talKZ

Cosimo Alfarano <kalfa@cs.unibo.it>
Stefano Zacchiroli <zack@cs.unibo.it>

2 ottobre 2002

Indice

Elenco delle figure

Elenco delle tabelle

1 Introduzione

Nell'ambito del Corso di Sistemi di Elaborazione 3 abbiamo realizzato una implementazione del noto programma `talk` (diffuso da BSD 4.2 in poi) e del corrispondente demone `talkd`.

2 Obiettivi

Esistono molte implementazioni della suite talk attualmente disponibili, la maggior parte delle quali si differenziano solamente nelle interfacce utente offerte o per l'aggiunta di servizi non standard (o comunque poco diffusi) come la possibilità di parlare in 3 o più.

La nostra implementazione si è voluta basare sull'implementazione addizionale di due opzioni non fornite dalla stragrande maggioranza di implementazioni da noi valutate: il logging e il supporto SSL nativo.

La necessità di **logging** emerge dal fatto che spesso capita di intrattenere lunghe sessioni talk e sarebbe desiderabile potere salvare in un semplice file di testo la sessione stessa distinguendo il testo scritto dai due partecipanti. A prima vista può sembrare una necessità banale, ma difficilmente ottenibile in altro modo: non è ad esempio possibile utilizzare programmi esterni come `screen` per ottenere lo stesso risultato in quanto le implementazioni testuali di talk utilizzano librerie (come le *ncurses*) che accedono direttamente al terminale.

Il supporto **SSL** è un'altra caratteristica desiderabile ed è auspicabile implementarla senza dovere ricorrere a tool esterni (come *stunnel*) avendo inoltre a disposizione un protocollo di accordo tra i due partecipanti per decidere se la comunicazione debba essere criptata o meno.

Un altro obiettivo importante di **talKZ** è stata la **compatibilità** con le attuali implementazioni di talk disponibili, sia per quanto riguarda il client che per quanto riguarda il server. È quindi possibile utilizzare qualsiasi combinazione di client e server sia standard che proveniente dalla suite **talKZ** .

3 Protocollo Talk

> The protocol used to communicate with the talk daemon is braindead.
>
> — *BUGS* section of the talk manpage

Abbiamo incontrato diverse difficoltà durante la realizzazione di **talKZ** , prima fra tutte quella di reperire documentazione sul funzionamento del protocollo talk. È importante sottolineare che, a differenza della stragrande maggioranza dei protocolli utilizzati nel mondo di Internet, il protocollo talk non è mai stato standardizzato da nessun ente. Questo ovviamente ha dato origine ad una proliferazione di diversi protocolli quasi sempre incompatibili tra loro.

Ciò nonostante l'implementazione che si può considerare affermatasi come standard *de facto* in ambiente Linux è quella realizzata per la prima volta su BSD 4.2; altri standard si sono invece affermati in ambiente SunOS/Solaris.

Per il resto della sezione parleremo di *protocollo talk* intendendo il protocollo utilizzato dall'implementazione BSD di talk e talkd.

3.1 Descrizione del protocollo

Il protocollo talk prevede due attori fondamentali: client e server. A differenza di quanto comunemente accade però, l'instaurazione di una sessione di talk prevede la presenza di due client e due server, ogni partecipante alla sessione infatti deve avere attivo sul proprio host un demone talkd ed eseguire un client talk per prendere parte alla comunicazione.

L'idea sottostante a tutto il protocollo è di avere un *luogo* nel quale custodire *inviti* di partecipazione ad una sessione talk.

Il luogo corrisponde nella pratica ad un demone talkd, un invito è invece costituito da informazioni su chi ha chiamato e su come contattarlo per iniziare una sessione.

Il protocollo prevede inolte una rigida distinzione tra *caller* (colui che per primo decide di iniziare una sessione) e *callee* (colui che risponde alla chiamata di un caller).

Lo scambio di messaggi tra client e server avviene via UDP ed ogni server talk [1] è in ascolto sulla *well known port* ([1], [2]) 518.

I messaggi che un client invia ad un server prendono il nome di *richieste* che possono essere di diversi tipi (tab. 1 a pag. 3). I messaggi che un client invia ad un server corrispondono sempre ad una richiesta e prendono il nome di *risposte*, anch'esse possono essere di diversi tipi (tab. 2 a pag. 4).

Tabella 1: *Richieste* (messaggi client—server)

tipo	**descrizione**
`LEAVE_INVITE`	deposito di un invito sul server
`LOOK_UP`	controllo per la presenza di un invito
`DELETE`	rimozione di un invito
`ANNOUNCE`	annuncio ad un utente

L'inizio del protocollo è affidato al caller che, non appena decide di iniziare una sessione talk con un altro utente, invia una richiesta di tipo `LOOK_UP` al demone talk dell'utente remoto.

[1] (*ntalk* per la precisione, si veda sez. 4 a pag. 5)

Tabella 2: *Risposte* (messaggi server—client)

tipo	**descrizione**
SUCCESS	richiesta completata con successo
NOT_HERE	callee non connesso *o* nessun invito per il caller
FAILED	richiesta *non* completata con successo
MACHINE_UNKNOWN	emphhostname del caller non valido
PERMISSION_DENIED	impossibile avvisare il callee
UNKNOWN_REQUEST	richiesta non supportata
BADVERSION	versione non supportata
BADADDR	campo `addr` non valido
BADCTLADDR	campo `ctl_addr` non valido

Il server remoto risponde con `NOT_HERE` ciò significa che l'utente remoto non ha ancora lasciato alcun invito per il caller (questo caso si verifica sempre per il caller).

Il caller ora invia due richieste: una richiesta di tipo `ANNOUNCE` al server remoto in modo che questi si incarichi di segnalare all'utente remoto la richiesta di comunicazione (solitamente scrivendo alcuni messaggi sulle *console*) e una richiesta di tipo `LEAVE_INVITE` al server locale (presso il caller) in modo che questi conservi l'invito. Prima di inviare la risposta `LEAVE_INVITE` al server local il caller si preoccupa inoltre di creare un socket TCP che utilizzerà per attendere connessione dall'utente remoto; l'indirizzo di questo socket viene salvato nell'invito lasciato al server locale.

Alla richista `LEAVE_INVITE` solitamente corrisponde una risposta di tipo `SUCCESS` , alla richiesta `ANNOUNCE` possono corrispondere invece diverse risposte: `SUCCESS` (l'utente remoto è stato avvisato), `NOT_HERE` (l'utente remoto non è connesso all'host remoto), `FAILED` (l'utente remoto è connesso, ma non è stato possibile avvisarlo[2]. Nel caso di risposta `NOT_HERE` o `FAILED` non è possibile per il caller contattare l'utente remoto e il protocollo viene abbandonato spedendo due richieste `DELETE` ai demoni locali e remoti in modo che l'uno non provi più ad avvisare l'utente remoto e che l'altro cancelli l'invito precedentemente depositato.

Nel caso invece in cui la risposta sia stata `SUCCESS` il protocollo procede normalmente e il caller si mette in attesa di connessioni[3] sul socket TCP precedentemente creato.

Appena l'utente remoto (il callee) si accorge della richiesta di talk da parte del caller inizia una parte di protocollo analoga a quella del caller, ovvero invia una richiesta di tipo `LOOK_UP` al server del caller. A differenza del caller però la risposta sarà di tipo `SUCCESS` [4] e conterrà tutte le informazioni

[2]ad esempio nel caso in cui l'utente non accetti messaggi sulle console

[3]mediante la system call *listen*

[4]a meno che l'invito non sia scaduto, in questo caso il callee ricopre il ruolo del caller e il protocollo riparte dall'inizio

che il caller aveva precedentemente lasciato nell'invito.

Il callee ora ha tutte le informazioni necessarie per contattare direttamente il caller, crea quindi un socket TCP, si connette[5] al caller e invia una richiesta di tipo `DELETE` ad ogni demone analogamente a quanto accadeva nel caso in cui il caller non riuscisse a contattare il callee.

Una volta stabilita una connessione TCP tra i due partecipanti ognuno di essi invia 3 caratteri all'altro, rispettivamente: il carattere di *elimina carattere*, quello di *elimina riga* e quello di *elimina parola*. Questo scambio di caratteri è necessario in quanto i due partecipanti probabilmente utilizzeranno terminali diversi e non è detto che questi terminali usino gli stessi caratteri per queste tre funzioni.

Il primo carattere corrisponde semanticamente all'usuale *backspace* o `CTRL-H`, il secondo a `CTRL-U` e il terzo a `CTRL-W` delle shell o di Emacs.

Abbiamo anche realizzato un **diagramma degli stati** che rappresenta schematicamente lo svolgimento del protocollo visto da parte dei client (fig. 1 pag. 6). È importante sottolineare che il diagramma è utile solamente come linea guida e non è considerabile come una descrizione formale del protocollo e nemmeno come riferimento per le implementazioni talk attualmente disponibili.

4 Implementazioni attuali

Come già detto esiste una moltitudine di diverse implementazioni di talk e altrettanti diversi protocolli, in questa sezione descriviamo le differenze tra i diversi protocolli e le conformità di **talKZ** a questi protocolli.

Il criterio di classificazione delle implementazioni è il formato delle richieste e delle risposte scambiate tra client e server in quanto la dinamica del protocollo è la stessa per tutte le implementazioni così come è identica la parte di protocollo dalla connessione TCP tra i due client in poi.

4.1 talk vs ntalk

La prima distinzione necessaria è quella tra *talk* e *ntalk*, due implementazioni diverse e incompatibili tra loro.

La differenza tra le due consiste nella well known port sulla quale il demone è in ascolto e nel formato dei dati scambiati tra client e server.

Il demone *talk* è in ascolto sulla porta UDP 517, mentro il demone *ntalk* è in ascolto sulla porta UDP 518. Il nome non è comunque sempre significativo, ad esempio i pacchetti talk e talkd delle più note distribuzioni Linux implementano in realtà il protocollo ntalk, anche se le recenti versioni degli stessi pacchetti supportano entrambi i protocolli.

[5] mediante la system call *connect*

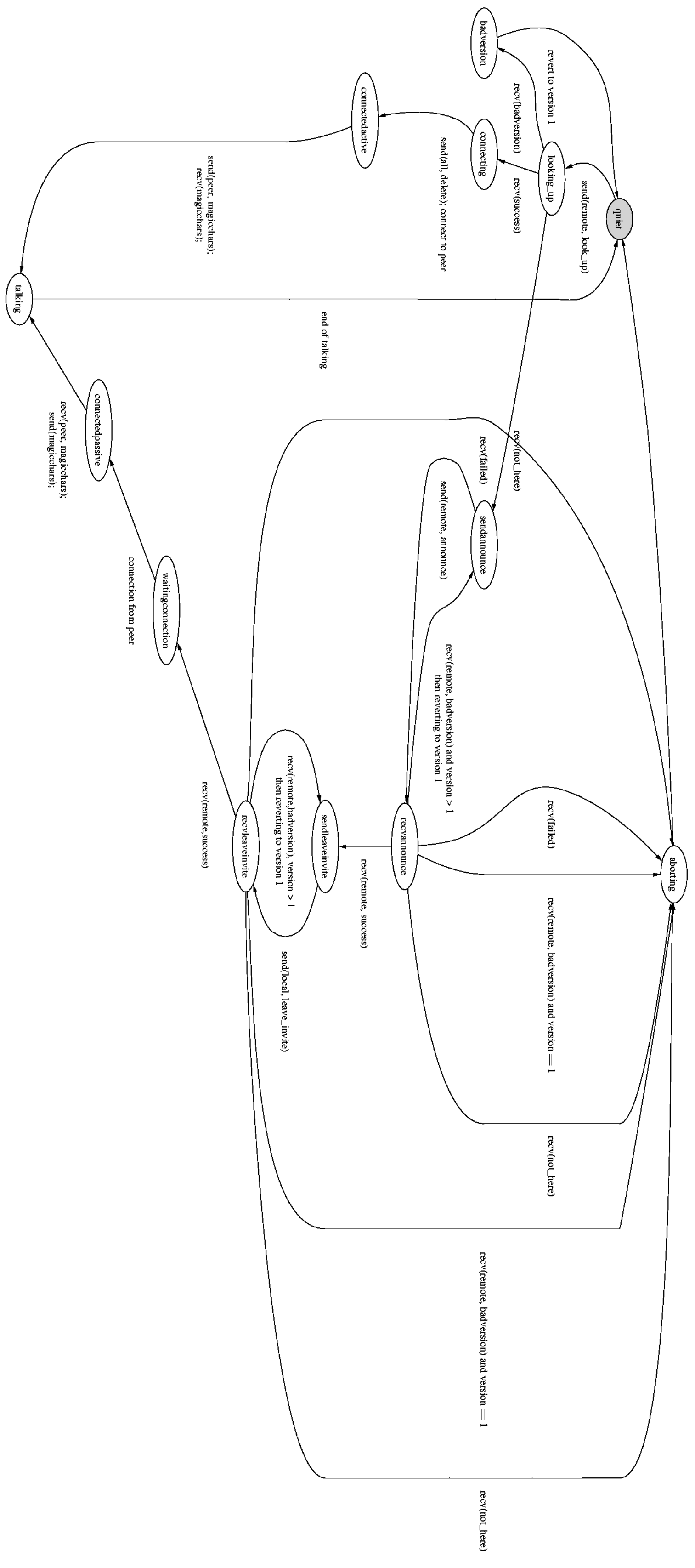

Figura 1: Diagramma degli stati del protocollo ntalk BSD

Per quanto riguarda il formato dei messaggi riportiamo la definizione delle struct C che vengono scambiate tra client e server (si veda tab. 3 a pag. 7 e tab. 4 a pag. 8), è importante notare che, sebbene i due formati siano semanticamente equivalenti (ovvero trasportino le stesse informazioni) l'ordinamento dei campi è tale da impedire compatibilità tra i due protocolli, l'unico modo per distinguere tra i due è appunto avere punti di accesso diversi (in questo caso due porte differenti).

Tabella 3: Formato delle richieste e delle risposte nel protocollo *talk*

```
typedef struct {
    char                type;
    char                l_name[9];
    char                r_name[9];
    unsigned short      pad;
    long                idnum;
    long                pid;
    char                r_tty[16];
    struct sockaddr_in  addr;
    struct sockaddr_in  ctl_addr;
} CTL_MSG;

typedef struct {
    char                type;
    char                answer;
    char                pad;
    long                idnum;
    struct sockaddr_in  addr;
} CTL_RESPONSE;
```

talKZ implementa solamente il protocollo ntalk, attualmente il più diffuso in ambiente Linux in quanto l'implementazione dei noti pacchetti talk e talkd di molte distribuzioni, sebbene supporti entrambi i protocolli, prova inizialmente ad utilizzare il protocollo ntalk.

4.2 BSD vs SunOS/Solaris

L'altra importante distinzione è tra l'implementazione originale BSD e quella SunOS/Solaris.

In ambiente SunOS/Solaris l'implementazione di talk utilizza un formato di messaggi diverso sia da quello previsto da talk che da quello pre-

Tabella 4: Formato delle richieste e delle risposte nel protocollo *ntalk*

```
typedef struct {
    unsigned char       vers;
    char                type;
    unsigned short      pad;
    long                idnum;
    struct sockaddr_in  addr;
    struct sockaddr_in  ctl_addr;
    long                pid;
    char                l_name[12];
    char                r_name[12];
    char                r_tty[16];
} CTL_MSG;

typedef struct {
    unsigned char       vers;
    char                type;
    char                answer;
    char                pad;
    long                idnum;
    struct sockaddr_in  addr;
} CTL_RESPONSE;
```

Tabella 5: Formato delle richieste e delle risposte nel protocollo SunOS/Solaris

```
typedef struct {
        u_char  type;
        char    l_name[NAME_SIZE_S];
        char    r_name[NAME_SIZE_S];
        u_char  pad;
        u_int32_t id_num;
        int32_t pid;
        char    r_tty[TTY_SIZE];
        struct  osockaddr addr;
        struct  osockaddr ctl_addr;
} CTL_MSG_S;

typedef struct {
        u_char  type;
        u_char  answer;
        u_short  pad;
        u_long id_num;
        struct  osockaddr addr;
} CTL_RESPONSE_S;

NAME_SIZE_S == 9, TTY_SIZE == 16
```

visto di ntalk. Si veda tab. 5 a pag. 9 per il formato dei messaggi nell'implementazione SunOS.

L'implementazione di **talKZ** supporta il formato di richieste e risposte dell'implementazione originale BSD.

5 TalKZ

talKZ è basato sull'implementazione del protocollo ntalk come definito dalla suite talk di BSD. Ciò nonostante è risultato necessario apportare alcune modifiche al protocollo per supportare le opzione che **talKZ** offre in aggiunta all'implementazione standard.

Per quanto riguarda il logging nessuna modifica è risultata necessaria in quanto questo viene gestito interamente dal client senza necessità di alcuno scambio di informazioni ne' con l'altro client ne' con nessuno dei due server.

La situazione risulta invece diversa per quanto riguarda il supporto SSL in quanto è necessario che i due client raggiungano un accordo sull'utilizzo o meno di SSL prima di iniziare a comunicare tra loro. Per implementare questa opzione abbiamo aggiunto alcune fasi al protocollo atte allo scambio di *capability* tra i due partecipanti, si veda sez. 5.1 a pag. 10.

Riportiamo anche il **diagramma degli stati** del protocollo **talKZ** (fig. 2 pag. 11, anch'esso, come nel caso del diagramma del protocollo talk standard, non è da considerarsi come una descrizione formale del protocollo, ma piuttosto come un insieme di linee guida per l'implementazione del protocollo stesso.

5.1 Protocollo

Il protocollo **talKZ** si differenzia dal protocollo ntalk BSD per due aspetti: il valore del campo **version** contenuto nelle richieste e nelle risposte, e lo scambio di **capability** tra i due peer dopo che si è instaurata la connessione TCP.

5.1.1 Versione

Sia il client che il server **talKZ** come default utilizzano un valore nel campo versione diverso dal valore standard 1 utilizzato dalle implementazioni ntalk BSD comunemente diffuse.

È importante sottolineare che questa differenza nel campo versione è utile nel caso sia supportata dal server, per diminuire il numero di messaggi scambiati in rete e inutili tempi di attesa, ma *non è assolutamente indispensabile* per l'utilizzo delle opzioni supportate da **talKZ** .

Per analizzare l'utilità di tale valore è necessario distinguere diversi casi:

client talkz - server ntalk BSD in questo caso il server risponderà `BADVERSION` alla prima richiesta del client che farà fallback alla versione 1 e da ora in poi scambierà messagi con il server utilizzando quest'ultima

client talk - server talkz in questo caso il server si accorgerà al primo messaggio ricevuto che il client è un client standard e si comperterà come un normale demone ntalk BSD, con l'unica differenza che, se l'altro client è un callee talkz, verra lui comunicato che il primo client non è talkz

client talkz - server talkz in questo caso tutti i messaggi scambiati sono con la nuova versione e il callee verrà informato del fatto che il caller è talkz

5.1.2 Capability

La parte addizionale di protocollo implementata da **talKZ** si aggiunge come parte finale del protocollo talk, ovvero dopo che i due partecipanti alla

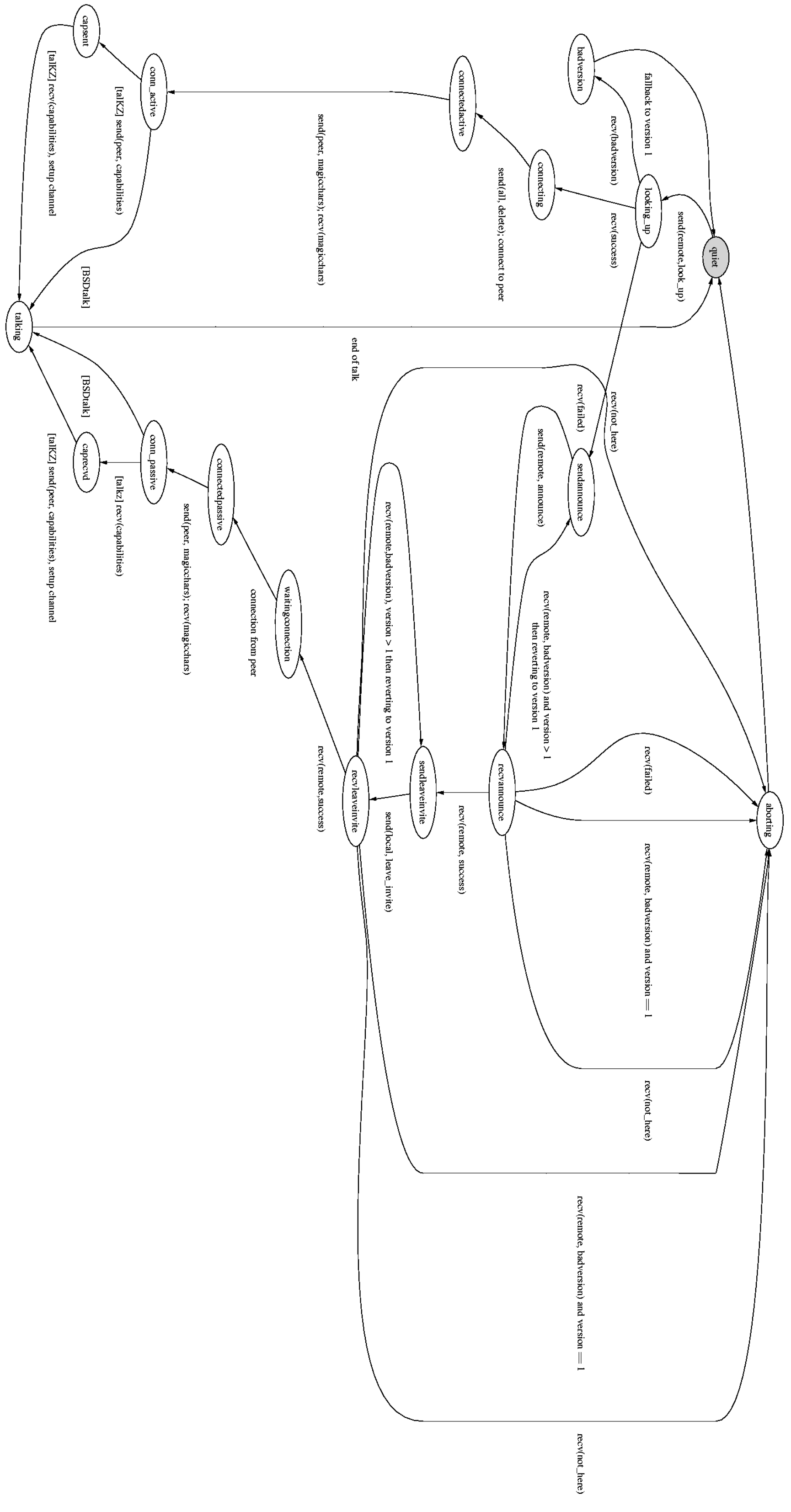

Figura 2: Diagramma degli stati del protocollo **talKZ**

sessione si sono scambiati i caratteri di cancella carattere, cancella riga, cancella parola.

Dopo questa fase infatti i due client entrano nella fase che abbiamo chiamato *scambio della capability.*

Questa fase è distinta a seconda del ruolo del client (caller o callee); il callee è colui che effettua la connessione *attiva* all'altro partecipante, appena la fase di scambio dei tre caratteri è terminata il callee invia un carattere di capability al caller e attende in risposta un carattera analogo. Il caller si comporta analogamente con la differenza che prima legge il carattere di capability e poi ne invia uno indietro al callee.

L'idea di base di questo protocollo è quella di avere uno dei due client (nel nostro caso il callee) che *propone* alcune capability all'altro client che *sceglie* quelle che è in grado di supportare tra quelle ricevute e invia le scelte indietro al mittente, questi può ora o accettare e inizare la comunicazioni utilizzando le capability ricevute oppure abbandonare e chiudere il socket.

Sia la scelta di quali capability inviare all'inizio che quella delle capability con le quali rispondere sono effettuate in base ai desideri dall'utente che possono essere specificati sia a riga di comando che con file di configurazione.

L'insieme di capability supportate dall'uno e dall'altro client è attualmente codificato in un unico carattere per motivi di semplicità implementativa e per il fatto che attualmente esiste una unica capability (il supporto o meno di SSL), può comunque essere esteso e codificato in un numero arbitrario di caratteri.

Per motivi di **compatibilità** la parte finale del protocollo (quella specifica di **talKZ**) non deve essere eseguita nel caso in cui anche solo uno dei due partecipanti alla comunicazione sia un client ntalk BSD. La soluzione a questo problema implementata in **talKZ** è diversa se vista dal lato del caller o dal lato del callee.

Il callee ha infatti modo[6] di conoscere la versione del caller in quanto il demone che conserva l'invito la include nella risposta; il caller non ha invece modo di conoscere la versione supportata dal callee in quanto dopo avere lasciato l'invito sul demone locale, non comunica più con nessuno fino a quando non riceve la connessione TCP dal callee.

Per risolvere il problema del caller abbiamo implementato un meccanismo di timeout sulla ricezione della capability, se queste non vengono ricevute entro un tempo limite, il caller assume che il callee sia un client ntalk BSD e che pertanto non supporti lo scambio di capability. Infine abbiamo fatto in modo che i caratteri di capability siano caratteri non stampabili o comunque non ottenbili da console in modo che un carattere inviato dal callee prima dello scadere del timeout non sia interpretato erroneamente come una capability.

[6]solo nel caso in cui il demone remoto sia talkz, in caso contrario si comporta come se il client locale non sia talkz

5.1.3 Correttezza stilistica

Nell'estendere il protocollo ci siamo posti alcuni problemi di correttezza stilistica dell'estensione stessa.

Quando il callee contatta il caller sulla porta TCP che il demone remoto fornisce, il caller funge a tutti gli effetti da *server* e il callee da *client*[7]. Il caller infatti resta in attesa di una (e una sola) connessione da parte del callee, successivamente riceve i *MagicChars* e solo dopo averli ricevuti spedisce i propri.

Nel caso il protocollo prenda il ramo del grafo dell'estensione **talKZ** , lo scambio di dati continua. Ci siamo assicurati quindi che la metafora client server fosse ancora valida, facendo proporre dal client le capability da lui supportate per poi ricevere risposta dal server sulla modalità di prosecuzione solo dopo che esso ne ha verificato il supporto.

Quindi, ad esempio, è il client (callee) a chiedere la connessione SSL, ed è il server (caller) a accettarla o meno, a seconda del supporto (o eventualmente dopo altre verifiche, nel caso le capability fossero ulteriormente estese).

5.2 Vantaggi

I vantaggi offerti da **talKZ** rispetto alle comuni implementazioni ntalk BSD diffuse possono quindi essere riassunti in due punti:

logging Il client **talKZ** offre all'utente la possibilità di salvare su file l'intero contenuto informativo di una sessione talk.

Attualmente il formato nel quale vengono salvate le sessioni è simile al formato utilizzato da molti client *IRC* ovvero i messaggi vengono salvati ogni volta che un partecipante termina la riga corrente con un carattere di *newline* o di *carriage return*, ogni riga viene salvata con un prefisso che identifica univocamente l'utente, viene inoltre aggiunto un simbolo particolare per distinguere l'utente locale (per risolvere problemi di omonimia tra utenti).

È importante sottolineare che questa feature è assolutamente indipendente dal protocollo **talKZ** ed è utilizzabile con qualsiasi combinazione di versioni sia per quanto riguarda i demoni che per quanto riguarda l'altro client.

SSL Il client **talKZ** offre inoltre all'utente la possibilità di avere una comunicazione criptata secondo lo standard de-facto SSL ([4]) in modo che non possa essere letta da persone indesiderate che abbiamo in qualche modo

[7]È buffo pensare che il chiamato durante la fase iniziare del protocollo diventi parte attiva durante la connessione TCP, e il chiamante, attivo all'inizio, entri passivamente in attesa della connessione

visibilità dei pacchetti IP scambiati tra i due partecipanti una volta che la connessione TCP è andata a buon fine.

È importante notare che è criptata solamente la comunicazione TCP tra i due client lo scambio di messaggi tra il client e i server non è criptato e difficilmente criptabile dato che SSL si basa sull'assunzione che ci sia una comunicazione *connection oriented*. Non è quindi consigliabile utilizzare **talKZ** con supporto SSL in ambiti che abbino seri requisiti di sicurezza, il protocollo da noi implementato infatti garantisce solamente la privacy della comunicazione, non l'autenticazione delle parti.

Questa opzione per ovvie ragioni non è utilizzabile nel caso in cui uno solo dei due client, supporti il protocollo **talKZ** .

6 Implementazione

6.1 Strumenti utilizzati

Abbiamo usato il linguaggio ad alto livello ed ad oggetti *Python* versione 2.2 con un particolare occhio alla progettazione seguendo il più possibile il paradigma *Object Oriented* ed alcuni suoi *Pattern*.

Per l'interfaccia grafica abbiamo scelto di utilizzare la libreria *ncurses*, è possibile però scrivere *frontend* alternativi.

6.2 Moduli

Le varie entità del protocollo e dell'interfaccia sono divise in alcuni moduli:

protocol La gestione del protocollo lato client, sia per caller che per callee, è controllata dal modulo protocol.py che rappresenta, sotto forma di classi, gli stati dell'automa (fig. 2 pag. 11) e rende pubblica una classe che gestisce automaticamente tutto il protocollo per caller e per callee, restituendo a chi gestisce la console il socket TCP per potere scrivere al proprio peer.

logger Il logging è gestito dal modulo logger.py che mette a disposizione una facility Logger e permette di registrare o meno i caratteri speciali di cancellazione concordati dal protocollo. È da notare che nel caso si scelga di loggare i caratteri speciali, nel file di log rimarranno anche i dati cancellati seguiti dal caratere speciale spedito, ma usando ad esempio il comando `cat` i dati cancellati durante la conversazione non compariranno, dato che la console unix interpreterà i caratteri come effettivamente dati in input, senza stamparli.

ui ui.py ncurses.py sono l'astrazione di una user interface e una implementazione usando le ncurses. È infatti possibile, usando altre librerie per disegnare *UI*, scrivere altri frontend semplicemente facendoli ereditare e usando la classe ui.UserInterface come interfaccia.

messages Il layer di astrazione dei pacchetti del protocollo è gestito dal modulo messages.py, che fornisce anche una *factory* come facility per il packing e unpacking dei pacchetti letti dai socket.

kzstruct Il layer di serializzazione è gestito da kzstruct.py e fa da ponte tra gli oggetti Request e Response python e la sua rappresentazione binaria spedita durante il protocollo.

peer peer.py è necessario per astrarre l'oggetto socket e gestire trasparentemente l'uso di SSL.

talkz talkz.py è il client che gestisce il protocollo e l'interfaccia utente da linea di comando

talkzd talkzd.py è il demone *ntalk* usabile traminte *inetd* o *standalone*, la sua funzione è quella di fare *storing* degli inviti e delle informazioni su come contattare il caller, nel caso sia contattato dal callee, o di ricerca dell'utente chiamato nel caso sia il demone remoto contattato dal caller. Il demone utilizza *syslog* come metodo di logging stantard, ma i messaggi di log possono essere scritti anche su *standard output*

misc altri moduli utilizzati dai precedenti sono debbugger.py per la gestione dei messaggi di debug, invites.py per astrarre le invitelist per il demone, announce.py per la ricerca dell utente chiamato ed eventualmente l'annuncio che il *caller* lo sta cercando.

6.3 Pattern

In fase di progettazione abbiamo scelto di utilizzare alcuni *design pattern*. In particolare:

Factory astrae il processo di fabbricazione dei pacchetti Request e Response per protocollo *ntalk*, gestendone la serializzazione e deserializzazione in modo semplice e veloce.

State astrae l'*automa* a stati finiti fig. 2 pag. 11 e rende visibile una classe capace di cambiare stato internamente a seconda del *path* all'interno del grafo dell'automa.

6.4 Code Metrics

I Moduli per il progetto sono 13 in totale.

Le Classi totali sono 54, concentrate per la maggior parte nei moduli protocol.py (23) e messages.py (13).

Le Linee di Codice sono comlessivamente 3517, concentrate in particolare in messages.py (880), protocol.py (770) e ncurses.py (457)

6.5 Problemi risolti e irrisolti

Il problema che ha occupato buona parte del tempo di progettazione è stato l'analisi del protocollo e la conseguente analisi delle sue estensioni garantendo la compatibilità con le versioni precedenti del protocollo, soddisfacentemente risolto.

Altri problemi rimasti insoluti sono però, per quanto riguarda il protocollo stesso, il passaggio in chiaro dei *MagicChars*, ovvero i caratteri speciali spediti per definire la cancellazione dei dati. Per garantire la privacy degli utenti, ci sarebbe piaciuto rendere sicura la transazione dei dati coprendola dal layer SSL.

A causa di un problema prettamente tecnico, di implementazione dei socket SSL di Python, non è stato possibile completare l'implementazione dell'estensione SSL, quindi anche se client e server concordano il protocollo esteso con capability SSL, in realtà non viene crittato nulla e i due client terminano riportando un messaggio di errore del layer SSL. Tutto il codice è comunque predisposto per funzionare, non appena il problema in Python verrà risolto. Sfortunatamente infetti gli oggetti SocketSSL Python non permettono di fare funzionare SSL in modalità server, rendendo impossibile la *listen/accept* sul socket SSL.

Abbiamo contattattato l'autore di una patch che risolverebbe il problema, siamo in attesa di una risposta.

7 Utilizzo di talKZ

Il pacchetto **talKZ** contiene due eseguibili fondamentali: `talkz.py` e `talkzd.py` che implementano il client ed il server **talKZ** rispettivamente.

Il server talkz può essere utilizzato sia in *standalone* mode che attraverso l'*internet super server* `inetd` ([5]), quest'ultimo modo di utilizzo è consigliato in quanto la modalità standalone è pensata principalmente per motivi di debugging.

Per utilizzare il demone talkz sul proprio host è sufficiente aggiungere una riga di questo tipo al file di configurazione `/etc/inetd.conf`:

```
ntalk dgram udp wait nobody.tty /usr/sbin/talkzd.py talkzd
```

L'utilizzo del client talkz è sostanzialmente analogo all'utilizzo dei comuni client talk, con l'aggiunta di alcune opzioni:

```
Usage:
     talkz.py [options] user[@host] [tty]
Options:
     -s | --ssl                    enable SSL
     -l <file> | --log=<file>      enable logging to file <file>
```

```
-h | --help                        show this help message
```

Le opzioni `-s` e `--ssl` indicano al client di richiedere che la comunicazione sia criptata con SSL, questa opzione sarà comunque attivata solamente se anche il partecipante remoto effettuerà la stessa richiesta.

Le opzioni `-l` e `--log` indicano invece al client di salvare il log della comunicazione sul file passato come argomento.

Le opzioni `-h` e `--help` mostrano all'utente il messaggio di help riportato qui sopra.

Per maggiori informazioni sulle opzioni offerte da **talKZ** si veda sez. 5.2 a pag. 13.

Il resto dei parametri passati a riga di comando sono analoghi a quelli delle usuali implementazioni talk, il primo può essere un semplice nome utente o una stringa del tipo *nomeutente@host* indicante l'utente con il quale si vuole iniziare una sessione talk. Nel caso in cui il nome dell'host non sia specificato viene assunto che l'utente con il quale si vuole parlare sia sulla macchina locale (*localhost*).

L'ultimo parametro permette di specificare una *tty* sulla quale il demone remoto tenterà di notificare la richiesta di inizio di una sessione talk, nel caso in cui questo parametro sia assente la notifica viene spedita a tutte le tty alle quali l'utente è connesso.

Riferimenti bibliografici

[1] IANA, Protocol Numbers and Assignment Services, `http://www.iana.org/assignments/port-numbers`

[2] RFC 1700, ASSIGNED NUMBERS, `http://www.rfc-editor.org/rfc/rfc1700.txt`

[3] Erich Gamma, Richard Helm, Ralph Jhonson, Jhon Vlissides - *Design Patterns* - Addison-Wesley

[4] Internet Draft - *The SSL Protocol Version 3.0*, Alan O. Freier, Philip Karlton, Paul C. Kocher - November 18, 1996 - `http://wp.netscape.com/eng/ssl3/draft302.txt`

[5] W. Richard Stevens - *UNIX Network Programming, Volume 1, Second Edition: Networking APIs: Sockets and XTI, Chapter 12.Daemon Processes and inetd Superserver* - Prentice Hall, 1998, ISBN 0-13-490012-X