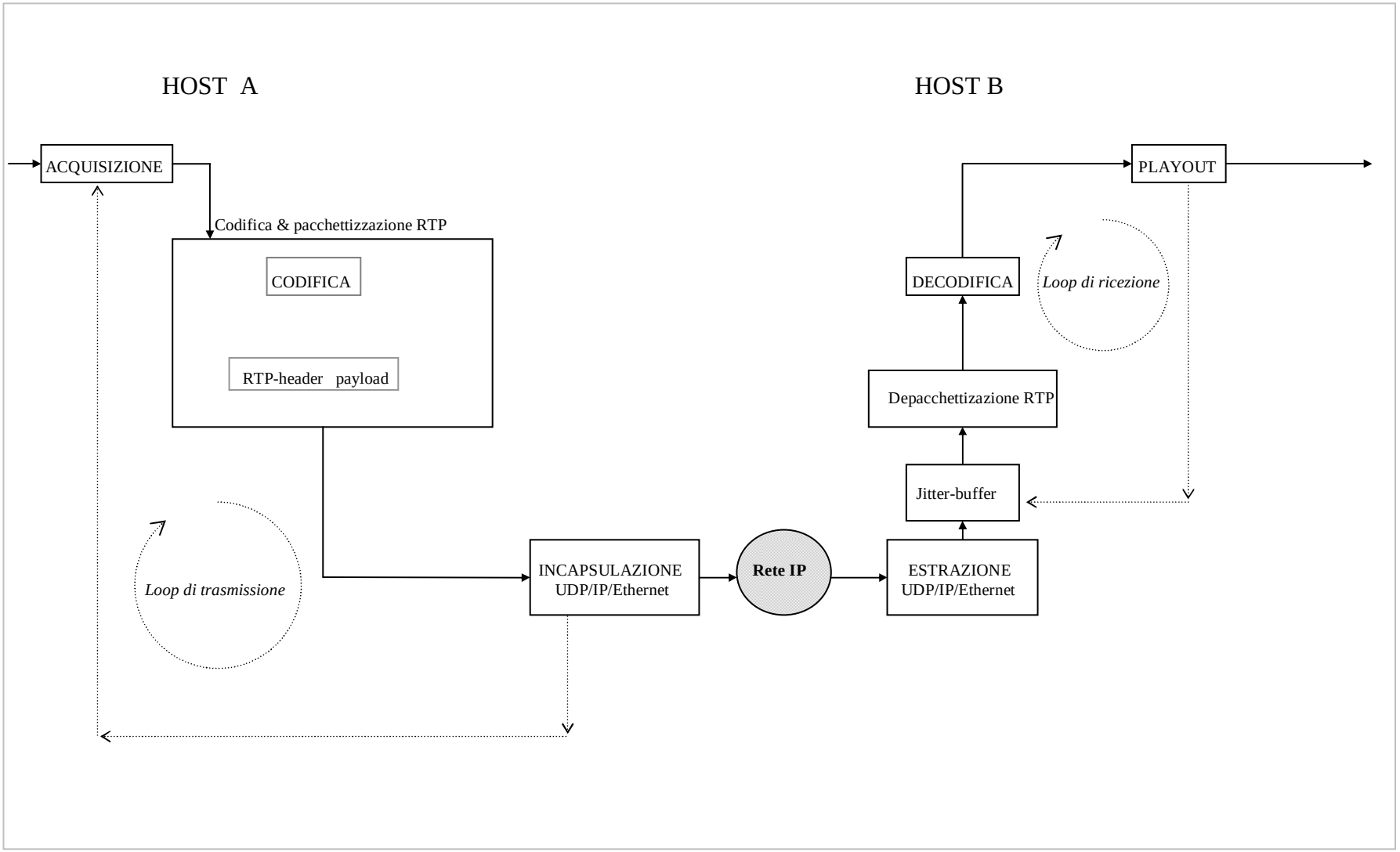


Fig. IV.4. Descrizione a blocchi dell'algoritmo di RTP-Speak.

IV.5 Prestazioni di RTP-Speak.

Il pacchetto software che compone RTP-Speak è stato testato su vari sistemi Linux e Unix. Le prove complete di ricetrasmissione sono state condotte sui PC connessi alla LAN del dipartimento di INFOCOM (INFOrmazione e COMunicazione). La topologia della LAN è a stella: ogni pacchetto trasmesso arriva ad uno *switch* [47] che lo dirotta verso la destinazione opportuna. La maggior parte delle prove è stata condotta su tre PC, le cui caratteristiche sono riassunte in tab. IV.5.

Indirizzo IP	SO	Processore
pcvoce2.ing.uniroma1.it	SuSE Linux 6.1	Pentium 166MHz
pcrondine.ing.uniroma1.it	SuSE Linux 6.2	486
Pcrondine.ing.uniroma1.it	SuSE Linux 6.3	486
pcsandro.ing.uniroma1.it	RedHat Linux 6.2	Pentium 133 MHz

Tab. IV.5.

I risultati ottenuti sono molto confortanti. In ogni caso, va fatta una certa distinzione tra i vari formati di codifica, in relazione alle prestazioni attualmente ottenute.
Nel caso si utilizzi la codifica *PCMμ* o *PCMA*, i risultati sono ottimi:

- La ricetrasmissione della voce avviene in maniera fluida e lineare. L'ascolto risulta gradevole.

- La perdita di pacchetti è sempre inferiore alla soglia limite del 5%, ma spesso, è intorno allo 0,4%. Non sono mai stati registrati burst of loss. Queste percentuali si riferiscono a flussi vocali di durata non inferiore a 20 minuti.

- Il jitter è sempre equalizzato.

Nel caso si utilizzi la codifica *G.729A* , i risultati sono i seguenti:

- La ricetrasmissione è meno fluida. Si avverte lo spezzettamento del flusso vocale dovuto ad un problema di sincronismo, non ancora completamente risolto.

- La perdita di pacchetti è sempre inferiore al 5%. Rispetto al G.711 si perde qualche pacchetto in più, ma la differenza è minima e l'ascoltatore non se ne accorge. Non sono mai stati registrati burst of loss.

- Il jitter è sempre equalizzato.

L'utilizzo del G.729A da luogo ad un flusso vocale non sempre gradevole, a causa della mancanza di fluidità. Il problema fondamentale cui bisogna far fronte è la gestione del ritardo endogenerato (15 ms) all'interno dei principali loop di ricetrasmissione (vedi fig. IV.4). Questo ritardo deve fare i conti con i tempi concessi dal sistema di doppia bufferizzazione dell'audio, secondo quanto detto nel par. IV.2.b. La cosa non è affatto insuperabile, anzi, i risultati ottenuti fin'ora sono molto confortanti. Infatti, sia il processo di codifica sia quello di decodifica funzionano correttamente. Ovviamente, la QoV è quella implicita al formato G.729A.

Nel complesso, RTP-Speak non ha mai generato problemi di overflow indipendentemente dalle differenti velocità dei processori delle macchine coinvolte (vedi tab. IV.5).

Per ciò che riguarda la QoV, i risultati pratici confermano le aspettative teoriche. Nonostante la notevole occupazione di banda, 64 kb/sec, la codifica PCM è preferibile, rispetto ad altre codifiche più stringenti, all'interno di una LAN. Infatti, una LAN è quasi sempre in grado di soddisfare tale richiesta di banda. Il PCM non solo riduce le richieste elaborative ma garantisce una qualità della voce eccellente. L'utilizzo del G.729A potrebbe essere preferito al quello del G.711 nel caso la trasmissione avvenga all'interno di Internet. In tal senso, una volta risolti i problemi di sincronismo, sarebbe interessante valutare le prestazioni dei due formati all'interno di Internet, sia in termini di ritardo che di perdita di pacchetti.

Il pregio principale di RTP-Speak è la sua *versatilità*. L'implementazione su più file separati ha permesso una separazione delle principali funzionalità, rendendo agevole la possibilità di modifiche e aggiunte.

Particolare riguardo è stato dedicato all'*interoperabilità* con altre soluzioni implementative. In particolare, si è fatto in modo di rendere ogni elaborazione indipendente dalla modalità di ricetrasmissione. In pratica, ogni pacchetto trasmesso o ricevuto è semplicemente una stringa di caratteri; solamente dopo la ricezione (o prima della trasmissione) la stringa ha il significato di RTP-header/payload. Non si fa nessuna assunzione a priori sulla lunghezza del pacchetto o sul significato dei vari campi. In questo modo l'implementazione delle varie routine elaborative è resa indipendente dal formato del pacchetto ricevuto o trasmesso. Tuttavia, va detto che non è stato ancora possibile testare sul campo l'interoperabilità, perciò le precedenti considerazioni vanno prese come considerazioni teoriche non ancora verificate. La verifica dell'interoperabilità necessita di un accorgimento: RTP-Speak assume che i dati siano trasmessi e ricevuti dalle porte 5004 e 5005, ma non è detto che sia sempre così, perciò è potrebbe essere necessario adeguare i numeri di porta.

Va da se che RTP-Speak è una versione molto sottile (circa 1300 righe di codice, circa 800 se si esclude la codifica) di un sistema di comunicazione VoIP, ma ciò ha i suoi vantaggi:

- E' possibile cogliere gli aspetti essenziali della trasmissione di voce su reti a commutazione di pacchetto, senza perdersi in decine di migliaia di righe di codice.
- E' possibile plasmarlo a proprio piacimento e dirottarlo verso un qualsiasi utilizzo.
- E' possibile sfruttare le sole proprietà di base della trasmissione vocale su reti IP, senza dover sopportare il carico di pesanti algoritmi.

I futuri sviluppi di RTP-Speak possono essere molteplici: si può costruire un applicativo più completo (secondo la scia del progetto di fig. IV.1), si può adattarlo in modo che funga da tester per codificatori audio su canali inaffidabili (quale è la rete IP), si può immaginare una sostra di e-mail vocale, e cosi via. Per quanto riguarda lo sviluppo legato alla fig. IV.1 molte indicazioni sono state gia date nel corso del capitolo e molte altre ne serviranno. Gran parte delle considerazioni fatte valgono anche nel caso lo sviluppo verta verso l'e-mail vocale.

Nel caso lo si voglia utilizzare come tester, va già bene, tuttavia, sarebbe opportuno inserire ulteriori funzionalità. La prima cosa da fare sarebbe di facilitare la modalità di inserimento del codificatore audio (attualmente l'unica modalità è quella di inserirlo dietro l'esempio del G.711). Inoltre, sarebbe opportuno poter variare, più parametri (es., la dimensione del buffer-audio, la modalità di I/O e quant'altro) direttamente dal lato dell'interfaccia utente, e magari tramite una GUI (anziché da prompt). La variabilità della dimensione del jitter-buffer può essere un modo per valutare la capacità di recupero, da parte del decodificatore, (es., interpolazione, ricostruzione,…) dei pacchetti persi, al variare del numero di pacchetti presenti nel buffer.

In ogni caso, vale la pena ribadire che, RTP-Speak, è già un'applicativo sufficientemente completo, in grado di garantire il trasporto di voce in tempo reale su reti IP.

Appendice A

Codici sorgente e manuale d'uso di RTP-Speak.

Questa appendice contiene tutti i programmi scritti ed adoperati per l'implementazione di RTP-Speak. Il software è scritto in linguaggio C ed è stato sviluppato su piattaforme LINUX e UNIX. La maggior parte del codice è conforme allo standard ANSI (American National Standard Institute) ma alcune parti sono legate alle tipiche chiamate di sistema UNIX; per cui, per l'applicabilità ad altri sistemi, ad esempio Windows, sono necessarie alcune modifiche. In ogni caso, il software è adatto a tutti i sistemi POSIX-compatibili supportanti i sockets.

Tutti i programmi sono stati testati con successo su varie piattaforme LINUX e UNIX. Il compilatore utilizzato è il *gcc*. Universalmente disponibile è la versione fornita dalla GNU [48].

L'appendice è organizzata nel seguente modo. Nell'app. A.1 sono riportati i listati completi di tutti i programmi, compreso il file Make [49], relativi alla versione di RTP-Speak che supporta i codificatori PCMA e PCMμ. Nell'app. A.2 si riporta un breve manuale d'uso di RTP-Speak. Nell'app. A.3 sono mostrati i listati della versione di RTP-Speak, che oltre al PCMA e al PCMμ, supporta il G.729. L'appendice A.3 è dedicata alla discussione delle modifiche apportate al software G.729, fornito dall'ITU-T, per renderlo applicabile alla comunicazione su reti IP; ciò sarà utile per ulteriori sviluppi del presente lavoro. Infine, nell'app. A.3 sono riportate alcune routine di utilità generale.

A.1 RTP-Speak (listati dei programmi).

Nel seguito si riportano i listati dei programmi che compongono RTP-Speak (supportante la codifica PCMA e PCMμ). Pur rischiando di perdere in leggibilità, si è scelto di riportare un buon numero di commenti in modo da rendere più agevole l'interpretazione delle varie istruzioni.

Il codice è organizzato su più file separati. Ogni programma riportato è preceduto dal nome del file (*.c, *.h, Makefile) che lo contiene. Tutti i codici sorgente sono stati depositati presso la segreteria didattica del dipartimento di INFOCOM (INFOrmazione e COMunicazione).

Makefile

```
#############################################################################
#############################################################################
#                                                                           #
# File Make per RTP-Speak.                                                  #
# Versione 1.0 (Maggio 2001)                                                #
#                                                                           #
#############################################################################
#############################################################################
#                                                                           #
# Autore: Daniele Mari.                                                     #
# e-mail: daniele.mari@tiscalinet.it                                        #
#                                                                           #
#############################################################################
#############################################################################

CC = gcc

# -O2
CFLAGS = -g -Wall -O2

LFLAGS =  -lm

# sorgenti per la parte di trasmissione
TSRCS = set_snd.c
```

```
# sorgenti per la parte di ricezione
RSRCS = set_snd.c

#########################################################################
Lin

COBJS = $(TSRCS:.c=.o) cg711.o

DOBJS = $(RSRCS:.c=.o) dg711.o

all: vmailto rcvmsg

cg711.o:     g711.c
 $(CC) -g -c -o cg711.o $(CFLAGS) -DCODER  g711.c

dg711.o:     g711.c
 $(CC) -c -o dg711.o $(CFLAGS) -DDECODER g711.c

rtpmail.o:   rtpmail.c
 $(CC) -g -c $(CFLAGS) rtpmail.c

openline.o:  openline.c
 $(CC) -c $(CFLAGS) openline.c

vmailto:     $(COBJS)  rtpmail.o
 $(CC) -g -o vmailto $(COBJS) rtpmail.o $(CFLAGS) $(LFLAGS)

rcvmsg: $(DOBJS) openline.o
 $(CC) -o rcvmsg $(DOBJS) openline.o $(CFLAGS) $(LFLAGS)


p-rcvmsg: $(DOBJS) openline.o
 purify $(CC) -o p-rcvmsg $(DOBJS) openline.o $(CFLAGS) $(LFLAGS)

SDSRC = $(RSRCS) g711.c openline.c

s-rcvmsg: $(SDSRC)
 #load $(CFLAGS) $(SDSRC)

#########################################################  Cleanup

lin-clean:
 -rm -f *.o
```

```
clean: lin-clean
```

rtpmail.c

```
/*
 ************************************************************************
 ************************************************************************
 * RTP-Speak.                                                           *
 * Versione 1.0 (Maggio 2001)                                           *
 * Principale routine di trasmissione.                                  *
 * Gestisce:                                                            *
 * 1. L'acquisizione da microfono                                       *
 * 2. La codifica                                                       *
 * 3. La trasmissione                                                   *
 ************************************************************************
 ************************************************************************
 * Autore: Daniele Mari                                                 *
 * e-mail: daniele.mari@tiscalinet.it                                   *
 ************************************************************************
 ************************************************************************
                                                                       */



#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <strings.h>                // bzero()
#include <sys/soundcard.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <signal.h>                 // signal()
#include <arpa/inet.h>              // inet_pton()
#include <sys/socket.h>             // struct sockadrr
#include <netinet/in.h>             // sockaddr_in
#include <netdb.h>                  // gethostbyname()
#include <stdlib.h>                 // perror()
#include "rtp.h"                    // RTP headers

#define       MAXLEN 350      /*
                               * massimo num di byte associabili ad
```

```
                                * un pacchetto RTP (è  consigliato non
                                * superare questa soglia)
                                */
static int loop = 0;

/*
 * Dichiarazioni globali per l'acqusizione audio
 *
 */

static char au_name[15] = "/dev/dsp";
int dsp_stereo = 0;      // mono
int dsp_speed = 8000;    // PCMA (defoult), PCMU, G729
extern int au_fd;
char *audio_buff;
int dsp_speed, aubuf_size;

typedef struct {
 char *encoder;
 int  rate;
 int  channel;
} enc_map;
static enc_map pt_map[128];

static int sockfd[2];    // output socket

/*
 * La funzione che segue stampa un breve help.
 */
static void usage (char *processname)
{

 fprintf(stderr,"\n");
 fprintf(stderr,"-----------------------------------------------\n");
 fprintf(stderr,"******************RTP-Speak******************\n");
 fprintf(stderr,"*************Version 1.0, May 2001************\n");
 fprintf(stderr,"\n");
 fprintf(stderr,">>>>>>>>>>>>>Designed by: Daniele Mari<<<<<<<<<<<\n");
 fprintf(stderr,"-----------------------------------------------\n");
 fprintf(stderr,"\n");
 fprintf ( stderr,
         "Usage: %s [-d devicename][-c codername] destination\n",
          processname);
 fprintf(stderr,"\t-d\t\tdevice name\n");
```

```
 fprintf(stderr,"\t-c\t\ttype of encoding name\n");
 fprintf(stderr,
         "\t\t\tAvailables coders are: PCMU, PCMA, G729\n");
 fprintf(stderr,"\tdestination\tIP address or host_name to
        contact\n");
fprintf(stderr, "\t escape character is : ^C\n");

 exit(1);

} /* usage */



/*
 * La funzione che segue verifica la presenza del tipo di codificatore
 * nella ‘pt-map’.
 * In caso di successo restituisce il PT corrispondente, altrimenti –1.
 */
static int search( enc_map *ptr, char *name)
{
 register int i;

 for(i=0;i<128;i++)
  if( !strcmp(ptr[i].encoder, name) ) return i;
 return -1;

} /* search */


/*
 * Qesta funzione inizializza la struttura (IPv4 sockaddr) del
 * destinatario con l'indirizzo IP (fornito da riga di comando)ed il
 * numero di porta (di defoult, 5004) di quest'ultimo.
 * L'indirizzo IP può essere sia in formato dotted-decimal
 * (151.100.9.29)che in formato hostname (pcvoce2.ing.uniroma1.it).
 */
static void set_addr( char *h, struct sockaddr *sa)
{

 struct hostent *hptr;
 struct sockaddr_in *servaddr=(struct sockaddr_in*)sa;

 // tipica inizializzazione a 0 della struttura 'sockaddr'
 bzero(servaddr,sizeof(*servaddr));
```

```
 servaddr->sin_family = AF_INET;
 servaddr->sin_port = htons(5004);

 // Se l'IPv4 è in formato hostname, lo si riconverte via DSN.
 if( (hptr = gethostbyname(h)) ) {
   servaddr->sin_addr = *(struct in_addr*)(hptr->h_addr_list[0]);
 }
 // Verifica se l'IPv4 è in formato dotted-decimal
 else if( (inet_pton(AF_INET,h,&servaddr->sin_addr)) <=0 ) {
   fprintf(stderr,
    "inet_pton error for %s. Maybe destination have no valid
     format\n", h);

   exit(1);
 }

} /* set_addr */


/*
 * Funzione che genera l'intestazione RTP.
 *
 * La variabile 'packet' è di output e conterrà 12 byte di
 * intestazione RTP, ai quali, seguirà il payload fornito dal
 * codificatore.
 *
 * La variabile 'map_position' è di input e serve ad indicare il
 * valore del payload-type.
 *
 * (N.B. IL PUNTATORE 'h' E' TALE DA CONSERVARE IL VALORE ASSEGNATOGLI
 * TRA UNA CHIAMATA E LA SUCCESSIVA ---> poiché, puntando
 * all'indirizzo di 'packet', il valore precedentemente assegnatogli
 * viene mantenuto).
 *
 * La variabile 'loop' è di input e serve a sapere se si stà trattando
 * il primo pacchetto o i successivi.
 *
 * La variabile 'numBp' è di input e serve ad indicare la lunghezza,
 * in byte, del payload in base al tipo di codifica.
 *
 * La variabile 'padding' è di input e serve ad indicare quanti byte
 * di padding sono presenti nel pacchetto.
 */
```

```
void rtp(char *packet, int map_position, int loop, int numBp, int
         padding)
{

 rtp_hdr_t *h = (rtp_hdr_t *)packet;

 /*
  * campi fissi.
  */

 // versione RTP
 h->version = RTP_VERSION;

 // padding-bit
 if( !padding )
  h->p = 0;
 else
  h->p = 1;

 // header extension
 h->x=0;

 // numero di CSRC
 h->cc=0;

 // marker-bit
 h->m=0;

 // payload type
 h->pt = map_position;

 /*
  * campi variabili
  */

 if( loop == 1 ) {

  // sequence number
  h->seq=htons( rand() );

  // timestamp
  h->ts=htonl( rand() );

  // syncronization source (SSRC) identifier
```

```
  h->ssrc=htonl( rand() );

 }
 else { // incremento dei campi variabili

  h->seq = htons( ntohs(h->seq) + 1 );

  h->ts = htonl( ntohl(h->ts) + numBp );

 }

 // contributing source (CSRC) supposti assenti

} /* rtp */


/*
 * Funzione di fine.
 * Stampa un breve messaggio di notifica.
 * E' innescata, in modo asincrono, dall'utente mediante
 * il comando: ^C
 */
static void stop(int signo)
{

 fprintf(stderr,"Trasmission interrupted\n");
 fprintf(stderr,"%d packets sent\n",loop);
 fprintf(stderr, "Bye :-)\n");
 fprintf(stderr, "\n");

 exit(0);

} /* stop */


/*
 * Funzione principale
 */
int main (int argc, char *argv[])
{
 static struct sockaddr_in servaddr;
 extern char *optarg;
 extern int optind,opterr,optopt;
 int map_position;       // Posizione del codificatore nella 'pt_map'.
```

```
char packet[MAXLEN];
int p,c,i;
const int on;
int padding = 0;        // # di padding-byte in un pacchetto
int numBp = 160;        // # byte per pacchetto (defoult: PCMA,20ms)
char cod_name[5]="PCMA"; // defoult: A-law (Europa)
extern void set_snd(char *audio_name, int open_mod, int dsp_stereo);
extern int g711(char type_of_conv[2], char *data_in, char *data_out,
                int padding);

// Elabora i comandi forniti da prompt.
while( (c = getopt(argc, argv, "d:c:h")) != -1 ) {
 switch(c) {
 case 'd':
   strncpy(au_name, optarg, 15);
   break;
 case 'c':
   strncpy(cod_name, optarg, 5);
   break;
 case '?':
 case 'h':        // h stà per help.
   usage(argv[0]);
   break;
 }
}

fprintf(stderr,"\n");
fprintf(stderr,"-----------------------------------------------\n");
fprintf(stderr,"********************RTP-Speak******************\n");
fprintf(stderr,"**************Version 1.0, May 2001*************\n");
fprintf(stderr,"\n");
fprintf(stderr,">>>>>>>>>>>>Designed by: Daniele Mari<<<<<<<<<<<\n");
fprintf(stderr,"-----------------------------------------------\n");
fprintf(stderr,"\n");
fprintf(stderr, "escape character is : ^C\n");
fprintf(stderr, "\n");


if(optind == argc) {
 fprintf(stderr,"Error: destination (IP address or hostname)
         missing\n");
 usage(argv[0]);
}
else {
```

```
 // Si inizializza l'indirizzo sockaddr a cui inviare i dati
 set_addr(argv[optind], (struct sockaddr *)&servaddr);

 // si creano due socket UDP-nonconnesse per l'RTP e l'RTCP
 for(i=0;i<2;i++) {
  sockfd[i] = socket(AF_INET, SOCK_DGRAM, 0);
  if( sockfd[i] < 0 ) {
   perror("socket");
   exit(1);
  }
  servaddr.sin_port = htons( ntohs(servaddr.sin_port) +i );
 }

} /* else */

// Messa in piedi dalla lista payload type
for (i=0; i<128; i++) {
 pt_map[i].encoder = "xxxx";
 pt_map[i].rate    = 0;
 pt_map[i].channel = 0;
}

pt_map[0].encoder="PCMU";pt_map[0].rate=8000;pt_map[0].channel=1;
pt_map[8].encoder="PCMA";pt_map[8].rate=8000;pt_map[0].channel=1;
pt_map[18].encoder="G729";pt_map[18].rate=8000;pt_map[18].channel=1;

i = search(pt_map, cod_name);
if( i == -1 ) {
 fprintf(stderr,"Connection impossible: %s unavailable.\n",cod_name);
 fprintf(stderr,"Availables coders are: PCMU, PCMA, G729,\n");
 exit(1);
}
else map_position = i;

/*
 ********************** PERCORSO DI TRASMISSIONE DI ***************
 ***************************OGNI SINGOLO PACCHETTO******************
                                                                    */

for(i=0;i<2;i++) {
if( setsockopt(sockfd[i], SOL_SOCKET, SO_BROADCAST, &on,
    sizeof(on)) < 0 )
 perror( "SO_BROADCAST" );
```

```
}

// Processo d'acqisizione, da microfono, dei dati audio
set_snd(au_name, O_RDONLY, dsp_stereo);

signal(SIGINT, stop);

// Elaborazione dei pacchetti
for( ; ; ) {

/*
 * Leggi la voce registrata.
 * PCM Lineare, 16 bits/campione ---> 20ms = 320 byte
 */
 if( ( p = read(au_fd, audio_buff,320)) == -1 ) {
    perror(au_name);
    exit(1);
   }
 if( p < 320 )
  padding = 320 - p;

 loop ++;

 // Elabora i dati in base al tipo di codifica.
 switch( map_position ) {
 case 0: // PCMU
 case 8: // PCMA

   // RTP-header
   rtp(packet, map_position, loop, numBp, padding);

   // Inizia le appropriate routine di codifica
   if( map_position == 8 )
    g711("la", audio_buff, packet+12, padding);
   else
    g711("lu", audio_buff, packet+12, padding);

   break;
 case 18:  // G.729
   numBp = 20;
   fprintf(stderr,
    "Sorry, %s coder is unavailable at moment\n",
    pt_map[18].encoder);
   exit(0);
```

```
    break;
  }

 // Spedisci l'attuale pacchetto
 if( sendto(sockfd[0], packet, numBp+12,
     0, (struct sockaddr *)&servaddr, sizeof(servaddr)) == -1 )
  perror("sendto");

 } /* for */


 return 0;

} /* main */
```

openline.c

```
/*
 ************************************************************************
 ************************************************************************
 * RTP-Speak.                                                           *
 * Principale routine di ricezione.                                     *
 * Versione 1.0 (Maggio 2001)                                           *
 * Gestisce:                                                            *
 * 1. La ricezione dall'interfaccia di rete                             *
 * 2. La decodifica                                                     *
 * 3. Il playout                                                        *
 ************************************************************************
 ************************************************************************
 * Autore: Daniele Mari                                                 *
 * e-mail: daniele.mari@tiscalinet.it                                   *
 ************************************************************************
 ************************************************************************
                                                                       */

#include <stdio.h>
#include <unistd.h>
#include <stddef.h>
#include <string.h>
```

```
#include <strings.h>          // bcopy(), bzero()
#include <sys/soundcard.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/stat.h>
#include <signal.h>           // signal()
#include <sys/socket.h>       // struct sockadrr
#include <netinet/in.h>       // sockaddr_in
#include <stdlib.h>           // perror()
#include "rtp.h"              // RTP headers


#define  MAXLEN 350     /*
                         * massimo num di byte associabili ad un
                         * pacchetto RTP (è consigliato non superare
                         * questa soglia)
                         */

#define RECV 1

/*
 * Dichiarazioni globali per l'acqusizione audio
 *
 */

static char au_name[15] = "/dev/dsp";

int dsp_stereo = 0;    // mono
int dsp_speed = 8000;  // PCMA (defoult), PCMU, G729
extern int au_fd;
char *audio_buff;
int dsp_speed, aubuf_size;

/*
 * Lista dei pacchetti ricevuti (n.b. la lista è dinamica.)
 * I pacchetti in lista sono equalizzati dal jitter.
 */
struct packlist {
 char data[MAXLEN];           // RTP-header + payload
 struct packlist *next;       // puntatore all'elemanto successivo
} *packinfo;
struct packlist *start, *last;     /*
                                    * puntatori al primo e all'ultimo
```

```
                                    * elemento della lista
                                    */
struct packlist *prev=0;       // puntatore al pacchetto già elaborato

// pacchetti ricevuti da equalizzare
char packet[MAXLEN];

static int sockfd[2];  // input socket


static int pcount = 0;



/*
 * Questa funzione inserisce in una lista i pacchetti in arrivo
 * secondo l'ordine dettato dal 'sequence number', jitter equalizzato,
 * in modo che possano essere passati al decodificatore.
 * La lista è aggiornata dinamicamente.
 */
static
void store(struct packlist *i, struct packlist **start,
           struct packlist **last, int lseq)
{

 struct packlist *old, *p;
 rtp_hdr_t *hpi = (rtp_hdr_t *)(i->data);
 rtp_hdr_t *hpp;

 /*
  * Se il pacchetto è in eccessivo ritardo, non va messo in lista.
  * 'lseq'  indica il sequence number dell'ultimo pacchetto ascoltato.
  */
 if( ntohs(hpi->seq) < lseq )
  goto no_store;

 p = *start;
 hpp =(rtp_hdr_t *)(p->data);

 if(!*last) { // primo elemento della lista (prima volta)
  i->next = NULL;
  *last = i;
  *start = i;
  return;
```

```
 }

 old = NULL;

 while(p) {
  if( ntohs(hpp->seq) < ntohs(hpi->seq) ) {
   old = p;
   p = p->next;
  }
  else {
   if(old) { // elemento mediano
    old->next = i;
    i->next = p;
    return;
   }
   i->next = p; // primo elemento
   *start = i;
   return;
  }
 } /* while */

 (*last)->next = i;  // ultimo elemento
 i->next = NULL;
 *last = i;

 no_store:

}  /* store */


/*
 * Funzione di fine.
 * Stampa un breve messaggio di notifica.
 * E' innescata, in modo asincrono, dall'utente mediante
 * il comando: ^C
 */
static void done(int signo)
{

 fprintf(stderr,"Trasmission interrupted\n");
 fprintf(stderr,"%d packets received\n",pcount);
 fprintf(stderr, "Bye :-)\n");
 fprintf(stderr, "\n");
```

```
 exit(0);

}


/*
 * Funzione principale
 */
int main(void)
{

 struct sockaddr_in servaddr, cliaddr;
 int clilen;
 int len;                /*
                          * # di byte per pacchetto (header+payload),
                          * defoult: 20ms per pacchetto.
                          */
 const int on = 1;
 int n,i;
 int padding = 0;        // # di byte di padding in un pacchetto
 int pad = 0;            // # byte di padding non decodificati
 rtp_hdr_t *hp;
 u_int16 lseq = 0;
 extern void set_snd(char *audio_name, int open_mod, int dsp_stereo);
 extern int g711(char type_of_conv[2], char *data_in, char *data_out,
        int padding);



 /*
  * Si aprono due socket UDP non connesse, per i pacchetti RTP ed
  * RTCP.
  */
 bzero(&servaddr, sizeof(servaddr));

 servaddr.sin_family = AF_INET;
 servaddr.sin_port   = htons(5004);
 servaddr.sin_addr.s_addr = htonl(INADDR_ANY);

 for(i=0;i<2;i++) {
  sockfd[i] = socket(AF_INET, SOCK_DGRAM, 0);
  if( sockfd[i] < 0 ) {
   perror("socket");
   exit(1);
```

```
  }
  servaddr.sin_port = htons( ntohs(servaddr.sin_port) + i );

 /*
  * Opzione SO_REUSEADDR. E' cautelativa nel caso sia stata già aperta
  * una connessione su questa porta. */
 if( setsockopt(sockfd[i], SOL_SOCKET, SO_REUSEADDR, &on,
     sizeof(on)) == -1 )
  perror("setsockopt (SO_REUSEADDR)");

 //  Si incrementa il socket-receive-buffer ad un valore ragionevole
 n = 240 * 1024;
 if(setsockopt(sockfd[i], SOL_SOCKET, SO_RCVBUF, &n, sizeof(n)) == -1)
  perror("setsockopt (SO_RCVBUF)");

 if(bind(sockfd[i],(struct sockaddr *)&servaddr,sizeof(servaddr)) < 0)
{
  perror("bind");
  exit(1);
 }
} /* for */

 start = last = NULL;

 fprintf(stderr,"\n");
 fprintf(stderr,"-------------------------------------------------\n");
 fprintf(stderr,"********************RTP-Speak******************\n");
 fprintf(stderr,"**************Version 1.0, May 2001*************\n");
 fprintf(stderr,"\n");
 fprintf(stderr,">>>>>>>>>>>>Designed by: Daniele Mari<<<<<<<<<<<\n");
 fprintf(stderr,"-------------------------------------------------\n");
 fprintf(stderr,"\n");
 fprintf(stderr, "\t escape character is : ^C\n");
 fprintf(stderr, "\n");


/*
 * Inizializzazione della scheda audio per il playback.
 */
 set_snd(au_name, O_WRONLY, dsp_stereo);

 signal(SIGINT, done);

 // Principale Loop di ricezione
```

```
while(RECV) {

 fill_buffer:

 clilen = sizeof(cliaddr);

 len = recvfrom(sockfd[0], packet, MAXLEN,
                0,(struct sockaddr *)&cliaddr, &clilen);

/*
 * Routine di immagazzinamento nella lista a jitter equalizzato.
 */
 packinfo = (struct packlist *)malloc(sizeof(struct packlist));
 if(!packinfo)
  fprintf(stderr,"Warning, no more memory for incoming packets\n");

 bcopy(packet , packinfo->data, len );

 store(packinfo, &start, &last, lseq);

 pcount++;

 // jitter-buffer (20 pacchetti)
  /*
   * momentaneamente, la soluzione è quella di riempire il buffer
   * con un numero di pacchetti '20' sufficiente ad eliminare il
   * jitter e, contemporaneamente, a non ritardare troppo l'emissione
   * dei pacchetti verso il decodeificatore.
   * Il Jitter-Buffer, una volta riempito con i primi '20' pacchetti,
   * ne conterrà sempre '20'.
   */
 if( pcount < 20 )
  goto fill_buffer;

 fprintf(stderr,"%d packet parsed from jitter-buffer\r",pcount);

 // aggiorna il puntatore'hp' al pacchetto da ascoltare.
 hp = (rtp_hdr_t *)(start->data);
 lseq = ntohs( hp->seq );

 // Individua il decodificatore appropriato attraverso il PT
 switch( hp->pt ) {
 case 0: // PCMU
 case 8: // PCMA
```

```
  // valuta il campo di padding
  if( hp->p == 1 ) {
  padding = start->data[160-1];
  pad = start->data[160-1] * 2;
 }

   /*
    * Innesca le appropriate routine di decodifica e
    * depacchettizzazione
    */
  if( hp->pt == 8 )
   g711("al",&(start->data[12]), audio_buff, padding);
  else
   g711("ul",&(start->data[12]), audio_buff, padding);

  break;
case 18:  // G.729

  fprintf(stderr,
   "Sorry, G.729 coder is unavailable at moment\n");
  exit(0);

  break;

}

/*
 * Playback.
 * Ogni pacchetto ricevuto è decodificato in
 * PCM lineare, 16 bits/campione ---> 20ms = 320 byte.
 * Attenzione al padding.
 */
if( write(au_fd, audio_buff, (320-pad) ) != (320-pad) ) {
 perror(au_name);
 exit(1);
}


/*
 * La lista, che verrà scandita per la memorizzazione, conterrà
 * sempre solo '20' pacchetti e non di più. In questo modo si
 * diminuirà notevolmente il tempo di ricerca del punto di
 * memorizzazione, arginando, si spera, il rischio di OVERFLOW.
```

```
   *
   * Si libera la memoria, con free(), per evitare che una lunga
   * conversazione venga meno per mancanza di memoria da allocare.
   */
 prev = start;
 start = start->next;
 free(prev);

 } /* while */

return 0;

} /* main */
```

set_snd.c

```
/*
 ********************************************************************
 ********************************************************************
 * Questo file contiene la funzione 'set_snd()' che realizza routine*
 * generali di setup, valide sia per la registrazione che per il    *
 * playback dell'audio.                                             *
 *                                                                  *
 * Versione 1.0 (Maggio 2001)                                       *
 *                                                                  *
 * La 'vm_toall.c' la richiama così:                                *
 *   set_snd(au_name, O_RDONLY, dsp_stereo);                        *
 *                                                                  *
 * la 'oplnall.c'  la richiama così:                                *
 *   set_snd(au_name, O_WRONLY, dsp_stereo);                        *
 ********************************************************************
 ********************************************************************
 * Autore: Daniele Mari                                             *
 * e-mail: daniele.mari@tiscalinet.it                               *
 ********************************************************************
 ********************************************************************
                                                                    */

#include <stdio.h>
#include <stdlib.h>          // perror()
#include <sys/ioctl.h>       // ioctl()
```

```
#include <fcntl.h>
#include <sys/soundcard.h>


int samplesize = 16;          // (defoult) PCM-lineare 16 bit/sample
extern int dsp_speed, aubuf_size;
extern char *audio_buff;
int au_fd;
int profile = APF_NORMAL;


void set_snd(char *audio_name, int open_mod, int dsp_stereo)
{
 int app;

 if( (au_fd = open(audio_name, open_mod, 0)) == -1 ) {
  perror(audio_name);
  exit(1);
 }

 // set-up del numeri di bit per campione.
 app = samplesize;
 ioctl(au_fd, SNDCTL_DSP_SAMPLESIZE, &samplesize);
 if( app != samplesize ) {
  fprintf(stderr, "Unable to set the samplesize (%d bit)\n",app);

  exit(1);
 }

 // set-up del profilo audio
 ioctl(au_fd,SNDCTL_DSP_PROFILE, &profile);

 // set-up del numero di canali  2stereo/1mono
 if( ioctl(au_fd, SNDCTL_DSP_STEREO, &dsp_stereo) == -1 ) {
  fprintf(stderr, "Unable to set stereo/mono\n");
  exit(1);
 }

 // set-up dei parametri di campionamento
 if( ioctl(au_fd, SNDCTL_DSP_SPEED, &dsp_speed) == -1 ) {
  fprintf(stderr, "Unable to set audio speed (sampling frequency)\n");
  exit(1);
 }
```

```
 // riscontro sulla taglia del buffer audio
 ioctl(au_fd, SNDCTL_DSP_GETBLKSIZE, &aubuf_size);
 if( aubuf_size < 1 ) {
  perror("SNDCTL_DSP_GETBLKSIZE");
  exit(1);
 }

 /*
  * Allocazione dinamica della memoria per il buffer audio.
  *
  * Dopo che registro vado a leggere da 'au_fd' 320 byte (20ms di PCM)
  * e li pongo in 'audio_buf'; quindi, se 'aubuf_size' di defoult è
  * minore dovrò allocare più memoria, altrimenti vanno bene
  * aubuf_size-byte.
  * Per il playback vale la stessa cosa.
  */
 if ( aubuf_size < 320 )
  aubuf_size = 320;
  if( (audio_buff = malloc(aubuf_size)) == NULL ) {
   fprintf(stderr, "Unable to allocate memory for I/O audio
           buffer\n");
   exit(1);
  }

 // messaggio per l'utente
 fprintf(stderr, "Speed %d Hz %d bps", dsp_speed, samplesize);
 if( dsp_stereo )
  fprintf(stderr, "(stereo)\n");
 else
  fprintf(stderr, "(mono)\n");
/*
 * N.B. quest'ultimi messaggi li ho scritti su 'stderr' anche se non
 * sono messaggi d'errore perchè 'stdout' potrebbe essere occupato,
 * per defoult, come canale d'uscita dei dati audio.
 */

} /* set_snd */
```

rtp.h

```
/*
```

```
 * rtp.h  --  RTP header file (RFC 1890)
 */
#include <sys/types.h>

/*
 * The type definitions below are valid for 32-bit architectures and
 * may have to be adjusted for 16- or 64-bit architectures.
 */
typedef unsigned char  u_int8;
typedef unsigned short u_int16;
typedef unsigned int   u_int32;
typedef          short int16;

/*
 * Current protocol version.
 */
#define RTP_VERSION    2

#define RTP_SEQ_MOD (1<<16)
#define RTP_MAX_SDES 255      /* maximum text length for SDES */

typedef enum {
    RTCP_SR   = 200,
    RTCP_RR   = 201,
    RTCP_SDES = 202,
    RTCP_BYE  = 203,
    RTCP_APP  = 204
} rtcp_type_t;

typedef enum {
    RTCP_SDES_END   = 0,
    RTCP_SDES_CNAME = 1,
    RTCP_SDES_NAME  = 2,
    RTCP_SDES_EMAIL = 3,
    RTCP_SDES_PHONE = 4,
    RTCP_SDES_LOC   = 5,
    RTCP_SDES_TOOL  = 6,
    RTCP_SDES_NOTE  = 7,
    RTCP_SDES_PRIV  = 8
} rtcp_sdes_type_t;

/*
 * RTP data header
 */
```

```
typedef struct {
    unsigned int version:2;   /* protocol version */
    unsigned int p:1;         /* padding flag */
    unsigned int x:1;         /* header extension flag */
    unsigned int cc:4;        /* CSRC count */
    unsigned int m:1;         /* marker bit */
    unsigned int pt:7;        /* payload type */
    u_int16 seq;              /* sequence number */
    u_int32 ts;               /* timestamp */
    u_int32 ssrc;             /* synchronization source */
    u_int32 csrc[1];          /* optional CSRC list */
} rtp_hdr_t;

/*
 * RTCP common header word
 */
typedef struct {
    unsigned int version:2;   /* protocol version */
    unsigned int p:1;         /* padding flag */
    unsigned int count:5;     /* varies by packet type */
    unsigned int pt:8;        /* RTCP packet type */
    u_int16 length;           /* pkt len in words, w/o this word */
} rtcp_common_t;

/*
 * Big-endian mask for version, padding bit and packet type pair
 */
#define RTCP_VALID_MASK (0xc000 | 0x2000 | 0xfe)
#define RTCP_VALID_VALUE ((RTP_VERSION << 14) | RTCP_SR)

/*
 * Reception report block
 */
typedef struct {
    u_int32 ssrc;             /* data source being reported */
    unsigned int fraction:8;  /* fraction lost since last SR/RR */
    int lost:24;              /* cumul. no. pkts lost (signed!) */
    u_int32 last_seq;         /* extended last seq. no. received */
    u_int32 jitter;           /* interarrival jitter */
    u_int32 lsr;              /* last SR packet from this source */
    u_int32 dlsr;             /* delay since last SR packet */
} rtcp_rr_t;

/*
```

```
 * SDES item
 */
typedef struct {
    u_int8 type;              /* type of item (rtcp_sdes_type_t) */
    u_int8 length;            /* length of item (in octets) */
    char data[1];             /* text, not null-terminated */
} rtcp_sdes_item_t;

/*
 * One RTCP packet
 */
typedef struct {
    rtcp_common_t common;     /* common header */
    union {
        /* sender report (SR) */
        struct {
            u_int32 ssrc;     /* sender generating this report */
            u_int32 ntp_sec;  /* NTP timestamp */
            u_int32 ntp_frac;
            u_int32 rtp_ts;   /* RTP timestamp */
            u_int32 psent;    /* packets sent */
            u_int32 osent;    /* octets sent */
            rtcp_rr_t rr[1];  /* variable-length list */
        } sr;

        /* reception report (RR) */
        struct {
            u_int32 ssrc;     /* receiver generating this report */
            rtcp_rr_t rr[1];  /* variable-length list */
        } rr;


        /* source description (SDES) */
        struct rtcp_sdes {
            u_int32 src;      /* first SSRC/CSRC */
            rtcp_sdes_item_t item[1]; /* list of SDES items */
        } sdes;

        /* BYE */
        struct {
            u_int32 src[1];   /* list of sources */
            /* can't express trailing text for reason */
        } bye;
    } r;
```

```
} rtcp_t;

typedef struct rtcp_sdes rtcp_sdes_t;
```

g711.c

```
/*
 ************************************************************************
 ************************************************************************
 * G.711 u-law, A-law and linear PCM conversions.                       *
 *                                                                      *
 * Code to implement the G711 A-law and u-law codes has been            *
 * released into the public domain by Sun Microsystems Inc, and         *
 * modified by Borge Lindberg.                                          *
 *                                                                      *
 ************************************************************************
 ************************************************************************
 * Versione 1.0 (Maggio 2001)                                           *
 * Autore del codice esterno al blocco 'BEGIN END': Daniele Mari.       *
 * e-mail: daniele.mari@tiscalinet.it                                   *
 ************************************************************************
 ************************************************************************
                                                                       */

 #include <stdio.h>
 #include <stdarg.h>
 #include <stdlib.h>
 #include <strings.h>          // bcopy()


 #ifdef CODER

 #define BLOCK_SIZE 320        // 20ms per pacchetto, 16 bits/campione
 #define OUT_SIZE   160        // 20ms per pacchetto,  8 bits/campione
 int k;

 #endif


 #ifdef DECODER

 #define BLOCK_SIZE 160        // 20ms per pacchetto,  8 bits/campione
```

```
#define OUT_SIZE   320       // 20ms per pacchetto, 16 bits/campione

int k;

#endif

union {
 unsigned char in_samples_char[BLOCK_SIZE];
 short in_samples_short[BLOCK_SIZE];
}inbuf;

union {
 unsigned char out_samples_char[BLOCK_SIZE];
 short out_samples_short[BLOCK_SIZE];
}outbuf;



/*********************************BEGIN*******************************/
/*
 * This source code (BEGIN - END) is a product of Sun Microsystems,
 * Inc. and is provided for unrestricted use.  Users may copy or
 * modify this source code without   charge.
 * SUN SOURCE CODE IS PROVIDED AS IS WITH NO WARRANTIES OF ANY KIND
 * INCLUDING THE WARRANTIES OF DESIGN, MERCHANTIBILITY AND FITNESS
 * FOR A PARTICULAR PURPOSE, OR ARISING FROM A COURSE OF DEALING,
 * USAGE OR TRADE PRACTICE.
 *
 * Sun source code is provided with no support and without any
 * obligation on
 *
 * the part of Sun Microsystems, Inc. to assist in its use,
 * correction,
 * modification or enhancement.
 *
 * SUN MICROSYSTEMS, INC. SHALL HAVE NO LIABILITY WITH RESPECT TO THE
 * INFRINGEMENT OF COPYRIGHTS, TRADE SECRETS OR ANY PATENTS BY THIS
 * SOFTWARE OR ANY PART THEREOF.
 *
 * In no event will Sun Microsystems, Inc. be liable for any lost
 * revenue or profits or other special, indirect and consequential
 * damages, even if Sun has been advised of the possibility
 * of such damages.
 *
```

```
 * Sun Microsystems, Inc.
 * 2550 Garcia Avenue
 * Mountain View, California  94043
 */

/*
 * g711.c
 *
 * u-law, A-law and linear PCM conversions.
 */

 /*
  * Function linear2alaw, linear2ulaw have been updated to correctly
  * convert unquantized 16-bit values.
  *
  * Tables for direct u- to A-law and A- to u-law have been corrected.
  * Borge Lindberg, Center for PersonKommunikation, Aolborg
  * University.
  * bli@cpk.auc.dk
  */

#define SIGN_BIT        (0x80)      /* Sign bit for a A-law byte. */
#define QUANT_MASK      (0xf)       /* Quantization field mask. */
#define NSEGS           (8)         /* Number of A-law segments. */
#define SEG_SHIFT       (4)         /*Left shift for segment number*/
#define SEG_MASK        (0x70)      /* Segment field mask. */

static short seg_aend[8] = {0x1F, 0x3F, 0x7F, 0xFF,
                                    0x1FF, 0x3FF, 0x7FF, 0xFFF};
static short seg_uend[8] = {0x3F, 0x7F, 0xFF, 0x1FF,
                                    0x3FF, 0x7FF, 0xFFF, 0x1FFF};


/* copy from CCITT G.711 specifications */
unsigned char _u2a[128] = {    /* u- to A-law conversions */
       1,    1,    2,    2,    3,    3,    4,    4,
       5,    5,    6,    6,    7,    7,    8,    8,
       9,    10,   11,   12,   13,   14,   15,   16,
       17,   18,   19,   20,   21,   22,   23,   24,
       25,   27,   29,   31,   33,   34,   35,   36,
       37,   38,   39,   40,   41,   42,   43,   44,
       46,   48,   49,   50,   51,   52,   53,   54,
       55,   56,   57,   58,   59,   60,   61,   62,
       64,   65,   66,   67,   68,   69,   70,   71,
```

```
       72,   73,   74,   75,   76,   77,   78,   79,
/* corrected:
       81,   82,   83,   84,   85,   86,   87,   88,
   should be: */
       80,   82,   83,   84,   85,   86,   87,   88,
       89,   90,   91,   92,   93,   94,   95,   96,
       97,   98,   99,   100,  101,  102,  103,  104,
       105,  106,  107,  108,  109,  110,  111,  112,
       113,  114,  115,  116,  117,  118,  119,  120,
       121,  122,  123,  124,  125,  126,  127,  128};

unsigned char _a2u[128] = {    /* A- to u-law conversions */
       1,    3,    5,    7,    9,    11,   13,   15,
       16,   17,   18,   19,   20,   21,   22,   23,
       24,   25,   26,   27,   28,   29,   30,   31,
       32,   32,   33,   33,   34,   34,   35,   35,
       36,   37,   38,   39,   40,   41,   42,   43,
       44,   45,   46,   47,   48,   48,   49,   49,
       50,   51,   52,   53,   54,   55,   56,   57,
       58,   59,   60,   61,   62,   63,   64,   64,
       65,   66,   67,   68,   69,   70,   71,   72,
/* corrected:
       73,   74,   75,   76,   77,   78,   79,   79,
   shold be: */
       73,   74,   75,   76,   77,   78,   79,   80,
       80,   81,   82,   83,   84,   85,   86,   87,
       88,   89,   90,   91,   92,   93,   94,   95,
       96,   97,   98,   99,   100,  101,  102,  103,
       104,  105,  106,  107,  108,  109,  110,  111,
       112,  113,  114,  115,  116,  117,  118,  119,
       120,  121,  122,  123,  124,  125,  126,  127};

static short search(short val,short *table,short size)
{
 short i;

 for (i = 0; i < size; i++) {
  if (val <= *table++)
   return (i);
 }

return (size);
} /* search */
```

```
/*
 * linear2alaw() - Convert a 16-bit linear PCM value to 8-bit A-law
 *
 * linear2alaw() accepts an 16-bit integer and encodes it as A-law
data.
 *
 *                        Linear Input Code       Compressed Code
 *            ------------------------            ---------------
 *            0000000wxyza                              000wxyz
 *            0000001wxyza                              001wxyz
 *            000001wxyzab                              010wxyz
 *            00001wxyzabc                              011wxyz
 *            0001wxyzabcd                              100wxyz
 *            001wxyzabcde                              101wxyz
 *            01wxyzabcdef                              110wxyz
 *            1wxyzabcdefg                              111wxyz
 *
 * For further information see John C. Bellamy's Digital
 * Telephony, 1982,
 * John Wiley & Sons, pps 98-111 and 472-476.
 */
unsigned char linear2alaw(short pcm_val)
/* 2's complement (16-bit range) */
{
 short        mask;
 short        seg;
 unsigned char aval;

 pcm_val = pcm_val >> 3;
 if (pcm_val >= 0) {
 mask = 0xD5;      /* sign (7th) bit = 1 */
 }
 else {
  mask = 0x55;     /* sign bit = 0 */
  pcm_val = -pcm_val - 1;
 }

 /* Convert the scaled magnitude to segment number. */
 seg = search(pcm_val, seg_aend, 8);

 /* Combine the sign, segment, and quantization bits. */

 if (seg >= 8)     /* out of range, return maximum value. */
  return (unsigned char)(0x7F ^ mask);
```

```
 else {
  aval =(unsigned char) seg << SEG_SHIFT;
  if (seg < 2)
   aval |= (pcm_val >> 1) & QUANT_MASK;
  else
   aval |= (pcm_val >> seg ) & QUANT_MASK;
 return (aval ^ mask);
 }
} /* linear2alaw */

/*
 * alaw2linear() - Convert an A-law value to 16-bit linear PCM
 *
 */
short alaw2linear(unsigned char a_val)
{
 short t;
 short seg;

 a_val ^= 0x55;

 t = (a_val & QUANT_MASK) << 4;
 seg = ((unsigned)a_val & SEG_MASK) >> SEG_SHIFT;
 switch (seg) {
 case 0:
  t += 8;
 break;
 case 1:
  t += 0x108;
 break;
 default:
  t += 0x108;

  t <<= seg - 1;
 }

 return ((a_val & SIGN_BIT) ? t : -t);
} /* alaw2linear */

#define BIAS (0x84)      /* Bias for linear code. */
#define CLIP  8159

/*
 * linear2ulaw() - Convert a linear PCM value to u-law
```

```
 *
 * In order to simplify the encoding process, the original linear
 * magnitude is biased by adding 33 which shifts the
 * encoding range from (0 - 8158) to(33 - 8191).
 * The result can be seen in the following encoding table:
 *
 *           Biased Linear Input Code           Compressed Code
 *           ------------------------           ---------------
 *           00000001wxyza                           000wxyz
 *           0000001wxyzab                           001wxyz
 *           000001wxyzabc                           010wxyz
 *           00001wxyzabcd                           011wxyz
 *           0001wxyzabcde                           100wxyz
 *           001wxyzabcdef                           101wxyz
 *           01wxyzabcdefg                           110wxyz
 *           1wxyzabcdefgh                           111wxyz
 *
 * Each biased linear code has a leading 1 which identifies the
 * segment number.
 * The value of the segment number is equal to 7 minus the number
 * of leading 0's. The quantization interval is directly available as
 * the four bits wxyz.
 * The trailing bits (a - h) are ignored.
 * Ordinarily the complement of the resulting code word is used for
 * transmission, and so the code word is complemented before it is
 * returned.
 * For further information see John C. Bellamy's Digital Telephony,
 * 1982,
 * John Wiley & Sons, pps 98-111 and 472-476.
 */
unsigned char linear2ulaw(short pcm_val)
/* 2's complement (16-bit range) */
{
 short       mask;
 short       seg;
 unsigned char     uval;

 /* Get the sign and the magnitude of the value. */
 pcm_val = pcm_val >> 2;
 if (pcm_val < 0) {
  pcm_val = - pcm_val;
  mask = 0x7F;
 }
 else {
```

```
  mask = 0xFF;
 }

 if( pcm_val > CLIP ) pcm_val = CLIP; /* clip the magnitude */
 pcm_val += ( BIAS >>2 );

 /* Convert the scaled magnitude to segment number. */
 seg = search(pcm_val, seg_uend, 8);

 /*
  * Combine the sign, segment, quantization bits;
  * and complement the code word.
  */
 if (seg >= 8)     /* out of range, return maximum value. */
  return (unsigned char)(0x7F ^ mask);
 else {
  uval = (unsigned char)(seg << 4) | ((pcm_val >> (seg + 1)) & 0xF);
  return (uval ^ mask);
 }

} /* linear2ulaw */

/*
 * ulaw2linear() - Convert a u-law value to 16-bit linear PCM
 *
 * First, a biased linear code is derived from the code word.
 * An unbiased output can then be obtained by subtracting 33 from
 * the biased code.
 *
 * Note that this function expects to be passed the complement of the
 * original code word. This is in keeping with ISDN conventions.
 */
short ulaw2linear(unsigned char u_val)
{
 short t;

 /* Complement to obtain normal u-law value. */
 u_val = ~u_val;

 /*
  * Extract and bias the quantization bits. Then
  * shift up by the segment number and subtract out the bias.
  */
 t = ((u_val & QUANT_MASK) << 3) + BIAS;
```

```
 t <<= ((unsigned)u_val & SEG_MASK) >> SEG_SHIFT;

 return ((u_val & SIGN_BIT) ? (BIAS - t) : (t - BIAS));
} /* ulaw2linear */


/*******************************END*********************************/


/*
  * Principale funzione.
  *
  * CODER:   data_in = audio_buff,         data_out = packet+12
  * DECODER: data_in = &(current->data[12]),  data_out = audio_buf
  *
  */
 int g711( char type_of_conv[2], char *data_in, char *data_out, int
           padding)
 {
  short           in_size, out_size,i;
  short           (*char_short_routine) (unsigned char uval) = NULL;
  unsigned char   (*short_char_routine) (short pcm_val)      = NULL;

  out_size = sizeof(char);
  in_size  = sizeof(char);

  // Valuta il tipo di conversione da innescare
  switch(type_of_conv[0]) {
  case 'u':
    out_size = sizeof(short);
    char_short_routine = ulaw2linear;
    break;
  case 'a':
    out_size = sizeof(short);
    char_short_routine = alaw2linear;
    break;
  case 'l':
    in_size = sizeof(short);
    switch(type_of_conv[1]) {
    case 'u':
      short_char_routine = linear2ulaw;
      break;
    case 'a':
      short_char_routine = linear2alaw;
```

```
    break;
   }
   break;
 }

 /*
  * Se 'padding = 0', la variabile 'padding' non ha alcun effetto.
  */
 bcopy( data_in, &inbuf, BLOCK_SIZE - padding );

 #ifdef CODER

 if( padding ) {
  for(i=0; i<( BLOCK_SIZE - padding ); i++)
   outbuf.out_samples_char[i]=(*short_char_routine)
                              (inbuf.in_samples_short[i]);
  for(k=i; k<( i+padding-1 ); k++)
   outbuf.out_samples_char[k]=0;    // zero-padding
  // l'ultimo ottetto dice quanti byte sono di padding
  outbuf.out_samples_char[k]= padding;
 }
 else // 'padding = 0' --> no effetto.
  for(i=0; i<( BLOCK_SIZE - padding ); i++)
   outbuf.out_samples_char[i]=(*short_char_routine)
                              (inbuf.in_samples_short[i]);


 #endif

 #ifdef DECODER

 for(i=0; i<( BLOCK_SIZE - padding ); i++)
  outbuf.out_samples_short[i]=(*char_short_routine)
                              (inbuf.in_samples_char[i]);

 #endif

 bcopy( &outbuf, data_out, OUT_SIZE );

 return 0;

} /* main function */
```

A.2 RTP-Speak (Manuale d'uso).

TITOLO

RTP-Speak

Sistema di comunicazione vocale tempo reale tra due host remoti

Versione 1.0 (Maggio 2001)
by Mari Daniele

DESCRIZIONE

Il pacchetto software è costituito da una serie di file in linguaggio C in grado di mettere in piedi una comunicazione tempo reale tra due host remoti.

Il software è rivolto a tutte le piattaforme POSIX-compatibili supportanti le sockets.

Le principali caratteristiche sono le seguenti:

- Interfaccia utente: *riga di comando*
- Acquisizione: *da microfono*
- Codifica: *PCMA, PCMμ*
- Modalità di comunicazione: *simplex real-time*
- Connessione: *unicast & broadcast (a scelta)*

CONTENUTO

Il pacchetto è costituito dai seguenti file:

- rtpmail.c
  Gestisce la trasmissione audio

- openline.c
  Gestisce la ricezione audio

- set_snd.c
  Realizza il setup della scheda audio in modo da poter registrare ed ascoltare file audio

- g711.c
  Realizza la codifica/decodifica PCMA, PCMU a 64kb/s

- rtp.h
  Definizioni per le intestazioni RTP ed RTCP

- Makefile
  Script per la compilazione ed il linkaggio

- readme.txt
  Manuale d'uso

Le routine principali sono: rtpmail.c ed openline.c
La 'rtpmail.c' è il CLIENT che gestisce la parte d'acquisizione dati; realizza l'acquisizione della voce (direttamente da microfono), la codifica (a scelta tra quelle disponibili), la pacchettizzazione RTP e la trasmissione.
La 'openline.c' è il SERVER che si pone in attesa di connessioni. I dati audio ricevuti vengono depacchettizzati (si toglie l'intestazione RTP), decodificati ed ascoltati.

COMPILAZIONE

Le istruzioni di compilazione che seguono si riferiscono all'uso del compilatore *gcc*. Nel caso non si disponga di tale compilatore sarà necessario apportare qualche modifica al Makefile, "settando" il compilatore in uso.

Il comandi per compilare e linkare i vari file sorgente è il seguente:

```
make all
```

Il codice è stato compilato e provato con successo su varie piattaforme Linux e Unix supportanti la suite TCP/IP.

USO

Lanciando `make all` si generano due eseguibili, uno per la ricezione e l'altro per la trasmissione, rispettivamente: `vmailto` e `rcvmsg`.

Affinché la comunicazione abbia successo devono essere lanciati entrambi i processi su host differenti.

Il comando che lancia il processo ricevente è:

```
$ rcvmsg
```

Il comando che lancia il processo trasmittente deve essere seguito dall'indirizzo IP del destinatario. L'indirizzo IP può essere sia in formato dotted-decimal che in formato hostname:

```
$ vmailto 151.100.9.29
```

oppure,

```
$ vmailto pcvoce2.ing.uniroma1.it
```

Attraverso i precedenti comandi il processo è lanciato con le sue caratteristiche di defoult. Le opzioni che modificano il defoult del programma riguardano il nome del dispositivo audio, il formato di codifica ed un help. Il defoult di dispositivo audio è *dev/dsp* mentre quello di codifica è il *PCMA*. Un esempio di setup delle opzioni è il seguente:

```
$ vmailto –d dev/audio –c PCMU 151.100.9.29
```

Per trasmettere in *broadcast* è sufficiente digitare uno degli indirizzi chiave previsti per il broadcast (151.100.9.255, 151.100.255.255, 151.255.255.255, 255.255.255.255).

Per lanciare un breve messaggio d'aiuto è sufficiente scegliere l'opzione "–h":

```
$ vmailto –h
```

in questo modo comparirà il seguente messaggio:

```
$ Usage: vmailto [-d devicename] [-c codername] destination

     -d               devicename
     -c               type of encoder
                      Available coders are: PCMA, PCMU, G729
     destination IP   address or host-name to contact

     escape character is ^C
```

Sia il processo trasmittente che quello ricevente vanno interrotti attraverso un comando da tastiera: ^C.

Appendice B

RTP-Speak (in progress).

La versione di RTP-Speak riportata in questa appendice è analoga a quella riportata nell'app. A.1, ma in più supporta la codifica G.729.

La ragione per cui le due versioni sono separate è la seguente. Questa versione pur essendo funzionante non è implementata in modo ottimo per ciò che riguarda la modularità. In particolare, alcune delle modifiche, che è stato necessario apportare ai codici sorgente del G.729 forniti dall'ITU-T per poter realizzare l'implementazione, vanno ottimizzate. Il criterio di ottimalità al quale si sta facendo riferimento è quello di rendere l'inserimento del G.729 modulare e versatile. In pratica è consigliabile che tutte le routine riguardanti il G.729 siano implementate in file esterni a "*rtpmail.c* " e ad "*openline.c* " (come si è fatto per il G.711, vedi app. A.1). La versione di RTP-Speak presentata in questa appendice raggiunge lo scopo per tutte i file forniti dall'ITU-T tranne che per due file (come vedremo nel seguito), per questa ragione si parla di versione "**in progress**". Una spiegazione più dettagliata in merito è fornita nella sezione B.1. Nella sezione B.2 saranno presentati i soli tre file di RTP-Speak che si differenziano dalla versione precedente. Anche in questo caso, il codice è organizzato su più file separati. Ogni programma riportato è preceduto dal nome del file (*.c, *.h, Makefile) che lo contiene. Tutti i codici sorgente sono stati depositati presso la segreteria didattica del dipartimento di INFOCOM (INFOrmazione e COMunicazione).

B.1 Modifiche ai codici sorgente della raccomandazione G.729

Per esigenza di sintesi, si eviterà di dare una descrizione dettagliata del codice sorgente del G.729 e si farà riferimento alle sole parti modificate. In ogni caso, si farà una breve premessa su come sono organizzati i codici. Per ulteriori approfondimenti si rimanda alla documentazione fornita dall'ITU-T nella raccomandazione.

In questa sede ci riferiremo al solo codice relativo all'Annex A della raccomandazione, poiché le considerazioni che si faranno valgono anche per le altre versioni.

Breve descrizione del pacchetto software del G.729A

Il pacchetto software è composto da 34 file in ANSI C, più due file Make uno per le routine di codifica e l'altro per quelle di decodifica:

1. basic_op.c
2. basic_op.h
3. oper_32b.c
4. oper_32b.h
5. pre_proc.c
6. post_pro.c
7. bits.c
8. util.c
9. filter.c
10. dspfunc.c
11. qua_lsp.c
12. lspdec.c
13. lspgetq.c
14. qua_gain.c
15. dec_gain.c
16. gainpred.c
17. de_acelp.c
18. pred_lt3.c
19. p_parity.c
20. dec_lag3.c
21. lpc.c
22. lpcfunc.c
23. coder.c
24. decoder.c
25. ld8a.h
26. cod_ld8a.c
27. acelp_ca
28. tab_ld8a.h
29. tab_ld8a.c
30. postfilt.c
31. pitch_a.c
32. taming.c
33. cor_func.c
34. typedef.h
35. coder.mak
36. decoder.mak

Per la compilazione dei file, i passi sono i seguenti:

1. E' necessario impostare i due file Make, ovvero scegliere le opzioni adatte al particolare SO in uso.
2. I file Make vanno lanciati tramite i comandi[29]:

```
$ make –f coder.mak
$ make –f decoder.mak
```

In questo modo si generano due eseguibili: `coder` e `decoder`.
Attraverso i due eseguibili è possibile lanciare le prove di codifica:

```
$ coder inputfile bitstreamfile
$ decoder bitstreamfile outputfile
```

[29] L'opzione "-f" serve ogni volta che si vuole dare un nome non standard al file Make [37].

L'*inputfile* deve essere un file vocale campionato ad 8000 Hz con 16-bit per campione (PCM lineare), l'*outputfile* generato dal decodificatore avrà lo stesso formato. Il *bitstreamfile* è generato dal codificatore.

Il bitsreamfile contiene, per ogni trama vocale da 10 ms, 82 word da 16-bit ciascuna, con il seguente significato:

- La prima word è di sincronizzazione.
- La seconda word contiene la lunghezza, in word, della trama sintetica, quindi 80.
- Le successive 80 word contengono i parametri di codifica (vedi tab. III.2).

Modifiche necessarie

La precedente descrizione del pacchetto software G.729, è sufficiente a mettre in evidenza alcune necessarie modifiche da apportare affinché l'implementazione sia adatta ad un'applicazione VoIP. Vediamole:

1. Non è proponibile che l'input e l'output siano dei file, indipendentemente dal fatto che il loro ingresso sia da riga di comando. Infatti, è ovvio che una conversazione vocale deve essere dinamica e non preregistrata. Un file non è adatto alla bufferizzazione dinamica.

2. Il formato di immagazzinamento del file sintetico (il bitsreamfile) è tale da rendere il file sintetico più grande di quello originale. Ad esempio, un'inputfile di 58,240 byte diventa un bitstreamfile di 59,696 byte; infatti, 58,240 byte corrispondono a 364 trame da 10 ms, quindi, considerando 82 word da 2 byte per ogni trama, risulta un file sintetico di 59,696 byte. Ciò è impensabile per un'applicazione VoIP, poiché è inutile trasmettere un file più grande dell'originale che, inoltre, è di qualità peggiore. Il problema sta nella rappresentazione del file sintetico, la quale non è adatta ad essere trasmessa[30]. Questa modalità rappresentativa dei parametri di codifica è "sovrabbondante" e va cambiata. La sovrabbondanza nella rappresentazione non implica un problema di decodifica, infatti le routine funzionano bene.

3. Ovviamente, le precedenti modifiche, una volta fatte, impongono un ripensamento dell'intero processo di compilazione dei file a partire dai file Make.

Un ultimo aspetto che necessità di modifiche è legato al discorso della proprietà di modularità accennata nell'introduzione. Questo aspetto non è immediatamente evidente come i precedenti, e per tale ragione è anche il più difficile da gestire. Diciamo subito che le considerazioni che seguono saranno chiare solo dopo uno studio attento del codice sorgente.

[30] Va da se che ciò è indipendente dal fatto che l'input e l'output siano dei file.

Il modo con cui sono correlati i vari file sorgente, e quindi le rispettive variabili, fa si che, affinché la codifica (o la decodifica) abbia successo, la routine "*coder.c*" (o la routine "*decoder.c*"), una volta avviata, non va mai interrotta, se non al termine dell'intero processo di codifica (o di decodifica). In particolare, l'interruzione comporta la perdita d'informazione legata alle routine di inizializzazione (in particolare le seguenti: *Init_Pre_Process(), Init_Coder_ld8a(), Set_zero(), Init_Decod_ld8(), Init_Post_Filter(), Init_Post_Process()*) ed alla variabile globale "*Word16 *new_speech*". Quest'ultima è quella che conserva l'avanzamento della finestra di analisi (vedi par. III.2). Ogni volta che la routine "*coder.c*" viene interrotta la variabile perde la memoria sull'avanzamento e la codifica non ha successo. Ad esempio, se si ha un file vocale e su di esso si fa la codifica e poi la decodifica, va tutto bene; ma se lo stesso file viene diviso in più parti da 20 ms ciascuna, su ogni parte si fa la codifica e la decodifica, e poi queste parti vengono messe insieme il risultato è un file ricostruito molto rumoroso. Diversamente, eseguendo le stesse operazioni senza riavviare più volte la routine "*coder.c*", va tutto bene. Quindi, in queste condizioni, nell'ambito di RTP-Speak, non è possibile pensare di innescare la "*coder.c*" e la "*decoder.c*" per ogni trama da elaborare, come si è fatto per il G.711.

Realizzazione delle modifiche

Le modifiche di cui si è parlato riguardano i seguenti file sorgente: *coder.c, decoder.c, bits.c, typedef.h*. Le modifiche sono state affrontate nel seguente modo:

- Attraverso banali cambiamenti ai file *coder.c* e *decoder.c* è stato agevole far sì che l'I/O fosse costituito da array piuttosto che da file. Ovviamente, gli array in questione sono rappresentati dal buffer-audio ed dal payload RTP, e non provengono da riga di comando.

- Per quanto riguarda il formato d'immagazinamento i cambiamenti sono stati più sostanziali e laboriosi. Il file *bits.c*, responsabile della modalità rappresentativa, è stato completamente eliminato. Nei nuovi file *coder.c* e *decoder.c* sono state inserite due funzioni, le quali, mediante operazioni bit-a-bit hanno il pregio di rappresentare 10 ms di voce con 10 byte (che corrisponde al giusto tasso di compressione del codificatore G.729), anziché 82 byte. Ovviamente, eliminando la sovrabbondanza, è cambiata la sola modalità rappresentativa, ma la qualità della voce ricostruita è rimasta inalterata. Attraverso queste modifiche, ripercorrendo l'esempio discusso nel precedente punto 2, si ha che il file sintetico, che nasce dall'originale, la cui dimensione è 58,240 byte, è di 3,640 byte, anziché 59,696 byte.

- Come annunciato in precedenza le modifiche relative all'inserimento modulare del G.729 in RTP-Speak sono efficaci ma non ottime. La scelta iniziale, di prova, è la seguente. Le nuove routine *coder.c* e *decoder.c* sono state inserite direttamente in RTP-Speak, rispettivamente in *rtpmail.c* e *openline.c*. In questo

modo, si è raggirato il problema della perdita delle informazioni relativa alle inizializzazioni. Una soluzione migliore è quella di riorganizzare le variabili critiche, come *new_speech*, ed implementare in un unico file separato le funzionalità di *coder.c* e *decoder.c*, seguendo la logica implementativa del G.711.

Prima di concludere questo paragrafo è opportuno fare alcune precisazioni. Le modifiche apportate al G.729A sono state organizzate in due pacchetti software distinti.

Un primo pacchetto è relativo alla sola modifica della rappresentazione dei parametri di codifica. Questo pacchetto mantiene l'I/O da file ed è indipendente da RTP-Speak. Il pregio è quello di poter ottenere un file sintetico effettivamente compresso. Per brevità, si eviterà di riportare il codice, il quale è comunque, disponibile presso la segreteria didattica del dipartimento di INFOCOM. Il software relativo è tutto in ANSI C.

Un secondo pacchetto è relativo ad RTP-Speak. Per brevità si riportano le sole routine principali, l'intero pacchetto è disponibile presso il dipartimento di INFOCOM.

B.2 RTP-Speak (in progress). Listati delle routine principali.

Il codice è organizzato su più file separati. Ogni programma riportato è preceduto dal nome del file (*.c, *.h, Makefile) che lo contiene.

Makefile

```
##############################################################################
##############################################################################
#                                                                            #
# File Make per RTP-Speak (in progress).                                     #
# Versione 1.1 (Maggio 2001)                                                 #
#                                                                            #
##############################################################################
##############################################################################
#                                                                            #
# Autore: Daniele Mari.                                                      #
# e-mail: daniele.mari@tiscalinet.it                                         #
#                                                                            #
##############################################################################
##############################################################################

CC = gcc

# -O2
CFLAGS = -g -Wall -O2

LFLAGS =  -lm

# sorgenti per la parte di trasmissione
TSRCS = \
 set_snd.c\
 acelp_ca.c\
 basic_op.c\
 cod_ld8a.c\
 dspfunc.c\
 filter.c\
 gainpred.c\
 lpc.c\
 lpcfunc.c\
 lspgetq.c\
```

```
 oper_32b.c\
 p_parity.c\
 pitch_a.c\
 pre_proc.c\
 pred_lt3.c\
 qua_gain.c\
 qua_lsp.c\
 tab_ld8a.c\
 util.c\
 taming.c\
 cor_func.c

# sorgenti per la parte di ricezione
RSRCS = \
 set_snd.c\
 basic_op.c\
 de_acelp.c\
 dec_gain.c\
 dec_lag3.c\
 dec_ld8a.c\
 dspfunc.c\
 filter.c\
 gainpred.c\
 lpcfunc.c\
 lspdec.c\
 lspgetq.c\
 oper_32b.c\
 p_parity.c\
 post_pro.c\
 pred_lt3.c\
 postfilt.c\
 tab_ld8a.c\
 util.c

#####################################################################Lin

COBJS = $(TSRCS:.c=.o) cg711.o

DOBJS = $(RSRCS:.c=.o) dg711.o

all: vmailto rcvmsg

# all objects files, for coder and decoder (G711)
cg711.o:     g711.c
```

```
 $(CC) -g -c -o cg711.o $(CFLAGS) -DCODER  g711.c

dg711.o:     g711.c
 $(CC) -c -o dg711.o $(CFLAGS) -DDECODER g711.c

# ..............................................


acelp_ca.o : acelp_ca.c typedef.h basic_op.h  ld8a.h
 $(CC) $(CFLAGS) -c  acelp_ca.c

basic_op.o : basic_op.c typedef.h basic_op.h
 $(CC) $(CFLAGS) -c  basic_op.c

cod_ld8a.o : cod_ld8a.c typedef.h basic_op.h  ld8a.h
 $(CC) $(CFLAGS) -c  cod_ld8a.c

de_acelp.o : de_acelp.c typedef.h basic_op.h  ld8a.h
 $(CC) $(CFLAGS) -c de_acelp.c

dec_gain.o : dec_gain.c typedef.h basic_op.h  ld8a.h tab_ld8a.h
 $(CC) $(CFLAGS) -c dec_gain.c

dec_lag3.o : dec_lag3.c typedef.h basic_op.h  ld8a.h
 $(CC) $(CFLAGS) -c dec_lag3.c

dec_ld8a.o : dec_ld8a.c typedef.h basic_op.h  ld8a.h
 $(CC) $(CFLAGS) -c dec_ld8a.c

dspfunc.o : dspfunc.c typedef.h basic_op.h  ld8a.h tab_ld8a.h
 $(CC) $(CFLAGS) -c  dspfunc.c

filter.o : filter.c typedef.h basic_op.h  ld8a.h
 $(CC) $(CFLAGS) -c  filter.c

gainpred.o : gainpred.c typedef.h basic_op.h ld8a.h  tab_ld8a.h\
oper_32b.h
 $(CC) $(CFLAGS) -c  gainpred.c

lpc.o : lpc.c typedef.h basic_op.h oper_32b.h ld8a.h  tab_ld8a.h
 $(CC) $(CFLAGS) -c  lpc.c

lpcfunc.o : lpcfunc.c typedef.h basic_op.h oper_32b.h ld8a.h\
            tab_ld8a.h
```

```
 $(CC) $(CFLAGS) -c  lpcfunc.c

lspgetq.o : lspgetq.c typedef.h basic_op.h ld8a.h
 $(CC) $(CFLAGS) -c  lspgetq.c

oper_32b.o : oper_32b.c typedef.h basic_op.h  oper_32b.h
 $(CC) $(CFLAGS) -c  oper_32b.c

p_parity.o : p_parity.c typedef.h basic_op.h  ld8a.h
 $(CC) $(CFLAGS) -c  p_parity.c

post_pro.o : post_pro.c typedef.h basic_op.h  ld8a.h tab_ld8a.h\
 oper_32b.h
 $(CC) $(CFLAGS) -c post_pro.c

pitch_a.o : pitch_a.c typedef.h basic_op.h ld8a.h   tab_ld8a.h\
oper_32b.h
 $(CC) $(CFLAGS) -c  pitch_a.c

pre_proc.o : pre_proc.c typedef.h basic_op.h oper_32b.h  ld8a.h\
              tab_ld8a.h
 $(CC) $(CFLAGS) -c  pre_proc.c

pred_lt3.o : pred_lt3.c typedef.h basic_op.h  ld8a.h tab_ld8a.h
 $(CC) $(CFLAGS) -c  pred_lt3.c

postfilt.o : postfilt.c typedef.h ld8a.h basic_op.h oper_32b.h
 $(CC) $(CFLAGS) -c postfilt.c

qua_gain.o : qua_gain.c typedef.h basic_op.h oper_32b.h  ld8a.h\
              tab_ld8a.h
 $(CC) $(CFLAGS) -c  qua_gain.c

qua_lsp.o : qua_lsp.c typedef.h basic_op.h  ld8a.h tab_ld8a.h
 $(CC) $(CFLAGS) -c  qua_lsp.c

tab_ld8a.o : tab_ld8a.c typedef.h ld8a.h tab_ld8a.h
 $(CC) $(CFLAGS) -c  tab_ld8a.c

util.o : util.c typedef.h ld8a.h  basic_op.h
 $(CC) $(CFLAGS) -c  util.c

taming.o : taming.c typedef.h ld8a.h  basic_op.h
 $(CC) $(CFLAGS) -c  taming.c
```

```
cor_func.o : cor_func.c typedef.h ld8a.h  basic_op.h
 $(CC) $(CFLAGS) -c  cor_func.c

# principali routine di RTP-Speak
rtpmail.o:   rtpmail.c typedef.h basic_op.h  ld8a.h
 $(CC) -g -c $(CFLAGS) rtpmail.c

openline.o:  openline.c typedef.h basic_op.h  ld8a.h
 $(CC) -c $(CFLAGS) openline.c

vmailto:     $(COBJS)  rtpmail.o
 $(CC) -g -o vmailto $(COBJS) rtpmail.o $(CFLAGS) $(LFLAGS)

rcvmsg: $(DOBJS) openline.o
 $(CC) -o rcvmsg $(DOBJS) openline.o $(CFLAGS) $(LFLAGS)

p-rcvmsg: $(DOBJS) openline.o
 purify $(CC) -o p-rcvmsg $(DOBJS) openline.o $(CFLAGS) $(LFLAGS)

SDSRC = $(RSRCS) g711.c openline.c

s-rcvmsg: $(SDSRC)
 #load $(CFLAGS) $(SDSRC)

############################################################  Cleanup

lin-clean:
 -rm -f *.o

clean: lin-clean
```

rtpmail.c

```
/*
 ************************************************************************
 ************************************************************************
 * RTP-Speak.                                                           *
 * Versione 1.0 (Maggio 2001)(in progress)                              *
 * Principale routine di trasmissione.                                  *
 * Gestisce:                                                            *
```

```
 * 1. L'acquisizione da microfono                                        *
 * 2. La codifica                                                        *
 * 3. La trasmissione                                                    *
 ************************************************************************
 ************************************************************************
 * Autore: Daniele Mari                                                  *
 * e-mail: Daniele.mari@tiscalinet.it                                    *
 ************************************************************************
 ************************************************************************
                                                                        */


#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <strings.h>                // bzero()
#include <sys/soundcard.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <signal.h>                 // signal()
#include <arpa/inet.h>              // inet_pton()
#include <sys/socket.h>             // struct sockadrr
#include <netinet/in.h>             // sockaddr_in
#include <netdb.h>                  // gethostbyname()
#include <stdlib.h>                 // perror()
#include "rtp.h"                    // RTP headers

                                /*
#define       MAXLEN 350        * massimo num di byte associabili ad
 * un pacchetto RTP (è  consigliato non
 * superare questa soglia)
                                                                      */
static int loop = 0;

/*
 * Dichiarazioni globali per l'acqusizione audio
 *
 */

static char au_name[15] = "/dev/dsp";
int dsp_stereo = 0;      // mono
int dsp_speed = 8000;    // PCMA (defoult), PCMU, G729
extern int au_fd;
```

```
char *audio_buff;
int dsp_speed, aubuf_size;

typedef struct {
 char *encoder;
 int  rate;
 int  channel;
} enc_map;
static enc_map pt_map[128];

static int sockfd;    // output socket

/*
 * La funzione che segue stampa un breve help.
 */
static void usage (char *processname)
{

 fprintf(stderr,"\n");
 fprintf(stderr,"-------------------------------------------------\n");
 fprintf(stderr,"******************RTP-Speak******************\n");
 fprintf(stderr,"**************Version 1.0, May 2001*************\n");
 fprintf(stderr,"\n");
 fprintf(stderr,">>>>>>>>>>>>>Designed by: Daniele Mari<<<<<<<<<<<\n");
 fprintf(stderr,"-------------------------------------------------\n");
 fprintf(stderr,"\n");
 fprintf ( stderr,
         "Usage: %s [-d devicename][-c codername] destination\n",
          processname);
 fprintf(stderr,"\t-d\t\tdevice name\n");
 fprintf(stderr,"\t-c\t\ttype of encoding name\n");
 fprintf(stderr,
         "\t\t\tAvailables coders are: PCMU, PCMA, G729\n");
fprintf(stderr,"\tdestination\tIP address or host_name to
        contact\n");

 exit(1);

} /* usage */


/*
 * Questa funzione raccoglie in 'payload' i soli 10 byte informativi
 * di ogni frame.
```

```
 *
 * Tali byte vengono presi dai 22 byte di 'prm', che, evidentemente,
 * contiene 12 byte ridondanti.
 *
 * L'informazione contenuta in ogni elemento di 'prm' è allocata in:
 *
 * parameter       position in prm[i]
 * L0+L1           1+7 lsb of prm[0]
 * L2+L3           5+5 lsb of prm[1]
 * P1              8 lsb of prm[2]
 * P0              1 lsb of prm[3]
 * C1              13 lsb of prm[4]
 * S1              4 lsb of prm[5]
 * F1+G1           3+4 lsb of prm[6]
 * P2              5 lsb of prm[7]
 * C2              13 lsb of prm[8]
 * S2              4 lsb of prm[9]
 * F2+G2           3+4 lsb of prm[10]
 *
*/
void info_synt( unsigned char *payload, Word16 prm[PRM_SIZE])
{

 payload[0] = prm[0]&255;
 payload[1] = prm[1]&255;
 payload[2] = ((prm[1]&768)>>2)+(prm[2]&63);
 payload[3] = (prm[2]&192)+((prm[3]&1)<<5)+(prm[4]&31);
 payload[4] = ((prm[4]&8160)>>5);
 payload[5] = ((prm[5]&15)<<4)+(prm[6]&15);
 payload[6]=  ((prm[6]&112)<<1)+(prm[7]&31);
 payload[7] = prm[8]&255;
 payload[8] = ((prm[8]&7936)>>5)+(prm[9]&7);
 payload[9] = ((prm[9]&8)<<4)+(prm[10]&127);

}


/*
 * La funzione che segue verifica la presenza del tipo di codificatore
 * nella ‘pt-map’.
 * In caso di successo restituisce il PT corrispondente, altrimenti –1.
 */
 static int search( enc_map *ptr, char *name)
 {
```

```
 register int i;

 for(i=0;i<128;i++)
  if( !strcmp(ptr[i].encoder, name) ) return i;
 return -1;

} /* search */


/*
 * Qesta funzione inizializza la struttura (IPv4 sockaddr) del
 * destinatario con l'indirizzo IP (fornito da riga di comando)ed il
 * numero di porta (di defoult, 5004) di quest'ultimo.
 * L'indirizzo IP può essere sia in formato dotted-decimal
 * (151.100.9.29)che in formato hostname (pcvoce2.ing.uniroma1.it).
 */
static void set_addr( char *h, struct sockaddr *sa)
{

 struct hostent *hptr;
 struct sockaddr_in *servaddr=(struct sockaddr_in*)sa;

 // tipica inizializzazione a 0 della struttura 'sockaddr'
 bzero(servaddr,sizeof(*servaddr));

 servaddr->sin_family = AF_INET;
 servaddr->sin_port = htons(5004);

 // Se l'IPv4 è in formato hostname, lo si riconverte via DSN.
 if( (hptr = gethostbyname(h)) ) {
   servaddr->sin_addr = *(struct in_addr*)(hptr->h_addr_list[0]);
 }
 // Verifica se l'IPv4 è in formato dotted-decimal
 else if( (inet_pton(AF_INET,h,&servaddr->sin_addr)) <=0 ) {
   fprintf(stderr,
    "inet_pton error for %s. Maybe destination have no valid
     format\n", h);

   exit(1);
 }

} /* set_addr */
```

```
/*
 * Funzione che genera l'intestazione RTP.
 *
 * La variabile 'packet' è di output e conterrà 12 byte di
 * intestazione RTP, ai quali, seguirà il payload fornito dal
 * codificatore.
 *
 * La variabile 'map_position' è di input e serve ad indicare il
 * valore del payload-type.
 *
 * (N.B. IL PUNTATORE 'h' E' TALE DA CONSERVARE IL VALORE ASSEGNATOGLI
 * TRA UNA CHIAMATA E LA SUCCESSIVA ---> poiché, puntando
 * all'indirizzo di 'packet', il valore precedentemente assegnatogli
 * viene mantenuto).
 *
 * La variabile 'loop' è di input e serve a sapere se si stà trattando
 * il primo pacchetto o i successivi.
 *
 * La variabile 'numBp' è di input e serve ad indicare la lunghezza,
 * in byte, del payload in base al tipo di codifica.
 *
 * La variabile 'padding' è di input e serve ad indicare quanti byte
 * di padding sono presenti nel pacchetto.
 */
void rtp(char *packet, int map_position, int loop, int numBp, int
         padding)
{

 rtp_hdr_t *h = (rtp_hdr_t *)packet;

 /*
  * campi fissi.
  */

 // versione RTP
 h->version = RTP_VERSION;

 // padding-bit
 if( !padding )
  h->p = 0;
 else
  h->p = 1;

 // header extension
```

```
 h->x=0;

 // numero di CSRC
 h->cc=0;

 // marker-bit
 h->m=0;

 // payload type
 h->pt = map_position;

 /*
  * campi variabili
  */

 if( loop == 1 ) {

  // sequence number
  h->seq=htons( rand() );

  // timestamp
  h->ts=htonl( rand() );

  // syncronization source (SSRC) identifier
  h->ssrc=htonl( rand() );

 }
 else { // incremento dei campi variabili

  h->seq = htons( ntohs(h->seq) + 1 );

  h->ts = htonl( ntohl(h->ts) + numBp );

 }

 // contributing source (CSRC) supposti assenti

} /* rtp */


/*
 * Funzione di fine.
 * Stampa un breve messaggio di notifica.
 * E' innescata, in modo asincrono, dall'utente mediante
```

```
 * il comando: ^C
 */
static void stop(int signo)
{

 fprintf(stderr,"Trasmission interrupted\n");
 fprintf(stderr,"%d packets sent\n",loop);

 exit(0);

} /* stop */


/*
 * Funzione principale
 */
int main (int argc, char *argv[])
{
 static struct sockaddr_in servaddr;
 extern char *optarg;
 extern int optind,opterr,optopt;
 int map_position;       // Posizione del codificatore nella ‘pt_map’.
 char packet[MAXLEN];
 int p,c,i;
 int padding = 0;        // # di padding-byte in un pacchetto
 int numBp = 160;        // # byte per pacchetto (defoult: PCMA,20ms)
 char cod_name[5]="PCMA"; // defoult: A-law (Europa)
 extern void set_snd(char *audio_name, int open_mod, int dsp_stereo);
 extern int g711(char type_of_conv[2], char *data_in, char *data_out,
                 int padding);


  unsigned char payload[10];
  extern Word16 *new_speech;    /* Pointer to new speech data */
  Word16 prm[PRM_SIZE];   /* Analysis parameters. */


 // Elabora i comandi forniti da prompt.
 while( (c = getopt(argc, argv, "d:c:h")) != -1 ) {
  switch(c) {
  case 'd':
    strncpy(au_name, optarg, 15);
    break;
  case 'c':
```

```
   strncpy(cod_name, optarg, 5);
   break;
 case '?':
 case 'h':        // h stà per help.
   usage(argv[0]);
   break;
 }
}

fprintf(stderr,"\n");
fprintf(stderr,"-----------------------------------------------\n");
fprintf(stderr,"********************RTP-Speak******************\n");
fprintf(stderr,"**************Version 1.0, May 2001*************\n");
fprintf(stderr,"\n");
fprintf(stderr,">>>>>>>>>>>>Designed by: Daniele Mari<<<<<<<<<<<\n");
fprintf(stderr,"-----------------------------------------------\n");
fprintf(stderr,"\n");


if(optind == argc) {
 fprintf(stderr,"Error: destination (IP address or hostname)
         missing\n");
 usage(argv[0]);
}
else {

 // Si inizializza l'indirizzo sockaddr a cui inviare i dati
 set_addr(argv[optind], (struct sockaddr *)&servaddr);

 // si crea una socket UDP-nonconnessa per l'RTP
  sockfd[i] = socket(AF_INET, SOCK_DGRAM, 0);
  if( sockfd < 0 ) {
   perror("socket");
   exit(1);
  }

} /* else */

// Messa in piedi dalla lista payload type
for (i=0; i<128; i++) {
 pt_map[i].encoder = "xxxx";
 pt_map[i].rate    = 0;
 pt_map[i].channel = 0;
}
```

```
 pt_map[0].encoder="PCMU";pt_map[0].rate=8000;pt_map[0].channel=1;
 pt_map[4].encoder="G723";pt_map[4].rate=8000;pt_map[4].channel=1;
 pt_map[8].encoder="PCMA";pt_map[8].rate=8000;pt_map[0].channel=1;
 pt_map[15].encoder="G728";pt_map[15].rate=8000;pt_map[0].channel=1;
 pt_map[18].encoder="G729";pt_map[18].rate=8000;pt_map[18].channel=1;
 pt_map[20].encoder="MELP";pt_map[20].rate=8000;pt_map[20].channel=1;

 i = search(pt_map, cod_name);
 if( i == -1 ) {
  fprintf(stderr,"Connection impossible: %s unavailable.\n",cod_name);
  fprintf(stderr,"Availables coders are: PCMU, PCMA, G731, G728, G729,
          MELP\n");
  exit(1);
 }
 else map_position = i;

/*
 ********************** PERCORSO DI TRASMISSIONE DI ***************
 ***************************OGNI SINGOLO PACCHETTO******************
                                                                    */

 // Processo d'acqisizione, da microfono, dei dati audio
 set_snd(au_name, O_RDONLY, dsp_stereo);

 Init_Pre_Process();
 Init_Coder_ld8a();
      Set_zero(prm, PRM_SIZE);

 signal(SIGINT, stop);

 // Elaborazione dei pacchetti
 for( ; ; ) {

 /*
  * Leggi la voce registrata.
  * PCM Lineare, 16 bits/campione ---> 20ms = 320 byte
  */
  if( ( p = read(au_fd, audio_buff,320)) == -1 ) {
     perror(au_name);
     exit(1);
    }
  if( p < 320 )
   padding = 320 - p;
```

```
  loop ++;

  // Elabora i dati in base al tipo di codifica.
  switch( map_position ) {
  case 0: // PCMU
  case 8: // PCMA

    // RTP-header
    rtp(packet, map_position, loop, numBp, padding);

    // Inizia le appropriate routine di codifica
    if( map_position == 8 )
     g711("la", audio_buff, packet+12, padding);
    else
     g711("lu", audio_buff, packet+12, padding);

    break;
  case 4: // G.723

    fprintf(stderr,
     "Sorry, %s coder is unavailable at moment\n",
     pt_map[4].encoder);
    exit(0);

    break;
  case 15:  // G.728
    numBp = 40;
    fprintf(stderr,
     "Sorry, %s coder is unavailable at moment\n",
     pt_map[15].encoder);
    exit(0);

    break;
  case 18:  // G.729
numBp = 20;

    // insert RTP-header
    rtp(packet, map_position, loop, numBp, 0);

     /* Loop for two "L_FRAME" speech data. */

   for(i=0;i<2;i++) {
```

```
   bcopy((Word16 *)audio_buff+i*80, new_speech, 2*L_FRAME);

   Pre_Process(new_speech, L_FRAME);

   Coder_ld8a(prm);

   info_synt(payload, prm); // bit packing

   bcopy(payload, packet+12+i*10, 10);

  }
    break;
  case 20: // MELP

    fprintf(stderr,
     "Sorry, %s coder is unavailable at moment\n",
     pt_map[20].encoder);
    exit(0);

    break;
  }

 // Spedisci l'attuale pacchetto
 if( sendto(sockfd, packet, numBp+12,
     0, (struct sockaddr *)&servaddr, sizeof(servaddr)) == -1 )
  perror("sendto");

 } /* for */


 return 0;

} /* main */
```

openline.c

```
/*
 ************************************************************************
 ************************************************************************
 * RTP-Speak.                                                           *
 * Principale routine di ricezione.                                     *
 * Versione 1.0 (Maggio 2001) (in progress)                             *
```

```
 * Gestisce:                                                          *
 * 1. La ricezione dall'interfaccia di rete                           *
 * 2. La decodifica                                                   *
 * 3. Il playout                                                      *
 **********************************************************************
 **********************************************************************
 * Autore: Daniele Mari                                               *
 * e-mail: daniele.mari@tiscalinet.it                                 *
 **********************************************************************
 **********************************************************************
                                                                      */

#include <stdio.h>
#include <unistd.h>
#include <stddef.h>
#include <string.h>
#include <strings.h>           // bcopy(), bzero()
#include <sys/soundcard.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/stat.h>
#include <signal.h>            // signal()
#include <sys/socket.h>        // struct sockadrr
#include <netinet/in.h>        // sockaddr_in
#include <stdlib.h>            // perror()
#include "rtp.h"               // RTP headers


                        /*
#define  MAXLEN 350      * massimo num di byte associabili ad un
                         * pacchetto RTP (è consigliato non superare
                         * questa soglia)
                         */

#define RECV 1

/*
 * Dichiarazioni globali per l'acqusizione audio
 *
 */

static char au_name[15] = "/dev/dsp";
```

```
int dsp_stereo = 0;    // mono
int dsp_speed = 8000;  // PCMA (defoult), PCMU, G729
extern int au_fd;
char *audio_buff;
int dsp_speed, aubuf_size;

/*
 * Lista dei pacchetti ricevuti (n.b. la lista è dinamica.)
 * I pacchetti in lista sono equalizzati dal jitter.
 */
struct packlist {
 char data[MAXLEN];           // RTP-header + payload
 struct packlist *next;       // puntatore all'elemanto successivo
} *packinfo;                        /*
struct packlist *start, *last;       * puntatori al primo e all'ultimo
                                     * elemento della lista
                                     */
struct packlist *prev=0;      // puntatore al pacchetto già elaborato

// pacchetti ricevuti da equalizzare
char packet[MAXLEN];

static int sockfd;  // input socket


static int pcount = 0;

/*
 * Questa funzione fa l'operazione inversa della 'info_synt' di
 * rtpmail.c.
 */

void info_extract( unsigned char payload[10], Word16 parm[PRM_SIZE+1])
{

 int i;
 unsigned short buff[10];

 for(i=0;i<10;i++)
  buff[i] = payload[i];

 parm[1]  = buff[0];
 parm[2]  = buff[1]+((buff[2]&192)<<2);
 parm[3]  = (buff[2]&63)+(buff[3]&192);
```

```
 parm[4]  = ((buff[3]&32)>>5)&1;
 parm[5]  = (buff[3]&31)+(buff[4]<<5);
 parm[6]  = ((buff[5]&240)>>4)&15;
 parm[7]  = (buff[5]&15)+(((buff[6]&224)>>1)&112);
 parm[8]  = buff[6]&31;
 parm[9]  = buff[7]+((buff[8]&248)<<5);
 parm[10] = (buff[8]&7)+(((buff[9]&128)>>4)&8);
 parm[11] = buff[9]&127;

}


/*
 * Questa funzione inserisce in una lista i pacchetti in arrivo
 * secondo l'ordine dettato dal 'sequence number', jitter equalizzato,
 * in modo che possano essere passati al decodificatore.
 * La lista è aggiornata dinamicamente.
 */
static
void store(struct packlist *i, struct packlist **start,
           struct packlist **last, int lseq)
{

 struct packlist *old, *p;
 rtp_hdr_t *hpi = (rtp_hdr_t *)(i->data);
 rtp_hdr_t *hpp;

/*
  * Se il pacchetto è in eccessivo ritardo, non va messo in lista.
  * 'lseq'  indica il sequence number dell'ultimo pacchetto ascoltato.
  */
 if( ntohs(hpi->seq) < lseq )
  goto no_store;

 p = *start;
 hpp =(rtp_hdr_t *)(p->data);

 if(!*last) { // primo elemento della lista (prima volta)
  i->next = NULL;
  *last = i;
  *start = i;
  return;
 }
```

```
 old = NULL;

 while(p) {
  if( ntohs(hpp->seq) < ntohs(hpi->seq) ) {
   old = p;
   p = p->next;
  }
  else {
   if(old) { // elemento mediano
    old->next = i;
    i->next = p;
    return;
   }
   i->next = p; // primo elemento
   *start = i;
   return;
  }
 } /* while */

 (*last)->next = i;  // ultimo elemento
 i->next = NULL;
 *last = i;

 no_store:

}  /* store */


/*
 * Funzione di fine.
 * Stampa un breve messaggio di notifica.
 * E' innescata, in modo asincrono, dall'utente mediante
 * il comando: ^C
 */
static void done(int signo)
{

 fprintf(stderr,"Trasmission interrupted\n");
 fprintf(stderr,"%d packets received\n",pcount);

 exit(0);

}
```

```
/*
 * Funzione principale
 */
int main(void)
{

 struct sockaddr_in servaddr, cliaddr;
 int clilen;
 int len;                /*
                          * # di byte per pacchetto (header+payload),
 * defoult: 20ms per pacchetto.
 */
 const int on = 1;
 int n;
 int padding = 0;        // # di byte di padding in un pacchetto
 int pad = 0;            // # byte di padding non decodificati
 rtp_hdr_t *hp;
 u_int16 lseq = 0;
 extern void set_snd(char *audio_name, int open_mod, int dsp_stereo);
 extern int g711(char type_of_conv[2], char *data_in, char *data_out,
        int padding);

 unsigned char payload[10];
 Word16 i;

 Word16  synth_buf[L_FRAME+M], *synth; /* Synthesis               */
 Word16  parm[PRM_SIZE+1];           /* Synthesis parameters       */
 Word16  Az_dec[MP1*2];        /* Decoded Az for post-filter  */
 Word16  T2[2];                /* Pitch lag for 2 subframes   */


 /*
  * Si apre una socket UDP non connessa, per i pacchetti RTP.
  */
 bzero(&servaddr, sizeof(servaddr));

 servaddr.sin_family = AF_INET;
 servaddr.sin_port   = htons(5004);
 servaddr.sin_addr.s_addr = htonl(INADDR_ANY);

 sockfd = socket(AF_INET, SOCK_DGRAM, 0);
 if( sockfd < 0 ) {
```

```
 perror("socket");
 exit(1);
}

/*
 * Opzione SO_REUSEADDR. E' cautelativa nel caso sia stata già aperta
 * una connessione su questa porta.
if( setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on)) ==
    -1 )
 perror("setsockopt (SO_REUSEADDR)");

//  Si incrementa il socket-receive-buffer ad un valore ragionevole
n = 240 * 1024;
if( setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &n, sizeof(n)) == -1 )
 perror("setsockopt (SO_RCVBUF)");

if( bind(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0
) {
 perror("bind");
 exit(1);
}

start = last = NULL;

fprintf(stderr,"\n");
fprintf(stderr,"-----------------------------------------------\n");
fprintf(stderr,"********************RTP-Speak******************\n");
fprintf(stderr,"**************Version 1.0, May 2001*************\n");
fprintf(stderr,"\n");
fprintf(stderr,">>>>>>>>>>>>Designed by: Daniele Mari<<<<<<<<<<<\n");
fprintf(stderr,"-----------------------------------------------\n");
fprintf(stderr,"\n");


/*
 * Inizializzazione della scheda audio per il playback.
 */
set_snd(au_name, O_WRONLY, dsp_stereo);
for (i=0; i<M; i++) synth_buf[i] = 0;
  synth = synth_buf + M;

  bad_lsf = 0;          /* Initialize bad LSF indicator */
  Init_Decod_ld8a();
  Init_Post_Filter();
```

```
  Init_Post_Process();


signal(SIGINT, done);

// Principale Loop di ricezione
while(RECV) {

 fill_buffer:

 clilen = sizeof(cliaddr);

 len = recvfrom(sockfd, packet, MAXLEN,
                0,(struct sockaddr *)&cliaddr, &clilen);

/*
 * Routine di immagazzinamento nella lista a jitter equalizzato.
 */
 packinfo = (struct packlist *)malloc(sizeof(struct packlist));
 if(!packinfo)
  fprintf(stderr,"Warning, no more memory for incoming packets\n");

 bcopy(packet , packinfo->data, len );

 store(packinfo, &start, &last, lseq);

 pcount++;

 // jitter-buffer (20 pacchetti)
  /*
   * momentaneamente, la soluzione è quella di riempire il buffer
   * con un numero di pacchetti '20' sufficiente ad eliminare il
   * jitter e, contemporaneamente, a non ritardare troppo l'emissione
   * dei pacchetti verso il decodeificatore.
   * Il Jitter-Buffer, una volta riempito con i primi '20' pacchetti,
   * ne conterrà sempre '20'.
   */
 if( pcount < 20 )
  goto fill_buffer;

 fprintf(stderr,"%d packet parsed from jitter-buffer\r",pcount);

 // aggiorna il puntatore'hp' al pacchetto da ascoltare.
 hp = (rtp_hdr_t *)(start->data);
```

```
lseq = ntohs( hp->seq );

// Individua il decodificatore appropriato attraverso il PT
switch( hp->pt ) {
case 0: // PCMU
case 8: // PCMA

  // valuta il campo di padding
  if( hp->p == 1 ) {
  padding = start->data[160-1];
  pad = start->data[160-1] * 2;
 }

   /*
    * Innesca le appropriate routine di decodifica e
    * depacchettizzazione
    */
  if( hp->pt == 8 )
   g711("al",&(start->data[12]), audio_buff, padding);
  else
   g711("ul",&(start->data[12]), audio_buff, padding);

  break;
case 4: // G.723

  fprintf(stderr,
   "Sorry, G.723 coder is unavailable at moment\n");
  exit(0);

  break;
case 15:  // G.728

  fprintf(stderr,
  "Sorry, G.728 coder is unavailable at moment\n");
  exit(0);

  break;
case 18:  // G.729

 // N. B. padding field deve essere sempre zero.

 for(i=0;i<2;i++) {

 bcopy(&(start->data[12+i*10]), payload, 10);
```

```
 info_extract(payload, parm); // bit depacketization

 /* the hardware detects frame erasures by checking if all bits
    are set to zero
  */
 parm[0] = 0;           /* No frame erasure */

 /* check pitch parity and put 1 in parm[4] if parity error */

 parm[4] = Check_Parity_Pitch(parm[3], parm[4]);

 Decod_ld8a(parm, synth, Az_dec, T2);

 Post_Filter(synth, Az_dec, T2);    /* Post-filter */

 Post_Process(synth, L_FRAME);

 bcopy(synth, (Word16 *)audio_buff+i*80, 2*L_FRAME);
 }
  break;
case 20: // MELP

  fprintf(stderr,
   "Sorry, MELP coder is unavailable at moment\n");
  exit(0);

  break;
}

/*
 * Playback.
 * Ogni pacchetto ricevuto è decodificato in
 * PCM lineare, 16 bits/campione ---> 20ms = 320 byte.
 * Attenzione al padding.
 */
if( write(au_fd, audio_buff, (320-pad) ) != (320-pad) ) {
 perror(au_name);
 exit(1);
}


/*
 * La lista, che verrà scandita per la memorizzazione, conterrà
```

```
   * sempre solo '20' pacchetti e non di più. In questo modo si
   * diminuirà notevolmente il tempo di ricerca del punto di
   * memorizzazione, arginando, si spera, il rischio di OVERFLOW.
   *
   * Si libera la memoria, con free(), per evitare che una lunga
   * conversazione venga meno per mancanza di memoria da allocare.
   */
 prev = start;
 start = start->next;
 free(prev);

 } /* while */

return 0;

} /* main */
```

Glossario

API	Application Program Interface
AR	Autoregression
ARPA	Advanced Research Project Agency
A/V	Audio/Video
COD	Coder
CPU	Control Unit Processing
CNG	Comfort Noise Gebnerator
CS-ACELP	Conijugate Structure Algebraic Code excited Linear Prediction
CSRC	Contributing Source Identifier
DEC	Decoder
DSP	Digital Signal Processor
FAC	Funzione di AutoCorrelazione
f_s	Sampling frequency
FTP	File Transfer Protocoll
GUI	Grafical User Interface
hbo	host byte order
IANA	Internet Assigned Numbers authority
I/O	Input/Output
IP	Internet Protocol
IGMP	Internet Group Managment Protocoll
ISO	International Standard Organizzation
ITU	International Telecommunication Union
LAN	Local Area Network
LP	Linear Prediction
LSB	Last Significant Bit
MRP	Multicast Routing Protocoll
MSB	Most Significant Bit
nbo	network byte order
OSI	Open Sistem Interconnection
P	Padding
PAM	Pulse amplitude Modulation
PBX	Private Branch Excange
PCM	Pulse Code Modulation
PCMA	PCM A-law

PCMμ	PCM μ-law
POSIX	Portable Operating System Interface
PSTN	Public Switched Telephone Network
PT	Payload Type
QoS	Quality of Service
QoV	Quality of Voice
RSVP	Resource ReSerVation Protocol
RSTP	Reliable Signalling Transport Protocoll
RTT	Round Trip Time
RTP	Real Time Transport Protocol
RTCP	Real Time Transport Control Protocol
RR	Receiver Report
SDES	Source Description items
SO	Sistema Operativo
SR	Sender Report
SSRC	Sincronizzation Source Identifier
STP	Server Processing Time
TCP	Trasmission Control Protocol
TCP/IP	Internet Protocol Suite
TFTP	Trivial File Transfer Protocoll
T_s	Sampling rate
UDP/IP	Voice over IP Suite
UDP	User Datagram Protocol
V	Version
VAD	Voice Activity detector
VoIP	Voice over IP
WAN	Wide Area Network
X	Extension

Bibliografia

[1] – G. Thomsen, Y. Jani, “Internet telephony: going like crazy,” IEEE Spectrum, May 2000, vol. 37, n° 5, pp. 52-58.

[2] – M. Decina, V. Trecodi, “Voice over Internet protocol and Human-Assisted E-Commerce,” IEEE Comm. Magazine, vol. 37, n° 9, Sept. 1999, pp. 64-67.

[3] – D. Cassing, “C6x solutions for voice over IP gateway,” IEEE Northcon/98 Conference proceedings, pp. 74-85.

[4] – A. S. Tanenbaum, “Reti di Computer,” Prentice Hall, Jackson Libri s. r. l., 1995.

[5] – D. Wu, Y. T. Hou, B. Li, Z. Wenwu, Z. Ya-Qin, H. J. Chao, “An end-to-end approach for optimal mode selection in Internet Communication: Theory and Application,” Selected Areas in Communnications, IEEE Journal on, Joune 2000, vol. 18, n° 6, pp. 977-995.

[6] – M. H. Sherif, A. Crossman, “Overview of speech packetisation,” IEEE Computer and Communication, Proceedings, Apr. 1995, pp. 296-304.

[7] – Dan Wing, B Thompson, Tmima Koren, “Tunnelling multiplexed compressed RTP (‘TCRTP’),” IETF, Internet Draft avt-group, 2000.

[8] - ITU-T Rec. H.323, “Packet based multimedia communications systems,” V. 2, 1998. Link: http://www.itu.int/itudoc/itu-t/rec/h/h323.html

[9] – Data Beam Corp, “A primer on the H.323 series standard version 2.0,” Link: http://www.iptelephony.org/

[10] – H. Schulzrinne, J. Rosenborg, “The IETF Internet telephony architecture and protocols,” IEEE Network, vol 13, n° 3, May 1999, pp. 18-23.

[11] – X. Xiao, L. Ni, “Internet QoS: A big picture,” IEEE Network, vol. 13, n° 2, Mar/Apr 1999, pp. 8-18.

[12] – B. Li, M. Hamdi, D. Jiang, X. Cao, Y. T. Hou, “QoS-enabled voice support in the next-generation Internet: issues, existing approaches and challenger,” IEEE Commm. Magazine, Apr. 2000, vol. 38, n° 4, pp. 54-61.

[13] – L. Zhang et al., “RSVP, a new Resource Reservation Prtocol,” IEEE Network, Sept. 1993, pp. 8-16.

[14] – R. Braden, L. Zhang , S. Berson, S. Herzog, S. Jamin, “Resorce ReSerVation Protocol functional signification,” RFC 2205, Sept. 1997.

[15] – B. Goodman, B. Atantic, “Internet telephony and modem delay,” IEEE Network, vol. 13 n° 3, May-June 1999, pp. 8-16.

[16] – C. Perkins, J. Crowcroft, “Effects of Interleaving on RTP header compression,” IEEE Infocom 2000, Nineteenth Annual Joint Conference of IEEE Computer and Communication Societies Proceedings, vol. 1, pp. 111-117.

[17] – T. J. Kostas, M. S. Borella, I. Sidhu, G. M. Shuster, J. Grabiec, J. Mahler, “Real-Time Voice over racket-switched networks,” IEEE Network, Jan.-Feb. 1998, vol.12 n°1, pp. 18-27.

[18] – T. Braun, “Internet protocols for multimedia communications. Part I: Ipng-the foundation of the Internet protocols,” IEEE Standards, vol. 44, July-Sept. 1997, pp. 85-90.

[19] – T. Braun, “Internet protocols for multimedia communications. Part II: Resource Reservation, Transport, and Application protocols,” IEEE Standards, vol. 44, Oct.-Dec. 1997, pp. 74-82.

[20] – http://www.protocols.com/pbook/tcpip.html

[21]– Information Science Institute University of Southern California, “Internet Protocol DARPA Internet program and protocol specification,” RFC 791, Sept. 1981.

[22] – S. Deerin, R. Hinden, “Internet Protocol, version 6 (specifications),” RFC 2460, Dec. 1998.

[23] – J. B. Postel, “User Datagram Protocol,” ISI, RFC 768, Aug. 1980.

[24] – J. B. Postel, “Trasmission Control Protocol,” ISI, RFC 768, Sept. 1981.

[25] – G. Bai, K. Long, W. Wang, S. Cheng, “A new reliable Signalling Transport Protocol RSTP for voice over IP,” IEEE Conferenc., 18-22 Oct. 1999, pp. 263-266.

[26] – P. P. Mishara, G. Wireles, H. Saran, “Capacity management and routing policies for voice over IP traffic,” IEEE Network, Mar.-Apr. 2000, vol. 14, n°2, pp. 20-27.

[27] – H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson, “RTP: A transport protocol for real-time applications,” STD1, Internet Engeneering Task Force, RFC 1889, Jan. 1996.

[28] – H. Schulzrinne, S. Casner, “RTP profile for Audio and Video Conferences whit minimal control,” IETF, Internet-Draft, July 2000.

[29] - D. L. Mills, “Network Time Protocol (version 3), specification, implementazion and analysis,” Network Working Groupe, RFC1305.

[30] – L. R. Rabiner, R.V. Shafer, “Digital processing of speech signals”, Prentice Hall, 1978.

[31] - T. W. Anderson, “The statisical analisis of time series,” Wiley, N. J., 1971.

[32] - KAY S. M., “Modern spectral estimation: Teory & Application,” Prentice Hall, N. J., 1988.

[33] – R. Salami, C. Laflamme, J. P. Adoul, A. Kataoka, S. Hayashi, T. Moriya, C. Lamblin, D. Massaloux, S. Proust, P. Kroon, Y. Shoham, “Design and description of CS-ACELP: a toll quality 8 kb/s speech coder,” IEEE Trans. Speech and Audio processing, vol. 6, n° 2, March 1998, pp. 116 – 130.

[34] ─ R. Salami, C. Laflamme, J. P. Adoul, A. Kataoka, S. Hayashi, C. Lamblin, D. Massaloux, S. Proust, P. Kroon, Y. Shoham, “Description of the proposed ITU-T 8 kb/s speech coding standard,” Proc. IEEE speech coding Wksp., Annapolis, MD, Sept. 1995, pp. 3 – 4.

[35] – A. Katoka, J. Ikedo and S. Hayashi, “LSP and Gain quantization for the proposed ITU-T 8-kb/s speech coding standard,” IEEE speech coding Wksp., Annapolis, MD, Sept. 1995, pp. 7 – 8.

[36] – D. Massaloux and S. Proust, “Spectral shaping in the proposed ITU-T 8 kb/s speech coding standard,” IEEE speech coding Wksp., Annapolis, MD, Sept. 1995, pp. 9 – 10.

[37] - A. V. Oppenheim, R. W. Shafer, “Discrete-Time signal processing,” Prentice Hall, Englewood Cliffs, N. J., 1989.

[38] – R. Salami, C. Laflamme, B. Bassette and J. P. Adoul, "ITU-T G.729 Annex A : Reduced complexity 8 kb/s CS-ACELP codec for digital simultaneus voice and data," IEEE communication magazine, Sept. 1997, vol. 35, n° 9, pp. 56 – 63.

[39] - C. Batini, L. Carlucci Aiello, M. Lenzerini, A. Marchetti Spaccamela, A. Miola, "Fondamenti di programmazione dei calcolatori elettronici", Franco Angeli, 7nd ed., 1994.

[40] - IEEE 1988, "Portable Operatine System Interface for Computer Enviroment (POSIX)," IEEE, Sept. 1988.

[41] - W. R. Stevens, "Unix Network Programming. Networking APIs: Sockets and XTI," Prentice Hall, vol.1, 2nd ed., 1998.

[42]- Opensound Systems™, "Programmers Guide. Version 1.11," 4Front Technologies. URL: http://www.opemsound.com

[43] - J. Reynolds, J. Postel, "Assigned Numbers," ISI RFC 1700, Oct. 1994.

[44] – W. R. Stevens, "Unix Network programming. Interprocess communications," Prentice Hall, vol.2, 2nd ed., 1998.

[45] - A. S. Tanenbaum, "Architettura del computer," Prentice Hall, Jackson Libri s.r.l., 1991.

[46] – H. S. Schildt, "C: la guida completa, terza edizione," Mc Graw-Hill Libri Italia, 2000.

[47] – http://www.infocom.uniroma1.it/switch/

[48] - http://www.gnu.org/software/gcc/gcc.html

[49] - M. Richard, S. McGrath, R. McGrath, "GNU Make. A program for directing recompilation," May 1998, http://www.gnu.org/manual/make-3.77/make.html.