



Dario Maggiorini

Introduzione alla

PROGRAMMAZIONE CLIENT - SERVER



Introduzione alla programmazione client-server

Dario Maggiorini

Introduzione alla programmazione client-server



© 2009 Pearson Paravia Bruno Mondadori S.p.A.

Le informazioni contenute in questo libro sono state verificate e documentate con la massima cura possibile. Nessuna responsabilità derivante dal loro utilizzo potrà venire imputata agli Autori, a Pearson Paravia Bruno Mondadori S.p.A. o a ogni persona e società coinvolta nella creazione, produzione e distribuzione di questo libro.

Per i passi antologici, per le citazioni, per le riproduzioni grafiche, cartografiche e fotografiche appartenenti all'approprietà di terzi, inseriti in quest'opera, l'editore è a disposizione degli aventi diritto non potuti reperire nonché per eventuali non volute omissioni e/o errori di attribuzione nei riferimenti.

I diritti di riproduzione e di memorizzazione elettronica totale e parziale con qualsiasi mezzo, compresi i microfilm e le copie fotostatiche, sono riservati per tutti i paesi.

LA FOTOCOPIATURA DEI LIBRI È UN REATO Le fotocopie per uso personale del lettore possono essere effettuate nei limiti del 15% di ciascun volume dietro pagamento alla SIAE del compenso previsto dall'art. 68, commi 4 e 5, della legge 22 aprile 1941 n. 633.

Le riproduzioni effettuate per finalità di carattere professionale, economico o commerciale o comunque per uso diverso da quello personale possono essere effettuate a seguito di specifica autorizzazione rilasciata da AIDRO, corso di Porta Romana n. 108, 20122 Milano, e-mail segreteria@aidro.org e sito web <http://www.aidro.org/>.

Copy-editing: Federica Sonzogno

Grafica di copertina: Nicolò Cannizzaro

Tutti i marchi citati nel testo sono di proprietà dei loro detentori.

978-88-7192-724-4

*A tutte le donne che lavorano nel campo dell'informatica,
perché non vengano più considerate "accettabili"
soltanto se più brave di qualunque uomo.*

Introduzione

Con l'esponenziale crescita di Internet, utenti e aspiranti programmatore si trovano di fronte alla possibilità di interagire con un numero sempre crescente di servizi e di accedere a quantità di informazioni senza precedenti. Oggi, scrivere programmi che utilizzano Internet non vuole più dire produrre software costituito da due componenti che fanno uso della rete come mezzo di trasporto per i dati; significa, piuttosto, creare programmi in grado di collaborare tra loro e di dialogare con architetture distribuite governate da una serie di standard. Questo libro vuole essere un accompagnamento per chi desidera imparare i rudimenti della programmazione di rete secondo un nuovo approccio. Il filo conduttore non è più costituito dal linguaggio e dalle operazioni usate per accedere alla trasmissione dati, bensí dai servizi e dalle modalità e strategie con le quali si intende usufruirne.

Il presente libro é diviso in quattro parti. Nella prima, introduttiva, vengono gettate le basi per la strutturazione di protocolli e servizi. Nella seconda parte vengono esplorate modalità e strategie con cui è possibile realizzare alcuni elementi software che interagiscono tra loro in rete, per poter usufruire dei più importanti servizi disponibili su Internet. Le

parti successive ripercorrono le stesse tappe della seconda proponendo due linguaggi diversi, Java e C, ma discutendo il codice necessario per ottenere il risultato voluto.

In particolare, la seconda parte e le successive si prestano a due modalità di lettura: potremmo definirle “verticale” e “orizzontale”. Nella modalità verticale i capitoli possono essere affrontati in maniera sequenziale, come in qualunque libro, permettendo al lettore di esaminare prima gli aspetti teorici e poi quelli pratici, eliminando al più un linguaggio di programmazione. Nella lettura orizzontale, invece, il lettore è invitato a leggere capitoli e paragrafi portando avanti parallelamente due o più sezioni; in tal modo, per ogni argomento affrontato verrà discussa sia la parte teorica che quella pratica e sarà possibile concentrarsi fin da subito sugli aspetti tecnici eliminando gli argomenti non di interesse senza sacrificare aspetti implementativi.

All'interno del testo si troveranno una serie di elementi evidenziati che hanno la funzione di identificare informazioni che possono dare spunto a riflessioni e/o approfondimenti.

Esercizio



Viene proposto un esercizio attinente all'argomento appena trattato.

 Questa informazione mette in evidenza un concetto utile per non cadere in errore o vuole chiarire le idee al lettore su un argomento spesso frainteso.

 Questa informazione è un dettaglio aggiuntivo rispetto al testo.

Propone un approfondimento o aiuta a contestualizzare l'argomento.

Il codice portato ad esempio verrà evidenziato tramite riquadri tratteggiati, all'interno dei quali potrebbero comparire definizioni di funzioni o classi,

```
1 #include <stdio.h>
2
3 main() {
4     printf("Hello, world!");
5 }
```

oppure veri e propri frammenti di programmi da analizzare; in tal caso, per facilitare la discussione, a sinistra verranno riportati anche i numeri di riga, che non fanno parte del programma.

```
1 #include <stdio.h>
2
3 main() {
4     printf("Hello, world!");
5 }
```

I

Protocolli e servizi

Capitolo 1

Modello client-server

Alla base di tutti i servizi fruibili tramite la rete troviamo un paradigma di accesso fondato su un meccanismo di *domanda e risposta*. Una entità in rete pone una domanda (ad esempio, “che ora è ?”) a cui qualcuno si occuperà di dare una risposta (del tipo, “sono le 10 e 15”). Ottenere la risposta equivale a fruire del servizio richiesto. Tale approccio viene formalizzato tramite un’architettura che prende il nome di *architettura client-server*.

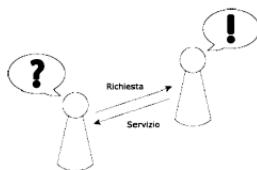


Figura 1.1 Client e server.

L'architettura client-server presuppone l'esistenza di due entità: una che offre un servizio e una che lo richiede. Queste due entità vengono normalmente identificate rispettivamente con i nomi *server* e *client* (Figura 1.1).

Come si può notare, non si è fatta nessuna ipotesi sulla natura degli attori in gioco, in quanto questo modello può non essere applicato solo a situazioni di natura informatica.

Se si considera la definizione molto generale appena data, sono sistemi client-server anche situazioni facenti parte della vita reale quali:

1. un risparmiatore e uno sportello bancomat
2. la segretaria di un'azienda e gli impiegati che ci lavorano
3. un correntista e la sua banca
4. lo sportello bancomat al primo punto e un istituto di credito.

Esercizio 1.1



Si prenda la lista appena riportata e, per ogni esempio, si identifichino:

1. l'entità server
2. l'entità client

3. il servizio erogato.

Nel primo caso ci si trova di fronte a un server che gestisce *un solo client alla volta* e lascia gli altri *in attesa*. Si parla di *server iterativo* (Figura 1.2); si tratta dello schema più semplice da realizzare.

Nel secondo caso, invece, un solo server si occupa di fornire un servizio *contemporaneamente* a più client. In questo caso si parla di *server concorrente* (Figura 1.3). Più avanti si vedrà che l'analogia è molto aderente anche alla realtà riscontrabile durante la stesura del codice nel senso che, sebbene il server rimanga *in contatto* con tutti i client *contemporaneamente*, lo scambio di dati con client diversi non avviene mai in maniera realmente parallela.

Nel terzo caso si assiste a un'ulteriore complicazione: il client richiede il servizio a un'entità, in questo caso la banca, che non si occupa in maniera diretta di erogarlo, ma si propone come *punto di riferimento* e successivamente fornisce un agente (l'operatore di sportello) che andrà a interfacciarsi con il cliente. Si parla in questo caso di *server multiplo* o *server multiprocesso* (Figura 1.4).

L'ultimo caso proposto serve a esemplificare la composizione di servizi, o meglio il concatenamento di sistemi client-server. Di per sé, la coppia utente/bancomat è un sistema client-server per il servizio *erogazione denaro*, mentre la coppia bancomat/

istituto di credito è un secondo sistema per il servizio *verifica disponibilità*.

A questo punto è lecito porsi la domanda se il bancomat sia un client o un server.

La risposta ovviamente è *tutti e due contemporaneamente*: occorre infatti sempre ricordare che le diciture di client e server sono una classificazione fatta da chi osserva il sistema per poter assegnare un ruolo alle varie entità in gioco. Questo non significa porre limitazioni alle loro funzionalità.

Non esistono operazioni che possono essere compiute solo da client o solo da server, come pure non esistono entità che possono operare in uno solo dei due ruoli; un'entità viene sempre classificata client o server in relazione a un'altra con la quale esiste un rapporto per l'erogazione di un servizio.



Inoltre, in ognuna delle situazioni presentate si osservano comportamenti diversi del server, generalmente dettati dalla cardinalità dei client e dalla natura del servizio erogato. Il comportamento di un server al fine di erogare il servizio viene anche detto *modello di servizio*. I modelli di servizio che verranno analizzati, largamente utilizzati nel campo della programmazione di rete, si suddividono, come già visto, in *iterativo, concorrente e multiprocesso*.

1.1 Modello di servizio iterativo

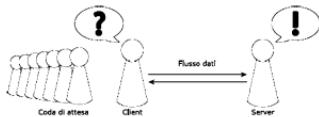


Figura 1.2 Server iterativo.

In questa situazione un server si occupa di fornire servizio a un singolo client, dedicandovisi completamente (Figura 1.2); si tratta del modello di servizio più semplice.

Come nel caso reale, quando un client fa richiesta di usufruire del servizio ci si può trovare di fronte a due casi:

1. il server è libero e il client può *entrare in servizio* immediatamente
2. il server è occupato ed è necessario aspettare.

Si assiste quindi alla formazione di una coda d'attesa, che, come tutte le risorse all'interno di un calcolatore, sarà limitata; possiamo perciò pensare che la lista appena vista possa essere ulteriormente specializzata nel seguente schema:

1. il server è libero e il client può *entrare in servizio* immediatamente
2. il server è occupato, quindi:
 - (a) c'è spazio in coda e il client può mettersi in attesa
 - (b) la coda è piena e il servizio viene rifiutato.

La gestione dell'eventuale coda risulta interamente a carico del sistema operativo; l'applicazione che implementa il server è completamente ignara del fatto che ci siano alcuni client in coda o se ce n'è qualcuno a cui in passato è stato rifiutato il servizio.

Il modello di servizio appena esposto risulta inoltre essere sensibile a *starvation*, situazione cioè in cui uno o più client potrebbero rimanere in attesa di servizio per un tempo virtualmente infinito. Per evitare questa condizione è consigliabile utilizzare un server iterativo solo nel caso in cui il tempo medio di servizio richiesto da un client sia piuttosto breve, o comunque predicibilmente limitato.

La starvation può essere il risultato di una cattiva organizzazione del programma o del protocollo, ma può anche essere creata in maniera maliziosa da un client intenzionato a bloccare la fornitura del servizio. Questi sono attacchi alla sicurezza di un sistema informatico prendono il nome di *Denial of Service* (DOS).

Esercizio 1.2

Si consideri la famiglia dei servizi erogabili tramite il modello di servizio iterativo. Si caratterizzino, facendo un esempio per ogni categoria, i servizi erogati nella vita reale che:

1. non è conveniente gestire in questo modo
2. non è possibile gestire in questo modo.

1.2 Modello di servizio concorrente

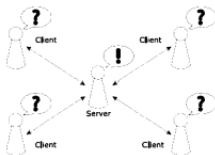


Figura 1.3 Server concorrente.

Questo modello di servizio è già stato caratterizzato come una situazione in cui un server offre servizio parallelamente a più client (Figura 1.3). È importante ricordare che la concorrenza dell'erogazione è un parallelismo nel mantenere i rapporti, non nell'inviare o ricevere dati. Il server è in grado di prestare attenzione a un insieme di client e di essere interpellato da uno qualsiasi di questi; tuttavia, durante l'invio e la ricezione di dati

a/da uno specifico client, il server non potrà essere interrotto con altre richieste di servizio.

Quando si utilizza un modello concorrente non si è più in presenza di una coda di attesa, in quanto tutti i client che fanno richiesta di servizio vengono ammessi immediatamente al sistema. Ciononostante, esiste comunque una limitazione al numero massimo dei client in servizio dettata dalla risorse disponibili; ogni sistema operativo, infatti, pone alcune limitazioni per quanto riguarda il numero massimo di canali di comunicazione assegnati a un dato processo.

Dal punto di vista implementativo occorre creare un programma in grado di accentrare su di sé tutti i canali di comunicazione aperti con i client; tale programma sarà responsabile di ricevere e gestire qualunque messaggio di richiesta di servizio indipendentemente da quale sia la sorgente. Sotto il profilo logico, invece, un sistema concorrente presenta la difficoltà di dover tenere traccia di tutti i contesti delle diverse connessioni. Quando viene ricevuto un messaggio da un client, questo non è quasi mai un'informazione isolata, ma dipende da cosa *si era detto prima*, ovvero dal contesto che si era venuto a creare durante la comunicazione. Si pensi per esempio a un server per il gioco degli scacchi. Questo server segue in parallelo più partite: riceve messaggi con le mosse dei giocatori e risponde con le proprie. Per ogni giocatore il server dovrà memorizzare una scacchiera virtuale la cui configurazione sarà il risultato di tutte le mosse precedenti: per essere valida, la mossa di un giocatore dovrà trovare riscontro sulla relativa scacchiera.

L'onere di memorizzare e associare le scacchiere (i contesti) ai giocatori (i client) è a carico del server.

Il modello di servizio concorrente è il più complicato per quanto riguarda l'implementazione.



Molto spesso, server implementati secondo questa filosofia hanno limitazioni di *scalabilità verso l'alto*, cioè il server non è in grado di accomodare un numero sempre crescente di client senza peggiorare la qualità del servizio offerto. Al crescere del numero di client, il tempo di risposta osservato da ognuno di essi cresce in maniera esponenziale; inoltre, la richiesta di risorse potrebbe arrivare a eccedere quanto reso disponibile dal sistema operativo portando al blocco totale del sistema. In questo caso deve essere cura del server rifiutare il servizio.

Esercizio 1.3



Si consideri la famiglia dei servizi erogabili tramite il modello di servizio concorrente e se ne diano cinque esempi presi dal mondo reale.

1.3 Modello di servizio multiprocesso/multithread

Questo modello di servizio corrisponde a una situazione in cui più server entrano in gioco contemporaneamente (Figura 1.4).

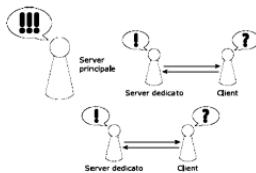


Figura 1.4 Server multiprocesso.

Nel mondo reale ciò implica il fatto di avere entità distinte e contemporaneamente attive, dal punto di vista della programmazione equivale ad avere più flussi esecutivi paralleli per la fornitura del servizio. Storicamente un flusso esecutivo all'interno di un sistema operativo prende il nome di *processo* (da qui proviene il nome del modello); a volte si usa il termine

equivalente *multithread* quando si fa uso di linguaggi (come ad esempio Java) che gestiscono i flussi esecutivi con thread anziché processi.

La differenza tra processi e thread sta nel fatto che questi ultimi condividono lo spazio di indirizzamento e, quindi, delle aree in cui vengono memorizzati i dati (le variabili). Oltre alle problematiche di gestione dei thread occorre gestire anche la sincronizzazione per l'accesso ai dati, diversamente le informazioni comuni all'interno del server potrebbero diventare inconsistenti.



Come già detto, in questo caso un client richiede il servizio a una entità astratta che provvederà a rendere disponibile un flusso esecutivo dedicato all'erogazione del servizio; la sequenza delle operazioni compiute è visibile nella Figura 1.5. Un nuovo client si presenta (a) e fa richiesta di fruizione di servizio (b); il server principale crea un nuovo server dedicato per il client (c) e la fruizione del servizio tra questi due avviene poi come nel caso di un server iterativo (d).

Occorre ricordare, però, che anche i processi/thread, in quanto ospitati dal sistema operativo, sono disponibili in quantità limitata. Dal punto di vista implementativo un server multiprocesso può risentire di problemi di scalabilità simili a quelli visti per il server concorrente (non è possibile creare un

nuovo processo a fronte di un nuovo client in arrivo). Per poter ovviare a questa limitazione, le implementazioni di server che vogliono soddisfare un numero molto grande di utenti adottano una tecnica mista. Il server principale dispone già di un insieme di processi *dormienti* e pronti all'uso: in questo caso, non sarà creato un server dedicato ma semplicemente *assegnato* uno di quelli esistenti. Essendo i server dedicati in numero limitato, i client potrebbero essere in soprannumero rispetto a questi ultimi, quindi si osserverà la formazione di gruppi di client attorno a ogni server, il quale può decidere di adottare lo calmamente una politica di tipo iterativo o concorrente. Compito del server principale diventa anche quello di dirigere i client in arrivo verso il server *meno affollato* in quel momento, in modo tale da massimizzare le prestazioni generali del sistema. Apache [1], un web server open source largamente utilizzato su piattaforme UNIX, adotta questo tipo di tecnica.

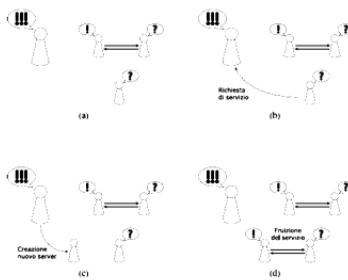


Figura 1.5 Lo schema di fruizione del servizio nel caso di un server multiprocesso.

Esercizio 1.4



Si consideri la famiglia dei servizi erogabili tramite il modello di servizio multiprocesso e se ne diano cinque esempi presi dal mondo reale.

Capitolo 2

Protocolli

Lo scambio dei dati all'interno di una rete avviene facendo uso di strutture sintattiche chiamate protocolli.

Consultando un dizionario si può trovare la definizione più classica di protocollo.

Protocollo:

1. *l'insieme delle norme che regolano lo svolgimento di manifestazioni, visite, ricevimenti ufficiali;*
2. *insieme di regole che governano la successione e lo scambio di informazioni fra due dispositivi comunicanti tra loro.*

Dizionario Garzanti.

Protocol:

1. *an original draft, minute, or record of a document or transaction;*
2. *a code prescribing strict adherence to correct etiquette and precedence (as in diplomatic exchange and in the military services);*
3. *a set of conventions governing the treatment and especially the formatting of data in an electronic communications system;*
4. *a detailed plan of a scientific or medical experiment, treatment, or procedure.*

Merriam-Webster dictionary.

Generalizzando la seconda definizione del Dizionario Garzanti, è possibile considerare come protocolli:

1. il codice morse
2. la lingua italiana
3. il dialetto milanese
4. le regole di buon comportamento
5. il formulario per la dichiarazione dei redditi.

Si può notare che non tutti questi esempi prevedono uno scambio di messaggi bidirezionale: il primo infatti non lo è, e il quinto non dovrebbe esserlo (si spera). Protocolli diversi possono operare a livelli distinti: il secondo e il terzo vincolano sulle strutture da usare mentre il quarto vincola su cosa è opportuno dire e cosa no. Infine, protocolli diversi potrebbero

usare messaggi simili, se non addirittura gli stessi, organizzati secondo regole diverse: è il caso del secondo e del terzo esempio.

Quando si progetta un sistema client-server, è necessario definire un protocollo per far *dialogare* due applicazioni, mentre il sistema operativo del calcolatore e l'architettura di rete si occuperanno *solo* di fornire la struttura per lo scambio dei messaggi.

2.1 Relazione tra protocolli e reti

Lo scopo di una rete di comunicazione è in sintesi quello di portare una informazione dal punto *A* al punto *B* di una maglia di connessioni. Per farlo ci sono una serie di elementi che devono dialogare tra loro: ogni architettura di rete definisce quindi una serie di protocolli di comunicazione necessari al suo funzionamento.

L'idea molto diffusa secondo la quale “*i protocolli fanno uso della rete*” non potrebbe essere più sbagliata; viceversa, sono le reti che basano la loro struttura sul concetto di protocollo. 

Si può implementare una rete telematica solo se prima si stabiliscono dei protocolli di comunicazione.

2.1.1 Il modello ISO-OSI

Dal punto di vista funzionale le reti sono standardizzate dalla *International Standard Organization* (ISO) [2] tramite il modello di riferimento *Open System Interconnection* (OSI) [3], che viene generalmente indicato con la sigla ISO-OSI.

Esiste una distinzione molto precisa tra modello e architettura.



Un modello descrive un sistema tramite una serie di funzionalità che questo assolve, un'architettura è la realizzazione pratica di un modello. Quello ISO-OSI è un modello di riferimento che definisce i livelli funzionali per una rete e ne stabilisce le relazioni interne, mentre TCP/IP [4] è un'architettura di rete che implementa il modello ISO-OSI facendo uso di un certo insieme di protocolli.

Il modello ISO-OSI suddivide le funzioni logiche di una rete in sette strati (livelli) come riportato nella Figura 2.1. Internet è una rete costituita dall'unione e cooperazione di una serie di architetture che aderiscono al modello ISO-OSI.

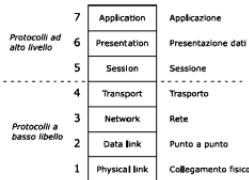


Figura 2.1 Il modello di riferimento ISO-OSI.

Brevemente, le funzioni definite nei vari livelli del modello ISO-OSI sono le seguenti.

Collegamento fisico

Utilizza una connessione fisica diretta per ottenere l'effettivo trasferimento dei dati.

Data link

Trasferisce dati tra entità fisicamente connesse tra loro e in modo indipendente dal mezzo fisico. È l'unico che, per puri motivi di abitudine, si preferisce chiamare con il nome non tradotto.

Rete

Permette lo scambio di dati tra entità anche non sono fisicamente connesse tra loro.

Trasporto

Permette di identificare entità distinte all'interno dello stesso nodo e permette a queste entità di collegarsi con altre su nodi remoti.

Sessione

Gestisce sessioni logiche tra applicazioni.

Presentazione

Rende i dati compatibili tra le varie piattaforme adottando una codifica indipendente dal calcolatore utilizzato.

Applicazione

Si compone delle applicazioni utente e interagisce con un essere umano.

All'interno del modello ISO-OSI ogni livello comunica unicamente con i due adiacenti: quello immediatamente superiore e quello inferiore. Fanno eccezione i livelli di collegamento fisico e applicazione, che andranno a interagire rispettivamente con l'hardware di rete e l'utente dell'applicazione.

Quando due nodi sulla rete *parlano tra di loro*, il software che implementa ognuno dei livelli osserva i messaggi generati dalla controparte dello stesso livello sull'altro nodo; quindi è come se, virtualmente, due livelli su nodi di rete distinti parlassero direttamente tra loro. Questo significa che ogni livello ha

bisogno di definire due cose: un suo specifico protocollo di comunicazione, nonché un suo sistema di indirizzamento per poter identificare la controparte con cui comunicare. Un esempio di comunicazione tra livelli diversi si può vedere nella Figura 2.2, i cui elementi saranno chiariti nel prossimo paragrafo.

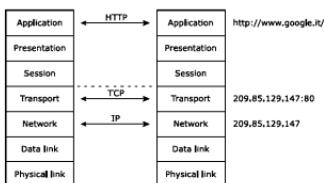


Figura 2.2 Esempio di comunicazione tra livelli all'interno del modello ISO-OSI.

2.2 Internet e i suoi protocolli

Nell'uso comune, il termine *internet* fa riferimento a una rete che impiega specifici protocolli per implementare i livelli funzionali di rete e di trasporto; nella fattispecie *Internet Protocol* (IP) [5, 6] per il livello di rete e *Transmission Control Protocol* (TCP) [7, 8, 9] e *User Datagram Protocol* (UDP) [10] per il livello di trasporto. Si parla spesso, come già detto, di architettura TCP/IP.

Quindi, TCP/IP non è un protocollo ma solo un'espressione molto abusata per indicare un certo tipo di reti o una famiglia di protocoli.



È bene fare attenzione alla sottile distinzione tra i termini “*internet*” e “*Internet*” (con l'iniziale maiuscola). Il primo si riferisce all'architettura di rete appena descritta, il secondo invece alla rete globale che tutti noi utilizziamo.



Nonostante il fatto che l'architettura TCP/IP sia basata sul modello ISO-OSI, questa non ne implementa tutti i livelli, come si può vedere nella Figura 2.3. Inoltre, per gli argomenti trattati in questo testo, sono di nostro interesse solo i livelli dal terzo al settimo.

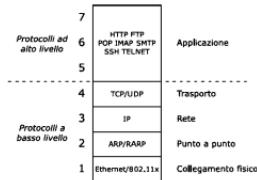


Figura 2.3 L'architettura TCP/IP con esempi dei protocolli implementati.

Come si può facilmente notare, *HyperText Transfer Protocol* [11] è un protocollo usato per far dialogare un'applicazione (tipicamente un browser) con un web server, ed è quindi un protocollo a livello applicazione.



Questo vuol dire che HTTP non ha tecnicamente nulla a che fare con la struttura interna di Internet, anche se è diventato luogo comune pensare che la rete sia funzionale solo e unicamente per la fruizione di pagine web.

2.2.1 Livello di rete

Il livello di rete caratterizza Internet sia a livello locale che geografico. In questo livello i dati provenienti dagli strati più alti della pila ISO-OSI vengono spediti in maniera indipendente

e raggiungono la destinazione senza coordinazione tra loro, facendo a volte percorsi diversi. È bene ricordare che su una rete di questo tipo la consegna delle informazioni non viene garantita nessun modo: i pacchetti potrebbero andare persi a causa della congestione degli apparati lungo il percorso; inoltre, pacchetti che compiono percorsi distinti sono soggetti a ritardi diversi e possono pertanto essere consegnati in un ordine differente da quello di invio.

Il protocollo principe per il livello di rete è l'IP, sviluppato da Jon Postel per conto della *Defense Advanced Research Projects Agency* (DARPA) [12]. Si tratta di un protocollo in cui non vi è una connessione permanente tra le due entità che dialogano; i protocolli con questo tipo di comportamento vengono anche definiti *senza connessione*. Inoltre, non sono implementati meccanismi per la ritrasmissione dei pacchetti persi; in questo caso si parla anche di *politica best effort*.

Il protocollo IP è stato sviluppato all'interno del  progetto *Advanced Research Projects Agency Network* (ARPANET). Nonostante sia opinione diffusa che ARPANET sia stato un progetto militare, in realtà si trattava di un progetto scientifico sovvenzionato dai militari: DARPA era interessata a fare sperimentazione su tecniche di commutazione di pacchetto. Quindi, dire che ARPANET

(e quindi anche Internet, sua diretta discendente) sia nata appositamente per scopi bellici è decisamente una forzatura.

Esistono due versioni del protocollo IP: 4(IPv4) [5] e 6(IPv6) [6]. Rispetto al primo il secondo è stato dotato di una capacità di indirizzamento più ampia per poter far fronte alla continua richiesta da parte di aziende e istituzioni; inoltre, è in grado di gestire facilmente utenti mobili ed è stato progettato per semplificare il più possibile le attività di gestione dell'infrastruttura di rete. Disgraziatamente, la disponibilità di IPv6 si limita a zone limitate di Internet a causa del fatto che apparecchiature di rete non sufficientemente recenti ne scarterebbero sistematicamente i pacchetti in quanto non riconosciuti. Per questo motivo all'interno del testo IPv6 non verrà trattato e in generale con IP si intenderà sempre IPv4.

Il protocollo IP ha la responsabilità di fornire un sistema di indirizzamento globale per i nodi della rete (indirizzi IP). Gli indirizzi IP sono numeri di 32 bit, ma vengono normalmente espressi con una notazione detta *decimale puntata* in cui l'indirizzo viene suddiviso in quattro gruppi da 8 bit, anche chiamati ottetti. Gli ottetti vengono scritti in sequenza intervallati da punti, da cui il nome della notazione. Questo vuol dire che il nodo di rete globalmente identificato dal numero 1436259743 sarà indicato con la dicitura

“159.149.155.85”; molto più gestibile da parte di un essere umano.

Gli indirizzi IP sono gestiti e assegnati ai vari soggetti che ne fanno richiesta da un’apposita organizzazione internazionale: la *Internet Assigned Numbers Authority* (IANA) [13].

È importante ricordare, inoltre, che non tutti gli indirizzi IP sono uguali tra loro; si differenziano infatti tra *pubblici* e *privati*. Gli indirizzi pubblici sono quelli utilizzati dai nodi connessi a Internet e assegnati dalla IANA; quelli privati, invece, vengono utilizzati senza bisogno di alcuna richiesta per la rete interna di una organizzazione. Gli indirizzi privati non possono essere utilizzati per specificare la destinazione di un pacchetto IP su Internet in quanto gli apparati di rete che la compongono sono configurati per scartare questo tipo di pacchetti; quindi, un nodo configurato con un indirizzo privato può essere raggiunto solo dall’interno della sua stessa rete a meno di non utilizzare tecniche quali il *Network Address Translation* (NAT) [14]. Gli intervalli destinati dalla IANA ad accogliere indirizzi privati sono riportati nelle righe della Tabella 2.1.

Indirizzo iniziale	Indirizzo finale
10.0.0.0	10.255.255.255
127.0.0.0	127.255.255.255
169.254.0.0	169.254.255.255
172.16.0.0	172.31.255.255
192.168.0.0	192.168.255.255

Tabella 2.1 Intervalli di indirizzi IP destinati dalla IA NA all’uso privato.



L'ultima riga della Tabella 2.1 chiarisce per quale motivo la stragrande maggioranza dei router A DSL viene pre-configure con un indirizzo del tipo 192.168.x.x per quanto riguarda la scheda di rete interna. Inoltre, spiega come mai avere un router A DSL è più sicuro che non avere un modem: i calcolatori all'interno della rete privata non sono raggiungibili dagli attacchi di sicurezza originati a partire dalla rete pubblica.

In particolare, tra gli indirizzi privati ne esiste uno con un significato speciale: 127.0.0.1, spesso indicato anche come *indirizzo di loopback*. Esso, come il nome suggerisce, può essere utilizzato da qualunque nodo per indicare *se stesso*, indipendentemente dall'indirizzo assegnato alla propria scheda di rete.

L'indirizzamento di Internet è strutturato in maniera gerarchica: nodi collegati allo stesso segmento di rete condividono una porzione del loro indirizzo IP (la parte a sinistra); in tal modo è possibile istituire strategie globali di instradamento per i pacchetti (*routing*) facendo uso di algoritmi ricorsivi. Per raggiungere il nodo con indirizzo 159.149.155.85 basterà prima raggiungere il segmento di rete che raggruppa tutti i nodi il cui indirizzo inizia con 159, successivamente identificare il sottoinsieme con i nodi con indirizzo 159.149 e così via.

Nonostante si senta parlare di altri protocolli, quali ad esempio *Internetwork Packet Exchange* (IPX) [15], *Datagram Delivery Protocol* (DDP) [16] o *Internet Datagram Protocol* (IDP) [17], Internet risulta essere basata esclusivamente su IP.

2.2.2 Livello di trasporto

All'interno del livello di trasporto possiamo trovare una serie di protocolli che hanno lo scopo di estendere il servizio fornito da IP. Al quarto livello della pila ISO-OSI Internet implementa tipicamente i protocolli TCP e UDP (perciò, sotto questo punto di vista, il termine TCP/IP, oltre che abusato, risulta anche incompleto).

L'estensione specifica del servizio offerto da IP dipende dal particolare protocollo che si intende analizzare. UDP non offre miglioramenti rispetto a IP, a parte il fatto di essere direttamente raggiungibile dalle applicazioni. TCP, invece, simula un sistema di trasporto orientato alla connessione, affidabile e con consegna ordinata delle informazioni; in questo modo diventa possibile fornire servizi con garanzie sulla consegna dei dati anche se lo strato sottostante non è affidabile e fa uso del protocollo IP. Inoltre, TCP implementa politiche di *controllo di flusso* e di *controllo di congestione*, grazie alle quali viene mantenuto sotto controllo il livello di carico della rete e viene garantita un'equa suddivisione della capacità trasmissiva. Le considerazioni appena fatte non implicano però che UDP sia un protocollo *inutile*. Infatti, la mancanza di controllo di flusso,

di affidabilità e della necessità di stabilire una connessione (operazioni che tipicamente aggiungono ritardo) lo rendono molto utile per i seguenti tre casi: la diffusione di contenuti multimediali in streaming (RTP [18], RTSP [19]); l'accesso a servizi in cui il tempo di fruizione è solitamente inferiore al tempo necessario per stabilire la connessione (DNS [20]) e la condivisione di dischi via rete (NFS [21]).

Uno dei compiti del livello di trasporto, come si vedrà in dettaglio più avanti, è anche quello di rendere possibile una connessione virtuale tra due applicazioni; gli indirizzi IP non sono sufficienti a questo scopo, in quanto identificano l'intero nodo e non una sua componente. Per poter fare questo, TCP e UDP estendono il sistema di indirizzamento di IP in maniera tale da portarlo a una granularità più fine; si parla di *indirizzi a livello di trasporto*. Gli indirizzi a livello di trasporto per la rete Internet sono numeri di 48 bit, di cui i primi 32 identificano il nodo e coincidono con il suo indirizzo IP e gli ultimi 16, che vengono generalmente indicati come *numero di porta*, rappresentano un riferimento al canale di comunicazione in possesso dell'applicazione. Gli indirizzi a livello di trasporto vengono generalmente rappresentati concatenando l'indirizzo del nodo al numero della porta divisi dal carattere ":"; ad esempio 159.149.155.85:80.

Ogni servizio standardizzato di cui è possibile fruire in rete è assegnato a una porta specifica (*Well Known Port* o *Registered Port*) dalla IANA.



Il fatto che un servizio sia standardizzato su una certa porta non comporta l'obbligo tassativo di erogarlo facendo uso di un certo indirizzo di trasporto. In generale, noto il servizio di cui si intende usufruire, questo verrà contattato utilizzando la porta standard. Nessuno vieta di erogare servizi su porte non standardizzate, ma questi risulteranno semplicemente più difficoltosi da contattare.

2.2.3 Livello di applicazione

A questo livello dello stack si collocano i protocolli utilizzati tra applicazioni per la fruizione dei servizi. Tra questi possiamo trovare protocolli per la gestione della posta elettronica (SMTP [22], POP [23], IMAP [24]), trasferimento file (FTP [25], TFTP [26]), condivisione di file system (NFS [21], SMB [27]), gestione di sessioni di lavoro (telnet [28], *Secure Shell* (SSH) [29]), trasporto (gopher [30], HTTP [11]) e ricerca di informazioni (DNS [20], whois [31]).

La specifica del protocollo per ogni servizio standardizzato di cui è possibile fruire in rete si può trovare sul sito web della *Internet Engineering Task Force* (IETF) [32] in documenti indicati con la dicitura *Request For Comments* (RFC).

Il sistema di indirizzamento dei protocolli applicativi si appoggia generalmente sul livello di trasporto, nel senso che nella maggior parte dei casi l'identificazione dell'entità da contattare avviene tramite un indirizzo IP (o il nome del nodo, riconducibile al suo indirizzo) e una porta.

Sebbene spesso le applicazioni definiscano una loro sintassi per l'identificazione di un client o di un server, non esistono dei veri e propri standard per un sistema di indirizzamento a questo livello; in tal caso, infatti, dovrebbe esserne stabilito uno per ogni possibile protocollo e metodo di accesso. È stato invece definito in tempi relativamente recenti un sistema più generale per l'identificazione di una singola risorsa all'interno della rete: si parla spesso di *Uniform Resource Identifier* (URI) [33]. Nella sua forma più generale un URI si compone di due parti: una modalità di accesso (protocollo o schema) e una parte dipendente da quest'ultima.

<scheme>:<scheme-specific-part>

Il sottoinsieme degli URI che accedono a informazioni organizzate in maniera gerarchica può essere identificato con una sintassi più specifica, come quella che segue:

<scheme>://<authority><path>?<query>

Scheme è il protocollo con cui accedere all'informazione, *authority* indica una entità responsabile per la gestione e la fornitura del servizio, *path* il percorso all'interno dello schema

gerarchico e *query* identifica una informazione che deve essere elaborata da una risorsa (come da esempio un parametro per una pagina web dinamica). A questo punto una stringa come la seguente:

`http://www.unimi.it/chiedove/
schedaPersonaXML.jsp?matricola=16088`

dovrebbe assumere un significato piuttosto chiaro.

Gli URI possono essere specializzati in due sottotipi a seconda di come vengono utilizzati. Se lo scopo principale è quello di definire un metodo di accesso per raggiungere un'informazione, si parla di *Uniform Resource Locator* (URL). Se, invece, l'obiettivo è quello di identificare una risorsa indipendentemente dal punto in cui essa si trova e dal fatto che sia o meno disponibile, si parla allora di *Uniform Resource Name* (URN). La differenza tra URL e URN è simile alla differenza tra “dove” e “chi”.

Gli URL vengono tipicamente utilizzati per indicare la posizione di una pagina web; in questo caso specifico il protocollo HTTP è utilizzato per accedere alle informazioni, l'autority è nodo che possiede la pagina e il *percorso* indica come arrivare al file che contiene le informazioni. Un esempio dell'uso di URN è invece il *Digital Object Identifier System* (DOI) [34]. Il sistema DOI viene utilizzato per recuperare informazioni su opere letterarie; in questo caso l'autority è

l'editore e il percorso indica un codice univoco con cui identificare l'opera.

Ad esempio, il DOI del documento che descrive come funziona il sistema stesso è il seguente:

doi:10.1000/182

È possibile reperire le informazioni relative a un DOI anche attraverso il sito web della *International DOI Foundation* utilizzando il seguente URL.

<http://dx.doi.org/10.1000/182>

Un URL non è necessariamente legato al protocollo HTTP; infatti, anche “<ftp://www.ietf.org/rfc/rfc2396.txt>” è un URL valido.



2.3 Protocolli e socket

Come fanno le applicazioni ad accedere alla struttura di rete per implementare i protocolli?

Il sistema operativo mette a disposizione del programmatore uno strumento chiamato *socket* per accedere ai moduli che implementano i protocolli a livello di trasporto. Una socket è il punto terminale di un sistema di comunicazione: due processi (client e server) creano una socket ciascuno e poi istruiscono l'infrastruttura di rete perché queste vengano associate tra loro e possano essere utilizzate per lo scambio di dati. Una volta che la comunicazione è stata instaurata, i due processi tratteranno le rispettive socket come comuni canali di *Input/Output* (I/O) e potranno scambiarsi messaggi utilizzando gli stessi criteri adottati per tutte le altre operazioni di I/O.

Questo vuol dire che il programmatore non dovrà fare altro che appoggiarsi sull'infrastruttura offerta dalle socket per scrivere codice in grado di aprire un canale di comunicazione, inviare messaggi secondo la sintassi richiesta dal protocollo implementato e reagire in maniera opportuna ai messaggi provenienti dalla rete.

Capitolo 3

Implementazione di un protocollo

In generale, l'implementazione di un protocollo è l'attività che, partendo dalle specifiche tecniche, permette di generare il codice che realizza il comportamento richiesto dal protocollo stesso.

All'interno delle specifiche di un protocollo vengono normalmente identificati:

1. il formato dei messaggi da inviare e ricevere (un lessico)
2. quali sono le sequenze di messaggi valide (una sintassi)
3. quando inviare e ricevere messaggi (un sistema *di turni*).

Esercizio 3.1



Per ognuno degli esempi visti all'inizio del Capitolo 2 si identifichino lessico e sintassi.

Lo scopo di questo capitolo è quello di fornire alcune indicazioni generali per l'organizzazione del codice in maniera tale per cui, partendo dalle specifiche, la stesura del programma possa essere la più semplice e organizzata possibile.

3.1 Strutturazione del codice

Come già detto nel capitolo precedente, il programmatore si appoggia all'infrastruttura offerta dalle socket e scrive del codice che, dopo aver aperto un canale di comunicazione, reagisce in maniera opportuna ai messaggi provenienti dalla rete. Come *reazione* a un messaggio il programma potrebbe decidere di aggiornare il valore delle sue variabili interne e/o spedire un messaggio di risposta; quindi, con una prima approssimazione, l'implementazione potrebbe assumere la forma delineata dallo pseudocodice presentato nella Figura 3.1.

```
programma {
    apertura_canale();
    while (true) { // ciclo infinito
        ricezione_messaggio();
        aggiornamento_variabili();
        spedizione_messaggio_di_risposta();
    }
}
```

Figura 3.1 Pseudocodice di un programma per la gestione di messaggi via rete.

La sequenza ricezione-aggiornamento-spedizione, se non implementata con la dovuta cura, può però portare a due situazioni difficili da gestire.

La prima situazione nasce dal tipico errore di includere tutto il codice all'interno di un'unica funzione; cosa, tra l'altro, sempre sconsigliata. All'inizio della stesura il programma è molto breve, quasi banale, e quindi sembra una soluzione ragionevole; progressivamente la funzione svolge sistematicamente un numero sempre maggiore di attività e diventa di fatto ingestibile non potendo più essere identificata con una specifica operazione. Nel caso in cui si debba modificare il programma in un secondo tempo, il programmatore sarà costretto a un lavoro lungo e oneroso per ricostruire il flusso delle operazioni.

La seconda situazione, invece, potrebbe essere generata da una sequenza non ottimale delle istruzioni eseguite. Un approccio tipico è quello di discriminare il contenuto del messaggio e poi, in base alla situazione (lo stato delle variabili interne) decidere quale reazione sia opportuno intraprendere. A seconda di quando lo si riceve, possono esserci però casi in cui uno stesso messaggio ha significati diversi; si pensi al messaggio *“buongiorno”* detto all'inizio o alla fine di una discussione. Il programmatore potrebbe essere costretto a svolgere operazioni correlate tra loro in zone del programma molto diverse: ad esempio, in fase di chiusura di una comunicazione potremmo avere un saluto oppure un'altra richiesta di servizio. Questo

rende spesso difficile l'identificazione degli errori e complicata la verifica di correttezza del software.

Le regole da seguire per evitare queste situazioni sono semplici.

1. Delimitare blocchi di codice distinti e ben separati per la gestione di ogni singolo messaggio (modularità).
2. Assegnare al programma uno stato (o configurazione) in cui esso si trova, in modo che sia facile associarvi i messaggi che, secondo le specifiche del protocollo, è lecito ricevere e le relative azioni da intraprendere.

La delimitazione di porzioni di codice specifiche per la gestione di ogni singolo messaggio, o categoria di messaggi, facilita enormemente la futura manutenzione del programma. Attenzione: *delimitare* non implica necessariamente *dislocare*, anche se sarebbe preferibile. La gestione di un certo messaggio dovrebbe essere svolta preferibilmente in una funzione o in un metodo separati; tuttavia, questo porta a un proliferare di definizioni che, se portate all'eccesso, rendono anch'esse di difficile lettura il codice. A volte può essere sufficiente, soprattutto per operazioni brevi e svolte una sola volta, delimitare le linee di codice con commenti ben visibili; l'importante è non spargere le chiamate riguardanti il messaggio in esame in punti arbitrari.

La definizione esplicita di uno stato all'interno del programma è utile per due motivi: il primo è che spesso le specifiche di un protocollo vengono date sotto forma di un automa a stati finiti, per cui, ricostruire gli stati assunti dal protocollo all'interno

del codice semplifica molto la verifica della sua correttezza; il secondo motivo è che lo stesso messaggio ricevuto in stati diversi può essere gestito in maniera distinta senza complicare ulteriormente il codice.

All'interno di questo libro non ci si soffermerà oltre sugli aspetti di modularità del software, che sono affrontati in molti testi di programmazione o ingegneria del software; si analizzerà invece in dettaglio la gestione degli stati per implementare un protocollo.

3.2 Stati di un programma

La struttura degli stati di un programma e la sua evoluzione da uno stato all'altro può essere rappresentata facendo uso di automi a stati finiti. Per una trattazione completa degli automi a stati finiti è possibile fare riferimento a [35].

Gli automi sono formalismi matematici utilizzati per descrivere l'evoluzione di sistemi in grado di assumere un insieme numerabile di configurazioni. Un automa a stati finiti è un sistema caratterizzato da un numero limitato di stati (o configurazioni) in cui si può trovare e che passa da una configurazione all'altra in base a una sollecitazione esterna (*un simbolo in ingresso*). Allo stesso modo, il programma che si sta andando a progettare passerà da uno stato all'altro in base ai messaggi in arrivo dalla rete.

Le transizioni di un automa possono essere rappresentata tramite un grafo (diagramma di transizione), si veda la Figura 3.2 per un esempio.

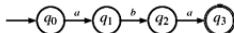


Figura 3.2 Esempio di diagramma di transizione per un automa a stati finiti.

In un diagramma di transizione i nodi rappresentano gli stati, e gli archi tutte e sole le possibili transizioni; le etichette sugli archi identificano i simboli in ingresso richiesti per effettuare la transizione. Nell'esempio appena proposto solo il messaggio *b* permette all'automa di evolvere dallo stato *q*₁ allo stato *q*₂. L'automa evolve dallo stato iniziale, identificato con una freccia entrante e tipicamente chiamato *q*₀, fino a uno stato finale, identificato da un doppio cerchio. In particolare, per quanto riguarda gli stati finali, non è detto che ce ne debba essere solo uno, come pure non è detto che uno stato terminale debba essere privo di vie di uscita.

Un programma rappresentabile con l'automa descritto nella Figura 3.2 può essere utilizzato per la gestione di un protocollo in cui la sequenza dei messaggi è fissa nonché nota a priori; come ad esempio una comunicazione che avviene quando si chiede che ora è: ci si saluta, viene fatta la domanda, viene fornita la risposta e si chiude con un ringraziamento.

I diagrammi di transizione possono anche presentare cicli, come nella Figura 3.3.

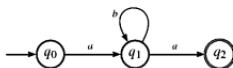


Figura 3.3 Esempio di diagramma di transizione con cicli.

Automi che presentano cicli sono necessari nello strutturare codice per la gestione di proto colli in cui la sequenza dei messaggi non è nota a priori; si pensi ad esempio alla comunicazione che avviene tra un cameriere e un cliente: dopo essersi accomodato al tavolo il cliente comincia a ordinare, finita una portata può ordinare nuovamente o chiedere il conto. Non esiste nessun vincolo sul numero delle portate che il cliente è in grado di ordinare prima di chiedere il conto; in alcuni casi neppure il cliente stesso lo sa a priori.

Come ultima variante, possono esserci situazioni in cui l'evoluzione del sistema non sia deterministica, come nella Figura 3.4.

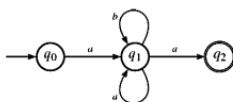


Figura 3.4 Esempio di diagramma di transizione non deterministico.

In questo caso ci si trova in presenza di almeno uno stato con due transizioni uscenti etichettate con lo stesso simbolo; a fronte di uno stesso messaggio in ingresso, l'evoluzione del programma può procedere verso più stati diversi.

Nella programmazione il non determinismo non esiste!



Il percorso intrapreso da un programma in cui, a fronte di un simbolo in ingresso si presentano più transizioni possibili, è determinato dallo stato interno dei suoi dati. Occorre ricordare che, nell'utilizzo di automi per descrivere il comportamento del software, l'obiettivo è ottenere uno schema per riuscire a gestire tutte le sequenze di messaggi valide, non fare un modello dell'evoluzione dello stato interno del software.

3.3 Gli automi applicati al modello client-server

Nel modello client-server ci si trova in presenza di due entità; questo vuol dire che dovranno essere usati due automi, uno per il client e l'altro per il server. Inoltre, è possibile supporre che i messaggi utilizzati dai due automi siano gli stessi; è infatti uso

comune definire un protocollo tramite un solo lessico (una sola specifica) comune a client e server.

Per *dare un senso* alla conversazione, l'evoluzione dei due automi dovrà però essere correlata. Sulla base della semplice osservazione per cui “*i messaggi dovranno essere mandati da server a client e viceversa*” è possibile stabilire che, nella transizione da uno stato all'altro ogni automa emetterà un messaggio verso la controparte in maniera tale da provocare un cambiamento di stato anche in quest'ultima.

Se si intende utilizzare il diagramma di stato di un automa come strumento per la progettazione di un programma che gestisce i messaggi di un protocollo, allora è necessario ricordare che una sequenza di messaggi ricevuti è sintatticamente valida se fa evolvere l'automa dallo stato iniziale fino a uno degli stati finali. Il protocollo descritto risulterà formalmente corretto se e solo se per ogni possibile sequenza di messaggi valida di ogni automa ne genererà una valida per l'altro automa.

Per rappresentare questa situazione è possibile agire sul diagramma di transizione ed estenderlo. Gli archi saranno etichettati con due simboli (messaggi) usando una dicitura del tipo “*a/B*”: il primo simbolo è quello responsabile della transizione, mentre il secondo è quello emesso verso l'altro automa. Per convenzione si farà uso del simbolo “.” per indicare l'assenza di un messaggio. Una transizione mancante del secondo simbolo significa che è possibile passare da uno stato all'altro senza emettere alcuno messaggio verso l'altro

automa, mentre una transizione mancante del primo implica che il cambio di stato può avvenire anche in assenza di input. Generalmente, quest'ultimo tipo di transizione è utile per formalizzare messaggi composti da più parti alle quali possono corrispondere una serie di azioni diverse. Anche la transizione fittizia che porta nello stato iniziale sarà etichettata; il primo simbolo sarà sempre un “”, in quanto l'automa si porta nello stato iniziale senza bisogno di ricevere nessun messaggio, il secondo, invece, sarà presente nel caso in cui l'automa debba iniziare la conversazione: al momento della partenza l'automa si pone nello stato iniziale e manda un messaggio alla controparte.

Si veda l'esempio riportato nella Figura 3.5, ottenuto estendendo con alcune etichette quello nella Figura 3.4.

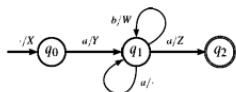


Figura 3.5 Diagramma di transizione etichettato: a ogni transizione un simbolo viene emesso e utilizzato come ingresso per un altro automa.

3.4 Fasi della progettazione

Nella progettazione di un programma per la gestione dei messaggi di un protocollo, innanzitutto occorre consultare le specifiche e capire quali siano i messaggi da gestire e le sequenze

valide. Spesso può essere utile descrivere il problema in forma discorsiva; in tal modo è possibile cominciare a individuare gli attori in gioco e come caratterizzare i messaggi che devono scambiarsi. È opportuno però fare una piccola puntualizzazione su questi ultimi. Per motivi di praticità i messaggi possono anche essere raggruppati in famiglie e resi parametrici; potrebbe infatti essere utile definire un messaggio del tipo “*desidererei N numeri casuali*”, dove N può variare di volta in volta. Al variare di N verranno generati un insieme di messaggi diversi ma che il programma dovrà essere in grado di trattare in modo analogo.



Un errore tipico è quello di pensare che a fronte di un messaggio parametrico si debba costruire un automa che *prenda una decisione*. L'automa deve solo permettere di fornire tutte le risposte ammissibili; il percorso da seguire lo deciderà il programma in base allo stato interno delle informazioni. Ad esempio, a fronte del messaggio “*vorrei N grammi di zucchero*”; è possibile ottenere risposta positiva o negativa, dipendentemente dalla disponibilità di magazzino, ma questo non è competenza né dell'automa né del protocollo. È vero comunque che condizioni diverse porteranno a evoluzioni distinte e, di conseguenza, la trasmissione di messaggi diversi verso l'altro automa; che dovranno poi essere gestiti.

Il secondo passo è quello di identificare le varie *condizioni* in cui gli attori coinvolti si possono trovare. Queste risulteranno essere in corrispondenza con gli stati degli automi e, di conseguenza, dei programmi da realizzare. Nella maggioranza dei casi la descrizione di uno stato sarà del tipo “*in attesa di ...*”; dove in pratica l'automa sta aspettando il prossimo messaggio per poter evolvere ulteriormente.

Nella terza e ultima fase si procede alla stesura dei due automi. Il modo più semplice per farlo è prima disporre graficamente gli stati e successivamente aggiungere le transizioni, ricordando che per ogni messaggio da emettere da parte di un automa deve essere presente almeno una transizione corrispondente nell'altro automa.

3.5 Esempio di progettazione

In questo paragrafo si progetterà una coppia di automi che descrivono il comportamento del protocollo per l'accesso alla biglietteria di una sala cinematografica. L'esempio verrà via via arricchito per analizzare una serie di possibili problematiche.

3.5.1 Problema di partenza

Il cliente si presenta allo sportello, richiede un biglietto; questo gli viene fornito e la comunicazione ha termine.

Prima fase: specifiche

Descrizione

Il cliente si presenta in biglietteria e saluta. Dopo aver avuto l'attenzione dell'operatore allo sportello, cosa segnalata dal fatto che anche questi saluta, richiede un biglietto. L'operatore consegna il biglietto e il cliente si allontana dopo aver salutato ed essere stato ricambiato dall'operatore.

Messaggi

- salve
- biglietto
- sì

In questo caso si è deciso di usare lo stesso messaggio per il saluto del cliente verso l'operatore e viceversa.

Seconda fase: definizione degli stati

Nella seguente tabella vengono elencati gli stati possibili per cliente e operatore. Per ragioni di praticità, a ogni stato sarà anche assegnata un'etichetta, meno significativa ma più semplice da indicare all'interno del diagramma degli stati.

Cliente	Operatore
c_0 in attesa di ricevere l'attenzione dell'operatore	o_0 in attesa dell'arrivo di un cliente
c_1 in attesa di ricevere il biglietto	o_1 in attesa della richiesta di biglietto
c_2 in attesa di essere salutato dall'operatore	o_2 in attesa di essere salutato dal cliente
c_3 servizio terminato	o_3 servizio terminato

Terza fase: stesura degli automi

È possibile quindi procedere a predisporre gli stati dei due automi (Figura 3.6).

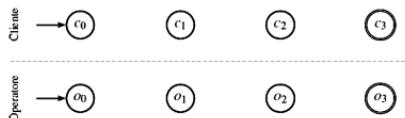


Figura 3.6 Il problema della biglietteria, stati senza transizioni.

Inizialmente i due automi si trovano entrambi nello stato iniziale: c_0 (in attesa di ricevere l'attenzione dell'operatore) per il cliente e o_0 (in attesa dell'arrivo di un cliente) per l'operatore.

L'operatore non dice nulla al cliente (non emette nessun messaggio) ed è in attesa di essere salutato; questo vuol dire che occorre aggiungere un'etichetta “ $\cdot/$ ” alla transizione fittizia entrante in o_0 . Per poter ricambiare il saluto del cliente è inoltre necessaria una transizione da o_0 a o_1 fatta scattare dal messaggio “*salve*” e che genera a sua volta un “*salve*”, per salutare il cliente.

Il cliente, invece, per prima cosa saluterà l'operatore, per cui occorre etichettare con “ $\cdot/salve$ ” l'ingresso nello stato iniziale c_0 , stato da cui il cliente uscirà per spostarsi in c_1 nel momento in cui verrà a sua volta salutato dall'operatore; per cui è necessario inserire un'ulteriore transizione da c_0 a c_1 fatta

scattare anch'essa dal messaggio “*salve*”. Temporaneamente, questa transizione sarà etichettata con “*salve/*”.

Il risultato è riportato nella Figura 3.7. L'emissione del messaggio “*salve*” all'ingresso in c_0 da parte del cliente farà evolvere l'operatore dallo stato o_0 allo stato o_1 con l'emissione dello stesso messaggio da parte dell'operatore, che farà evolvere il cliente dallo stato c_0 allo stato c_1 ; queste dipendenze sono state indicate con le frecce tratteggiate, che non rappresentano le transizioni.

La fase introduttiva alla conversazione, in cui client e server entrano in contatto e cominciano a sincronizzarsi viene anche chiamata fase di *handshake* (stretta di mano). 

A questo punto il cliente e l'operatore si trovano rispettivamente negli stati c_1 e o_1 dopo essersi sincronizzati; è quindi possibile procedere alla fruizione del servizio.

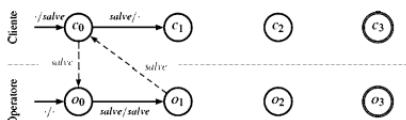


Figura 3.7 Biglietteria, fase di sincronizzazione.

Il cliente, entrando nello stato c_1 procede con la richiesta di usufruire del servizio mandando il messaggio “*biglietto*” all’operatore, e poi si mette in attesa di ottenere il servizio ricevendo il messaggio “*si*”; questo si traduce nell’estendere l’etichetta della transizione da c_0 a c_1 a “*salve/biglietto*” e aggiungere una transizione dallo stato c_1 allo stato c_2 innescata dalla ricezione del messaggio “*si*”.

L’operatore, invece, deve ricevere la richiesta di servizio e spostarsi nello stato successivo, o_2 , dove consegnerà il biglietto mandando il messaggio “*si*” al cliente; questo si traduce nell’aggiungere una transizione dallo stato o_1 allo stato o_2 innescata dalla richiesta di servizio e che prevede l’invio del messaggio di risposta.

Il risultato è rappresentato nella Figura 3.8.

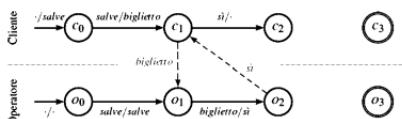


Figura 3.8 Biglietteria, fase di fruizione del servizio.

Infine, una volta ottenuto il servizio, il cliente saluta l’operatore, che ricambia, e i due attori terminano la propria attività. Questo vuol dire che il cliente nello stato c_2 deve mandare il messaggio “*salve*” al quale l’operatore risponderà e dopo tutti e due gli automi dovranno trovarsi nei rispettivi stati finali (c_3

e o_3); per ottenere questo comportamento occorre emettere il messaggio “*salve*” nella transizione da c_1 a c_2 e aggiungere due transizioni, da c_2 a c_3 e da o_2 a o_3 , innescate dal messaggio “*salve*”. Il risultato finale è raffigurato nella Figura 3.9.

Come già anticipato, si può notare che nella Figura 3.9 i messaggi “*salve*” in apertura e chiusura della comunicazione hanno significati diversi in quanto associati a due transizioni distinte. Tecnicamente, dovrebbero essere trattati come messaggi non correlati tra loro nonostante abbiano la stessa dicitura.

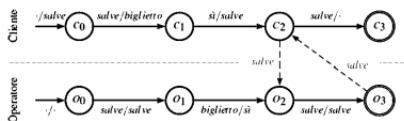


Figura 3.9 Biglietteria, fase di chiusura della connessione.

3.5.2 Introduzione di messaggi parametrici

È possibile apportare una piccola modifica all'esempio appena visto facendo richiedere al cliente N biglietti.

Lo svolgimento risulta essere pressoché identico a quello già visto. L'unica differenza è che bisogna parametrizzare il secondo messaggio. Questo non ha comunque nessuna ricaduta sulla struttura; d'altronde, come già puntualizzato, non è compito dell'automa prendere decisioni.

Descrizione

Il cliente si presenta in biglietteria e, dopo aver avuto l'attenzione dell'operatore allo sportello, richiede N biglietti; l'operatore consegna i biglietti e il cliente si allontana dopo aver salutato.

Messaggi

- salve
- N biglietti
- sì

Stati

Cliente	Operatore
c_0 in attesa di ricevere l'attenzione dell'operatore	o_0 in attesa dell'arrivo di un cliente
c_1 in attesa di ricevere i biglietti	o_1 in attesa della richiesta di biglietti
c_2 in attesa di essere salutato dall'operatore	o_2 in attesa di essere salutato dal cliente
c_3 servizio terminato	o_3 servizio terminato

Automi

La soluzione, banale, è rappresentata nella Figura 3.10.

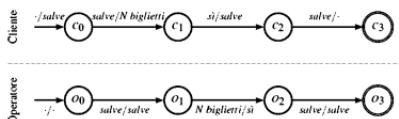


Figura 3.10 Biglietteria con messaggio parametrico.

3.5.3 Introduzione del non determinismo

Estendendo ulteriormente il caso in esame, è epossibile prevedere la possibilità che l'operatore non abbia a disposizione N posti liberi in sala. Se non sono disponibili, l'operatore fornisce una risposta negativa e non consegna i biglietti; diversamente si comporta come al solito. Indipendentemente dall'esito della transazione il cliente saluta e si allontana.

Descrizione

Il cliente si presenta in biglietteria e saluta. Dopo aver avuto l'attenzione dell'operatore allo sportello, richiede N biglietti. Se sono disponibili l'operatore consegna i biglietti, altrimenti comunica l'esito negativo della richiesta. Il cliente, ottenuta la risposta (qualunque essa sia), si allontana dopo aver salutato ed essere stato ricambiato dall'operatore.

Messaggi

Occorre aggiungere un ulteriore messaggio per comunicare la non disponibilità dei biglietti e ciò si ripercuote anche sulla struttura degli automi.

- salve
- N biglietti
- sì
- no

Stati

La configurazione degli stati dell'operatore non cambia; nonostante il fatto per cui lo stato o_1 può generare due risposte diverse a seconda della disponibilità lo stato successivo, per quanto detto nelle specifiche, è comunque quello in cui l'operatore si mette in attesa di essere salutato dal cliente che se ne va (o_2).

Cliente	Operatore
c_0 in attesa di ricevere l'attenzione dell'operatore	o_0 in attesa dell'arrivo di un cliente
c_1 in attesa di ricevere i biglietti	o_1 in attesa della richiesta di biglietti
c_2 in attesa di essere salutato dall'operatore	o_2 in attesa di essere salutato dal cliente
c_3 servizio terminato	o_3 servizio terminato

Automi

Il risultato è rappresentato nella Figura 3.11. Come si può osservare, l'automa relativo all'operatore può emettere due risposte diverse dallo stato o_1 ; di conseguenza il cliente, quando si trova nello stato c_1 , deve poter gestire sia il messaggio “*si*” che “*no*”.

Come regola generale, quando in un automa, a fronte di un messaggio in ingresso, esistono più transizioni possibili, tutti i messaggi emessi dalle transizioni devono trovare riscontro nelle transizioni uscenti dallo stato corrente dell'altro automa.



Un errore tipico in questa situazione è considerare come due stati distinti quelli in cui si viene a trovare l'operatore dopo aver fornito la risposta; assumendo che, siccome ci sono due possibili esiti, allora ci saranno anche due stati, come descritto nella Figura 3.12.

In realtà non si tratta di un vero errore. Semplicemente, siamo in presenza di uno stato ridondante: non esiste una vera differenza, dal punto di vista dell'automa, nel trovarsi nello stato o_2 piuttosto che nello stato o_4 . Un programma sviluppato sulla base di questo diagramma di transizione funzionerà correttamente ma presenterà due stati distinti (due zone di codice) preposti alle stesse operazioni, costringendo probabilmente il programmatore a duplicare il codice con i conseguenti problemi per l'identificazione di eventuali errori.

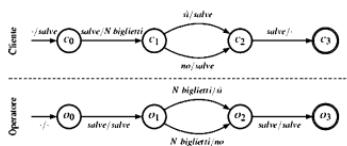


Figura 3.11 Biglietteria con operatore non deterministico.

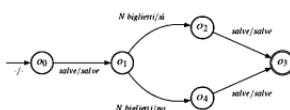


Figura 3.12 Operatore non deterministico sub-ottimale.

Quando, in un diagramma di transizione, due o più stati presentano esattamente le stesse transizioni in uscita (etichette e stati di destinazione), allora probabilmenet si è di fronte a una ridondanza e gli stati possono essere rappresentati come uno solo.



3.5.4 Esempio completo

Si può rendere non deterministico anche l'automa relativo al cliente. In caso di non disponibilità dei biglietti il cliente può decidere di salutare e andarsene oppure di effettuare un'altra richiesta.

Descrizione

Il cliente si presenta in biglietteria e saluta. Dopo aver avuto l'attenzione dell'operatore allo sportello richiede N biglietti. Se sono disponibili, l'operatore consegna i biglietti, altrimenti comunica l'esito negativo della richiesta. Il cliente, in caso di risposta positiva, si allontana dopo aver salutato ed essere stato ricambiato dall'operatore; in caso di risposta negativa, invece, ha la facoltà di effettuare una nuova richiesta oppure decidere se salutare e andarsene dopo essere stato ricambiato dall'operatore.

Messaggi

- salve
- N biglietti
- sì
- no

Stati

In questo caso è corretto introdurre un nuovo stato: dopo aver effettuato la prima richiesta il cliente potrebbe dover salutare o fare una nuova richiesta; essendo i messaggi ammissibili in ricezione diversi occorrono due stati distinti all'interno dell'operatore.

Cliente	Operatore
c_0 in attesa di ricevere l'attenzione dell'operatore	o_0 in attesa dell'arrivo di un cliente
c_1 in attesa di ricevere i biglietti	o_1 in attesa della richiesta di biglietti
c_2 in attesa di essere salutato dall'operatore	o_2 in attesa di essere salutato dal cliente (biglietti disponibili)
c_3 servizio terminato	o_3 servizio terminato
	o_4 in attesa di essere salutato o di una nuova scelta del cliente (biglietti non disponibili si primo tentativo)

Automi

Una prima soluzione è proposta nella Figura 3.13.

Per poter estendere il cliente è sufficiente aggiungere una transizione che da c_1 riporta nello stesso stato e ha come effetto l'emissione di un 'ulteriore richiesta a fronte di una risposta negativa. In questo modo, quando il cliente si trova nello stato c_1 e ottiene una risposta negativa non è più forzato a salutare

e terminare. Nel server, invece, occorre inserire due transizioni uscenti da o_4 . In caso di risposta negativa il server deve essere in grado di gestire l'operazione di saluto oppure un'ulteriore richiesta di servizio; infatti, tutte e due queste transizioni uscenti sono innescate dallo stesso messaggio “ N biglietti”.

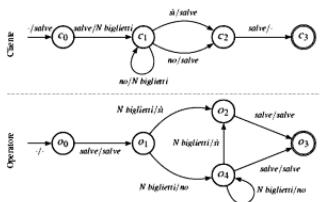


Figura 3.13 Biglietteria, soluzione completa.

È importante notare che il quantitativo di posti da richiedere non interessa l'automa; per un corretto funzionamento del software il numero di posti richiesti dovrà di volta in volta diminuire, ma questo non influisce sul corretto funzionamento del protocollo.



Tuttavia, la soluzione proposta, seppur corretta, pone un problema di carattere pratico. Se a livello di programma ogni stato viene realizzato con un suo frammento di codice si ha una ripetizione nell'implementazione di o_1 e o_4 ; infatti, in tutti e due gli stati occorre effettuare il controllo se a fronte di una richiesta di posti c'è disponibilità in sala. L'ideale, dal punto

di vista della programmazione, sarebbe avere un unico punto in cui viene eseguita quest'operazione; in via teorica sarebbe possibile pensare di unire tra loro gli stati o_1 e o_4 , ottenendo quanto riportato nella Figura 3.14.

Osservando attentamente, però, anche questa soluzione presenta un problema non secondario: avendo unito gli stati o_1 e o_4 nello stato $o_{1,4}$, per non perdere di funzionalità è stato necessario inserire una transizione innescata dal messaggio “*salve*” da $o_{1,4}$ a o_3 ; per cui, in questo momento, anche due messaggi di saluto consecutivi portano l'automa nello stato finale. Formalmente, possiamo dire che è stato esteso l'insieme delle sequenze di messaggi valide riconosciute del protocollo.

Dal punto di vista pratico questo non dovrebbe essere un problema, visto che il cliente non è in grado di generare la sequenza in questione; tuttavia questo ci espone a potenziali problemi di sicurezza, in quanto, in un sistema più complesso l'operatore potrebbe essere indotto a comportamenti anomali da clienti costruiti appositamente per fornire sequenze di messaggi riconosciute come valide dal programma ma non facenti parte del protocollo. Inoltre, da un punto di vista più formale questo automa non è più una valida implementazione del protocollo richiesto, in quanto è in grado di generare (attraversando gli stati o_0 , $o_{1,4}$ e o_4) una stringa costituita da due messaggi di saluto consecutivi, la quale non è riconosciuta come valida dal cliente.

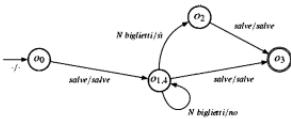


Figura 3.14 Biglietteria, operatore con un singolo stato decisionale.

In realtà il vero problema risiede nella strutturazione del protocollo, il quale obbliga ad avere un primo momento in cui si deve gestire sicuramente una richiesta di servizio e un secondo in cui si può ricevere una nuova richiesta di servizio oppure un saluto. Dal punto di vista implementativo, se non si vuole incorrere nei problemi appena descritti, ci sono due vie di uscita: la prima è quella di variare le specifiche del protocollo; ciò non sempre è possibile (e comunque è sempre sconsigliato) in quanto le specifiche potrebbero essere state scritte da terzi. La seconda via è quella di *rilassare* i vincoli e implementare un operatore che accetti anche sequenze di messaggi non previste dal protocollo, come quello proposto nella Figura 3.14.

La soluzione consigliata è quella rappresentata nella Figura 3.13, a patto di dislocare il codice per la gestione di una richiesta in un unico punto e farlo condividere alle sezioni di codice relative ai due stati.

Esercizio 3.2



Modificare il protocollo per l'accesso alla biglietteria in maniera tale che sia l'operatore a salutare per primo.

Esercizio 3.3



Si estenda il protocollo per l'accesso all biglietteria in maniera tale da rendere noto il numero di posti rimasti in sala.

Descrizione

Il cliente si presenta in biglietteria e saluta. Dopo aver avuto l'attenzione dell'operatore allo sportello, cosa segnalata dal fatto che anche questi saluta, richiede N biglietti. Se sono disponibili, l'operatore consegna i biglietti altrimenti comunica l'esito negativo della richiesta e i posti attualmente disponibili in sala. Il cliente, in caso di risposta positiva, si allontana dopo aver salutato ed essere stato ricambiato dall'operatore; in caso di risposta negativa, invece, può effettuare una nuova richiesta oppure decidere se salutare e andarsene dopo essere stato ricambiato dall'operatore.



Esercizio 3.4

Si estenda il protocollo prodotto nell'esercizio precedente in modo da poter gestire eventuali condizioni di errore nel caso in cui il cliente aumenti il numero di biglietti richiesti, anzichè diminuirlo.

Descrizione

Esattamente come l'esercizio precedente ma, a fronte di una risposta negativa, se il cliente aumenta o mantiene invariato il numero di biglietti richiesti, l'operatore segnala la situazione anomala e si rimette in attesa di una nuova richiesta o di un saluto.



Esercizio 3.5

Si progetti, facendo uso di automi a stati finiti, il protocollo per la fruizione di una partita di scacchi in cui client e server giocano uno contro l'altro. Si supponga che il client abbia sempre il bianco e che quindi inizi sempre per primo.

Descrizione

La partita inizia con una mossa del giocatore bianco. Durante la partita un giocatore invia la sua mossa all'altro giocatore. Il secondo giocatore riceve la mossa e la valuta; la mossa potrebbe essere scorretta, corretta o scacco matto. Il risultato della valutazione viene comunicato al primo giocatore. In caso di mossa corretta, tocca al secondo giocatore muovere; in caso di mossa scorretta il primo giocatore deve rifare la sua mossa e in caso di scacco matto la partita termina per entrambi i giocatori.

Messaggi

- mossa
- valida
- non valida
- scacco matto

3.6 Gestione di messaggi nonprevisti

La costruzione di un sistema client-server combinando il cliente (client) nella Figura 3.13 e l'operatore (server) nella Figura 3.14 solleva una questione importante: cosa fare nel momento in cui si riceve un messaggio per cui non è prevista una transizione in uscita dallo stato attuale. Questi messaggi fuori sequenza, come già detto, possono avere due origini: possono essere il

risultato di un errore di progettazione o di un programmatore *pigro*, oppure essere causati intenzionalmente da controparti interessate a portare il software in uno stato instabile al fine di sottrarre informazioni o bloccare la fornitura di un servizio con un attacco *Denial of Service* (DoS).

Per prevenire in particolare il secondo caso è opportuno intercettare e gestire qualunque messaggio in qualunque stato; se il messaggio non risulterà essere tra quelli validi, il programma viene generalmente bloccato (o riportato alla situazione di partenza) in modo da non compromettere l'integrità dei dati al suo interno.

Esercizio 3.6



Si estenda ulteriormente il protocollo prodotto nell'Esercizio 3.4 in modo da terminare immediatamente la comunicazione a fronte di una condizione di errore.

Questo impedisce di mantenere l'operatore occupato indefinitamente da un cliente che sta effettuando un attacco DoS.

II

Programmazione di rete

Capitolo 4

Programmazione con le socket

4.1 Socket

Una socket, come già detto, è il punto terminale di un sistema di comunicazione tra due entità in rete.

Consultando un dizionario in lingua inglese, una definizione di socket è: “*an electrical device receiving a plug or light bulb to make a connection*”. Il termine può quindi essere equiparato al concetto di *presa elettrica* nella lingua italiana. Infatti, una presa elettrica può essere anche vista come il punto terminale dell’infrastruttura che mette in comunicazione un elettrodomestico con la rete del centrale; in modo in cui l’energia viene prodotta o veicolata fino all’utilizzatore è ininfluente al fine della fruizione dell’servizio (ad esempio, azionare la lavatrice o accendere il forno). Le socket, viste come sistema di comunicazione via rete seguono esattamente la stessa filosofia.

In generale si parla indifferentemente di socket sia al femminile sia al maschile. Nel presente testo si preferisce utilizzare il genere femminile per via dell'analogia con “la presa elettrica”.



Da un punto di vista più tecnico, una socket è un’interfaccia all’infrastruttura di comunicazione che il kernel di un sistema operativo mette a disposizione dei processi per accedere ai servizi del livello di trasporto dello stack ISO-OSI.

Una socket può assumere due modalità di funzionamento: *connessa* o *non connessa*. Nel primo caso è messa in relazione diretta con un’altra socket e si stabilisce un rapporto mittente/destinazione permanente; nella seconda modalità, invece, la socket può essere usata per inviare messaggi sulla rete fornendo ogni volta l’indirizzo della destinazione finale, che può variare tra le diverse spedizioni.

4.2 Creazione di una socket

Al momento della creazione di una socket, il sistema operativo dovrà creare una serie di strutture dati al suo interno per poterla gestire. Occorrono tre tipi di informazioni: la modalità con cui si vuole far identificare la socket in rete (un sistema di indirizzamento da utilizzare, o *dominio*), la modalità con cui si

vuole trasferire i dati e il protocollo a livello di trasporto che si intende utilizzare.

4.2.1 Dominio

Un dominio specifica un sistema di indirizzamento e condiziona la modalità nella quale la socket verrà identificata e raggiunta. Il dominio di una socket è messo in corrispondenza con il protocollo a livello di rete (il terzo dello stack ISO-OSI) utilizzato per instradare le informazioni.

I domini più comuni sono:

Unix

La socket non accede alla rete ma riceve e manda dati tramite un particolare file su disco.

Questo vuol dire che sarà possibile costruire un canale di comunicazione solo con altre entità all'interno dello stesso sistema operativo e che la socket verrà identificata tramite una stringa contenente il nome del file.

Internet

In corrispondenza del protocollo IP [5].

La socket verrà identificata tramite l'indirizzo IP del nodo più un numero di 16 bit (porta).

Questo è il dominio normalmente utilizzato per implementare applicazioni che sfruttano servizi presenti su Internet.

Internet versione 6

In corrispondenza del protocollo IPv6 [6].

La socket verrà identificata tramite l'indirizzo IPv6 del nodo più un numero di porta.

Altri domini vengono implementati solo in alcune architetture e sono tipicamente in correlazione con protocolli a loro omonimi; qui di seguito ne vengono riportati alcuni piuttosto noti.

- Appletalk (protocollo DDP [16])
- IPX [15]
- X25[36]
- ATM [37]

Altri, infine, sono specifici di certe architetture e servono tipicamente per le comunicazioni che rimangono all'interno del sistema operativo: esempi di questi domini sono *netlink* per linux [38] e *netgraph* per FreeBSD [39].

Il numero di domini che una socket potrebbe supportare dipende unicamente dall'implementazione del kernel del sistema operativo; tuttavia, il linguaggio di programmazione adottato potrebbe non renderli tutti accessibili al programmatore.

All'interno di questo testo è stata operata la scelta di trattare solo socket operanti in dominio internet in quanto l'attenzione viene rivolta principalmente alla fruizione di servizi su Internet e non alle strategie impiegate nelle varie implementazioni.

4.2.2 Modalità di trasferimento dati

La modalità di trasferimento dati stabilisce in quale modo il ricevente potrà fruire dei dati inviati dal mittente. Usando un termine più formale possiamo dire che fornisce un'astrazione del canale di comunicazione.

Byte-stream

Nella modalità byte-stream i dati vengono fruiti in maniera sequenziale; tutti i messaggi inviati vengono accodati in una sequenza di byte, realizzando quindi un flusso continuo fino alla chiusura del canale. Questa modalità preserva l'ordine di invio dei dati e garantisce l'arrivo di tutte le informazioni.

Datagram

La modalità datagram suddivide i dati inviati in unità indipendenti e li affida al livello di rete. Normalmente, ogni messaggio generato dall'applicazione verrà associato a un pacchetto e inviato, a meno che il messaggio in questione non sia più grande della dimensione massima di un pacchetto a livello di rete; in questo caso la socket provvederà autonomamente a suddividerlo in due segmenti. Il destinatario si vedrà recapitare una sequenza di pacchetti che dovrà interpretare e da cui potrà estrarre dei messaggi; non vengono garantiti però né l'ordine di consegna né il fatto che un messaggio inviato raggiunga effettivamente la destinazione.

La scelta della modalità di trasferimento dei dati dipende unicamente dal servizio da erogare o di cui si intende usufruire. In generale, servizi quali mail e web vengono erogati tramite una modalità byte-stream in quanto perdita e consegna disordinata delle informazioni non sono tollerate. Diversamente, servizi multimediali per la consegna di audio e video sono in grado di tollerare la perdita di una frazione delle informazioni ma prediligono una consegna uniforme dei dati nel tempo; quindi vengono gestiti con una modalità datagram.

Le socket di tipo datagram possono essere utilizzate sia in modalità connessa che non, mentre le socket di tipo byte-stream possono essere utilizzate solo e unicamente in modalità connessa.

Occorre non confondere il concetto di socket datagram (o byte-stream) con quello di modalità non connessa (o connessa): il primo termine riguarda la modalità di fruizione dei dati; il secondo indica invece in che modo il sistema operativo dovrà gestire i pacchetti a livello di trasporto.

4.2.3 Protocollo

La scelta del protocollo è vincolata dalla modalità di trasferimento dati che si è deciso di utilizzare.

Nel caso più protocolli siano disponibili per implementare una data modalità, sarà possibile scegliere liberamente tra questi, diversamente (e nei sistemi moderni questa è la regola) ci si troverà di fronte a un solo protocollo per ogni possibile scelta. Per questo motivo in molti linguaggi e piattaforme di recente creazione la scelta di modalità e protocollo viene fatta con un unico parametro se non è addirittura implicita nel *tipo di socket* che si sta creando.

Nei sistemi operativi moderni ci troviamo di fronte fondamentalmente due protocolli: TCP e UDP.

TCP

TCP è un protocollo (normalmente l'unico) con cui i sistemi operativi moderni implementano una modalità di trasferimento byte-stream. Esso, oltre che garantire la consegna ordinata dei dati, effettua anche una serie di controlli sullo stato della rete e implementa meccanismi di *controllo di flusso* e *controllo di congestione*: il primo aiuta a non saturare le risorse del nodo ricevente mentre il secondo ha lo scopo di non saturare le risorse degli apparati lungo il percorso, in maniera tale da prevenire la perdita di pacchetti ed evitare di doverli ritrasmettere.

UDP

UDP è un protocollo (normalmente l'unico) con cui i sistemi operativi moderni implementano una modalità di trasferimento datagram. Diversamente da TCP, non implementa nessuna politica di controllo del traffico di rete in caso di congestione.

Chi studia le dinami che del traffico in rete è solito dire che UDP *uccide* TCP. La strategia di controllo di congestione di TCP è tale per cui la capacità trasmissiva di rete da parte di ogni singolo flusso dati diminuisce in presenza di congestione, cioè in presenza di altri flussi di traffico. Se il canale trasmissivo viene utilizzato unicamente da flussi gestiti con TCP si ha un'equa distribuzione delle risorse e contemporaneamente la quantità di pacchetti persi e ritrasmessi è minima. Se, invece, il canale è condiviso da flussi TCP e UDP che trasmettono dati in maniera in discriminata, ci si può trovare di fronte a un circolo vizioso. Nel momento in cui vengono persi alcuni pacchetti a causa della congestione, il protocollo TCP diminuirà l'impegno della rete e libererà risorse che verranno immediatamente occupate da pacchetti UDP, senza quindi abbassare il livello di congestione. Iterando questo comportamento, la rete converge verso una situazione in cui tutta la banda viene utilizzata dal protocollo UDP. Per questo motivo sarebbe quindi opportuno non utilizzare il protocollo UDP se si intende erogare un servizio in cui si riversano sulla rete dati in maniera incontrollata. I sistemi di streaming multimediale, che in quanto datagram utilizzano il protocollo UDP, non inviano dati senza alcun controllo ma pongono alcuni limiti applicativi al quantitativo di bit inviati ogni secondo. Infatti, quando si fruisce di contenuti

multimediali in rete, troviamo generalmente un'indicazione della banda utilizzata.

4.3 Indirizzi: numeri e nomi

Durante il normale uso di applicazioni di rete, è molto più comune usare indirizzi in forma alfanumerica, come “www.latimes.com” piuttosto che “63.241.84.101”, se non altro perché i primi sono molto più semplici da ricordare dei secondi. Le socket, però, utilizzano indirizzi in formato numerico (intesi come numeri di 32 bit), per cui, al fine di utilizzare il nome alfanumerico di un host per comporre un indirizzo di trasporto, occorre effettuare una conversione di formato.

In questo capitolo, con l'espressione *indirizzo numerico* si intende il numero di 32 bit utilizzato internamente dal sistema; la forma decimale puntata (63.241.84.101) non è un indirizzo in formato numerico; tant'è vero che anch'essa è soggetta a una conversione di natura aritmetica.



Il *Domain Name Service* (DNS) [20] è un servizio non centralizzato e organizzato in maniera gerarchica la cui funzione è principalmente quella di convertire nomi di host in indirizzi numerici e viceversa. Questo servizio viene interpellato

tramite la rete da una parte di sistema operativo che prende il nome di *resolver*. Per poter effettuare una conversione un programmatore non è quindi costretto a interagire con il DNS ma può limitarsi a usare il resolver, anche perché, diversamente, avrebbe comunque il problema di capire quale sia l'indirizzo del DNS server da utilizzare.

Il resolver, a fronte di una richiesta effettuata sulla base di un nome alfanumerico o di un indirizzo, restituisce tutte le informazioni relative all'host, tra cui il nome principale, la lista dei nomi secondari e la lista degli indirizzi IP a cui fa riferimento.

Non è garantito che ci sia una corrispondenza uno-a-uno tra i nomi e gli indirizzi IP. L'associazione è lasciata alla totale discrezione del gestore della rete; infatti in alcuni casi uno stesso nome potrebbe corrispondere più indirizzi (8 nel caso di www.cnn.com) oppure più nomi potrebbero risultare associati allo stesso indirizzo IP. Quest'ultimo caso è tipico per web server virtuali multipli gestiti da un singolo nodo. Questa non univocità non deve essere vista come un problema, ma come l'opportunità di accedere agli stessi contenuti attraverso più canali.

Esiste un equivalente alfanumerico dell'indirizzo IP 127.0.0.1, ed è localhost. Il resolver si occupa internamente della conversione tra i due.



4.4 Identificazione di una socket

Per poter erogare o fruire di un servizio occorre identificare in maniera univoca la socket per renderla raggiungibile, o raggiungerla, e poter creare il canale di comunicazione. Come già discusso nel Paragrafo 2.2.2 gli indirizzi al livello di trasporto risolvono questo problema. L'associazione di una socket a un indirizzo di trasporto prende il nome di *binding*.

I vari linguaggi di programmazione fanno assunzioni diverse sul formato da utilizzare per rappresentare un indirizzo a livello di trasporto. Alcuni, come il linguaggio C, massimizzano la compatibilità a scapito della semplicità d'uso; altri, invece, come Java, suppongono che il livello di rete faccia uso esclusivo di tecnologia IP, semplificando notevolmente la scrittura del codice a discapito della flessibilità di funzionamento in alcuni contesti.

4.5 Binding

L'operazione di binding può avvenire in modo esplicito o implicito.

Un binding esplicito viene richiesto appositamente dall'utente per associare una socket a un indirizzo di trasporto locale. Inoltre, quando il binding viene fatto in maniera esplicita il sistema operativo verifica che l'indirizzo di trasporto richiesto sia effettivamente disponibile; in caso contrario verrà generato un errore. Normalmente, l'associazione viene fatta solo a livello di porta però, nel caso sul nodo siano presenti più, schede di rete e si voglia ricevere, dati attraverso solo una di esse, è possibile effettuarla anche a livello di indirizzo IP. Nel caso in cui più schede di rete siano presenti e si faccia un bind generico a una porta, allora tutti gli indirizzi trasporto risulteranno allocati a un'unica socket.

Trovare una porta disponibile non è scontato, soprattutto all'interno di un sistema multiutente; per evitare continui tentativi fino a che non se ne trova una libera, è possibile, per convenzione, effettuare il binding sulla porta numero 0: in tal modo il sistema operativo assegnerà alla socket il primo indirizzo di trasporto non ancora allocato.



Le modalità byte-stream (TCP) e datagram (UDP) possiedono due sistemi di binding disgiunti. Infatti, se una socket effettua una bind esplicita alla porta 80 con protocollo TCP, la porta 80 con protocollo UDP risulterà ancora libera per essere utilizzata.

Il binding implicito avviene invece autonomamente da parte del sistema operativo all'atto dell'istituzione del canale di comunicazione o al primo invio di dati nel caso di una socket non connessa. Se il binding avviene in maniera implicita l'indirizzo di trasporto associato alla socket sarà dato dall'indirizzo IP della scheda di rete dalla quale transitano i dati e da una porta libera assegnata dal sistema operativo.

Quella di binding è un'operazione necessaria per tutte e due le socket che partecipano a una comunicazione. Il ricevente fa normalmente uso di binding esplicito e si rende disponibile, il chiamante può decidere se fare un binding esplicito e poi contattare l'altra socket oppure cominciare a inviare dati e lasciare il compito al sistema operativo. Questo dipende dal fatto che il canale di comunicazione è sempre bidirezionale, quindi, durante la sua creazione anche la socket che viene contattata deve sapere dove inviare la risposta.

4.6 Porte e servizi standard

Non tutti i numeri di porta possono essere trattati allo stesso modo, inoltre, i servizi vengono generalmente standardizzati su specifiche porte dalla IANA; quindi non è opportuno fare bind su porte completamente arbitrarie.

Questo non vuol dire che durante l'evoluzione delle tecnologie non ci siano state collisioni, anzi alcune di loro possono anche risultare piuttosto critiche da gestire.

Ad esempio, una molto diffusa architettura di file sharing adotta come suo standard interno le porte 4662 TCP e 4672 UDP, che però la IANA mette effettivamente a standard rispettivamente come *OrbitNet Message Service* (OMS) e *Remote File Access Server* (RFA).

La IANA suddivide le porte in tre gruppi a seconda del numero:

Well Known Ports

Sono le porte con numeri da 1 a 1023.

Queste porte non dovrebbero essere utilizzate se non precedentemente registrate presso la IANA. I servizi associati a questo gruppo svolgono funzioni importanti ai fini del buon funzionamento dell'rete.

Molto spesso possono essere allocate solo da utenti privilegiati (l'amministratore di sistema).

Registered Ports

Sono le porte con numeri da 1024 a 49151.

Queste porte non dovrebbero essere utilizzate se non precedentemente registrate presso la IANA. I servizi associati a questo gruppo sono proposti e sviluppati da organizzazioni esterne e singoli utenti; svolgono generalmente funzioni accessorie al buon funzionamento della rete. La IANA gestisce la loro standardizzazione come un servizio verso la comunità di Internet.

Dynamic Ports

Sono le porte con numeri da 49152 a 65535.

Possono essere usate liberamente dagli utenti a qualunque titolo.

Nonostante queste direttive, nella maggior parte dei sistemi operativi moderni un'operazione di bind alla porta 0 associa la socket a una porta superiore a 1023 senza distinzione tra gli ultimi due gruppi. 

Per lo stesso motivo visto prima, per cui porte byte-stream e datagram sono disgiunte per quanto riguarda il binding, lo standard prevede un'associazione separata per servizi associati a protocolli TCP e UDP. Molto spesso un servizio viene associato

alla stessa porta con tutte e due le modalità, ma esistono alcune eccezioni; ad esempio, la porta 514 è associata al servizio *shell* nella modalità TCP e *syslog* [40] in quella UDP.

Quando si intende allocare una porta standard è molto più conveniente usare il nome del servizio anziché il numero (e questo rende anche il software più flessibile in caso lo standard subisca variazioni). I sistemi operativi moderni possiedono un file di configurazione contenente un estratto delle tabelle di standardizzazione della IANA; per i sistemi UNIX è “/etc/services”, mentre per i sistemi Windows è “\WINDOWS\system32\drivers\etc\services”. I linguaggi di programmazione offrono spesso funzionalità che partendo dal nome di un servizio interrogano questo file di configurazione e ne estraggono la porta relativa.

4.7 Associazione di due socket

L’associazione di due socket porta queste ultime a lavorare in modalità connessa: vengono messe in relazione stretta e i sistemi operativi dei nodi ospitanti si accordano per stabilire un canale di comunicazione permanente tra loro.

Non è sempre strettamente necessario che due socket siano connesse tra loro per poter comunicare: lo è solo nel caso in cui le socket siano di tipo byte-stream.

Nel caso byte-stream l'associazione di due socket implica un accordo tra le due parti e deve essere collaborativa. Nel caso datagram la stessa operazione può essere considerata quasi unilaterale: ogni socket imposta nella sue strutture dati l'equivalente di una *destinazione preassegnata* per i pacchetti in uscita e una *sorgente accettabile* per quelli in ingresso.

Lo stato delle socket all'interno del sistema con associato il loro attuale stato (associato, in attesa o in fase di chiusura) può essere visualizzato facendo uso dei comandi netstat o lsof [41] (quest'ultimo, però, solo su sistemi UNIX piuttosto recenti).

4.7.1 Operazioni lato ricevente

Si considera come ricevente la socket che viene contattata al fine di creare un canale di comunicazione.

È opportuno non confondere il concetto di ricevente con quello di server (e chiamante con client). Il ricevente è l'entità che rimane in attesa e viene contattata dal chiamante, e cioè non corrisponde sempre con la fruizione del servizio, in altre parole, non è detto che sia sempre il client a effettuare la chiamata (si veda ad esempio il caso del servizio FTP nella modalità di funzionamento *attiva* [25]). Nel caso specifico, in seguito alla richiesta di trasferimento di un file, il client si mette in attesa di essere ri-contattato

dal server per poter creare un canale di comunicazione che verrà usato appositamente per il trasferimento dati.

Se ci si trova a operare con una socket di tipo byte-stream quest'ultima deve essere dichiarata come *disponibile* per ricevere richieste di connessione. Nel momento in cui una socket byte-stream viene dichiarata disponibile perde alcune caratteristiche, ad esempio non può più essere usata per effettuare una chiamata verso altre socket in attesa e per mandare e ricevere dati. Quando verrà contattato dal chiamante, il ricevente dovrà generare ogni volta una nuova socket su cui effettuare lo scambio di informazioni, mentre quella esistente continuerà a rimanere in attesa di altri chiamanti. Tutte le socket create in conseguenza a una chiamata condivideranno lo stesso indirizzo di trasporto di partenza ma saranno associate a indirizzi distinti come controparte.

Se invece la socket è di tipo datagram, il ricevente può mettersi immediatamente in condizione di ricevere dati perché l'associazione da parte del chiamante sarà lui inconsapevole. Al più, il ricevente potrebbe considerare l'idea di associarsi al chiamante come conseguenza della ricezione del primo pacchetto; in tal caso l'associazione avviene secondo le stesse modalità del chiamante.



Nonostante tutto, il numero di porta continua a essere un problema. Se si sta erogando o usufruendo di un servizio standard, la porta è nota a priori alle due entità (ammesso che ci sia preventivamente messi d'accordo sul servizio). Se, invece, si sta utilizzando una porta arbitraria (soprattutto se assegnata dal sistema) non esiste nessun modo per comunicarla al chiamante in maniera automatica; d'altronde, la connessione non è ancora stata instaurata. Ciò che si può fare durante lo sviluppo è utilizzare sempre la stessa porta oppure far stampare a video quella utilizzata dal ricevente e utilizzare il valore come parametro per il programma che implementa il chiamante.

4.7.2 Operazioni lato chiamante

Il lato chiamante è il caso più semplice; basterà creare una socket e poi richiedere al sistema operativo di associarla alla controparte. Dopo questa operazione le funzionalità di lettura e scrittura della socket risulteranno abilitate e i dati potranno fluire via rete.

4.8 Trasferimento dati

L'operazione di trasferimento dati tra due socket avviene in maniera diversa a seconda che le due socket siano connesse o meno. Nel caso di socket di tipo datagram è possibile implementare uno scambio di informazioni con una situazione mista; cioè, ricevere dati su una socket connessa anche se la sorgente non lo è, e viceversa.

4.8.1 Socket connesse

Per quanto riguarda le socket connesse l'invio dei dati non ha bisogno di specificare ogni volta la destinazione, in quanto questa informazione viene implicitamente e permanentemente svolta durante l'operazione di associazione.

Nel caso particolare in cui le due socket adottino anche una politica byte-stream, è di solito possibile utilizzare le stesse funzionalità usate per accedere ai normali canali di I/O.

4.8.2 Socket non connesse

Nel caso in cui la socket sia in modalità non connessa l'invio dei dati avviene specificando ogni volta l'indirizzo di trasporto della socket da raggiungere. La ricezione, di conseguenza,

viene sempre accompagnata da informazioni relative all'indirizzo di trasporto di chi li ha spediti. Questo fa sì che una stessa socket possa essere usata per scambiare informazioni con un numero arbitrario di entità in rete.

4.9 Chiusura del canale

La chiusura del canale può avvenire in maniera esplicita, tramite apposita richiesta o in maniera implicita, da parte del sistema operativo al termine del processo che gestisce il canale stesso. In generale, per evitare un uso ingiustificato delle risorse è opportuno effettuare sempre una chiusura esplicita della socket.

Particolare attenzione va prestata nel caso si stia implementando un server: se al termine dell'erogazione del servizio la socket utilizzata per lo scambio dei dati non viene chiusa continuerà a occupare risorse; quindi, dopo aver gestito un certo numero di client il processo non riuscirà a farsi allocare altri canali dal sistema operativo e rimarrà in uno stato di blocco.

Capitolo 5

Implementazione di sistemi client-server

In questo capitolo si vedrà come strutturare e concatenare tra loro le operazioni viste nel Capitolo 4 per ottenere i vari modelli di servizio descritti nel Capitolo 1.

Per prima cosa si analizzeranno i vincoli di cui tenere conto nel gestire i messaggi usati per trasferire informazioni da client a server e viceversa; successivamente, verranno presentate le sequenze di operazioni necessarie nei vari contesti al fine di ottenere un sistema client-server.

La struttura delle operazioni da eseguire verrà presentata tramite diagrammi a blocchi in cui ogni rettangolo rappresenta un'operazione; le frecce a tratto pieno indicheranno la sequenza temporale delle operazioni mentre le frecce tratteggiate verranno usate per sottolineare le interazioni tra client e server.

È importante notare che i diagrammi a blocchi qui presentati rappresentano una sequenza di operazioni e non hanno nessuna corrispondenza con gli automi discussi nel Capitolo 3, che descrivono invece uno scambio di messaggi. Gli automi possono essere al più visti come un dettaglio di quanto avviene all'interno di un'operazione di scambio dati tra client e server.



5.1 Formato dei messaggi inviati

Nel capitolo precedente ci si è limitati a parlare delle modalità con cui le socket inviano e ricevono dati; nulla è stato detto sul loro formato, se non che si tratta di sequenze di byte.

Il formato è un problema relativo, in quanto le specifiche di un protocollo prevedono anche il formato dei pacchetti: nell'implementare un sistema client-server, il formato dei dati è imposto a priori. La questione si pone in maniera più significativa nel momento in cui si è interessati a sviluppare un proprio protocollo e ci si trova a dover stabilire un lessico. In generale è necessario veicolare qualunque tipo di dato all'interno di una sequenza di byte scambiata in rete: ciò può essere effettuato senza problemi a patto di tenere conto di alcune limitazioni.

I contenuti di tipo binario la cui interpretazione non dipende né dal sistema operativo né dall'hardware sottostante ma da software sviluppato da terzi, quali ad esempio immagini, suoni e video, non costituiscono affatto un problema: possono essere inviati in rete così come memorizzati sul disco. I contenuti che possono invece essere fonte di problemi sono quelli in cui i tipi di dati trasportati assumono un significato per il programmatore: caratteri, stringhe, numeri e dati strutturati.

5.1.1 Caratteri

La trasmissione di caratteri non è generalmente fonte di problemi: grazie allo standard ASCII [42, 43] è possibile associare univocamente il valore di un byte a un carattere alfanumerico.

I sistemi operativi moderni, tuttavia, possono usare caratteri secondo la codifica unicode [44, 45] che fa uso di 32 bit per carattere e quindi solo un sottoinsieme di simboli sarà associabile a un singolo byte; fortunatamente, i linguaggi di programmazione mettono a disposizione una serie di funzioni per effettuare conversioni tra i formati.

5.1.2 Stringhe

Le stringhe sono sequenze di caratteri, quindi valgono le stesse considerazioni viste nel paragrafo precedente.

Un problema da non sottovalutare quando si manipolano stringhe è il formato del cosiddetto *ritorno a capo*, o *new line*. Si tratta di una sequenza di caratteri che istruiscono il sistema operativo, quando la stringa viene rappresentata a video, di effettuare un ritorno a capo. Dal punto di vista del risultato grafico questa operazione può comporsi di due azioni: tornare all'inizio della riga (*Carriage Return* o CR) e passare alla riga successiva (*Line Feed* o LF). Disgraziatamente, ogni sistema operativo adotta una propria convenzione: per i sistemi UNIX (compresi Linux, FreeBSD e MacOS X) il ritorno a capo è costituito dal solo carattere LF; per la maggior parte degli altri sistemi (tra cui Windows, MS-DOS, OS/2 e Symbian OS) è costituito dalla sequenza CR e LF; infine, per una minoranza di sistemi oramai poco usati (tra i quali MacOS versione 9) solo dal carattere CR. Ogni protocollo dovrebbe specificare chiaramente il formato del ritorno a capo da usare quando si inviano messaggi contenenti del testo.

Nella maggioranza dei casi, per aumentare il più possibile la compatibilità, si sceglie di adottare la convenzione dei sistemi non UNIX e terminare la stringa con la sequenza CR+LF; si vedano, ad esempio, le specifiche di HTTP [11], SMTP [22] e POP [23].



Come ultima nota, occorre prestare attenzione quando si converte un buffer in una stringa: di norma, non si può

ipotizzare che la ricezione di una sequenza di byte dalla rete riempia il buffer, per cui è sempre bene tenere conto del numero di byte letti per identificare l'inizio e la fine dei dati.

Disgraziatamente, la flessibilità sul formato dei dati si ferma alle stringhe; si vedrà ora che è in generale un azzardo trasferire dati in rete senza prima stabilire un formato byte per-byte oppure, più semplicemente, convertire l'informazione nella sua rappresentazione alfanumerica.

5.1.3 Valori numerici

Ricadono in questa categoria, numeri interi, numeri in virgola mobile, date, puntatori (se supportati dal linguaggio) e qualunque altra informazione numerica che in memoria possa occupare più di 8 bit.

L'hardware del calcolatore (la CPU) stabilisce in quale sequenza i byte che compongono un valore numerico vengono scritti all'interno della memoria; ovviamente, ogni CPU è libera di usare un proprio formato per estensione e ordine dei byte. Di conseguenza, mandare una informazione numerica via rete non offre nessuna garanzia di preservare il valore trasmesso, a meno che le CPU sulle quali vengono eseguiti client e server non siano compatibili; cosa assolutamente non predibile.

Per ovviare a questo inconveniente è necessario definire alcune funzioni per la codifica da e verso un formato stabilito per il

trasferimento, riordinando i byte, oppure convertire il numero in una stringa contenete la sua rappresentazione alfanumerica, e inviarlo come tale.

A volte, per aumentare l'efficienza, è consigliabile convertire valori numerici facendo uso di una rappresentazione esadecimale; in tal modo verranno usati mediamente meno caratteri e sarà possibile fissare a priori un numero di cifre da utilizzare, dando quindi al risultato una dimensione fissa.

5.1.4 Dati strutturati

Con dati strutturati si intendono tutti i raggruppamenti di informazioni che all'interno dei programmi definiscono dei tipi di dati estesi, quali ad esempio array, strutture e classi.

Come per i valori numerici, non è possibile inviare in rete array o strutture così come sono contenute nella memoria locale; tuttavia la conversione a un formato più consono non è propriamente banale e richiede che il programmatore implementi alcune funzioni apposite per ogni tipo di classe o per tipo di contenuto dell'array. Tali funzioni, seguendo l'approccio più semplice possibile, dovranno mettere in sequenza il contenuto dei vari campi all'interno di un array di byte; se i campi non hanno una lunghezza fissa dovranno essere intervallati da delimitatori. Un'alternativa può essere quella di trasformare il dato strutturato in un documento XML [46],

che comunque è un'informazione di tipo testuale, quindi una stringa, e inviarlo in tale forma.

Per quanto riguarda le classi, invece, la situazione è ancora più delicata, perché queste includono anche codice eseguibile; inoltre, non è detto che un'istanza venga memorizzata in maniera continuativa all'interno della memoria centrale. Tuttavia, i linguaggi moderni prevedono la possibilità di effettuare, su classi con certi requisiti, una *serializzazione* dei contenuti, per cui l'istanza di una classe può essere trasformata in una sequenza di byte, inviata via rete e poi *deserializzata* dalla controparte al fine di ottenere una copia esatta dell'oggetto di partenza.

5.2 Gestione dei messaggi

La gestione dei messaggi deve tenere conto del fatto che a seconda della modalità di trasferimento dati i messaggi potrebbero essere uniti tra loro o spezzati in più parti.

È sbagliato pensare che in qualunque caso a un'operazione di scrittura su una socket corrisponda una e una sola operazione di lettura. Questo è vero solo nel caso di socket di tipo datagram, ammesso che non vengano persi pacchetti.



Quando il mittente manda dati più velocemente di quanto il sorgente è in grado di leggerli si osservano fenomeni di accodamento: con socket di tipo byte-stream più messaggi verranno ricevuti in una volta sola nella prossima operazione di lettura; con socket datagram, invece, grazie alla diversa gestione effettuata dal sistema operativo, la ricezione avviene comunque un messaggio alla volta. La frammentazione dei messaggi si può presentare indipendente da eventuali accorgimenti presi dal programmatore e dipende dalla configurazione degli apparati di rete attraversati dai pacchetti nonché dall'implementazione dello stack di protocolli del sistema operativo.

Esistono varie strategie per poter essere sicuri di ricevere correttamente i messaggi dalla rete nonostante siano necessarie più operazioni di lettura e tutte coinvolgono il lessico dei messaggi: è possibile utilizzare messaggi di dimensione fissa, delimitare il termine di un messaggio o preporre un valore numerico riportante la sua dimensione. Ogni metodo ha, ovviamente, i suoi pro e contro. Il primo è molto semplice da implementare ma decisamente poco flessibile: la delimitazione del termine di un messaggio obbliga il programma a leggere dalla socket un byte alla volta (con conseguente calo di prestazioni); infine, in caso di messaggi molto brevi, indicare la lunghezza potrebbe portare a uno spreco di risorse.

Un'alternativa è data anche dalla gestione di un buffer intermedio. La gestione di un buffer intermedio prevede l'allocazione di una zona di memoria relativamente grande all'interno della quale depositare i dati in arrivo dalla rete e

in cui tenere traccia, tramite contatori, dei dati che devono essere ancora forniti all'applicazione. Nel momento in cui l'applicazione richiede un messaggio, preleva i dati dal buffer e aggiorna i contatori; quando il buffer è vuoto la zona di memoria viene nuovamente riempita prelevando le informazioni dalla socket.

5.3 Implementazione di un client

L'implementazione di un client è fondamentalmente semplice: esso deve infatti creare una socket, associarla eventualmente alla socket del server, scambiare i dati per la fruizione del servizio e infine chiudere il canale. La sequenza di queste operazioni viene descritta nella Figura 5.1, in cui sono rappresentate le due varianti facenti uso di socket con connessione (a) e senza connessione (b). Come è possibile notare, si tratta delle operazioni viste nel capitolo precedente a cui si è data una sequenza logica e temporale.

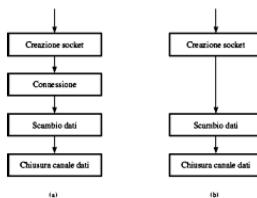


Figura 5.1 Diagrammi delle operazioni di un client con socket connesse (a) e non connesse (b).

Durante un processo di sviluppo è molto più comune richiedere ai programmati l'implementazione di un client piuttosto che di un server: quest'ultimo viene generalmente pianificato e realizzato all'inizio del progetto e difficilmente modificato in seguito, anche per garantire compatibilità nel tempo. I client, invece, devono soddisfare le necessità degli utenti finali e possono essere soggetti a frequenti aggiornamenti.



5.4 Implementazione di un server iterativo

Il diagramma a blocchi delle operazioni svolte da un server iterativo (Paragrafo 1.1) è proposto nella Figura 5.2, dove sono rappresentate due varianti: una con socket connesse (a) e l'altra non connesse (b).

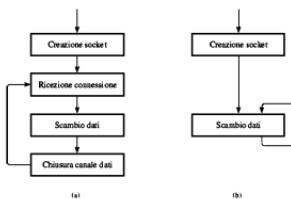


Figura 5.2 Diagrammi delle operazioni di un server iterativo con socket connesse (a) e non connesse (b).

5.4.1 Server iterativo con socket connesse

Come si può notare, osservando la Figura 5.2(a), la prima differenza visibile rispetto al diagramma relativo al client, nella Figura 5.1 (a), è che in questo caso ci si trova in presenza di un ciclo. In generale un server non viene implementato per soddisfare le richieste di un solo client e poi terminare: una volta chiuso il canale dati con il client attualmente in servizio ci si deve porre nuovamente in attesa di una nuova richiesta di connessione.

È possibile mettere fianco a fianco il diagramma del server con quello del client e individuare una serie di relazioni tra le operazioni svolte dai due. Come si può vedere nella Figura 5.3, la richiesta di connessione del client interagisce con il server in maniera monodirezionale (è il client a prendere l'iniziativa), mentre lo scambio dati viene effettuato in maniera bidirezionale. Nel momento in cui lo scambio dei messaggi termina, quando cioè le implementazioni degli automi che gestiscono il protocollo arrivano entrambe a uno stato finale, sia il client che il server chiudono il canale di comunicazione. Il client terminerà, mentre il server tornerà in attesa della prossima richiesta di servizio.

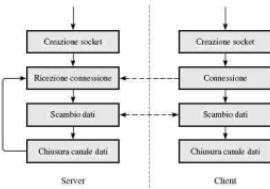


Figura 5.3 Diagramma delle operazioni congiunte di client e server iterativo con socket connesse.

5.4.2 Server iterativo con socket non connesse

Nel caso di un server iterativo facente uso di socket non connesse il ciclo per l'erogazione del servizio non viene effettuato tornando ad aspettare una richiesta ma si svolge sullo scambio dei dati: il server si pone semplicemente in attesa di ricevere un messaggio dopo l'altro dalla rete, essendo la socket non connessa non esiste un canale dati dedicato a un client che necessita di essere chiuso. Mettendo in relazione i diagrammi delle Figure 5.2(b) e 5.1(b) si ottiene la Figura 5.4.

Anche se non viene più effettuata l'operazione di connessione questo non vuol dire che non esiste più una coda di attesa: prima i client rimanevano in coda per attendere di essere connessi, ora per attendere il messaggio di risposta dal server.



5.5 Implementazione di un server concorrente

Un server concorrente, come già discusso nel Paragrafo 1.2, presenta il problema di dover gestire in maniera parallela un certo numero di eventi che possono verificarsi in qualunque ordine. Il relativo diagramma a blocchi è rappresentato nella Figura 5.5.

In questo caso non ha senso parlare di una variante con socket non connesse; infatti, se dallo schema della Figura 5.5 vengono eliminate le operazioni di connessione e chiusura del canale, potrà seguire solo uno scambio dati e lo stato di attesa concorrente sarà inutile, per cui il diagramma risulterà identico a quello di un server iterativo senza connessione.

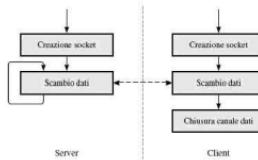


Figura 5.4 Diagramma delle operazioni congruenti di client e server iterativo con socket non connesse.

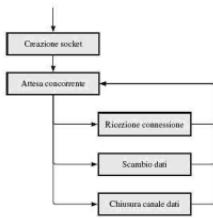


Figura 5.5 Diagramma delle operazioni di un server concorrente.

Le difficoltà implementative risiedono tutte nel blocco che effettua l'attesa concorrente; infatti, il programma dovrà essere in grado di gestire indiscriminatamente la richiesta di connessione da parte di un nuovo client, un messaggio (scambio dati) proveniente da un client che ha effettuato una connessione precedentemente oppure la disconnessione di un client che ha terminato la fruizione del servizio. Inoltre, come già puntualizzato, ogni client deve avere un suo contesto all'interno della comunicazione con il server, cioè, per ogni client il server deve mantenere un automa separato e gestire una sequenza di messaggi specifica di quel client.

Affiancando lo schema a blocchi nella Figura 5.5 a quello di un client si ottiene la Figura 5.6.

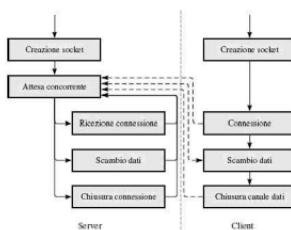


Figura 5.6 Diagramma delle operazioni congiunte di client e server concorrente.

Nonostante le evidenti difficoltà implementative, questo tipo di server si rende necessario nel momento in cui client diversi devono accedere contemporaneamente a informazioni comuni avendo anche la possibilità di modificarle. Il server ha il compito di rendere disponibili i dati che verranno manipolati tramite un protocollo di accesso; un'applicazione tipica è rappresentata dai *DataBase Management System* (DBMS) come *MySQL* [47] e *PostgreSQL* [48].

5.6 Implementazione di un server multiprocesso/multithread

Un sistema operativo è un'architettura software che si occupa di gestire e rendere accessibili le risorse hardware di un calcolatore. Tra le varie modalità si incontrano spesso sistemi operativi definiti *multitask* (dall'inglese *task* , attività assegnata), dove cioè più entità software fanno un uso concorrente delle risorse. Un task all'interno di un sistema operativo prende storicamente il nome di *processo* . Un processo può essere visto come un programma in esecuzione associato a un contesto (lo stato dei suoi dati interni). I processi all'interno di un calcolatore possono essere considerati alla stregua di organismi viventi che condividono uno stesso spazio: ognuno vive di vita propria ma interagisce con gli altri e compete per le risorse comuni.

I linguaggi di programmazione più evoluti danno la possibilità al programmatore di gestire flussi esecutivi paralleli; a volte però, a seconda del sistema operativo ospite (Windows piuttosto che UNIX) o del linguaggio di programmazione utilizzato (Java piuttosto che C), queste esecuzioni parallele possono essere gestite come processi oppure *thread*.

Un thread si differenzia da un processo in quanto si colloca a una granularità più fine; in generale un processo potrebbe suddividere internamente la sua esecuzione in thread. La differenza fondamentale tra processi e thread è che, mentre i processi sono sempre entità distinte tra loro, i thread appartenenti a uno stesso processo condividono un unico spazio di indirizzamento o, più semplicemente, hanno in comune variabili e codice. Avere informazioni condivise tra i thread rende possibile implementare con questo modello una serie di servizi che precedentemente, avendo a disposizione solo processi, dovevano essere erogati con un modello concorrente. Per questo motivo alcuni linguaggi di nuova concezione e con una gestione dei thread molto sviluppata (come ad esempio Java) non forniscono al programmatore gli strumenti necessari per implementare un server concorrente perché sarebbe l'equivalente *più complicato* di ciò che si potrebbe ottenere con un insieme di thread.

Un server multiprocesso, già descritto nel Paragrafo 1.3, si compone di più unità esecutive: una (il server principale) si occupa di gestire le richieste di servizio dei client mentre le altre (i server dedicati) si prendono carico dell'erogazione del servizio

secondo un modello iterativo. Il diagramma blocchi per questo tipo di server è rappresentato nella Figura 5.7.

Paradossalmente, nonostante la presenza di più unità esecutive, questo modello risulta piuttosto semplice da implementare; infatti, al di là della creazione di un nuovo processo, la sequenza delle operazioni è del tutto identica al caso iterativo (Figura 5.2). La difficoltà della gestione del processo (o del thread) creato dipende principalmente dal sistema operativo e dal linguaggio utilizzati.

Il diagramma ottenuto affiancando un client a un server multiprocesso è presentato nella Figura 5.8.

Diversamente da quanto avviene nel caso di un server concorrente, potrebbe avere senso parlare di socket non connesse. Il diagramma relativo a client e server multiprocesso facenti uso di socket non connesse è rappresentato nella Figura 5.9.

Con socket non connesse è comunque possibile utilizzare server dedicati per suddividere il carico di lavoro richiesto al sistema; tuttavia, in generale, l'uso di socket non connesse complica la gestione del codice senza dare particolari benefici.

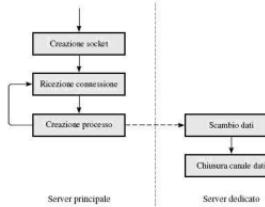


Figura 5.7 Diagramma delle operazioni di un server multiprocesso con socket connesse.

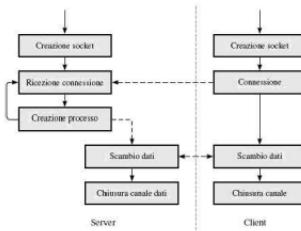


Figura 5.8 Diagramma delle operazioni congruente di client e server multiprocesso con socket connesse.

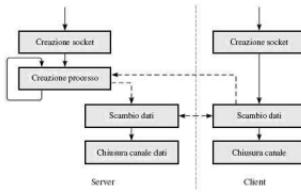


Figura 5.9 Diagramma delle operazioni congruente di client e server multiprocesso con socket non connesse.

In manzitutto, non essendoci una fase di connessione, il server principale è in grado di creare un'istanza di un server dedicato

solo nel momento in cui viene ricevuto il primo messaggio: il server principale gestisce il primo messaggio e il server dedicato tutti gli altri. Con questo tipo di approccio occorre fare attenzione a non creare inconsistenza nei dati e garantire la corretta gestione dei messaggi. Inoltre, il server dedicato non potrà utilizzare, per erogare il servizio al proprio client, la stessa socket usata dal server, altrimenti i messaggi verranno ricevuti da tutti e due i processi. Il server dedicato ne dovrà creare un'altra come succede anche nel caso in cui si fa uso di *accept*. Il fatto di dover aprire una seconda socket per ogni client fa perdere parte dei benefici dati dalle socket non connesse, e cioè che più mittenti possano inviare messaggi a un solo ricevente tramite la medesima socket. Di fatto, si è ottenuto un risultato identico a quello prodotto nel caso di socket connesse, solo che il programmatore è costretto a fare i conti con un canale non affidabile.

Infine, si fa notare che, essendo le socket non connesse, il server dedicato non ha modo di sapere se il client è ancora funzionante oppure no, a meno di non mandare messaggi periodici per sincerarsene. In un caso il server potrebbe essere ancora attivo nonostante il client abbia terminato la sue esecuzione, nell'altro, l'invio di messaggi fa un uso apparentemente ingiustificato delle risorse di rete.

Per i motivi appena elencati, in questo testo si è scelto di non esaminare nel dettaglio server multiprocesso basati su socket non connesse.

Capitolo 6

Accesso alle risorse web

All'interno di questo capitolo verranno analizzate le strategie più comuni per accedere a uno dei principali servizi erogati su Internet: il Web. Nel farlo, non verranno più discusse le problematiche di connessione con il server, ma ci si concentrerà sullo scambio di messaggi necessario per la fruizione del servizio.

6.1 Servizio web

Il Web è oramai il servizio principe di Internet, tant'è vero che per i non addetti ai lavori i due termini tendono, erroneamente, a confondersi.

Diversamente da quanto spesso si pensa, questo servizio non è funzionale a *scaricare e visualizzare pagine html*, ma piuttosto all'interrogazione di risorse a cui potrebbero essere associate tali pagine.

contenuti. Nell'uso comune molte risorse sono associate a contenuti testuali che possono essere visualizzati in formato grafico da un client specializzato (*web browser*).

La fruizione del servizio nella sua forma di base è molto semplice: il client si collega al server e richiede di accedere a una risorsa identificata tramite un URI, il server fornisce una risposta eventualmente corredata di un contenuto e chiude la connessione. Il servizio appena descritto è standardizzato sulla porta 80 e il protocollo adottato a livello applicazione è HTTP [11].

Attenzione! Non è detto che la risorsa:



1. sia associata a una pagina web
2. sia localizzata sullo stesso nodo del server
3. esista fisicamente al momento della connessione.

Il server che si occupa di erogare il servizio prende generalmente il nome di *web server*. Esistono implementazioni di web server liberamente disponibili e tra queste la più diffusa è sicuramente *apache* [1].

6.2 Protocollo HTTP

Il protocollo HTTP definisce la struttura dei messaggi con cui effettuare l'accesso a una risorsa. Questo protocollo prevede solo due tipi di messaggi: uno per la richiesta e l'altro per la risposta; entrambi sono composti da una serie di stringhe ASCII terminate dalla sequenza “\r\n” seguite, eventualmente, dal contenuto associato o da associare alla risorsa.

In generale, un messaggio è composto da:

1. una stringa iniziale
2. un header, in cui compaiono una serie di direttive (opzioni)
3. una stringa vuota per segnalare la terminazione dell'header
4. il contenuto in formato binario.

Si noti che è stata fatta una distinzione netta tra risorsa e contenuto a essa associato. Nel caso, ad esempio, di una pagina web dinamica, come vedremo, il contenuto che viene restituito al client non è la pagina (risorsa) identificata dall'URL bensì il risultato della sua elaborazione.



La stringa iniziale distingue un messaggio di richiesta da uno di risposta.

Le opzioni che compongono l'header hanno un formato di tipo “campo: valore” e definiscono i parametri per il funzionamento di client e server; dove non diversamente specificato le opzioni sono tutte facoltative e, dove necessario, possono anche comparire più volte. In [11] viene definito l’insieme minimo dei parametri da gestire, tuttavia, client o server possono definire a piacimento nuove opzioni; campi ricevuti ma non supportati vengono ignorati da ambo le parti.

Infine, il contenuto del messaggio, che può presentarsi anche in forma binaria, è costituito dai dati scambiati; nel caso specifico di una risposta potrebbe trattarsi del testo di una pagina web, di un’immagine o di un file.

Il contenuto non è esclusiva di un messaggio di risposta: un messaggio di richiesta potrebbe, infatti, portare con sé alcuni dati per permettere alla risorsa di espletare un servizio, come l’aggiunta di un messaggio a un forum o l’invio di dati di fatturazione a un sito di e-commerce.

6.2.1 Messaggio di richiesta

In un messaggio di richiesta la linea iniziale si compone di tre elementi intervallati da spazi e ha il formato che segue:

METODO URL VERSIONE

Il metodo di accesso definisce la modalità con cui si richiede di accedere all'URL specificato come secondo elemento; i vari metodi di accesso si vedranno in dettaglio nel prossimo paragrafo.

L'URL può essere espresso in maniera sia relativa che assoluta. In un URL relativo vengono omessi il protocollo e il nome dell'host perché il primo è sottinteso e il secondo coincide con il nodo verso il quale è stato aperto il canale di comunicazione; tuttavia, lo standard raccomanda di usare sempre URL assoluti (completi), anche se a volte ridondanti, per favorire le operazioni svolte dal server.

Il terzo elemento, che ha un formato del tipo “HTTP/X.Y”, specifica la versione del protocollo HTTP implementata; serve principalmente a dare un'indicazione al server su quali opzioni il client è in grado di riconoscere e sulle modalità gestione del canale.



Nonostante l'ultima versione di HTTP sia la 1.1, è preferibile usare, almeno inizialmente, la stringa “HTTP/1.0” per rendere più semplice la gestione della socket. La versione 1.0 prevede, se non diversamente specificato, la chiusura del canale appena terminato il messaggio di risposta, mentre la versione 1.1 ha come comportamento predefinito quello di tenere la connessione aperta fino allo scadere di un timeout per lasciare al client la possibilità di inviare altre richieste. Il secondo approccio permette al client di non doversi collegare nuovamente, ma rende la chiusura della socket un evento potenzialmente asincrono, portando a un codice più complesso.

Le seguenti opzioni sono tra quelle di uso più comune all'interno di un messaggio di richiesta.

Host

Specifica quale indirizzo e porta sono stati utilizzati per contattare il server; in questo modo è possibile per un nodo implementare più web server virtuali i cui nomi alfanumerici sono tutti associati allo stesso indirizzo IP. Il valore di questo campo deve essere l'indirizzo a livello di trasporto del servizio; il nome del nodo può essere espresso in formato alfanumerico o in forma decimale puntata, il numero di porta non è necessario se si usa quella standard (80). Questa opzione diventa obbligatoria con HTTP 1.1 e superiori.

Content-Type

Specifica il formato del contenuto. Il valore di questo campo deve essere un identificatore secondo il formato MIME [49]; è obbligatorio solo se viene inviato al server del contenuto.

Content-Length

Questa opzione è obbligatoria solo se vengono inviati contenuti al server. Specifica la dimensione in byte dei dati che seguiranno l'header.

User-Agent

Fornisce al server informazioni riguardo il client: i tipi e le versioni di software e sistema operativo; queste informazioni possono essere usate per personalizzare il servizio o per raccogliere statistiche. Il valore di questo campo può essere una qualsiasi stringa.

6.2.2 Messaggio di risposta

In un messaggio di risposta la linea iniziale si compone di tre elementi intervallati da spazi e ha un formato come segue:

VERSIONE CODICE COMMENTO

La versione indicata come primo elemento svolge la stessa funzione del suo equivalente nella linea iniziale di un messaggio di richiesta.

Il codice presente come secondo elemento identifica il risultato della richiesta: se è andata a buon fine oppure no e perché; si tratta di un numero decimale di tre cifre, delle quali la prima identifica una classe di risposte e le altre due uno specifico risultato. Le possibili classi di risposta sono elencate nella Tabella 6.1.

Codice	Significato della classe
1xx	Implementata solo da HTTP 1.1 e superiori viene usata dal server per mandare informazioni addizionali al client; non viene seguita dal contenuto ma da un ulteriore messaggio di risposta che potrebbe a sua volta ricadere in questa stessa classe.
2xx	La richiesta è andata a buon fine.
3xx	La risorsa esiste ma deve essere raggiunta tramite un URL diverso.
4xx	La richiesta del client non è appropriata: la risorsa non esiste, il contenuto non può essere acceduto o il formato del messaggio di richiesta non è corretto.
5xx	Si è verificato un errore all'interno del server.

Tabella 6.1 Significato della prima cifra del codice in un messaggio di risposta.

Tra tutti i possibili codici di risposta, quelli più comuni da gestire sono i seguenti.

200 (OK)

La richiesta è andata a buon fine.

301 (Moved Permanently)

La risorsa deve essere raggiunta tramite un URL diverso, che verrà specificato nell'header tramite l'opzione *Location*.

400 (Bad Request)

Il messaggio di richiesta non è sintatticamente corretto.

403 (Forbidden)

Il client non ha il permesso di accedere alla risorsa.

404 (Not Found)

La risorsa non esiste o non è stata trovata.

Il terzo elemento è una e una stringa di commento fornita al client; in caso di errore dà una descrizione del perché questo si è verificato.

Le seguenti opzioni sono tra quelle di uso più comune all'interno di un messaggio di risposta.

Content-Type

Ha lo stesso significato che assume nell'header di un messaggio di richiesta, ma non è obbligatoria. Se non viene fornita, l'onere di identificare il tipo di dato spetta al client.

Content-Length

In un messaggio di risposta questa opzione è facoltativa. Se non viene specificata, il contenuto si intende finito con la chiusura del canale da parte del server.

Last-Modified

Specifica quando il contenuto è stato modificato l'ultima volta; questa informazione può essere usata dal client per capire se usare una copia nella cache locale anziché trasferire nuovamente il contenuto.

Server

Fornisce al client informazioni riguardo il server: i tipi e le versioni di software e sistema operativo. Il valore di questo campo non ha un formato prefissato.

6.3 Accesso a una risorsa

Nell'effettuare la richiesta di accesso a una risorsa, l'unico aspetto che rimane da chiarire è come indicare il metodo di accesso. Come si può osservare nella Tabella 6.2, le specifiche di HTTP identificano una serie di metodi di accesso per permettere non solo la consultazione ma anche la manipolazione delle risorse, tuttavia, nella maggioranza dei casi solo i metodi di consultazione sono davvero di interesse; per questo motivo all'interno di questo testo ci si limiterà a trattare i metodi HEAD, GET e POST.

Metodo di accesso	Operazione associata
OPTIONS	Richiede al server quali siano le modalità di fruizione di una certa risorsa, tra le quali anche i metodi di accesso utilizzabili, che verranno elencati tramite l'opzione <i>Allow</i> .
GET	Accede al contenuto associato alla risorsa.
HEAD	Identico al metodo GET, ma il messaggio di ritorno è privo del contenuto; può essere utile

Metodo di accesso	Operazione associata
	per conoscere la dimensione dei dati senza effettivamente richiederne l'invio sulla rete.
POST	Accede al contenuto fornendo alla risorsa informazioni aggiuntive per il suo funzionamento.
PUT	Richiede che il contenuto che segue l'header venga memorizzato all'interno della risorsa specificata.
DELETE	Richiede la cancellazione della risorsa.
TRACE	Il server risponde con un messaggio il cui contenuto è il messaggio di richiesta; viene utilizzato per fare un test sulla raggiungibilità di un sito attraverso uno o più proxy.
CONNECT	Questo metodo può essere utilizzato solo con un proxy: richiede di deviare la connessione corrente verso un altro indirizzo di trasporto usando il proxy come ponte (<i>tunneling</i>).

Tabella 6.2 Metodi di accesso a una risorsa.

6.3.1 HEAD

L'automa relativo al protocollo implementato da un client che emette un messaggio di richiesta HEAD e ne gestisce la relativa risposta è rappresentato nella Figura 6.1. Come già presentato all'interno della Tabella 6.2, lo scopo del metodo di accesso HEAD è quello di ottenere dal server solo il risultato della richiesta e l'header.

Il client si trova inizialmente nello stato H_1 e manda verso il server la stringa iniziale, che comincerà con “HEAD”, una serie facoltativa di opzioni e una stringa vuota per poi trovarsi nello stato H_3 . Successivamente, il client dovrà accettare il messaggio di risposta che, indipendentemente dal codice del risultato, sarà composto a sua volta da una stringa recante il risultato della richiesta, una serie facoltativa di opzioni e una stringa vuota. Alla fine del ciclo il client si troverà di nuovo nello stato H_1 e potrà effettuare una successiva richiesta.

È possibile fare una prova con il metodo HEAD usando il comando *telnet*. Il comando telnet è un client generico in grado di scambiare stringhe attraverso una socket; sul video verranno visualizzati i messaggi ricevuti dal server e sulla socket saranno inviate le stringhe digitate dall'utente. Telnet accetta come parametri un indirizzo di trasporto indicando prima l'indirizzo IP del nodo e poi il numero della porta; se il numero di porta

non viene fornito sarà utilizzata la porta standard per il servizio telnet (23). In questo caso è possibile collegarsi a un qualunque sito web sulla porta 80 e digitare un messaggio di richiesta HEAD nella sua forma più semplice seguito da due ritorni a capo (il secondo per la stringa vuota di terminazione dell'header).

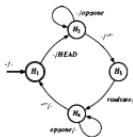


Figura 6.1 Automa degli stati di un client che fa uso del protocollo HTTP e invia richieste HEAD.

```
telnet www.lila.it 80
```

Seguiranno i messaggi di avvenuta connessione.

```
Trying 62.149.130.198...
Connected to www.lila.it.
Escape character is '^]'.
```

E quindi sarà possibile inviare il messaggio (seguito da due ritorni a capo)

```
HEAD http://www.lila.it/ HTTP/1.0.
```

al quale il server risponderà con l'header della risposta, di cui si riportano solo le parti più significative.

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/6.0
Last-Modified: Tue, 30 Sep 2008 10:25:37 GMT
```

```
Content-Type: text/html; charset=utf-8
Content-Length: 21723
```

Si noti come, nonostante non venga trasmesso il contenuto, sono comunque presenti nell'header le opzioni *Content-Type* e *Content-Length*.

Se il sistema operativo è Windows, il testo digitato potrebbe non essere visualizzato per via dell'implementazione del client. È facile però trovare software di libera distribuzione con cui sostituire il comando telnet di sistema, come puttytel [50] o il telnets contenuto nel pacchetto cygwin [51].



6.3.2 GET

Nella Figura 6.2 viene esteso l'automa della Figura 6.1 in modo da inviare anche richieste GET. Come si può notare, nonostante sia raddoppiato il numero degli stati, la differenza concettuale rispetto al caso precedente è piuttosto ridotta: lo stato G_4 , alla ricezione della stringa vuota, anziché fare come lo stato H_4 e portarsi nello stato iniziale HG_1 , si porta nello stato G_5 . Nello stato G_5 l'automa si predisponde a ricevere un messaggio corrispondente al contenuto per la dimensione specificata dall'opzione *Content-Length*.

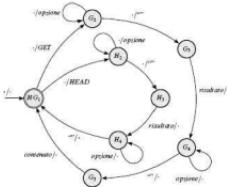


Figura 6.2 Automa degli stati di un client che fa uso del protocollo HTTP e invia richieste HEAD e GET.

Per effettuare una prova è possibile usare ancora una volta telnet, ma in questo caso con GET anziché HEAD; in questo modo all'header seguirà il contenuto della pagina principale del sito.

6.4 Contenuti statici e dinamici

I contenuti di cui è possibile fruire tramite il protocollo HTTP si dividono in due categorie: statici e dinamici. Il contenuto associato a una risorsa si dice dinamico se varia nel tempo o in base al contesto; un contenuto statico, invece, è sempre lo stesso indipendentemente dalle condizioni. Un esempio di risorsa con contenuto dinamico è, ad esempio, l'URL di un motore di ricerca: il suo contenuto sarà ogni volta diverso a seconda della ricerca effettuata. I parametri in base ai quali cambia il contenuto di una risorsa possono essere indipendenti dall'utente oppure sotto il suo controllo, in quest'ultimo caso all'accesso il client fornisce alcuni parametri alla risorsa per istruirla su come generare il contenuto.



Come già accennato in precedenza, quando si parla di pagine web si fa spesso la distinzione tra pagine statiche e dinamiche. Le prime sono file che, quando referenziati, vengono trasmessi al client; le seconde veri e propri programmi che vengono eseguiti dal server, il contenuto mandato al client è l'output ottenuto dall'esecuzione.

È possibile fornire parametri di funzionamento a una risorsa dinamica tramite il metodo GET aggiungendoli in coda all'URL: viene aggiunto un carattere “?”, seguito da una o più coppie “parametro=valore” intervallate da caratteri “&”, come negli esempi che seguono.

`http://www.unimi.it/chiedove/
schedaPersonaXML.jsp?matricola=16088`

`http://www.google.it/search?hl=en&q=client-server`

Nel primo caso si accede alla risorsa *schedaPersonaXML.jsp* dicendo che si è interessati alla scheda del dipendente con matricola 16088, nel secondo si chiede al server *google.it* attraverso la sua risorsa *search* di costruire una pagina in inglese (*hl=en*) in cui compaiano dei link a pagine web nelle quali sia presente il termine “client-server” (*q=client-server*).

Questa strategia, pur essendo molto semplice da implementare, ha il difetto di rendere gli URL illeggibili ed eventuali bookmark di un web browser potrebbero non essere più validi a distanza di tempo, ad esempio nel caso di un sito di e-commerce, perché un certo articolo potrebbe non essere più disponibile. L'alternativa è usare il metodo di accesso POST.

6.4.1 POST

Il metodo di accesso POST viene usato da un web browser, ad esempio, nel momento in cui l'utente sottmette un form all'interno di una pagina web. I parametri che con il metodo GET andrebbero in coda all'URL vengono invece spediti nello stesso identico formato come contenuto del messaggio.

Aggiungendo all'automa nella Figura 6.2 gli stati relativi all'invio di una richiesta POST, si ottiene la Figura 6.3. Come si può notare, è stato sufficiente aggiungere una sequenza di stati P_2 e P_3 che, dopo aver inviato al server il contenuto della richiesta, rincongiungono l'esecuzione allo stato G_3 , quello in cui il client è in attesa di ricevere un messaggio di risposta comprensivo di dati finali.

Per poter usare POST, però, le opzioni *Content-Type* e *Content-Length* diventano necessarie; la prima, si ricordi, identifica il tipo del contenuto che verrà inviato dopo l'header e la seconda la sua lunghezza. In particolare, per quanto riguarda il tipo di contenuto, i due valori più tipici sono “*application/*

x-www-form-urlencoded”, se vengono inviate coppie del tipo parametro/valore come nel caso della sottomissione di un form, e “*application/octet-stream*”, se vengono inviati dati binari come nel caso dell’aggiunta di una immagine in un archivio.

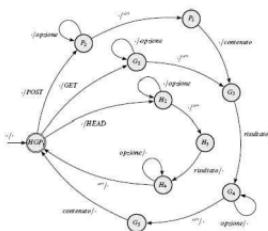


Figura 6.3 Automa degli stati di un client per la consultazione di risorse tramite il protocollo HTTP.

6.5 Connessione persistente e non persistente

Negli esempi proposti fin qui, il client si è sempre trovato nello stato iniziale dopo aver effettuato la richiesta e nulla è stato detto circa la necessità o meno per effettuare la richiesta successiva, di aprire un nuovo canale con il server.

La connessione si dice non persistente se viene chiusa dal server dopo l'invio del messaggio di risposta e persistente altrimenti, permettendo al client di inviare una successiva richiesta attraverso la stessa socket.

Il comportamento predefinito per HTTP è quello di usare connessioni non persistenti nella versione 1.0 e persistenti nelle versioni 1.1 e superiori. Se il comportamento predefinito non soddisfa le necessità del client, è possibile richiedere esplicitamente un certo tipo di connessione con l'opzione *Connection*. Per richiedere esplicitamente la chiusura o meno del canale di comunicazione l'opzione *Connection* può assumere due valori: *Close* e *Keep-Alive*. Nel primo caso il server chiuderà il canale al termine del messaggio di risposta mentre nel secondolo manterrà aperto.

È sempre meglio essere cauti con l'opzione *Keep-Alive*: i server sono generalmente configurati per chiudere la socket dopo un certo periodo di inattività (per prevenire attacchi DoS), per cui è consigliabile forzare il *Keep-Alive* solo nei casi in cui si faranno una serie di accessi in rapida successione.



6.6 Web proxy

Un proxy è un server *intermedio* tra un client e un qualunque web server, il servizio viene generalmente erogato sulla porta 8080 e ne esistono varie implementazioni; la più famosa, tra quelle di libera distribuzione, è probabilmente *squid* [52].

Un proxy svolge la funzione di cache per gli utenti di una rete locale verso il resto di Internet mantenendo una copia locale dei contenuti in transito. A fronte di una richiesta il proxy controlla la sua cache e cerca di soddisfarla immediatamente; nel caso questo non sia possibile esso richiede il contenuto al server e lo passa al client dopo aver aggiornato la cache. Se tutti gli utenti di una rete locale fanno uso dello stesso proxy si riducono i tempi di accesso ai contenuti e il numero di accessi alla rete pubblica (con relativo risparmio economico).

La cache che ogni browser mantiene localmente sul disco svolge una funzione identica a quella di un proxy, ma limitata a un solo utente.



Naturalmente, non tutti i contenuti possono essere inseriti nella cache seguendo le stesse politiche; ad esempio, la pagina principale di un quotidiano dovrebbe avere una permanenza limitata mentre la pagina web relativa a un'operazione di home-banking è giusto che non venga neppure memorizzata su di un proxy. Per imporre particolari politiche di memorizzazione i web server fanno uso di due opzioni all'interno degli header di risposta: *Expires* e *Cache-Control*. La prima opzione determina fino a quando il contenuto potrà ritenersi valido all'interno della cache, mentre la seconda permette di stabilire se il contenuto può essere inserito nella cache e per quanto tempo può rimanervi.

L'uso di un proxy è molto semplice: anziché aprire un canale di comunicazione con il server, il client si collega al proxy; se si fa uso di un URL assoluto il messaggio di richiesta potrà essere esattamente lo stesso che nel caso di una connessione diretta. Il proxy ha la facoltà di inserire delle proprie opzioni nell'header del messaggio di risposta; alcune di queste potrebbero non essere standard e in tal caso, per convenzione, i parametri dovrebbero avere nomi che iniziano con la stringa “X-Cache”, come nell'esempio che segue.

```
$ telnet proxy.private.net 8080
Trying 192.168.2.254...
Connected to proxy.private.net.
Escape character is '^]'.
GET http://netdev.usr.dico.unimi.it/ HTTP/1.0

HTTP/1.0 200 OK
Server: Apache/2.2.0 (FreeBSD)
Last-Modified: Wed, 04 Jun 2008 10:10:13 GMT
Content-Type: text/html
Content-Length: 10283
X-Cache: MISS from proxy.private.net
X-Cache-Lookup: MISS from proxy.private.net:8080
...
```

Come si può sono state inserite le opzioni *X-Cache* e *X-Cache-Lookup* le quali, oltre a dare informazioni sul proxy, comunicano che il contenuto che segue è frutto di un “*cache miss*”; il proxy, cioè, è stato costretto a richiederlo al server originario. Infatti, accedendo nuovamente alla stessa risorsa, che ora si troverà nella cache, si ottiene un “*cache hit*” e nell'header saranno visibili le opzioni riportate qui di seguito.

```
X-Cache: HIT from proxy.private.net  
X-Cache-Lookup: HIT from proxy.private.net:8080
```

È possibile, infine, forzare una connessione permanente o non permanente con il proxy in dipendentemente dal tipo di connessione che quest'ultimo usa con il server. Per farlo, è sufficiente inserire nell'header l'opzione *Proxy-Connection* seguendo esattamente le stesse regole dell'opzione *Connection* vista prima.

6.7 Cookie

La connessione tra cliente web server si definisce *stateless*, ovvero, ogni accesso a una risorsa è indipendente da quelli che lo hanno preceduto. Il meccanismo dei cookie [53] permette a un web server di associare uno stato a ogni client e tenere traccia delle attività dei singoli utenti. In questo modo il client sarà in grado di ottenere contenuto personalizzato e di gestire sessioni di lavoro anche ad distanza di tempo, come ad esempio non perdere il contenuto del carrello memorizzato in un sito di e-commerce.

Un server può assegnare a un client uno più cookie facendo uso dell'opzione *Set-Cookie*. Il valore associato a *Set-Cookie* si compone di una serie variabile di elementi “nome=valore” intervallati da caratteri “;”, un nome può identificare un cookie oppure esprimere un criterio su come utilizzarlo.

Domain

Specifica il dominio per cui il cookie è valido (ad esempio “.google.it”), se non viene specificato vale solo per il server che lo ha generato.

Path

Un sottoinsieme di URL per cui il cookie è valido, se non specificato solo l’URL della risorsa che lo ha generato.

Max-Age

Il tempo di validità del cookie espresso in secondi. Se non viene specificato, il cookie dovrà essere scartato alla terminazione del client; in questo caso si usa il termine “cookie di sessione”.

Expires

Ha lo stesso significato di *Max-Age*, ma il tempo di validità viene espresso come una data di scadenza. Non fa parte dello standard ma della proposta originale scritta da netscape; viene usato comunque da molti siti.

È cura del client memorizzare in maniera permanente (su disco) i cookie con relativi attributi.

Nel momento in cui un client effettua una richiesta a un server che soddisfa i criteri per uno o più cookie in suo possesso, questi possono essere allegati al messaggio di richiesta facendo uso dell’opzione *Cookie*.

 Il client ha la facoltà di decidere se mandare o meno un cookie; parimenti, il server effettuerà un controllo per verificare se è effettivamente stato generato da lui.

Un server è in grado di richiedere la cancellazione di un cookie inviandolo nuovamente al client e specificando “*Max-Age: 0*”.

 Alcuni siti, soprattutto se di e-commerce, mandano con ogni pagina un cookie dal nome sempre uguale ma della durata di pochi secondi (1 o 2); questo meccanismo serve a evitare un doppio click accidentale: se nella richiesta di una pagina compare quel cookie, allora molto probabilmente è stata involontaria e può essere ignorata.

6.8 Interrogazione di motori di ricerca

Un motore di ricerca è probabilmente la più grande fonte di informazioni con cui è possibile interagire su Internet.

Il messaggio di richiesta in sé non presenta enormi problemi: si può semplicemente utilizzare i metodi GET o PUT con parametri dipendenti dal motore di ricerca che si sta usando.

Disgraziatamente, i motori di ricerca non pubblicano quasi mai i dettagli relativi al modo in cui effettuare un'interrogazione; è compito del programmatore capire quali siano i parametri analizzando l'URL che compare nel browser e il testo del form all'interno della pagina principale. Mentre l'analisi dell'URL può essere semplice quella del form può rivelarsi un po' complicata: occorre infatti isolare il tag FORM e i vari campi INPUT al suo interno. Per i dettagli su come interpretare i vari tag si può fare riferimento alle specifiche di HTML [54, 55].

```

<form action="http://www.google.com/search" method="get">
  <input name="q" type="text" value="<?php echo $q ?>" />
  <input maxlength="250" name="q" title=" cerca con Google" value="" />
  <input name="hl" type="hidden" value="it" />
  <input name="meta" type="hidden" value="<?php echo $meta ?>" />
  <input type="checkbox" checked="checked" value="forex" /> cerca forex
  <input type="checkbox" value="forex" /> cerca forex
  <input id="lang" type="radio" name="hl" value="it" checked="checked" /> cerca forex pagine in italiano
  <input id="lang" type="radio" name="hl" value="en" /> cerca forex pagine in inglese
  <input type="checkbox" value="forex" /> cerca forex pagine provenienti da Italia
</form>

```

Figura 6.4 Form di interrogazione estratto da <http://www.google.it/>.

Nella Figura 6.4 viene rappresentato il form estratto dalla pagina principale di Google.

Come si può osservare, in assenza del tag “*method=post*”, il metodo associato alla richiesta è GET; inoltre, sono presenti una serie di parametri dai nomi poco intuitivi, dei quali alcuni anche nascosti. In generale, quello che interessa è il parametro *q* che conterrà il testo dell’interrogazione, mentre gli altri parametri sono lasciati alla sperimentazione del programmatore.

Per una discussione completa, anche se non ufficiale, di come interagire con Google è possibile consultare [56].

Una trattazione simile, ma molto più semplice, può essere fatta per wikipedia.it, il cui form è riassunto nella Figura 6.5

```
<form action="/wiki/Speciale:Ricerca">
  <input type="text" name="search" />
  <input type="submit" name="go" value="Vai" />
</form>
```

Figura 6.5 Form di interrogazione estratto da <http://www.wikipedia.it/>.

Nel caso di Wikipedia occorre però fare attenzione: il messaggio di risposta non porta con sé il contenuto relativo ai risultati della ricerca ma è una redirect (codice 301) a una risorsa contenente la pagina cercata o una pagina di selezione se la ricerca ha dato un risultato ambiguo.

La parte più difficoltosa dell'interrogazione è però l'interpretazione del risultato; in generale le pagine web potrebbero essere:

- poco strutturate (se non totalmente destrutturate)
- variabili nel tempo con il layout del sito web
- composte da tag obsoleti o non sintatticamente corretti
- corredate da codice javascript.

Tutto questo rende estremamente difficile l'estrazione di informazioni se non identificando delle stringhe di testo sempre presenti all'interno della pagina e isolando il contenuto di

interesse. Questa operazione porta a implementazioni di client estremamente orientati a un dato motore di ricerca e che richiedono aggiornamento immediato nel caso in cui il gestore del sito decida di cambiare la struttura delle pagine.

Anche per ovviare a questi tipi di problemi è stato proposto lo standard XHTML [57]: una versione di HTML con una struttura più formale e un controllo più rigido dei tag; purtroppo, i siti che utilizzano XHTML per il formato dei loro contenuti sono ancora una minoranza.

6.9 RSS

Quella del *Really Simple Sindication* (RSS) [58] è una strategia che tenta di risolvere il problema della generale assenza di strutturazione delle pagine web.

Con RSS il contenuto di una certa risorsa (o gruppo di risorse) viene riassunto e reso disponibile in formato XML con un namespace definito appositamente; in pratica si tratta di un documento XML con vincoli precisi sui tag da utilizzare. All'interno di un documento RSS vengono definiti alcuni canali di distribuzione, identificati da un titolo, descrizione e un URL di riferimento (il sito web che lo ha generato), all'interno dei canali sono presenti una serie di elementi (*item*) a loro volta caratterizzati da titolo, descrizione e l'URL specifico a cui l'*item* fa riferimento. Un esempio di documento RSS è presentato nella Figura 6.6.

```
<rss version="1.0" encoding="ISO-8859-1" >
<rss version="1.0">
<channel>
<title>W3Schools - Free web building tutorials</title>
<link>http://www.W3Schools.com</link>
<description>Free web Building tutorials</description>
<item>
<title>HTML Tutorial</title>
<link>http://www.W3Schools.com/html/</link>
<description>New Web tutorial on W3Schools</description>
</item>
<item>
<title>CSS Tutorial</title>
<link>http://www.W3Schools.com/css/</link>
<description>New Web tutorial on W3Schools</description>
</item>
</channel>
</rss>
```

Figura 6.6 Esempio di documento RSS.

Per le sue caratteristiche, RSS risulta particolarmente adatto a distribuire informazioni relative a prime pagine di quotidiani e aggiornamenti di blog; queste informazioni possono essere fruite e rielaborate molto facilmente e senza vincoli di interfaccia grafica.

La distribuzione di un documento RSS prende anche il nome di *feedRSS* per via del fatto che viene spesso adottato per strutturare contenuti con una grande variabilità nel tempo; in lingua inglese *to feed* significa anche *alimentare, fornire in maniera continuativa*.



6.9.1 Podcast

Un podcast non è altro che un feed RSS all'interno del quale i riferimenti dei singoli item sono risorse a cui è associato un contenuto multimediale. Per farlo si usano i tag *enclosure* o *guid*

perché il tag *link* viene spesso destinato a una pagina web con la descrizione del contenuto stesso, anche per motivi di copyright.

Il nome deriva dalla prima applicazione in questo ambito sviluppata da Apple per i dispositivi *iPod*: in questo caso, per semplificare la fruizione, sono stati anche proposti tag aggiuntivi all'interno del namespace [59].

Nell'uso comune si differenziano i feed RSS in base al tipo di contenuto distribuito; si parla, oltre che di podcast, di *webcast*, *videocast* e *shoutcast*.



6.10 Web service

I web service rappresentano unametodologia di accesso a risorse web in cui la finalità non è quella di accedere a del contenuto ma piuttosto di richiedere l'esecuzione di una procedura sul server. I web service hanno cambiato radicalmente la prospettiva con cui si usavano certi servizi: da architettura per la distribuzione di informazioni, il Web è diventato un'infrastruatura con la quale realizzare un sistema distribuito [60].

Per poter richiedere l'esecuzione di codice remoto, la richiesta identifica come risorsa la procedura che deve essere eseguita

e il contenuto del messaggio sarà costituito dai parametri da utilizzare. Uno dei problemi principali da affrontare è quello di come trasformare qualunque tipo (o quasi) di parametro e di risultato in un formato indipendente dalla piattaforma e utile alla trasmissione; questa operazione prende il nome di *marshalling* ed è concettualmente molto simile a quella di serializzazione, ma non limita agli oggetti. Le informazioni relative ai parametri vengono, nel caso dei web service, trasformate in una loro rappresentazione XML [46] con un namespace prefissato.

La trattazione dei web service è molto ampia e complessa. Il loro utilizzo si basa su un protocollo applicativo intermedio chiamato SOAP [61] per il quale i linguaggi di programmazione più evoluti forniscono interfacce già pronte. Per questi motivi si preferisce rimandare il lettore a testi specifici sull'argomento quale, ad esempio, [62].

III

Linguaggio Java

Capitolo 7

Programmazione con le socket

7.1 Socket

Il linguaggio Java tratta le socket come canali di I/O incapsulati all'interno di apposite classi. Dal punto di vista della programmazione questo approccio ha il vantaggio di nascondere tutti i dettagli implementativi e di lasciare al programmatore solo il compito di mettere in atto la connessione e gestire i messaggi scambiati.

Esistono varie classi che permettono la manipolazione delle socket, ma, tutte demandano le funzionalità dibase, l'interfaccia verso il canale di I/O, all'istanza di una sottoclasse di *SocketImpl* referenziata da un campo privato dell'oggetto. Ognuna delle classi analizzate in questo capitolo renderà disponibili al programmatore solo un sottoinsieme delle funzionalità implementate da *SocketImpl*; così facendo è

possibile associare ogni classe a uno specifico contesto d'uso (ad esempio, lato server o lato client), permettendo di ridurre gli errori in fase di programmazione.

L'idea alla base di questa scelta è che il programmatore può utilizzare classi ad alto livello mentre l'applicazione fornisce in maniera trasparente la corretta estensione di *SocketImpl* per poter accedere alla rete. L'implementazione di base di *SocketImpl* accede in maniera diretta alla rete, mentre in alcune situazioni potrebbe essere necessario utilizzare un proxy o autenticarsi. Estendendo la classe *SocketImpl* in maniera opportuna ed utilizzandone il risultato all'interno delle classi a più alto livello è possibile effettuare queste operazioni senza bisogno di modificare il codice delle applicazioni.

In questo libro non ci si occuperà di estendere la classe *SocketImpl*, ma verrà trattato l'uso delle classi ad alto livello.

7.2 Creazione di una socket

La creazione di una socket avviene tramite la creazione di un'istanza della classe *Socket*. Nella Figura 7.2 è illustrato l'uso più semplice possibile della classe *Socket*.

```
import java.net.Socket;
public class Socket extends Object;
```

Figura 7.1 Definizione della classe *Socket*.



La classe *SocketImpl* non è una superclasse della classe *Socket*; bensì, la classe *Socket* contiene al suo interno un’istanza di una sottoclasse di *SocketImpl*.

```
1 import java.net.Socket;
2 public class esempio {
3     public void main(String[] args) {
4         Socket s; // la socket
5         s = new Socket();
6         // altro codice
7     }
8 }
```

Figura 7.2 Uso di un costruttore della classe *Socket*.

Ovviamente, Java non mette a disposizione un solo tipo di socket: è infatti possibile disporre di alcune varianti a seconda della modalità con cui si intende effettuare lo scambio dei dati.

7.2.1 Dominio

Il linguaggio Java, a differenza di altri, è stato introdotto in tempi relativamente recenti. Gli sviluppatori hanno abbracciato l’idea, ormai consolidata, per cui, in futuro, l’infrastruttura di Internet sarà basata unicamente su IP.

Per questa ragione Java effettua una scelta molto drastica: solo il dominio Internet viene supportato.

Con la versione 1.4 del linguaggio è stata introdotta la possibilità di usare anche il dominio relativo alla versione 6 del protocollo IP; tuttavia questo non ha richiesto l'introduzione di una variante della classe *Socket*, ma solo l'estensione della classe usata per la rappresentazione di un indirizzo a livello di trasporto.

Neppure il dominio *unix* viene supportato; questo, infatti, utilizza risorse specifiche del calcolatore (il *file system*), in netto contrasto con la filosofia di indipendenza dalla piattaforma di Java. Non potendo presumere che il sistema operativo ospitante disponga di un disco o di un file system, Java non mette a disposizione socket in dominio unix.

All'interno di questo libro ci si concentrerà unicamente su socket in dominio Internet per IPv4.

7.2.2 Modalità di trasferimento dati

La classe *Socket* definisce una socket di tipo byte-stream.

Per poter creare una socket in modalità datagram occorre fare uso di un'istanza della classe *DatagramSocket*.

La classe *DatagramSocket* possiede al suo interno una diversa implementazione di socket ottenuta tramite un'estensione della classe virtuale *DatagramSocketImpl*. Il funzionamento di

quest'ultima è molto simile a quello di *SocketImpl*, fatta eccezione per i metodi usati per inviare e ricevere messaggi.

7.2.3 Protocollo

Per gli stessi motivi descritti nel Paragrafo 7.2.1, Java pone forti vincoli sui protocolli utilizzabili.

Essendo il protocollo di una socket vincolato dalla modalità di trasferimento dati, Java associa in maniera univoca il protocollo TCP a tutte le istanze della classe *Socket* e il protocollo UDP a tutte quelle di *DatagramSocket*.

7.3 Indirizzi: numeri e nomi

Il linguaggio Java, utilizzando il meccanismo dell'*overloading*, mette a disposizione diverse versioni di costruttori e metodi per permettere sempre al programmatore di usare nomi in formato alfanumerico.

Tuttavia, la classe *InetAddress* può essere utilizzata per effettuare conversioni esplicite che poi verranno usate come parametri di costruttori e metodi. La classe *InetAddress* non è dotata di un costruttore; è possibile crearne un'istanza solo tramite una chiamata a uno dei suoi metodi statici, usati per effettuare una richiesta esplicita al server DNS. Se la chiamata va a buon fine e non viene sollevata nessuna eccezione, l'istanza

(o le istanze) restituite contengono tutte le informazioni relative al nodo interessato. I metodi di più comune utilizzo per questo scopo sono presentati nella Figura 7.3.

Il metodo *getByName* restituisce un’istanza di *InetAddress*, mentre *getAllByName* restituisce un array contenente le istanze relative a tutte le possibili conversioni del nome fornito come parametro. Come già accennato nel Paragrafo 4.3, non è possibile supporre che ci sia un solo indirizzo associato a un certo nome. Il terzo metodo, *getLocalHost*, restituisce un’istanza contenente le informazioni sul nodo locale e viene utilizzato per ricavare l’indirizzo IP associato alla scheda di rete.

```
public class InetAddress extends Object {
    public static InetAddress getByName(String host);
    public static InetAddress[] getAllByName(String host);
    public static InetAddress getLocalHost();
    ...
}
```

Figura 7.3 Alcuni metodi per la conversione di nomi in istanze di *InetAddress*.

L’indirizzo in forma numerica può essere estratto da un’istanza di *InetAddress* facendo uso del metodo *getAddress*, il quale, però, non restituirà un numero di 32 bit ma un array di 4 byte corrispondente all’indirizzo in forma decimale puntata.

Nella Figura 7.4 viene presentato un programma che stampa il primo indirizzo associato a un nome.

```

1 import java.net.InetAddress;
2 import java.net.UnknownHostException;
3 public class esempio {
4     public static void main(String[] args) {
5         String nome = "www.cnn.com"; // da convertire
6         try {
7             InetAddress ia = InetAddress.getByName(nome);
8             byte[] ip = ia.getAddress();
9             System.out.println("Indirizzo: " + ip[0] & 0xFF + "."
10                         + ip[1] & 0xFF + "."
11                         + ip[2] & 0xFF + "."
12                         + ip[3] & 0xFF);
13         } catch (UnknownHostException ue) {
14             ue.printStackTrace();
15         }
16     }
17 }
18
19
20
21
22

```

Figura 7.4 Uso di *getByName* per la conversione di un indirizzo.

Le righe di codice dalla 13 alla 16 possono sembrare un po' oscure; in realtà sono necessarie per convertire i byte in interi e stampare i numeri in essi contenuti senza segno. Diversamente, essendo Java privo di tipi numerici senza segno, il risultato dell'esecuzione sarebbe stato l'indirizzo in notazione decimale puntata “64.-20.29.120”, palesemente errato.

Se si è invece interessati a tutti i possibili indirizzi, allora occorre scorrere l'array restituito dal metodo *getAllByName*. Per fare questo è sufficiente modificare opportunamente la riga 11 e inserire le righe da 12 e 16 all'interno di un ciclo, come illustrato nella Figura 7.5.

```

11 InetAddress[] ias = InetAddress.getAllByName(nome);
12 for (int i = 0; i < ias.length; i++) {
13     byte[] ip = ias[i].getAddress();
14     System.out.println("Indirizzo: " + ip[0] & 0xFF + "."
15                         + ip[1] & 0xFF + "."
16                         + ip[2] & 0xFF + "."
17                         + ip[3] & 0xFF);
18
19
20
21
22

```

Figura 7.5 Stampa di tutti gli indirizzi possibili riportati da *getAllByName*.

7.4 Identificazione di una socket

Metodi e costruttori che manipolano socket, al fine di costruire canali di comunicazione, possono fare uso della classe *InetSocketAddress*, estensione della classe astratta *SocketAddress* per memorizzare indirizzi a livello di trasporto. I costruttori di *InetSocketAddress* sono riportati nella Figura 7.6.

```
public class InetSocketAddress extends SocketAddress {
    public InetSocketAddress(InetAddress addr, int port);
    public InetSocketAddress(String hostname, int port);
    public InetSocketAddress(int port);
    ...
}
```

Figura 7.6 I costruttori della classe *InetSocketAddress*.

In tutti i casi l'istanza viene creata combinando un indirizzo a livello di rete con un numero di porta. Nel primo caso viene usata un'istanza di *InetAddress*, contenente un indirizzo a livello di rete, nel secondo si indica il nome del nodo lasciando al costruttore l'onere della conversione e nel terzo viene indicato solo il numero di porta utilizzando come indirizzo a livello di rete quello del calcolatore locale.

```
InetSocketAddress isa = new InetSocketAddress(1024);
```

è equivalente a:

```
InetAddress ia = InetAddress.getLocalHost();
InetSocketAddress isa = new InetSocketAddress(ia, 1024);
```

A questo punto è facile creare un’istanza di *InetSocketAddress* che identifichi un servizio erogato sulla porta 80 del nodo www.google.it, come rappresentato nella Figura 7.7.

```
1 import java.net.InetAddress;
2 import java.net.InetSocketAddress;
3 public class esempio {
4     public static void main(String[] args) {
5         String nome = "www.google.it";
6         try {
7             InetAddress ia = InetAddress.getByName(nome);
8             InetSocketAddress isa = new InetSocketAddress(ia, 80);
9         } catch (Exception e) {
10             e.printStackTrace();
11         }
12     }
13 }
```

Figura 7.7 Istanza dell’indirizzo di trasporto che identifica il servizio HTTP di www.google.it.

Oppure, in forma ancora più concisa, è possibile sostituire le righe 11 e 12 con:

```
InetSocketAddress isa = new InetSocketAddress(nome, 80);
```

7.5 Binding

Il binding esplicito di una socket viene effettuato facendo uso dell’omonimo metodo *bind*.

```
public class Socket extends Object {
    ...
    public void bind(SocketAddress indirizzo) throws IOException;
    ...
}
```

Figura 7.8 Definizione del metodo *bind*.

Per poter usare questo metodo basterà quindi creare un’istanza di *InetSocketAddress* e utilizzarla come parametro.

Nella Figura 7.9 viene proposto il codice necessario per creare una socket e associarla alla porta 7000.

La riga 9 si occupa della creazione della socket, la riga 10 stabilisce l'indirizzo di trasporto al quale fare l'associazione e la 12 effettua il bind. La riga 14 mette in attesa il programma per 120 secondi; durante tale intervallo di tempo è possibile usare il comando netstat per controllare che la porta sia stata effettivamente allocata. Nel nostro caso il risultato sarà simile al seguente:

```
1 import java.net.Socket;
2 import java.net.InetSocketAddress;
3
4 public class esempio {
5
6     public static void main(String[] args) {
7
8         try {
9             Socket s = new Socket();
10            InetSocketAddress isa = new InetSocketAddress(7000);
11            // Indirizzo sul nodo locale
12            s.bind(isa);
13
14            Thread.sleep(120 * 1000);
15        } catch (Exception e) {
16            e.printStackTrace();
17        }
18    }
19 }
20 }
```

Figura 7.9 Associazione di una socket alla porta 7000.

Active Internet connections (including servers)

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	(state)
tcp4	0	0	*.7000	*.*	CLOSED

Lo stato della socket (CLOSED) non va letto come “chiuso” ma bensì come “disconnesso”. La socket è stata creata ma non è ancora stata associata a una controparte. Nel momento in cui la socket verrà associata lo stato passerà da CLOSED a CONNECTED.

Per quanto riguarda invece la possibilità di effettuare un binding alla prima porta disponibile sarà sufficiente porre a zero il parametro. Tuttavia ci si troverà di fronte a un nuovo problema: quello di capire a quale porta la socket è stata associata.

Per rispondere a questa domanda è possibile fare uso del metodo *getLocalPort*.

È semplice modificare il programma nella Figura 7.9 in modo da fare allocare una porta dal sistema operativo e poi stamparne il numero; si veda a questo riguardo la Figura 7.10.

```
try {
    10   Socket s = new Socket();
    11   InetSocketAddress iss = new InetSocketAddress(0);
    12   // legge automaticamente il numero della porta
    13   s.bind(iss);
    14   System.out.println("Porta assegnata: " + s.getLocalPort());
    15
    16   Thread.sleep(100 + 1000);
    17 }
```

Figura 7.10 Associazione di una socket a una porta assegnata dal sistema.

7.6 Porte e servizi standard

Nel momento in cui è necessario contattare o erogare un servizio standard è preferibile utilizzare come riferimento all'interno del codice il nome e non il numero di porta. Questa scelta permette di dare più flessibilità al software in caso di variazione di uno standard.

Disgraziatamente, Java non offre questa funzionalità; in base allo stesso principio di indipendenza dalla piattaforma per cui sceglie di non supportare le socket in dominio unix, Java decide di non fare affidamento su un database esterno, dipendente dall'architettura del calcolatore, per le associazioni tra numero di porta e nome di un servizio.

L'unica opzione lasciata al programmatore è quella di creare proprie classi di servizio o file di configurazione in cui fare quest'associazione o, diversamente, cablare il numero di porta all'interno del codice.

7.7 Associazione di due socket

Come già anticipato nel Paragrafo 7.1, Java mette a disposizione più classi per accedere alle funzionalità offerte dalle socket, a seconda dell'uso per cui queste vengono impiegate. La distinzione tra chiamante e ricevente viene fatta tramite l'uso rispettivo delle classi *Socket* e *ServerSocket*. Queste due classi contengono entrambe un'istanza della stessa sottoclasse di *SocketImpl*, ma permettono l'acceso a diversi insiemi di funzionalità.

Diversamente, nel caso di socket datagram, a causa della simmetria della comunicazione, non esiste una vera e propria distinzione tra chiamante e ricevente, per cui viene resa disponibile solo la classe *DatagramSocket*, al cui interno troveremo l'istanza di una sottoclasse di *DatagramSocketImpl*.

7.7.1 Operazioni lato ricevente

Se la socket è di tipo datagram (classe *DatagramSocket*) non sono necessarie ulteriori operazioni per poterla utilizzare, a meno che non la si voglia associare a uno specifico mittente. Questa operazione è analoga a quanto fatto dal chiamante, per cui il lettore può fare riferimento al paragrafo successivo.

Se, invece, la socket è di tipo byte stream (classe *ServerSocket*), allora il programma deve mettersi in attesa di una richiesta di associazione tramite una chiamata al metodo *accept* (Figura 7.11). La chiamata ad *accept* è bloccante, arresta cioè l'esecuzione del programma fino al verificarsi di un evento, che sarà la richiesta di associazione da parte di un'altra socket. Il valore restituito, alla ripresa dell'esecuzione, sarà una nuova socket da utilizzarsi per scambiare dati.

È possibile in questo modo modificare il programma proposto nella Figura 7.10, avendo cura di definire la variabile *s* come un'istanza di *ServerSocket* e aggiungendo in coda una chiamata al metodo *accept*. Il risultato finale è riportato nella Figura 7.12.

Il programma terminerà nel momento in cui un chiamante effettuerà la connessione.

```
public class ServerSocket extends Object {
    ...
    public Socket accept() throws IOException
    {
        ...
    }
}
```

Figura 7.11 Definizione del metodo *accept*.

```
1 import java.net.ServerSocket;
2 import java.net.Socket;
3 import java.net.InetSocketAddress;
4 public class esempio {
5     public static void main(String[] args) {
6         try {
7             ServerSocket s = new ServerSocket();
8             InetSocketAddress isa = new InetSocketAddress(0);
9             s.bind(isa);
10            s.listen(1);
11            System.out.println("Porta allocata: " + s.getLocalPort());
12            Socket data_socket = s.accept();
13            data_socket.close();
14        } catch (Exception e) {
15            e.printStackTrace();
16        }
17    }
18 }
```

Figura 7.12 Messa in attesa di una socket byte-stream.

Per fare una prova pratica è possibile compilare e lanciare il programma scritto fin qui; esso si bloccherà subito dopo aver prodotto un output simile al seguente:

```
Porta allocata: 50226
```

però il numero sarà adiverso.

Usando nuovamente netstat, comparirà una riga recante la seguente informazione:

```
Active Internet connections (including servers)
```

Proto	Recv-Send-Local	Foreign	(state)
	Q Q	Address Address	
tcp4	0 0*	.50226 *.*	LISTEN

La colonna relativa allo stato della socket conferma che il programma è effettivamente in attesa di una chiamata.

Successivamente, tramite una seconda finestra con un prompt di comandi si può usare telnet per contattare il programma in attesa, avendo l'accortezza di usare il giusto numero di porta.

```
telnet localhost 50226
```

Nel momento in cui il comando telnet riuscirà a collegarsi, si osserverà la terminazione del programma in attesa.

Esercizio 7.1



Si estenda il programma proposto nella Figura 7.12 in maniera tale da stampare le informazioni relative all'indirizzo di trasporto del chiamante.

7.7.2 Operazioni al chiamante

Il chiamante non deve fare altro che effettuare una richiesta di associazione a una socket remota. Questa funzione viene svolta dal metodo *connect* della classe *Socket*.

```
public class Socket extends Object {  
    ...  
    public void connect(SocketAddress indirizzo) throws IOException  
    public void connect(SocketAddress indirizzo, int timeout) throws IOException  
    ...  
}
```

Figura 7.13 Alcune definizioni del metodo *connect*.

Il metodo *connect* della classe *Socket* segue la stessa logica di *bind* nel caso di *ServerSocket*, con la differenza che l'indirizzo fornito come parametro deve essere quello della socket alla quale associarsi.

Prendendo come punto di partenza il programma rappresentato nella Figura 7.7 è possibile aggiungere una chiamata a *connect* per effettuare la connessione a www.google.it.

```
9     String nome = "www.google.it";
10    try {
11        InetAddress ia = InetAddress.getByName(nome);
12        InetSocketAddress isa = new InetSocketAddress(ia, 80);
13
14        Socket s = new Socket();
15        s.connect(isa);
16    }
17    catch (Exception e) {
18        e.printStackTrace();
19    }
```

Figura 7.14 Collegamento al servizio HTTP di www.google.it.

Esercizio 7.2



Si modifichi il programma proposto nella Figura 7.14 in maniera tale da poterlo utilizzare per sbloccare il programma in attesa realizzato per l'Esercizio 7.1.

7.8 Trasferimento dati

Il linguaggio Java adotta due approcci molto diversi se le socket sono connesse o meno. Nel primo caso, che viene fatto coincidere con le socket di tipo byte-stream, vengono rese disponibili dall'istanza della classe due istanze di canali di I/O monodirezionali: una di *InputStream* e una di *OutputStream*. Nel secondo caso, invece, che viene fatto coincidere con socket di tipo datagram, sono disponibili due metodi: *send* e *receive*, che si occupano di scambiare con la controparte il contenuto di un'istanza della classe *DatagramPacket*, all'interno della quale si troveranno i dati le informazioni relative a mittente e destinatario.

7.8.1 Socket connesse

Le classi che implementano socket funzionanti in maniera connessa, come ad esempio *Socket*, mettono a disposizione del programmatore due metodi: *getInputStream* e *getOutputStream*. Questi metodi restituiscono entrambi un'istanza di una classe per la gestione di un canale monodirezionale da o verso la socket acui si è connessi.

```
public class Socket extends Object {
    ...
    public OutputStream getOutputStream() throws IOException;
    public InputStream getInputStream() throws IOException;
    ...
}
```

Figura 7.15 Definizioni dei metodi *getInputStream* e *getOutputStream*.

Le due classi definiscono rispettivamente un metodo *write* e un metodo *read*, con i quali è possibile inviare o ricevere delle sequenze di byte.

```
public abstract class OutputStream extends Object {
    ...
    public void write(byte[] buffer) throws IOException
    public void write(byte[] buffer, int offset, int dim_buffer)
        throws IOException
    ...
}
```

Figura 7.16 Definizioni del metodo *write*.

Nella sua forma più semplice il metodo *write* invia sulla rete i dati contenuti in un buffer (un array di byte); è anche possibile richiedere l'invio di un sottoinsieme dell'array facendo uso di parametri aggiuntivi.

```
public abstract class InputStream extends Object {
    ...
    public int read(byte[] buffer) throws IOException
    public int read(byte[] buffer, int offset, int dim_buffer)
        throws IOException
    ...
}
```

Figura 7.17 Definizioni del metodo *read*.

Il metodo *read*, invece, sospende l'esecuzione del processo fino all'arrivo del prossimo messaggio dalla rete. Alla disponibilità di dati viene restituito il numero di byte letti dalla socket. Il numero di byte ricevuti potrebbe essere minore di quanto richiesto con il parametro *dim_buffer* o della dimensione del buffer; questo, però, non è indicativo di un errore, semplicemente la dimensione del messaggio in arrivo risulta essere inferiore di quella del buffer. In questo caso *dim_buffer*

è da interpretare com un *numero massimo accettabile* di byte in ingresso. In generale, la politica migliore è quella di usare il valore restituito dal metodo *read* per delimitare la porzione di memoria utilizzata.

È possibile, quindi, estendere ulteriormente il programma rappresentato nella Figura 7.12 come illustrato nella Figura 7.18.

```
12     Socket data_socket = s.accept();
13
14     int dim_buffer = 1024;
15     byte buffer[] = new byte[dim_buffer];
16     InputStream is = data_socket.getInputStream();
17     int r = is.read(buffer);
18     int r = is.read(buffer);
19     String str = new String(buffer, 0, r - 2);
20
21     System.out.println("Ricevuta stringa '" + str +
22                         "' usando " + r + " byte");
23 }
24 catch (Exception e) {
25     e.printStackTrace();
26 }
27 }
```

Figura 7.18 Ricezione di una stringa su una socket connessa.

Per provare il programma è possibile operare in maniera simile a quanto fatto nel Paragrafo 7.7.1, ma questa volta il comando telnet non terminerà appena collegato al programma in attesa, bensì entrerà in uno stato di apparente blocco. Sarà possibile far evolvere la situazione digitando una qualunque stringa di caratteri e terminando con un ritorno a capo; il programma realizzato nell'esempio stamperà un messaggio relativo alla stringa e terminerà, provocando anche la terminazione di telnet.

```
Porta allocata: 61877
Ricevuta stringa 'messaggio di prova', usando 20 byte
```

Il significato della riga 23 può sembrare un po' oscuro; tuttavia si tratta di un'operazione necessaria in quanto il messaggio

invia^{to} dal chiamante è ricevuto sotto forma di un array di byte, cosa diversa dai caratteri, e risulta costituito unicamente da tutti e soli i dati digitati. Per prima cosa va convertito in una stringa, altrimenti non potrebbe essere rappresentato facilmente a video. In secondo luogo, comprende anche il carattere di ritorno a capo, che in questo caso è dannoso perchè rovina il formato dell'output.

Esercizio 7.3



Modificare il programma proposto nella Figura 7.18 in maniera tale da far visualizzare tramite il comando telnet un *prompt*, cioè una stringa che dia informazioni su come procedere.

Anche con una socket di tipo byte-stream non è per nulla garantito che a fronte di un invio di N byte ci si trovi a osservare una ricezione di esattamente N byte all'interno di un solo messaggio. Questo comportamento, apparentemente anomalo, può dipendere dalla frammentazione effettuata a livello di rete oppure dalla disponibilità dei buffer dei sistemi operativi dei nodi



ricevente o trasmittente.

Esercizio 7.4



Estendere la classe *InputStream* in maniera tale da definire un metodo *readAll* tale per cui l'operazione di lettura non termini se non alla consegna di tutti i byte richiesti.

7.8.2 Socket non connesse

I due metodi implementati da classi che gestiscono socket non connesse (in questo caso *DatagramSocket*) sono *send* e *receive*.

```
public class DatagramSocket extends Object {
    ...
    public void receive(DatagramPacket packet) throws IOException;
    public void send(DatagramPacket packet) throws IOException;
    ...
}
```

Figura 7.19 Definizioni dei metodi *send* e *receive*.

Questi due metodi, al contrario del caso visto in precedenza, non utilizzano array di byte ma istanze di una classe apposita

per lo scambio dati: *DatagramPacket*. La classe *DatagramPacket* incapsula al suo interno un buffer, costituito da un array di byte, l'informazione relativa al quantitativo di dati da inviare o ricevere e gli indirizzi delle socket mittente e ricevente.

```
public class DatagramPacket extends Object {  
    ...  
    public DatagramPacket(Byte[] buffer, int bufferSize);  
    public DatagramPacket(Byte[] buffer, int offset, int limit);  
    ...  
}
```

Figura 7.20 Alcuni costruttori della classe *DatagramPacket*.

Il metodo *send* esamina l'istanza di *DatagramPacket* e invia i dati contenuti nel buffer all'indirizzo specificato. Il metodo *receive*, invece, blocca l'esecuzione del programma fino a che non ci sono dati in arrivo e, quando questi sono disponibili, riempie il contenuto dell'istanza di *DatagramPacket* con il messaggio e le informazioni relative al mittente. Tali informazioni possono essere poi utilizzate per rispondere al messaggio ricevuto.

Normalmente, non è conveniente creare un'istanza di *DatagramPacket* ogni volta che si intende inviare un messaggio, soprattutto dal punto di vista delle prestazioni, ma è più pratico crearne una sola, con un buffer eventualmente sovrardimensionato, e impostare di volta in volta i vari parametri tramite i metodi *setData*, *setLength* e *setSocketAddress*.



Se il buffer con cui è stato istanziato il pacchetto non è sufficiente a contenere tutte le informazioni il messaggio proveniente dalla rete verrà troncato.

A questo punto è possibile scrivere una versione non orientata alla connessione del programma appena visto per la ricezione di una stringa.

Ovviamente, nel programma rappresentato nella Figura 7.21 è stata tolta la chiamata al metodo *accept* per l'associazione. Inoltre, facendo ora uso di una modalità non connessa non è più possibile utilizzare il comando telnet per verificarne il funzionamento.

Esercizio 7.5



Implementare, facendo uso di una socket in modalità non connessa, un programma che mandi una stringa all'esempio della Figura 7.21 e se ne verifichi il buon funzionamento.

```
10      DatagramSocket d = new DatagramSocket(); //  
11      ...  
12      int din_buffer = 1024;  
13      byte buffer[] = new byte[din_buffer];  
14      DatagramSocket dp = new DatagramSocket(buffer,din_buffer);  
15      dp.setSoTimeout(1000);  
16      String str = new String(dp.getData(),0,dp.getLength()-2);  
17      System.out.println("Allevata stringa " + str +  
18          " usando " + dp.getLength() + " bytes");  
19      }  
20  } catch (Exception e) {  
21     e.printStackTrace();  
22  }  
23 }  
24 }
```

Figura 7.21 Ricezione di una stringa su una socket non connessa.

7.9 Chiusura del canale

La chiusura esplicita del canale viene normalmente effettuata facendo uso del metodo *close*. Da quel momento in poi la socket non potrà più essere utilizzata per trasferire dati.

In alternativa, la chiusura implicita del canale viene effettuata dal garbage collector quando l'istanza della socket non risulta più visibile dal flusso esecutivo (*out of scope*).

Capitolo 8

Implementazione di sistemi client-server

In questo capitolo si affrontano le problematiche proposte nel Capitolo 5 utilizzando gli strumenti che il linguaggio Java mette a disposizione.

Diversamente dal capitolo precedente, dove è stato fatto un uso piuttosto prolioso dei metodi relativi a binding e connessione, da questo momento in poi si adotteranno costruttori per le classi *Socket*, *ServerSocket* e *DatagramSocket* tali per cui non sarà più necessario effettuare esplicitamente queste operazioni.

Si richiama l'attenzione del lettore sul fatto che, per motivi di spazio e comprensibilità, anche in questo capitolo la gestione delle eccezioni è ridotta al minimo e tutto il codice è inserito all'interno di un unico blocco *try* e *catch*. Ciononostante, si ricorda che il controllo e la gestione degli errori è fondamentale

per il buon funzionamento di ogni programma: quindi è buona norma catturare ogni tipo di eccezione in maniera tale da poter identificare correttamente le situazioni anomale e intraprendere le adeguate azioni correttive.

8.1 Formato dei messaggi inviati

Come già visto, le classi *OutputStream* e *InputStream* permettono di inviare e ricevere informazioni tramite array costituiti di elementi di tipo *byte*. La manipolazione di questi array può avvenire in due modi: un elemento alla volta tramite il suo indice oppure facendo uso della classe *Arrays*.

È importante non confondere le classi *Array* e *Arrays* (con la “s” finale); la prima è un’astrazione dell’accesso agli elementi, mentre la seconda permette di copiare, confrontare, assegnare e ordinare l’array o un suo sottoinsieme.



```
abstract class Arrays extends Object {
    ...
    static Object[] copyOf(Object[] originale, int nuova_lunghezza);
    static Object[] copyOfRange(Object[] originale, int inizio, int fine);
    static void fill(Object[] array, int inizio, int fine, Object o);
    static void sort(Object[] array, int inizio, int fine);
    ...
}
```

Figura 8.1 Alcuni metodi per la manipolazione di array in *Arrays*.

8.1.1 Caratteri

Il tipo nativo *char* di Java rappresenta un carattere unicode[44], per cui non può essere inviato sulla rete senza prima essere convertito secondo una codifica ASCII [42]. Fortunatamente, la conversione non presenta grossi problemi: può essere fatta con il cast a un tipo *byte*.

```
System.out.println((byte) '0'); // stampa "48"
```

8.1.2 Stringhe

Il linguaggio Java mette a disposizione del programmatore le due operazioni di *Carriage Return* (CR) e *LineFeed* (LF) tramite i caratteri di "\r" e "\n" rispettivamente. L'interpretazione di questi due caratteri è totalmente indipendente dal sistema operativo ospite, ciò vuol dire che è a carico del programmatore usare la *sequenza corretta* per ottenere un ritorno a capo durante le operazioni di I/O. Per questo motivo, tutti i metodi della classe *PrintStream* e delle sue sottoclassi, la cui istanza più usata è forse *System.out*, che prevedono output testuale vengono forniti in due versioni: con ritorno a capo e senza (come ad esempio *println* e *print*). Il ritorno a capo prodotto dai metodi con suffisso "ln" sarà congruente con quanto usato dal sistema.

È possibile usare il metodo *System.getProperties()* per sapere quale sia la convenzione utilizzata dal sistema operativo ospite.



In generale, è sempre consigliabile mantenere in memoria istanze della classe *String* prive di ritorni a capo, eventualmente togliendoli quando il messaggio arriva via rete. A questo proposito può essere utile usare il metodo *replace* della classe *String* per eliminare CR e LF dai dati, come nell'esempio che segue, dove viene chiamato il metodo *replace* due volte per sostituire tutte le occorrenze prima di "\r" e poi di "\n" con una stringa vuota. Il risultato finale è assegnato alla variabile *str*.

```
String str = stringa_dalla_rete.replace("\r", "").replace("\n", "")
```

È possibile creare una stringa di caratteri partendo da un array di byte e viceversa. La classe *String* supporta queste operazioni rispettivamente tramite il metodo *getBytes* e costruttori che accettano un array di byte come parametro. Dato un buffer, comunque, non è mai consigliabile convertire tutti i byte in una stringa, in quanto potrebbero esserci dati relativi a una ricezione precedente dopo quelli ricevuti con l'ultima lettura, per cui è sempre meglio fare uso del costruttore della classe *String* che limita il numero di byte da utilizzare.

8.1.3 Valori numerici

La conversione di un valore numerico in una stringa può essere effettuata in vari modi, il più semplice dei quali è probabilmente quello riportato qui di seguito:

```
String stringa = "" + numero;
```

dove la variabile *numero* può essere, in realtà, di un qualunque tipo.

In alternativa, è possibile usare il metodo statico *toString* di una delle classi che Java rende disponibili per incapsulare i tipi di base; in questo caso però si è vincolati a usare la corretta classe di incapsulamento.

```
double numero_decimale = 3.14159265;  
String stringa = Double.toString(numero_decimale);
```

La conversione in formato esadecinale può essere fatta con gli equivalenti metodi *toHexString*.

Per convertire una stringa in un valore numerico è invece necessario fare uso dei metodi di conversione all'interno delle classi di incapsulamento menzionate sopra. Questi metodi hanno un nome del tipo *parseTYPE*, dove *TYPE* può essere “*Byte*”, “*Double*”, “*Float*”, “*Int*”, “*Long*”, “*Short*” o “*Boolean*” a seconda del contesto.

```
String stringa = "1.41421356";
double numero_decimale = Double.parseDouble(stringa);
```

8.1.4 Dati strutturati

In Java è possibile definire solo classi, per cui l'unica opzione possibile è la serializzazione.

Per definire una classe serializzabile è sufficiente dichiararla come implementazione dell'interfaccia *Serializable*; l'interfaccia non richiede la definizione di metodi ma solo quella di un campo intero *serialVersionUID*, dove deve essere memorizzato un numero di versione della classe che verrà usato per garantire che il ricevente effettui la deserializzazione su una classe identica a quella di partenza. Il trasferimento di oggetti serializzati avviene tramite istanze della classi *ObjectOutputStream* e *ObjectInputStream*, ricavabili a partire dalle istanze di *OutputStream* e *InputStream* di una socket.

Un esempio della spedizione di una classe serializzabile viene proposto nella Figura 8.2.

```

1 import java.net.Socket;
2 import java.io.Serializable;
3 import java.io.ObjectOutputStream;
4
5 public class esempio {
6     public static void main(String[] args) {
7         ...
8         try {
9             // classe da inviare
10            ClasseLocale dg_inviare = new ClasseLocale();
11            // socket già connesso al server
12            Socket s = new Socket("host", porta);
13            // canale per l'invio di oggetti serializzati
14            ObjectOutputStream os = new ObjectOutputStream(s.getOutputStream());
15            os.writeObject(dg_inviare);
16            // invio
17            os.writeObject(dg_inviare);
18        } catch (Exception e) {
19            e.printStackTrace();
20        }
21    }
22 }
23
24 public class ClasseLocale implements Serializable {
25     ...
26     static final long serialVersionUID = 1L;
27     ...
28     // campi e metodi della classe
29 }
30

```

Figura 8.2 Invio su di una classe serializzata tramite una socket.

8.2 Gestione dei messaggi

Come già discusso, i problemi di gestione dei messaggi si presentano in modo più marcato con socket di tipo byte-stream e per poterli affrontare in maniera agevole è opportuno che il lessico del protocollo sia stato ben progettato.

Se il messaggio da ricevere è di lunghezza fissata, o possiede quantomeno una parte iniziale di lunghezza nota (*header*), una buona strategia è quella di effettuare una prima ricezione facendo una serie di chiamate al metodo *read* fino a che esattamente il numero di byte necessari è stato ricevuto nel buffer, successivamente dai dati ricevuti ricavare l'informazione relativa alla dimensione totale del messaggio e poi effettuare una successiva serie di letture dalla socket per il totale dei byte restanti. Un esempio di come implementare un metodo che

legga esattamente il numero di byte richiesti da una socket è riportato nella Figura 8.3.

```
1 import java.io.InputStream;
2 import java.io.IOException;
3 ...
4     public int readAllInputStream(InputStream is, byte[] buffer, int dimensione)
5             throws IOException {
6         int r = 0;
7         while (r < dimensione) {
8             int t = is.read(buffer, r, dimensione - r);
9             if (t < 0) return r; // socket chiuso
10            r += t;
11        }
12        return r;
13    }
14 }
```

Figura 8.3 Metodo per leggere esattamente un quantitativo di byte da una socket.

Se invece il messaggio ha una lunghezza variabile e fa uso di un terminatore (come, ad esempio, una sequenza di stringhe testuali in arrivo da un web server), la cosa più semplice è leggere un carattere alla volta e depositarlo nei singoli elementi del buffer fino alla ricezione del terminatore.

8.3 Implementazione di un client

In base a quanto già visto nel capitolo precedente è possibile sostituire le diciture nella Figura 5.1 con i nomi delle chiamate necessarie e ottenere la Figura 8.4, dove sono stati riportati solo i passi più significativi.

Durante la discussione del codice, per favorire la comprensione e semplificare la struttura dei programmi, la fase di scambio dati verrà collocata all'interno del costruttore di una classe separata (*FruizioneServizio*), mentre il codice relativo alla gestione della connessione con il server risiederà all'interno del corpo

principale del programma. È lecito chiedersi come mai si preferisca usare una classe separata anziché un metodo all'interno della classe principale; il motivo è per favorire la riutilizzabilità del codice: un metodo (probabilmente anche statico, visto che viene spesso richiamato dal metodo *main*) necessita di essere copiato, mentre una classe può essere istanziata in un nuovo programma senza bisogno di ulteriori accorgimenti.

Si esamineranno ora separatamente i due casi in cui il client fa uso di socket connesse o meno.

8.3.1 Client con socket connesse

Facendo riferimento alla Figura 8.4(a) e ricordando quanto già appreso, è possibile cominciare a scrivere il codice relativo alle prima due fasi, presentato nella Figura 8.5.

Come è possibile notare, alla riga 13 si è fatto uso del costruttore di *Socket* che include anche l'operazione di connessione al server. Per poter usare questo costruttore si è costretti a inserire alcune variabili dove memorizzare il nome del nodo a cui connettersi e la porta dove è in attesa il server (righe 9 e 10). Seguendo l'approccio adottato già a partire dal Paragrafo 7.4, sono stati assegnati alle variabili valori codificati all'interno del programma; purtroppo però, e questo dovrebbe essere emerso anche durante lo svolgimento dell'Esercizio 7.2, il *cablare* valori all'interno del codice non è mai una buona scelta. Se i valori

dai quali dipende il funzionamento di un programma sono fissati in maniera statica all'interno del suo codice, si osserverà una riduzione di fiessibilità e, molto probabilmente, al variare di qualche condizione esterna, quale l'indirizzo del server da contattare, sarà necessaria una nuova compilazione dei sorgenti. Una soluzione molto più efficace consiste nel richiedere all'utente di fornire i parametri in linea dicomando, dando al più dei valori di default.

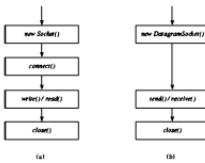


Figura 8.4 Diagrammi delle operazioni di un client con socket connesse (a) e non connesse (b).

```
1 import java.net.Socket;
2 public class Client {
3     ...
4     public static void main(String args[]) {
5         ...
6         try {
7             // parametri
8             String nome_host = "localhost";
9             int porta = 7000;
10            ...
11            // creazione socket
12            Socket s = new Socket(nome_host, porta);
13        }
14    }
15 }
```

Figura 8.5 Implementazione della prima e seconda fase di un client con socket connesse.

Un esempio di come sia possibile gestire parametri in linea di comando è presentato nella Figura 8.6.

```

8     // parametri
9     String nome_host = "localhost";
10    int porta = 7000;
11
12    // acquisizione parametri
13    if (args.length != 2)
14      throw new IllegalArgumentException(
15        "Numero parametri non corretto");
16
17    nome_host = args[0];
18    porta = Integer.parseInt(args[1]);
19
20    if (porta <= 0)
21      throw new IllegalArgumentException(
22        "Porta non valida");
23
24    // creazione socket

```

Figura 8.6 Esempio di gestione di parametri in linea di comando.

Per una questione stilistica, è preferibile sollevare un'eccezione anziché ricorrere a un *return*; questo perché le eccezioni permettono di gestire gli errori interni alla stregua di quelli relativi a socket e canali di I/O, mentre con il secondo approccio il flusso esecutivo viene semplicemente interrotto.

Della terza fase, come già anticipato, ci si occuperà separatamente, mentre la quarta, la chiusura del canale di comunicazione, risulta essere abbastanza banale. La parte conclusiva del codice del client è presentata nella Figura 8.7.

```

26   Socket s = new Socket(nome_host, porta);
27
28   // scambio dati
29   new FruizioneServizio();
30
31   // chiusura canale
32   s.close();
33
34   i catch (Exception e) {
35     e.printStackTrace();
36   }
37 }
38

```

Figura 8.7 Implementazione della terza e quarta fase di un client con socket connesse.

Il costruttore della classe *FruizioneServizio* accetta come parametro un'istanza di *Socket* già connessa al server e termina nel momento un cui il servizio è terminato. Si noti come la

nuova istanza creata non viene associata a nessuna variabile perché, in questo caso, non si ha interesse a recuperare all'interno del programma principale eventuali dettagli su come si sia fruito del servizio.

A questo punto il codice che permette al client di collegarsi al server è completo ed è possibile concentrarsi sulla parte di scambio dei dati implementando la classe *FruizioneServizio*.

Come primo passo si andrà a implementare un client per interagire con il programma realizzato nella Figura 7.18. In questo caso il protocollo a livello applicativo è estremamente semplice: il client manda una stringa al server e termina. Operativamente, il programma dovrà aspettare che l'utente inserisca una stringa, mandarla al server e terminare. Una possibile implementazione viene proposta nella Figura 8.8.

```
1 import java.net.Socket;
2 import java.io.IOException;
3 import java.io.BufferedReader;
4 import java.io.BufferedWriter;
5 import java.io.InputStreamReader;
6 import java.io.OutputStream;
7
8 public class FruizioneServizio {
9
10    public FruizioneServizio(Socket socket) throws IOException {
11        InputStreamReader isr = new InputStreamReader(socket.getInputStream());
12        BufferedReader br = new BufferedReader(isr);
13
14        String stringa = br.readLine(); // ritorno a capo con incluso
15        // il carattere che la stringa termini con "\r\n"
16        stringa += "\r\n";
17
18        OutputStream os = socket.getOutputStream();
19        os.write(stringa.getBytes(), 0, stringa.length());
20
21    }
22 }
```

Figura 8.8 Fruizione del servizio di spedizione di una stringa con socket connesse.

Come si può notare, la maggior parte del codice si occupa di operazioni che non hanno a che vedere con la rete: le righe dalla 11 alla 14 servono ad acquisire quanto digitato dall'utente,

mentre nella riga 16 viene aggiunta la sequenza di ritorno a capo per rendere i dati compatibili con quanto atteso dal server.

Esercizio 8.1



Modificare il programma proposto nella Figura 8.8 in maniera tale da usarlo come client per il server sviluppato per l'Esercizio 10.7. Il client deve prima ricevere dal server una stringa e visualizzarla, poi attende che l'utente digitò una stringa e la invia al server.

Ovviamente, un client in grado di inviare una singola stringa non risulta molto utile ai fini pratici. Per renderlo più flessibile è opportuno estenderne il funzionamento inserendo un ciclo infinito di lettura e scrittura: il client attende una stringa in input dall'utente, la invia al server e poi si mette in attesa della risposta; al suo arrivo la risposta viene visualizzata e il client si mette nuovamente in attesa di input da parte dell'utente. Per ottenere questo comportamento, è sufficiente inserire le righe dalla 14 alla 19 in un ciclo infinito avendo cura di spostare la definizione della variabile *stringa* al di fuori di quest'ultimo. Inoltre, è necessario definire una variabile *buffer* come array di byte per contenere la risposta in arrivo dal server. Il codice risultante è proposto nella Figura 8.9.

Il costruttore viene dichiarato in grado di sollevare eccezioni di tipo *IOException*, per far sì che eventuali errori di lettura e scrittura vengano gestiti dal chiamante.

```
9  public FruizioneServizio(Socket socket) throws IOException {
10
11     InputStream is = new InputStreamReader(socket);
12     BufferedReader br = new BufferedReader(new InputStreamReader(is));
13     OutputStream os = socket.getOutputStream();
14     PrintWriter pw = new PrintWriter(os);
15
16     int dimBuffer = 1024;
17     byte[] buffer = new byte[dimBuffer];
18     String stringa;
19     int i;
20
21     while (true) {
22         // input dell'utente
23         stringa = br.readLine();
24         stringa += "\n";
25         // invia del messaggio al server
26         os.write(stringa.getBytes("UTF-8", stringa.length()));
27         // visualizzazione risposta
28         i = is.read(buffer, 0, dimBuffer);
29         stringa = new String(buffer, 0, i);
30         System.out.println(stringa);
31     }
32 }
```

Figura 8.9 Fruizione del servizio di scambio di stringhe con socket connesse.

Nonostante il client appena proposto funzioni, esso presenta un errore di progettazione non indifferente: non termina mai. Sotto un aspetto formale sarebbe come aver progettato un protocollo il cui automa non presenta stati finali. Da un punto di vista teorico questo non è un problema, ma all'atto pratico non è applicabile: il client dovrà terminare, se non al ricevimento di un messaggio di chiusura quantomeno di fronte alla chiusura della socket. Nella Figura 8.10 viene esteso il costruttore in modo da terminare quando viene rilevata la chiusura della socket; per fare questo si è considerato il fatto che il metodo *read* della classe *InputStream* restituisce un numero di byte letti inferiore a zero se la socket è stata chiusa dalla controparte. Leggendo attentamente il codice, però, si può notare anche che si è ipotizzato che la socket possa essere chiusa solo durante la fase di lettura, ovvero, che il server chiuda la connessione a fronte di una richiesta del client.

```
27 // visualizzazione risposta  
28 r = is.read(buffer, 0, dim_buffer);  
29 if (r > 0)  
30 strings = new String(buffer, 0, r);  
31 System.out.println(strings);
```

Figura 8.10 Rilevamento del fatto che il server ha chiuso la connessione.

Esercizio 8.2



Modificare il programma proposto fino alla Figura 8.10 in maniera tale da usarlo come client per il server sviluppato per l'Esercizio 7.3. Il client deve prima ricevere dal server una stringa e visualizzarla, poi effettuare il ciclo per inviare stringhe al server e attendere le relative risposte.

Esercizio 8.3



Modificare il programma proposto fino alla Figura 8.10 in maniera tale da far terminare il client nel momento in cui l'utente digita una stringa composta unicamente dal carattere “.”.

8.3.2 Client con socket non connesse

Confrontando tra loro i diagrammi nella Figura 8.4, è piuttosto semplice, partendo dal codice proposto nel paragrafo precedente, ricavare un client che faccia uso di socket non connesse e possa usufruire del server per la ricezione di una stringa proposta nella Figura 7.21.

Innanzitutto, rispetto alla Figura 8.5 occorre utilizzare la classe *DatagramSocket*, in quanto le socket TCP degli esempi precedenti non supportano la modalità non connessa. La socket creata non sarà associata a una controparte, tuttavia va mantenuta la gestione dei parametri perché necessari al metodo *send*. Per quanto riguarda, invece, la classe che si occupa della fruizione del servizio nella Figura 8.8, non sono più necessarie le due istanze per i canali di I/O e le chiamate a *send* e *receive* faranno uso di un'istanza di *DatagramPacket* per accedere al buffer.

Le modifiche appena descritte, per la parte di gestione della socket e per la fruizione del servizio sono presentate rispettivamente nelle Figure 8.11 e 8.12.

In particolare, per quanto riguarda la classe che si occupa della fruizione del servizio, è stato necessario estendere il numero di parametri del costruttore: se la socket non è connessa è necessario rendere noto anche l'indirizzo di trasporto a cui inviare il messaggio. È importante ricordare che

DatagramSocket non è una sottoclasse di *Socket*, quindi si è costretti anche a cambiare il tipo del primo parametro.

```
1 import java.net.DatagramSocket;
2 import java.net.InetSocketAddress;
3
4 public class ClientNonconnesso {
5
6     public static void main(String args[]) {
7
8         try {
9             // parametri
10            String nome_host = "localhost";
11            int porta = 7000;
12
13            // acquisizione parametri
14            if (args.length != 2) {
15                throw new IllegalArgumentException(
16                    "Numero parametri non corretto");
17            }
18
19            nome_host = args[0];
20            porta = Integer.parseInt(args[1]);
21            if (porta < 0)
22                throw new IllegalArgumentException(
23                    "Porta non valida");
24        }
25
26        // creazione socket
27        DatagramSocket ds = new DatagramSocket();
28
29        // scambio dati
30        InetSocketAddress iss =
31            new InetSocketAddress(nome_host, porta);
32        new FrasiInLineaServer(ds, iss);
33
34        // chiusura canale
35        ds.close();
36
37    } catch (Exception e) {
38        e.printStackTrace();
39    }
40 }
41 }
```

Figura 8.11 Client con socket non connesse.

In questo caso non è possibile implementare un equivalente del programma richiesto per l'Esercizio 8.1 in quanto, mancando la fase di connessione, il server non è in grado di *prendere l'iniziativa* e mandare un *prompt* al client.

L'aggiunta di un ciclo di lettura e scrittura (Figura 8.13) viene invece semplificata. Infatti, essendo la socket non connessa, non è più necessario (possibile) controllare se il server ha chiuso il canale di connessione; però, non avendo un riscontro dell'effettiva consegna dei pacchetti a livello di trasporto, tutti i messaggi da un certo momento in avanti potrebbero essere persi perché inviati a un indirizzo non più associato a una socket.

La soluzione appena proposta e rappresentata nella Figura 8.13 presenta però due gravi problemi.

```

8 public class FruizioneServizio {
9     public DatagramService(DatagramSocket socket,
10                         InetAddress iaa) throws IOException {
11         try {
12             InputStreamHeader iis = new InputStreamHeader(system.in);
13             BufferHeader br = new BufferHeader(iis);
14             br.setLength(10);
15             String strings = br.readline();
16             strings += "\r\n";
17             byte[] buffer = strings.getBytes();
18             DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
19             dp.setSocketAddress(iaa);
20             socket.send(dp);
21         } catch (IOException e) {
22         }
23     }
24 }

```

Figura 8.12 Fruizione del servizio di spedizione di una stringa con socket non connesse.

```

8 Public class FruizioneServizio {
9     public DatagramService(DatagramSocket socket,
10                         InetAddress iaa) throws IOException {
11         try {
12             int dimBuffer = 1024; // per la gestione della risposta
13             byte[] buffer; // per la gestione della risposta
14             InputStreamHeader iis = new InputStreamHeader(system.in);
15             BufferHeader br = new BufferHeader(iis);
16             br.setLength(dimBuffer);
17             String strings;
18             while (true) {
19                 strings = br.readline();
20                 strings += "\r\n";
21                 buffer = strings.getBytes();
22                 DatagramPacket dp = new DatagramPacket(buffer,
23                     buffer.length, iaa);
24                 dp.setSocketAddress(iaa);
25                 socket.send(dp);
26             }
27         } catch (IOException e) {
28             byte[] buffer = new byte[dimBuffer];
29             dp = new DatagramPacket(buffer, dimBuffer);
30             socket.receive(dp);
31             strings = new String(buffer, 0, dp.getLength());
32             system.out.println(strings);
33         }
34     }
35 }
36 }
37 }

```

Figura 8.13 Fruizione del servizio di scambio di stringhe con socket non connesse.

Il primo riguarda la sicurezza: per essere assolutamente certi dell'autenticità della risposta è necessario confrontare l'indirizzo di trasporto del pacchetto in partenza con l'indirizzo di quello in arrivo. Diversamente, un altro programma malintenzionato potrebbe intercettare la richiesta e rispondere al posto del server, influenzando in maniera anomala il comportamento del client.



Esercizio 8.4

Modificare il programma proposto nella Figura 8.13 in maniera tale da prevenire attacchi di sicurezza.

Il secondo problema, invece, riguarda l'affidabilità: il client implementa un protocollo di tipo *stop-and-wait* (in cui cioè vi è un'alternanza stretta nello scambio di messaggi) su un canale inaffidabile; se un messaggio viene perso il sistema si blocca irrimediabilmente. Purtroppo, con gli strumenti presi in esame fin qui non è ancora possibile risolvere questo problema.

Esercizio 8.5



Modificare il programma proposto nella Figura 8.13 in maniera tale da far terminare il client nel momento in cui l'utente digita una stringa composta unicamente dal carattere “.”.

Esercizio 8.6



Si modifichi il codice riportato nella Figura 8.11 in maniera tale da aggiungere un terzo parametro alla linea di comando per permettere all’utente di scegliere se il client deve utilizzare socket con connessione o senza connessione.

Sempre nella Figura 8.13 è possibile osservare una continua riallocazione delle variabili relative al *buffer* e al messaggio. In generale è una pessima abitudine fare ipotesi sulle dimensioni del messaggio in arrivo: il messaggio in partenza ha la dimensione della stringa scritta dall’utente ma quello in arrivo potrebbe anche essere una sua manipolazione, per cui è meglio definire un buffer che si stima sufficientemente grande ed effettuare la ricezione con una nuova istanza di *DatagramPacket*, più capiente.

8.4 Implementazione di un server iterativo

Come nel caso del client, per semplificare la comprensione del codice, la fase di scambio dati verrà collocata nel costruttore di una classe apposita (*ErogazioneServizio*).

8.4.1 Server iterativo con socket connesse

Combinando tra loro i diagrammi di client e server iterativi e sostituendo le operazioni con le opportune chiamate, si ottiene quanto rappresentato nella Figura 8.14.

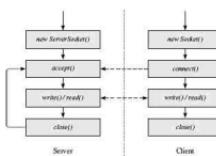


Figura 8.14 Diagramma delle operazioni congruenti di client e server iterativo con socket connesse.

In modo simile a quanto già visto analizzando la struttura del client, sono quattro le operazioni che è possibile tradurre in codice. Il programma parte facendo l'analisi dei parametri e creando un'istanza della classe *ServerSocket*. In questo caso ci si trova di fronte a necessità diverse e il testo del programma cambia, come si può osservare nella Figura 8.15.

```
8 Public class ServerIterativo {
9     public static void main(String args[]) {
10         try {
11             // parametri
12             int porta = 0;
13
14             // acquisizione parametri
15             if (args.length > 1) {
16                 throw new IllegalArgumentException(
17                     "Numero parametri non corretto");
18             }
19             if (args.length == 1) {
20                 porta = Integer.parseInt(args[0]);
21                 if (porta < 0)
22                     throw new IllegalArgumentException(
23                         "Porta non valida");
24             }
25
26             // creazione socket
27             ServerSocket ss = new ServerSocket(porta);
28             if (ss != null) {
29                 System.out.println("Porta allocata: " +
30                     ss.getLocalPort());
31             }
32         }
```

Figura 8.15 Gestione dei parametri e creazione della socket per un server iterativo.

Innanzitutto non è più necessaria una variabile con il nome di un nodo a cui collegarsi; secondariamente, quello della porta diviene un parametro facoltativo, per cui l'unico vincolo è che il numero di parametri non sia superiore a 1 (riga 13). Il valore di default per la porta è 0, quindi, in assenza di parametri si lascia che sia il sistema operativo a selezionarne una. Se è specificata una porta sulla quale mettersi in ascolto (riga 18), allora viene interpretato il parametro (riga 19); in questo caso anche il valore 0 è e accettabile (riga 20) perché l'utente potrebbe voler esplicitamente lasciare la scelta al sistema. La riga 29 viene eseguita solo nel caso un cui il valore della porta sia uguale a 0 e comunica all'utente la porta alla quale la socket è stata associata.

Fino a questo punto il programma è nella fase iniziale: sono state infatti solo allocate le risorse per permettere l'erogazione del servizio.

Il codice relativo al ciclo di gestione dei client viene proposto nella Figura 8.16.

```
32      // ciclo di gestione dei client
33      while(true) {
34          // ricezione connessione
35          Socket s = server.accept();
36
37          // scambio dati
38          new KlientInneinServizio(s);
39
40          // chiusura canale dati
41          s.close();
42
43      }
44  } catch (Exception e) {
45      e.printStackTrace();
46  }
47 }
```

Figura 8.16 Il ciclo di gestione dei client di un server iterativo con socket connesse.

Come si può notare, la socket principale non viene mai chiusa e il programma non termina mai; all'interno del ciclo (righe dalla 35 alla 42) viene per prima cosa effettuata una chiamata al metodo *accept* (riga 36). Tale chiamata blocca l'esecuzione fino a che un client non effettuerà una connessione, nel momento in cui l'esecuzione riprende viene erogato il servizio tramite la socket dati appena creata (riga 39) e successivamente vengono rilasciate le risorse (riga 42) per tornare in attesa del prossimo client.

Avendo terminato tutti i passaggi necessari a gestire i client, la classe di erogazione del servizio diventa a questo punto l'unica preoccupazione; si tratta di usare i metodi *write* e *read* in maniera opportuna per scambiare dati, secondo un protocollo stabilito, con la classe di fruizione del servizio implementata dal client. Nella Figura 8.17 viene presentato il codice in grado di dialogare con il client proposto nella Figura 8.9: il client manda un messaggio al server chelo rimanda al client.

Il servizio appena descritto prende il nome di “echo” [63], standardizzato dalla IANA sulla porta 7 sia per UDP che per TCP.

```
 6 public class ErogazioneServizio {
 7     public ErogazioneServizio(Socket socket) throws IOException {
 8         int dim_buffer = 1024;
 9         byte[] buffer = new byte[dim_buffer];
10         int i;
11         int r;
12         InputStream is = socket.getInputStream();
13         OutputStream os = socket.getOutputStream();
14         while (true) {
15             r = is.read(buffer, 0, dim_buffer);
16             if (r < 0)
17                 os.write(buffer, 0, r);
18             else
19                 return;
20         }
21     }
22 }
23 }
24 }
```

Figura 8.17 Erogazione del servizio echo con socket connesse.

Nonostante la classe di erogazione del servizio sia molto breve, vale la pena fare qualche commento. Il servizio non entra nel merito dei dati: legge dalla socket una sequenza di byte e ve li riscrive, per cui non è necessario adattare la sequenza per il ritorno a capo. Inoltre, il valore di ritorno del metodo *read* viene usato come discriminante: se sono stati letti dei dati e l'espressione alla riga 18 risulta vera il buffer viene rispedito al mittente; diversamente, l'erogazione del servizio termina.

È assai improbabile trovare oggi nodi su Internet con il servizio echo disponibile. Il servizio echo, infatti, può facilmente essere usato per un attacco di tipo DoS: un client malintenzionato manda messaggi indiscriminatamente in maniera tale da utilizzare tutta la banda disponibile e prevenire l'erogazione di altri servizi.



Esercizio 8.7

Modificare il programma proposto nella Figura 8.17 in maniera tale da stampare informazioni sulle stringhe in transito.

Esercizio 8.8



Modificare il programma proposto nella Figura 8.17 in maniera tale per cui nel momento in cui il client invia un messaggio composto unicamente dal carattere “.” il server eroga il servizio e poi termina.

Esercizio 8.9



Implementare un server iterativo per l'erogazione del servizio *chargen* [64]. Se ne verifichi il buon funzionamento facendo uso del comando telnet.

Esercizio 8.10



Implementare un server iterativo per l'erogazione di un servizio di calcolo secondo la notazione polacca inversa (RPN). Un'operazione si compone di tre messaggi in sequenza: due numeri che possono essere interi o in virgola mobile e l'operazione da effettuare. Quando riceve il messaggio relativo all'operazione il server calcola il risultato, lo manda al client e chiude la connessione. Se ne verifichi il buon funzionamento facendo uso del comando telnet.

8.4.2 Server iterativo con socket non connesse

Combinando tra loro i diagrammi di client e server iterativi facendo uso di socket non connesse e operando le opportune sostituzioni alle diciture dei blocchi si ottiene il diagramma congiunto rappresentato nella Figura 8.18. Si noti, in particolare, la simmetria: mentre nel caso di socket connesse si osservavano da una parte *Socket* e dall'altra *ServerSocket*, in questo si fa uso di *DatagramSocket* sia lato client che lato server.

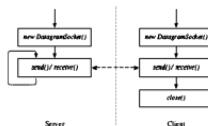


Figura 8.18 Diagramma delle operazioni congiunte di client e server iterativo con socket non connesse.

Un server di questo tipo può essere ricavato facilmente modificando i sorgenti proposti nelle Figure 8.15 e 8.16 facendo una serie di considerazioni simili a quelle viste per un client con socket non connesse. La socket deve essere un'istanza di *DatagramSocket* e l'operazione di ricezione della connessione non è più necessaria; il ciclo di gestione dei client, infine, viene effettuato sulla creazione dell'istanza all'interno della classe principale.

La Figura 8.19 mostra la differenza rispetto al codice visto nel caso di socket connesse.

```
1 import java.net.DatagramSocket;
2 ...
3     // creazione socket
4     DatagramSocket ds = new DatagramSocket(porta);
5 ...
6     // ciclo di gestione dei client
7     while(true) {
8         // ricezione dati
9         new ErogazioneServizio(ds);
10    }
11 }
```

Figura 8.19 Server iterativo con socket non connesse.

Relativamente alla classe che si occupa dell'erogazione del servizio ci si trova di fronte a un cambiamento: mancando la fase di ricezione della connessione ogni messaggio deve essere gestito come un'informazione a sé stante in quanto non è detto che due messaggi successivi provengano dallo stesso client. Le alternative possibili sono due: ricevere il messaggio all'interno del ciclo di gestione dei client e demandare solo la risposta alla classe di erogazione del servizio oppure svolgere tutte e le operazioni all'interno di quest'ultima. La prima soluzione

è sconsigliata in quanto complica la struttura del programma e operazioni di lettura e scrittura correlate tra loro sarebbero collocate in punti diversi del programma. La seconda è di più facile implementazione e permette di mantenere inalterati i parametri del costruttore, per cui negli esempi si è scelto di seguire questa strada.

Il codice del costruttore risulta essere quasi identico alla versione facente uso di socket connesse se non per il fatto che gestisce un solo messaggio (Figura 8.20).

```
5 public class ErogazioneServizio {
6     public ErogazioneServizio(DatagramSocket socket) throws IOException {
7         dimBuffer = 1024;
8         byte[] buffer;
9         Buffer = new byte[dimBuffer];
10        Buffer = Buffer;
11        DatagramSocket dp = new DatagramPacket(buffer, buffer.length);
12        socket.receive(dp);
13        socket.send(dp);
14    }
15 }
```

Figura 8.20 Erogazione del servizio echo con socket non connesse.

Nell'esempio l'istanza di *DatagramPacket* utilizzata per ricevere il messaggio viene immediatamente inviata sulla rete senza modifiche. Ciò è possibile solo in questo caso specifico: all'interno di *DatagramPacket* è presente un'unica informazione riguardante un indirizzo a livello di trasporto e questa non è stata modificata, per cui l'indirizzo di provenienza verrà usato come indirizzo di destinazione.

Esercizio 8.11



Simodifichi il codice relativo a un server iterativo facente uso di socket connesse in maniera tale da aggiungere un secondo parametro alla linea di comando per permettere all'utente di scegliere se il server deve utilizzare socket con connessione o senza connessione.

8.5 Implementazione di un server concorrente

Il linguaggio Java non ha la possibilità di manipolare processi ma solo thread che, a differenza dei processi, condividono un unico spazio di indirizzamento. L'avere un unico spazio di indirizzamento rende possibile erogare tramite un insieme di thread servizi che prevedono condivisione di informazioni tra i client e rende inutile la complicazione imposta dalla fase di attesa concorrente. Per questo motivo non sono disponibili meccanismi per implementare un server concorrente e l'erogazione di servizi concorrenti viene affidata a server multithread.

8.6 Implementazione di un server multiprocesso/multithread

Il diagramma congiunto che descrive l'interazione tra server multithread e client è rappresentato nella Figura 8.21.

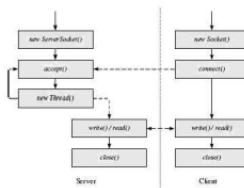


Figura 8.21 Diagramma delle operazioni congiunte di client e server multithread.

Il linguaggio Java permette la gestione dei thread tramite la creazione di istanze di classi specializzate. Per essere eseguita in un thread separato una classe può seguire due approcci: implementare l'interfaccia *Runnable* oppure estendere la classe *Thread*. Nel primo caso sarà necessario creare un'istanza di *Thread* utilizzando come parametro un'istanza della classe sviluppata, nel secondo la classe potrà essere istanziata direttamente. In entrambi i casi la classe che implementa il nuovo thread deve prevedere un metodo *run* che verrà eseguito in un flusso esecutivo separato nel momento in cui sarà fatta una chiamata al metodo *start* dell'istanza. Nel momento in cui il metodo *run* termina, il thread viene bloccato e le risorse allocate saranno rese nuovamente disponibili.

Il codice di un server di questo tipo si ottiene con variazioni minime partendo da quello di un server concorrente. Per prima cosa la classe *ErogazioneServizio* deve essere eseguita in un thread a parte. Per ottenere questo effetto è sufficiente definirla come estensione della classe *Thread* e spostare il codice nel costruttore all'interno del metodo *run*, come descritto nella Figura 8.22, ottenuta a partire dalla Figura 8.17.

```
1 import java.net.Socket;
2 import java.io.InputStream;
3 import java.io.OutputStream;
4
5 public class ErogazioneServizio extends Thread {
6
7     private Socket socket;
8
9     public ErogazioneServizio(Socket socket) {
10         this.socket = socket;
11     }
12
13     public void run() {
14         try {
15             int dimBuffer = 1024;
16             byte[] buffer = new byte[dimBuffer];
17             int r;
18
19             InputStream is = socket.getInputStream();
20             OutputStream os = socket.getOutputStream();
21
22             while (true) {
23                 r = is.read(buffer, 0, dimBuffer);
24                 if (r > 0)
25                     os.write(buffer, 0, r);
26                 else
27                     return;
28             }
29         } catch (Exception e) {
30             e.printStackTrace();
31         }
32     }
33 }
```

Figura 8.22 Erogazione del servizio echo in un thread separato.

Come si può notare, sono presenti anche altre modifiche: innanzitutto è stato necessario aggiungere un campo privato per permettere al metodo *run* di avere visibilità del parametro passato al costruttore (righe 7 e 10); secondariamente, non è più possibile sollevare eccezioni da inviare alla classe principale, per cui la gestione degli errori deve essere fatta localmente (righe dalla 29 alla 31).

Per quanto riguarda invece il server principale, per attivare un server dedicato è sufficiente una chiamata al metodo *start* subito dopo aver creato l'istanza, come nell'esempio della Figura 8.23.

```
32     // ciclo di gestione dei client
33     while(true) {
34         // accettazione connessione
35         Socket s = ss.accept();
36         // scambio dati
37         Thread t = new ErogazioneServizio(s);
38         t.start();
39     }
40     // chiusura canale dati
41     ss.close();
42 }
43 }
```

Figura 8.23 Gestione dei client in un server multithread.

Nel caso i thread relativi ai server dedicati dovessero accedere in maniera concorrente alle risorse è importante garantire la mutua esclusione tramite la direttiva *synchronized*.



In realtà è possibile semplificare ulteriormente lo schema con un accorgimento molto semplice: la partenza del nuovo thread viene innescata dall'interno del costruttore e non dal server principale, come nella Figura 8.24. Questo minimo stratagemma permette di non effettuare nessun cambiamento al codice del server principale, quindi può essere usato quello proposto nella Figura 8.17.

```
9     public ErogazioneServizio(Socket socket) {
10        this.socket = socket;
11    }
12 }
```

Figura 8.24 Erogazione di un servizio in un thread con partenza autonoma.

Si noti che, in questo modo il server non solo è indipendente dal servizio offerto, ma non è neppure più vincolato al modello adottato.



Esercizio 8.12



Thread multipli possono essere usati per risolvere il problema dell'alternanza stretta.

Si implementi un client per il servizio echo che sia in grado di attendere dati parallelamente dalla socket con cui è collegato al server e dall'utente.

Capitolo 9

Accesso alle risorse web

La comparsa di Java è di poco posteriore alla nascita del Web, non c'è quindi da meravigliarsi se all'interno del linguaggio sono presenti molte classi appositamente studiate per fruire di questo tipo di servizio. In questo capitolo si discuterà di come utilizzare queste specifiche classi e di quali accorgimenti occorre tenere conto.

9.1 Servizio web

Java mette a disposizione una serie di classi per astrarre la connessione con un web server, le più importanti sono: *URL*, *URLConnection* e *HttpURLConnection*.

La classe *URL* (Figura 9.1) è un contenitore per un URL: un riferimento a una risorsa accedibile via Web. Tra i suoi metodi quello più utile in questo caso è *openConnection*, che crea una

connessione con il server e restituisce un'istanza della classe *URLConnection*.

```
public final class URL extends Object {
    ...
    URLConnection openConnection();
    ...
}
```

Figura 9.1 La classe *URL* e il suo metodo principale.

La classe *URLConnection* (Figura 9.2) rappresenta l'astrazione di una generica connessione con un web server. I metodi di questa classe permettono di controllare la modalità con cui effettuare l'accesso alla risorsa. Una volta impostati tutti i parametri si fa uso del metodo *connect* per aprire fisicamente il canale e da quel momento in poi sarà possibile accedere al contenuto relativo alla risorsa; tale contenuto potrebbe essere di natura testuale ma anche binario.

```
public abstract class URLConnection extends Object {
    ...
    void setRequestProperty(String campo, String valore);
    int getConnectTimeout();
    String getHeaderFieldNames();
    InputStream getInputStream();
    OutputStream getOutputStream();
    ...
}
```

Figura 9.2 La classe *URLConnection* e i suoi metodi principali.

In ultimo, la classe *HttpURLConnection* (Figura 9.3) è una sottoclasse di *URLConnection* implementa metodi e campi specifici del protocollo HTTP.

```
public abstract class HttpURLConnection extends URLConnection {
    ...
    void setRequestMethod(String metodo);
    int getResponseCode();
    String getResponseMessage();
    ...
}
```

Figura 9.3 La classe *HttpURLConnection* e i suoi metodi principali.

In realtà il metodo *openConnection* di URL restituisce solo in rare occasioni una classe che non è un'istanza di *HttpURLConnection*, e anche quando succede è per richiesta del programmatore.



Il client deve gestire il canale di comunicazione e un buffer per ospitare i dati in arrivo dal server; tali dati potrebbero essere utilizzati internamente al client stesso o forniti in input a un'altra applicazione.

Un client per il servizio web non implementa necessariamente un'interfaccia grafica e non sempre ha come obiettivo la navigazione dei contenuti, tant'è vero che esistono web browser puramente testuali, come lynx [65] e w3m [66], come pure client per il solo ascolto di musica via Web, come winamp [67].



9.2 Protocollo HTTP

Tutti i dettagli del protocollo HTTP sono nascosti dalle classi che operano l'astrazione; non è quindi strettamente necessario conoscere la conformazione dei messaggi, ma solo le funzionalità disponibili. Rimane comunque il problema di come gestire il contenuto in arrivo dal server, ma anche questo non presenta particolari difficoltà in quanto basta fare uso di un buffer. La classe *URLConnection* definisce infatti due metodi: *getInputStream* e *getOutputStream* con le stesse finalità dei metodi omonimi della classe *Socket*.

9.2.1 Messaggio di richiesta

Una volta ottenuta un'istanza della classe *HttpURLConnection* è possibile impostare tutti i parametri con cui effettuare la richiesta con il metodo *setRequestProperty* e aprire la connessione con il server invocando il metodo *connect*. Nella Figura 9.4 viene proposto il codice di un client che si collega alla risorsa indicata come primo parametro dell'applicazione.

```
1 import java.net.URL;
2 import java.net.HttpURLConnection;
3
4 public class WebClient {
5
6     public static void main(String args[]) {
7
8         try {
9             // creazione del canale di comunicazione
10            URL u = new URL(args[0]);
11            HttpURLConnection uuc = (HttpURLConnection) u.openConnection();
12
13            // impostazione delle opzioni
14            uuc.setRequestProperty("User-Agent", "CustomClient v1.0");
15
16            // invio della richiesta
17            uuc.connect();
18
19        } catch (Exception e) {
20            e.printStackTrace();
21        }
22    }
23}
```

Figura 9.4 Invio di un messaggio di richiesta.

9.2.2 Messaggio di risposta

Quando la chiamata al metodo *connect* restituisce il controllo al programma, è stata effettuata la connessione al server e i dati relativi a intestazione e header sono stati già analizzati dalla classe *HttpURLConnection*. La Figura 9.5 mostra le informazioni relative alla linea iniziale e all'header dopo che è stata effettuata la connessione.

Nelle righe dalla 23 alla 26 vengono estratti codice di risposta e commento; all'interno del ciclo dalla riga 33 in poi vengono estratte tutte le opzioni dell'header. La variabile *i* è usata come indice e il ciclo termina quando il suo valore supera il numero delle opzioni disponibili (alla riga 33 il valore di ritorno di *getHeaderFieldKey* è null); se invece esiste un campo con un certo indice, il relativo valore viene estratto con il metodo *getHeaderField* e visualizzato (righe 34 e 35).

```
22     // ricezione della risposta
23     System.out.println("Codice di risposta: " +
24         " " + huc.getResponseCode());
25     System.out.println("Commento: " +
26         " " + huc.getResponseMessage());
27
28     System.out.println("Header:");
29     String campo;
30     String valore;
31     int i = 0;
32
33     while ((campo = huc.getHeaderFieldKey(i)) != null) {
34         valore = huc.getHeaderField(i);
35         System.out.println(campo + " => " + valore);
36         i++;
37     }
38 }
```

Figura 9.5 Interpretazione di un messaggio di risposta.

9.3 Accesso a una risorsa

L'ultimo elemento necessario per accedere a una risorsa è il metodo di accesso; la sua impostazione è abbastanza banale e prevede l'uso del metodo *setRequestMethod* alla riga 16. Negli esempi che seguono si userà come base di partenza il programma ottenuto unendo il codice delle Figure 9.4 e 9.5.

In realtà, il codice prodotto finora è già in grado di funzionare correttamente perché esiste un valore di default per il metodo di accesso: GET



9.3.1 HEAD

Per usare il metodo HEAD è sufficiente modificare il codice come indicato nella Figura 9.6.

```
16     huc.setRequestMethod("HEAD");
```

Figura 9.6 Possibile impostazione per un test del metodo HEAD.

Eseguendo il programma così ottenuto si otterrà un output simile a quello che segue; per chiarezza, sono state riportate solo le parti più significative dell'header.

```
Codice di risposta: 200
Commento: OK
Header:
Server -> Microsoft-IIS/6.0
Last-Modified -> Fri, 10 Oct 2008 12:45:53 GMT
Content-Length -> 22559
Content-Type -> text/html
Connection -> close
```

Esercizio 9.1



Si modifichi il client in maniera tale da utilizzare l'opzione *Accept* per richiedere solo contenuti di tipo *text/html* e se ne verifichi il buon funzionamento.

9.3.2 GET

Per inviare una richiesta di tipo GET occorre innanzitutto cambiare il metodo di accesso alla riga 16 e successivamente estendere il codice per effettuare la ricezione del contenuto. Le variazioni necessarie sono proposte nella Figura 9.7.

Alla riga 40 è stata introdotta una nuova variabile *dimensione* in cui memorizzare il quantitativo di byte da leggere; tale variabile viene assegnata trasformando in un intero il valore del campo *Content-Length*. Il ciclo nelle righe dalla 49 alla 54 legge il contenuto dal canale di comunicazione e lo visualizza; la condizione di permanenza all'interno del ciclo è che ci siano ancora byte da ricevere; infatti, la variabile *dimensione* viene ogni volta decrementata del numero di byte letti (riga 53) e il ciclo termina quando questa arriva al valore 0. Il quantitativo di byte (variabile *da_leggere*) va calcolato di volta in volta alla riga 50: deve essere il quantitativo minimo tra la dimensione del buffer e il numero di byte ancora da ricevere.

Eseguendo il programma così composto si ottiene in output il testo della pagina principale di <http://www.lila.it/>.

```
Codice di risposta: 200
Commento: OK
Header:
Server -> Microsoft-IIS/6.0
Last-Modified -> Fri, 10 Oct 2008 12:45:53 GMT
Content-Length -> 22559
Content-Type -> text/html
Connection -> close Contenuto:
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional...
...
```

```

1 /**
2  * Import java.io.InputStream
3  */
4
5     hu.setReadBufferSize("8K");
6
7     // Identificazione della lunghezza del contenuto
8     int dimensione = Integer.parseInt(hu.getHeaderField("Content-Length"));
9
10    // ricezione del contenuto
11    System.out.println("Dimensione:");
12    System.out.println(dimensione);
13
14    InputStream is = hu.getInputStream();
15    int daLeggere = dimensione;
16    byte[] buffer = new byte[dimensione];
17
18    int r, daLeggere;
19
20    do {
21        daLeggere = Math.min(dimensione, dim_buffer - 1);
22        r = is.read(buffer, 0, daLeggere);
23        System.out.println(new String(buffer, 0, r));
24        dimensione -= r;
25    } while (dimensione > 0);
26
27
28 } catch (Exception e) {
29     e.printStackTrace();
30 }
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46

```

Figura 9.7 Modifiche al client per la ricezione del contenuto.

Se però il server non dovesse fornire l'opzione *Content-Length* nell'header, la situazione si complicherebbe leggermente: il termine del contenuto corrisponderebbe alla chiusura del canale.

In tal caso non è più possibile mantenere inalterata l'assegnazione della variabile *dimensione*: se l'opzione *Content-Length* non esiste, il valore di ritorno di *getHeaderField* è null e la chiamata a *parseInt* solleva un'eccezione. Una soluzione facilmente praticabile è assegnare inizialmente a *dimensione* il massimo valore possibile (*Integer.MAX_VALUE* o $2^{32} - 1$) e impostarlo diversamente solo se specificato nell'header, in questo modo il client si predisponde per leggere una sequenza virtualmente infinita di byte. A questo punto però il problema diventa la terminazione del ciclo di lettura, che deve dipendere, oltre che dal numero di byte letti anche dal fatto che il canale sia stato chiuso dal server. La Figura 9.8 mostra il codice relativo ai cambiamenti appena descritti.

Esercizio 9.2



Utilizzare il metodo GET per ottenere il contenuto relativo a un’immagine e anziché visualizzarlo a video lo si memorizzi sul disco. Se ne verifichi il buon funzionamento visualizzando l’immagine.

```
39 // identificazione della lunghezza del contenuto
40 String dimensione_m = huc.getHeaderField("Content-Length");
41 int dimensione_m_i = -1;
42 if (dimensione_m != null) Integer.parseInt(dimensione_m);
43 ...
44 do {
45     da_lepaga = Math.min(dimensione, dim_buffer);
46     r = s.getInputStream().read(da_lepaga);
47     if (r > 0) out.print(new String(buffer, 0, r));
48     dimensione -= r;
49 } while (dimensione > 0 && r > 0);
```

Figura 9.8 Ricezione del contenuto in assenza di *Content-Length*.

9.4 Contenuti statici e dinamici

L’accesso a contenuti dinamici con il metodo GET è piuttosto banale: basta aggiungere i vari parametri in coda all’URL da contattare e in questo caso lo si può specificare sulla linea di comando.

```
java WebClient "http://www.google.it/search?hl=en&q=client"
```

Nei sistemi operativi UNIX i caratteri “?” e “&” hanno un significato particolare per l’interprete dei comandi; in tal caso è opportuno racchiudere il parametro tra doppi apici.



9.4.1 POST

Per poter fare un esempio del metodo POST non è possibile utilizzare nuovamente Google, in quanto il server accetta solo richieste con GET. Yahoo, invece, può essere usato anche con il metodo POST; il codice per effettuare l’interrogazione è proposto nella Figura 9.9.

È possibile notare un certo numero di cambiamenti. Innanzitutto è stata inserita una variabile *contenuto* per memorizzare i parametri (righe 16 e 17); successivamente, nelle righe dalla 21 alla 24, sono state aggiunte all’header le due opzioni *Content-Type* e *Content-Length*. L’invio del contenuto (riga 30) avviene tramite un’istanza di *OutputStream* ricavata da una chiamata al metodo *getOutputStream* alla riga 29; il canale di output così ottenuto non può essere usato a meno che non lo si comunichi preventivamente a *HttpURLConnection* tramite una chiamata al metodo *setDoOutput* (riga 25).

```

10    // creazione del canale di comunicazione
11    URL u = new URL("http://it.search.yahoo.com/web");
12    HttpURLConnection httpURLConnection = (HttpURLConnection) u.openConnection();
13
14    // impostazione delle opzioni
15    String contenuto_s = "p(\"client-server\"");
16    byte[] contenuto = contenuto_s.getBytes();
17
18    HttpURLConnection huc = (HttpURLConnection) u.openConnection();
19    huc.setRequestProperty("User-Agent", "CustomClient v1.0");
20    huc.setRequestProperty("Content-Type",
21        "application/x-www-form-urlencoded");
22    huc.setRequestProperty("Content-Length",
23        Integer.toString(contenuto.length));
24    huc.setDoOutput(true);
25
26    // invio della richiesta
27    huc.connect();
28    OutputStream os = huc.getOutputStream();
29    os.write(contenuto, 0, contenuto.length);
30

```

Figura 9.9 Richiesta con metodo POST al motore di ricerca di Yahoo.



Esercizio 9.3

Verificare, e possibilmente anche gestire, l'errore restituito da Google nel momento in cui si usa il metodo POST.

9.5 Connessione persistente e non persistente

L'astrazione di *HttpURLConnection* gestisce autonomamente le connessioni persistenti, compresi timeout ed eventuali riconnessioni. Per richiedere una connessione non persistente è sufficiente assegnare al campo “*Connection*” il valore “*Close*”; questa operazione può essere giustificata se si vuole esplicitamente ridurre il numero di canali di I/O utilizzati da un thread, altrimenti si rinuncia al beneficio di una connessione potenzialmente più rapida senza di fatto semplificare il codice.

9.6 Web proxy

Tecnicamente, la modalità con cui si usufruisce del servizio non subisce nessuna modifica a fronte dell'uso di un proxy; Java mette comunque a disposizione varie modalità, più o meno invasive per il programmatore. L'infrastruttura di Java permette infatti di impostare un proxy perché sia usato in maniera trasparente dalle varie classi sia a livello di singola connessione che a livello globale di sistema.

Per usare un proxy limitatamente a una connessione è possibile fare uso della classe *Proxy* e usare una variante del metodo *openConnection* della classe *URL* come descritto nella Figura 9.10.

```
10 // creazione del canale di comunicazione
11 URL u = new URL("http://[0]:");
12 HttpURLConnection uc = (HttpURLConnection) u.openConnection();
13 uc.setProxy(new Proxy(Proxy.Type.HTTP, new InetSocketAddress("proxy.private.net", 8080)));
14 Proxy p = new Proxy(Proxy.Type.HTTP, l);
15 uc.setProxy(p);
16 uc.openConnection();
```

Figura 9.10 Apertura di un canale di comunicazione tramite un proxy.

Nell'esempio proposto viene creata un'istanza della classe *Proxy* alla riga 14 definendo il tipo di proxy (in questo caso HTTP) e l'indirizzo a livello di trasporto sul quale il server è in attesa. Questa istanza viene poi utilizzata come parametro nella chiamata del metodo *openConnection* (righe 15 e 16) e fa sì che il servizio venga fruito usando come intermediario l'indirizzo specificato alla riga 13; il resto del programma rimane

inalterato. L'uso di un proxy per una singola connessione può essere utile in situazioni per cui un certo servizio deve essere obbligatoriamente fruito attraverso uno specifico punto di accesso (ad esempio un firewall) mentre il resto del traffico segue un percorso standard.

L'impostazione di un proxy a livello di infrastruttura, invece, permette di condizionare il comportamento di tutte le classi eseguite (anche quelle implementate da terze parti e per cui non si dispone dei sorgenti) senza modificare la struttura del programma. Per farlo è necessario modificare la configurazione dell'ambiente di esecuzione tramite il metodo statico *setProperty* della classe *System*, come indicato nella Figura 9.11.

```
10 // impostazione del proxy
11 System.setProperty("http.proxyHost", "proxy.private.net");
12 System.setProperty("http.proxyPort", "8080");
13
14 // creazione del canale di comunicazione
```

Figura 9.11 Impostazione di un proxy a livello globale.

Le due proprietà *http.proxyHost* e *http.proxyPort* definiscono rispettivamente indirizzo dell'host e porta dove il proxy è in attesa di connessioni.

Se il proxy non è disponibile, nel primo caso verrà sollevata un'eccezione mentre nel secondo il server sarà



contattato in maniera diretta, senza dare alcun tipo di segnalazione.

Indipendentemente dal metodo utilizzato l'output ottenuto sarà identico ai casi con connessione diretta, a meno di opzioni inserite nell'header dal proxy.

È possibile, in linea di principio, fare anche un uso esplicito del servizio di proxy, collegandovi un'istanza di *Socket* e inviando un messaggio di richiesta HTTP costruito dal programmatore. Questo metodo, oltre a essere scomodo, non permette di usare le funzionalità di *HttpURLConnection*, per cui non ci sono molti benefici nell'adottarlo.

9.7 Cookie

Il linguaggio Java, nelle sue versioni più recenti, mette a disposizione del programmatore una serie di classi e interfacce pensate appositamente per la gestione dei cookie tra cui la classe *HttpCookie*, per contenere le informazioni, e le interfacce *CookieStore* e *CookiePolicy*, per memorizzare le istanze di *HttpCookie* e stabilire quali cookie possono essere memorizzati e quali no. Un programmatore è tenuto a scrivere le proprie classi aderenti a queste interfacce per poter gestire i cookie ricevuti da un web server.

Una volta realizzate le classi appena menzionate è anche possibile far gestire dall'ambiente di esecuzione i cookie relativi a qualunque connessione HTTP tramite la classe *CookieManager*, come nell'esempio riportato nella Figura 9.12, che però non comprende il codice di classi per *store* e *policy*.

```
1 CookieStore cs = new CookieStore_locale();
2 CookiePolicy cp = new CookiePolicy_locale();
3 
4 CookieManager cm = new CookieManager(cs, cp);
5 
6 cookieHandler.setDefault(cm);
```

Figura 9.12 Esempio di utilizzo di *CookieManager*.

Per demandare a Java la gestione di tutti i cookie si crea un'istanza della classe *CookieManager* (riga 4) usando come parametri due istanze di classi che implementano le interfacce *CookieStore* (riga 1) e *CookiePolicy* (riga 2). L'istanza di *CookieManager* così ottenuta è segnalata al sistema, alla riga 6, tramite il metodo statico *setDefault* della classe *CookieHandler*. Da quel momento in poi tutte le istanze di *HttpURLConnection* accederanno in maniera trasparente a un unico sistema di gestione dei cookie.

9.8 Interrogazione di motori di ricerca

L'interrogazione di un motore di ricerca, come già visto, si riduce in realtà a simulare la compilazione di un form e raccogliere il risultato per estrarre dei dati. La vera difficoltà è invece rappresentata dall'estrazione delle informazioni: non

essendoci una struttura prefissata non è possibile andare *a colpo sicuro*.

In realtà i motori di ricerca tendono a essere piuttosto fiscali sull'header del messaggio di richiesta, questo per verificare l'identità del client ed evitare attacchi di sicurezza o di essere usati da altri motori di ricerca senza ricevere credito. Potrebbe capitare che, nonostante il codice del risultato sia positivo, la pagina non contenga i dati attesi: ciò può dipendere dalla conformazione dell'header della richiesta o dalla mancanza di cookie. Purtroppo, è piuttosto difficile reperire informazioni ufficiali su questo argomento e i vincoli cambiano da un sito all'altro.



L'unico approccio in grado di funzionare quasi sempre è quello di identificare alcune stringhe presenti all'interno della pagine, come titoli o tag in posizioni fisse, e da lì estrarre l'informazione che interessa isolandola con delimitatori o adottando tecniche basate sul metodo *split* della classe *String*. Ad esempio, Google fa precedere a tutti i link relativi ai risultati la stringa "<h3 class=r><a href=""", per cui il codice proposto nella Figura 9.13 può essere usato per estrarli dal testo della pagina.

```
1 static void extrai_link(string buffer) {
2     String prensabolo = "<h3 class=r><a href=""";
3     int g = 0;
4     int inicio, fine;
5     String line;
6     String[] tokens;
7     while (buffer.indexOf(prensabolo, g) != -1) {
8         inicio = buffer.indexOf(prensabolo, g) +
9             prensabolo.length();
10        fine = buffer.indexOf("</a>", inicio);
11        line = buffer.substring(inicio, fine);
12        System.out.println(line);
13        g = fine;
14    }
15 }
```

Figura 9.13 Estrazione dei link relativi ai risultati da una pagina di Google.

Nel codice proposto si usa la variabile *c* per scorrere il buffer e localizzare i vari preamboli fino a che ci sono preamboli successivi alla posizione corrente di *c* (riga 8), il link inizia alla fine del preambolo (righe 9 e 10) e finisce in corrispondenza dei doppi apici che identificano la fine del link (riga 11). Nella riga 12 si estraе la sottostringa relativa al link e in quella successiva viene aggiornata la posizione identificata da *c* per poter iterare il ciclo.

Se Google dovesse cambiare la struttura delle sue pagine questo codice non funzionerà più.



Esercizio 9.4



I risultati di Google, se non diversamente specificato, vengono presentati a gruppi di 10. Si estenda il client sviluppato fin qui in maniera tale da effettuare una serie di richieste successive e ottenere tutte le pagine. Si utilizzino sia connessioni non persistenti che persistenti.

Esercizio 9.5



Si inserisca nel codice la funzione proposta per l'estrazione dei link e la si usi in sequenza su tutte le pagine restituite da Google.

Esercizio 9.6



Si implementi un client in grado di collegarsi al sito di Wikipedia e di effettuare la ricerca di un termine fornito dall'utente; l'output del programma deve essere il contenuto della pagina HTML. Si faccia attenzione al fatto che, in prima battuta, Wikipedia risponde con un messaggio che ridirige a un altro URL.

9.9 RSS

Java mette a disposizione alcune classi già pronte per il trattamento di documenti XML. Sono disponibili due tipi di approcci contrapposti: *Simple API for XML* (SAX) e *Document Object Model* (DOM). SAX legge un documento sotto forma di una sequenza di byte e genera un evento nel momento in cui

viene incontrato un tag; DOM, invece, trasferisce il documento in memoria e lo trasforma in un albero di oggetti concatenati tra loro. Nel caso di feed RSS, considerando anche la loro dimensione contenuta, è consigliabile usare DOM perché effettua un controllo sulla consistenza del documento e permette anche di fare ricerche all'interno dell'albero.

Per poter costruire un albero DOM occorrono tre elementi: una classe *Document* per rappresentare l'albero, una classe *DocumentBuilder* responsabile per la conversione del documento e una classe *DocumentBuilderFactory* per creare istanze di *DocumentBuilder* con specifiche caratteristiche. La sequenza delle operazioni necessarie è proposta nella Figura 9.14.

Per prima cosa occorre un'istanza di *DocumentBuilderFactory*. Questa classe non possiede un costruttore pubblico e quindi si fa uso del suo metodo statico *newInstance* (righe 48 e 49); a partire dall'oggetto appena creato si genera un'istanza di *DocumentBuilder* (riga 50) e, infine, l'istanza della classe *Document* viene creata da *DocumentBuilder* attraverso il metodo *parse* e partendo dai dati in arrivo dal server. Come si può notare, in questo caso non è più necessario l'uso di un buffer, l'unico parametro di *parse* è l'istanza di *InputStream* associata al canale di comunicazione.

```
4 import javax.xml.parsers.DocumentBuilder;
5 import javax.xml.parsers.DocumentBuilderFactory;
6 import org.w3c.dom.Document;
7
8 /**
9  * ricezione del contenuto
10 * System.out.println("contenuto");
11 * InputStream ix = hub.getInputStream();
12 */
13 DocumentBuilderFactory dfr;
14 DocumentBuilder builder;
15 dfr = DocumentBuilderFactory.newInstance();
16 DocumentBuilder builder = dfr.newDocumentBuilder();
17 Document doc = builder.parse(ix);
```

Figura 9.14 Creazione di un albero DOM a partire dal contenuto di una risorsa web.

Il realtà, il metodo *parse* accetta come parametro anche una stringa contenente un URL; quindi, a rigor di logica, anche *HttpURLConnection* è ridondante.



Una volta ottenuto l'albero DOM la classe *Document* rende disponibili una serie di metodi per rintracciare informazioni al suo interno: è possibile, partendo da un qualunque nodo dell'albero, ottenere la lista dei nodi suoi figli tramite *getChildNodes* o estrarne nome, valore e tipo facendo uso di *getnodeName*, *getNodeValue* e *getNodeType* rispettivamente. L'istanza di *Document* ottenuta da *DocumentBuilder* riferenzia la radice dell'albero e può essere usata per iniziare la scansione.

Vale però la pena fare un'ultima osservazione: i feed RSS sono sempre di dimensioni piuttosto limitate e usano pochi tag, per cui, anche utilizzare metodi basati sul riconoscimento di sottostringhe, come nel caso di motori di ricerca, potrebbe essere una strada percorribile.

Esercizio 9.7



Si selezioni un quotidiano dotato di feed RSS e si scriva un client in grado di estrarre i titoli delle singole notizie dal feed.

IV

Linguaggio C

Capitolo 10

Programmazione con le socket

10.1 Socket

Il linguaggio C tratta le socket come comuni canali di I/O. Per via della strutturazione del linguaggio questo vuol dire che ci si troverà a dover gestire dei numeri interi. Dal punto di vista della programmazione questo approccio ha il vantaggio di rendere le socket facili da gestire e di permettere di scrivere codice indipendente dalla sorgente dati. Dal punto di vista del programmatore non importa se le informazioni arrivano dalla rete, da un file o da altri processi collocati nello stesso calcolatore; una volta che il canale di comunicazione è attivo diventa quasi indistinguibile dagli altri.

La manipolazione di socket e altri descrittori di file avviene tramite richieste al sistema operativo (*system call*). Una system call permette a un processo che viene eseguito in modalità

non privilegiata di richiedere la modifica di strutture interne del sistema operativo, tra le quali si trovano anche le strutture di controllo per i canali di I/O da e verso il file system, altri processi e la rete.

Come filosofia generale, system call e funzioni di sistema adottano una strategia piuttosto semplice per le segnalazioni di eventuali errori: si discrimina l'esito in base al valore restituito dalla chiamata. Se il risultato è un valore di tipo numerico, in caso di errore verrà restituito al chiamante un numero minore di zero eventualmente dipendente dall'errore; se invece il risultato è un puntatore, in caso di errore verrà restituito un valore nullo (NULL). L'identificazione dell'errore specifico potrà in entrambi i casi avvenire facendo uso di funzioni apposite (come ad esempio *perror*) oppure variabili di ambiente (come *sys_errlist* e *sys_nerr*).

10.2 Creazione di una socket

La creazione di una socket avviene tramite una system call omonima.

```
-----  
#include <sys/socket.h>  
int socket(int domain, int type, int protocol);  
-----
```

Figura 10.1 Prototipo della system call *socket*.

L'uso tipico della system call *socket* è illustrato nella Figura 10.2.

```

1 #include <sys/socket.h>
2 int main(int argc, char ** argv, char ** envp) {
3     int domain = 0; // 
4     int tipo = 0; // 
5     int protocollo = 0; // serve assegnare dei valori
6     int nr; // la socket
7
8     s = socket(domain, tipo, protocollo);
9
10    if (s < 0)
11        perror("socket");
12    else
13        return 0;
14 }

```

Figura 10.2 Uso della system call *socket*.

Alla system call *socket* è possibile aggiungere la gestione di eventuali errori includendo l'header “*stdio.h*” e facendo uso della funzione *perror*.

```

11     s = socket(domain, tipo, protocollo);
12     if (s < 0)
13         perror("socket"); // "socket: descrizione dell'errore"
14     else
15         return 0;

```

Figura 10.3 Gestione di un errore in una chiamata a *socket*.

Provando a eseguire il programma la creazione della socket non andrà a buon fine a causa della non corretta combinazione dei tre parametri, generando un errore come segue:

```
socket: Address family not supported by protocol family
```

Si vedrà ora come impostare i tre parametri in modo opportuno.

10.2.1 Dominio

Il dominio di una socket può essere impostato facendo uso di costanti definite in “*sys/socket.h*”. I nomi di queste costanti

hanno la caratteristica di iniziare con il prefisso “AF_”, con il significato di *Address Family* (famiglia di indirizzamento).

Facendo riferimento ai domini discussi nel Paragrafo 4.2.1 le costanti di interesse sono:

AF_UNIX per le socket in dominio unix

AF_INET per le socket in dominio Internet

AF_INET6 per le socket in dominio Internet versione 6 (IPv6).

All'interno di questo libro ci si concentrerà unicamente sul dominio AF_INET.

Su alcune piattaforme è possibile trovare, in aggiunta a quelle citate sopra, costanti simili ed equivalenti ma con il prefisso “PF_”, con il significato di *Protocol Family* (famiglia di protocolli).

È possibile quindi cominciare ad assegnare un valore a una delle variabili nelle righe dalla 5 alla 7 nella Figura 10.3 per ottenere la Figura 10.4.

```
5 int dominio = AF_INET;
6 int tipo = 0;
7 int protocollo = 0; // serve assegnare dei valori
```

Figura 10.4 Impostazione del valore di dominio.

Provando a eseguire nuovamente il programma sarà possibile osservare un errore diverso.

```
socket: Protocol not supported
```

10.2.2 Modalità di trasferimento dati

La modalità di trasferimento può essere impostata facendo uso di ulteriori costanti definite sempre nell'header “sys/socket.h”.

Il linguaggio C permette di accedere a tre modalità:

SOCK_STREAM per le socket byte-stream

SOCK_DGRAM per le socket datagram

SOCK_RAW per accedere a funzionalità di basso livello.

La modalità SOCK_RAW è disponibile solo all'amministratore di sistema e permette l'accesso diretto all'hardware della scheda di rete; in questo modo è possibile intercettare tutte le informazioni in transito a qualunque livello dello stack ISO-OSI e generare traffico scrivendo sequenze di byte direttamente sul canale trasmissivo. All'interno di questo testo non verranno trattate socket di questo tipo.

Nella Figura 10.5 sono stati modificati i parametri al fine di creare una socket di tipo datagram.

```
5    int domain   = AF_INET;
6    int type     = SOCK_DGRAM;
7    int protocol = 0; // serve assegnare dei valori
```

Figura 10.5 Impostazione della modalità di trasferimento.

Eseguendo il programma risultante si potrebbe non osservare nessun errore, ma solo perché incidentalmente il numero 0 potrebbe essere identificativo di un protocollo compatibile con i parametri impostati.

10.2.3 Protocollo

Purtroppo i protocolli non sono codificati con costanti, ma vengono associati a valori numerici tramite un file di configurazione, /etc/protocols per i sistemi UNIX. Per poter interrogare questi file di configurazione è possibile usare una serie di funzioni, quella che può tornare utile in questo caso è *getprotobynam*e, definita nell'header “netdb.h”.

```
#include <netdb.h>
struct protoent *getprotobynam(char *name);
```

Figura 10.6 Prototipo della funzione *getprotobynam*.

La funzione *getprotobynam* restituisce un puntatore a una struttura *protoent* che contiene nei suoi campi tutte le informazioni relative al protocollo cercato.

```
struct protoent {
    char name; // nome ufficiale
    char *p_aliases; // lista degli alias
    int p_proto; // numero del protocollo
};
```

Figura 10.7 Definizione della struttura *protoent*.

Dei suoi campi è possibile utilizzare *p_proto* come terzo parametro per la creazione della socket, come nella Figura 10.8.

Esercizio 10.1



Quale errore si ottiene dal sistema operativo se si usa il portocollo UDP per una socket byte-stream (o viceversa, il protocollo TCP per una socket datagram)?

```
12 struct protoent *pe; // informazioni sul protocollo
13
14 pe = getprotobynumber("udp");
15 if (pe == NULL)
16     perror("getprotobynumber");
17     return -1;
18
19
20
21 protocollo = pe ->p_proto;
```

Figura 10.8 Impostazione del numero di protocollo.

Esercizio 10.2



Si scriva un programma in C che prenda come parametro il nome di un protocollo e ne stampi tutte le informazioni correlate.

Il codice completo per la creazione di una socket e che riassume tutte le operazioni descritte è riportato nella Figura 10.9.

```

1 #include <sys/socket.h>
2 #include <sys/types.h>
3 #include <stropts.h>
4
5 int main(int argc, char ** argv, char ** envp) {
6     int domain = AF_INET;
7     int tipo = SOCK_DGRAM;
8     int protoselle = 0;
9
10    int si;           // la socket
11
12    struct protoent * p; // informazioni sul protocollo
13
14    if (argc == 1) {
15        p = getprotobyname("udp");
16    } else if (argc == 2) {
17        p = getprotobyname(argv[1]);
18    } else {
19        exit(1);
20    }
21
22    protoselle = p->p_proto;
23
24    si = socket(domain, tipo, protoselle);
25
26    if (si < 0)
27        perror("socket()");
28
29    return -1;
30}

```

Figura 10.9 Codice per la creazione di una socket datagram.

Per motivi di spazio, d'ora in poi la gestione degli errori verrà riportata nel codice solo per le funzioni di nuova introduzione o quando utile ai fini della comprensione del testo. Il lettore, per buona pratica di programmazione, è invitato comunque a effettuare sempre il controllo di eventuali condizioni di errore nei programmi da lui prodotti.



10.3 Indirizzi: numeri e nomi

Nel linguaggio C occorre fare un uso esplicito del resolver. In questo paragrafo ci si limiterà a trasformare il riferimento a un nodo, sia esso un nome o un indirizzo in forma decimale puntata in un numero di 32 bit; nel prossimo paragrafo questo numero sarà utilizzato per comporre un indirizzo a livello di trasporto.

I sistemi operativi che aderiscono allo standard POSIX memorizzano gli indirizzi IP in una variabile di tipo *in_addr_t*, definita nell'header “netinet/in.h”. Questo non è un problema, perché il tipo *in_addr_t* risulta semplicemente essere una ridefinizione di un intero senza segno a 32 bit.

```
typedef uint32_t in_addr_t;
```

Figura 10.10 Definizione del tipo *in_addr_t* in sistemi POSIX.

Il tipo *uint32_t* è a sua volta una ridefinizione del tipo “unsigned long int” e viene supportato da tutti i sistemi POSIX tramite l'header “*stdint.h*”. I sistemi operativi moderni, soprattutto quelli basati su UNIX, tra cui Linux e FreeBSD, adottano già di base numeri interi della dimensione di 32 bit, per cui, usare una variabile di tipo *int* dove ne è richiesta una di tipo *uint32_t* (*o in_addr_t*) non produce nessun errore nel programma ma solo avvertimenti da parte del compilatore.



Il caso più semplice è quello in cui si parte da un indirizzo in forma decimale puntata: si può fare uso della funzione *inet_addr* definita nell'header “arpa/inet.h”.

```
#include <arpa/inet.h>
in_addr_t inet_addr(const char * inaddr);
```

Figura 10.11 Prototipo della funzione *inet_addr*.

Nel programma presentato nella Figura 10.12 si utilizza *inet_addr* per convertire un indirizzo nel suo corrispettivo numerico.

```
1 #include <stropts/in.h>           // per in_addr_t
2 #include <sys/types.h>            // per in_addr
3 #include <stropts.h>              // per perror e printf
4
5 int main(int argc, char ** argv, char ** envp) {
6
7     char * nome_nod = "#2.112.217.055"; // da convertire
8     in_addr_t indirizzo; // va bene anche int
9
10    indirizzo = inet_addr(nome_nod);
11
12    if(indirizzo == INADDR_NONE) {
13        perror("inet_addr");
14        return -1;
15    }
16
17    printf("Indirizzo (32 bit): %u\n", indirizzo);
18
19    return 0;
20 }
```

Figura 10.12 Uso di *inet_addr* per la conversione di un indirizzo.

Se invece l'indirizzo da convertire viene dato sotto forma di nome alfanumerico è necessario seguire un approccio diverso: occorre richiedere al resolver di effettuare una richiesta esplicita al server DNS per avere le informazioni relative al nodo, tra cui il suo indirizzo. La richiesta viene effettuata in conseguenza all'uso della funzione *gethostbyname*, definita nell'header “netdb.h”.

```
#include <stropts.h>
struct hostent * gethostbyname(char *name);
```

Figura 10.13 Prototipo della funzione *gethostbyname*.

La funzione *gethostbyname* restituisce un puntatore a una struttura *hostent* già allocata in memoria e contenente tutte le informazioni utili.

```

struct hostent {
    char * h_name;           // Nome ufficiale
    char ** h_aliases;       // Lista degli alias
    int h_length;            // Lunghezza del nome
    int h_addrtype;          // Tipo di indirizzo
    int h_bтыpe;              // Numero di indirizzi
    char ** h_addr_list;     // Lista degli indirizzi
};


```

Figura 10.14 Definizione della struttura *hostent*.

Come si può intuire leggendo i commenti nella Figura 10.14, i dati che interessano sono contenuti nel campo *h_addr_info*, tuttavia si può rimanere un po' confusi dal fatto che il tipo di tale campo sia un puntatore a un array di puntatori a caratteri. Occorre fare due considerazioni: la prima è che, come già accennato nel Paragrafo 4.3, non è possibile supporre che ci sia un solo indirizzo associato a un certo nome, quindi il campo *h_addr_info* dovrà obbligatoriamente essere un puntatore per permettere la gestione di un array di valori. La seconda considerazione è che la funzione *gethostbyname* è stata progettata per gestire qualunque tipo di indirizzi a livello di rete; quindi, il campo *h_addr_info* va usato come un riferimento a un array di puntatori a zone di memoria dove sono contenuti indirizzi.

Presumendo che questi indirizzi siano di 32 bit, al più verificando che il campo *h_length* abbia un valore di 4, è possibile estrarre il primo indirizzo numerico con un semplice *cast*.

```
in_addr_t indirizzo = *((in_addr_t *) (hostent -> h_addr_1
```

oppure anche:

```
int indirizzo = *((int *) (hostent -> h_addr_list[0]))
```

Nel dettaglio: si è preso il primo elemento dell'array, si è forzato il tipo a un puntatore a *in_addr_t* (o *int*) e da questo se ne è preso il valore contenuto nella zona di memoria puntata.

Nella Figura 10.15 viene presentato un programma analogo a quello della Figura 10.12 che stampa il primo indirizzo associato a un nome.

```
1 #include <sys/types.h>           // per in_addr_t
2 #include <sys/socket.h>          // per socket e htons
3 #include <sys/sockio.h>           // per perror e printf
4
5 int main(int argc, char ** argv, char ** envp) {
6
7     char * nome = "www.parlamento.it"; // da convertire
8     in_addr_t indirizzo;             // va bene anche long
9     struct hostent * he;
10
11    he = gethostbyname(nome);
12
13    if (he == NULL) {
14        perror("gethostbyname");
15        return -1;
16    }
17
18    indirizzo = *(in_addr_t *) (he-> h_addr_list[0]);
19    printf("%Indirizzo (%d bit): %u\n", indirizzo);
20
21    return 0;
22 }
```

Figura 10.15 Uso di *gethostbyname* per la conversione di un indirizzo.

Se si è invece interessati a tutti i possibili indirizzi, allora occorre scorrere l'array puntato da *h_addr_info* fino a che non viene incontrato il valore NULL; è sufficiente inserire le righe 19 e 20 della Figura 10.15 all'interno di un ciclo, come illustrato nella Figura 10.16.

```
19    for (i = 0; he-> h_addr_list[i] != NULL; i += 1) {
20        indirizzo = *(in_addr_t *) (he-> h_addr_list[i]);
21        printf("%Indirizzo %d (%d bit): %u\n", i, indirizzo);
22    }
23 }
```

Figura 10.16 Stampa di tutti gli indirizzi possibili riportati da *gethostbyname*.

Esercizio 10.3



Si scriva un programma che, dato come parametro il nome di un nodo, stampi tutte le informazioni riportate dal resolver.

La funzione *gethostbyname* è in grado di convertire correttamente anche indirizzi espressi nella notazione decimale puntata; nei sistemi operativi moderni, non è necessario usare altro. Tuttavia, in sistemi di vecchia realizzazione occorre necessariamente usare la funzione *inet_addr* nel caso di indirizzi in forma decimale puntata; è buona norma, per favorire la portabilità del codice, usare prima *gethostbyname*. In caso questa fallisca utilizzare *inet_addr* e solo se tutte e due danno esito negativo segnalare un errore all'utente.

Esercizio 10.4



Si scriva un programma che, dato come parametro il nome di un nodo, ne faccia la conversione in formato numerico, usando prima *gethostbyname* e poi, in caso questa fallisca, *inet_addr*.

10.4 Identificazione di una socket

System call e funzioni che manipolano socket al fine di costruire canali di comunicazione fanno uso di una struttura *sockaddr* definita in “sys/socket.h” per memorizzare indirizzi a livello di trasporto. Purtroppo, nonostante questa struttura sia utilizzata quasi esclusivamente per TCP e UDP, è stata progettata per accogliere qualunque tipo di indirizzo, per cui in realtà, come si può vedere nella Figura 10.17, nel definirla ci si limita ad allocare spazio per i singoli casi specifici.

```
struct sockaddr {
    unsigned char sa_len;      // dimensione
    unsigned char sa_family;    // tipo di indirizzo;
    char        sa_data[16];    // dati specifici dell'indirizzo
};
```

Figura 10.17 Definizione della struttura *sockaddr*.

La struttura che viene definita per memorizzare indirizzi trasporto utilizzabili con TCP e UDP è invece *sockaddr_in*, dichiarata in “netinet/in.h” e riprodotta nella Figura 10.18.

```
struct sockaddr_in {
    unsigned short sin_port;   // porta
    unsigned char sin_family; // tipo di indirizzo
    in_port_t sin_port;       // porta
    struct in_addr sin_addr; // indirizzo dell'host
    char        sin_zero[8];  // riempimento
};
```

Figura 10.18 Definizione della struttura *sockaddr_in*.

Queste due definizioni, usando opportunamente i campi *sa_data* e *sin_zero* fanno sì che la dimensione dei due tipi sia identica (16 byte). In tal modo, essendo i primi due rispettivi

campi coincidenti, è possibile usare una variabile di tipo *sockaddr_in* dove sia richiesta una variabile di tipo *sockaddr*, a patto che il campo *sin_family* (o *sa_family*) sia impostato in maniera corretta, e cioè con il valore AF_INET. Assegnare il valore AF_INET a *sin_family* permette alle varie funzioni di interpretare correttamente il formato degli ultimi 14 byte.

Resta da chiarire un dubbio che riguarda i tipi di alcuni campi di *sockaddr_in*. Il tipo *in_port_t* è in realtà una ridefinizione di un intero senza segno di 16 bit (cosa prevedibile) e può essere assegnato senza grossi problemi tramite variabili di tipo *int*, a patto che queste non assumano valori superiori o uguali a 65536 (2^{16}). Il campo *sin_addr* risulta stranamente essere una struttura di tipo *in_addr* anziché di tipo *in_addr_t*. Anche in questo caso ci si trova di fronte a un'apparente complicazione al fine di garantire compatibilità; infatti, la definizione della struttura *in_addr* contiene un solo campo, per cui il suo assegnamento risulterà comunque un'operazione piuttosto banale.



Figura 10.19 Definizione della struttura *in_addr*.

Qualche volta, esaminando codice scritto da altri, anzichè l'assegnamento:

```
indirizzo.sin_addr.s_addr = valore;
```



ci si trova di fronte a un'operazione del tipo:

```
((int) indirizzo.sin_ddr) = valore;
```

Il codice, in questo caso, risulta corretto (trascorando eventuali avvertimenti da parte del compilatore) in virtù del fatto che la struttura è composta da un solo campo. In ogni caso si sconsiglia caldamente il secondo metodo.

Prima di procedere ad assegnare valori ai vari campi della struttura è necessario fare una puntualizzazione riguardo al campo *sin_port*. Nonostante il tipo *in_port_t* sia una ridefinizione di un intero a 16 bit non è ugualmente possibile assegnare il numero di porta in maniera diretta, ad esempio nel modo seguente:

```
inet_addr_t.sin_port = 80; // errore
```

Le informazioni numeriche hanno infatti due tipi di rappresentazioni: una che si può definire “di rete”, usata per lo scambio di informazioni e all’interno dei pacchetti trasmessi, e una definita tipicamente “di host”, usata per immagazzinare i dati nella memoria del calcolatore. La differenza tra queste due rappresentazioni è l’ordine che viene dato ai byte.

Esistono una serie di funzioni per la conversione da un formato all’altro. Tali funzioni coprono tutte e quattro le possibili combinazioni per convertire da rappresentazione interna a

rappresentazione di rete (il nome della funzione inizia con *hton*) e viceversa (*ntohl*) sia interi di 16 bit (*short*, il nome della funzione termina in “s”) che interi a 32 bit (*long*, “l”). Queste combinazioni sono riassunte nella Tabella 10.1.

	Short (16 bit)	Long (32 bit)
Da interna a rete	<i>hton</i>	<i>htonl</i>
Da rete a interna	<i>ntoh</i>	<i>ntohl</i>

Tabella 10.1 Funzioni per la conversione di tipi numerici da rappresentazione interna a rappresentazione di rete e viceversa.

```

1 #include <sys/types.h>
2 #include <netdb.h>
3 #include <stdio.h>
4
5 int main(int argc, char ** argv, char ** envp) {
6     char * nome = "www.google.it";
7
8     int domain = AF_INET; // tipo di indirizzo trasporto
9     int addrlen = 0; // da ricevere da nome
10    struct sockaddr_in sdi; // da convertire in nome
11
12    struct hostent * he = gethostbyname(nome);
13
14    if (he == NULL) {
15        indirizzo = <(in_addr_in *)he->h_addr_list[0]>;
16
17        struct sockaddr_in indirizzo;
18
19        indirizzo.sin_family = domain;
20        indirizzo.sin_addr.s_addr = indirizzo;
21        indirizzo.sin_port = htons(80);
22
23        return 0;
24    }

```

Figura 10.20 Impostazione dell’indirizzo di trasporto che identifica il servizio HTTP di www.google.it.

A questo punto, facendo uso della funzione *gethostbyname* descritta precedentemente, è possibile scrivere un programma che assegna tutti i valori necessari ai campi di una struttura *sockaddr_in*. Nella Figura 10.20 vengono impostati i dati di un indirizzo a livello di trasporto che identifica un servizio erogato sulla porta 80 del nodo www.google.it.

10.5 Binding

Il binding esplicito di una socket viene effettuato utilizzando l'omonima system call *bind*.

```
1 #include <sys/types.h>
2 #include <sys/socket.h>
3 int bind(int socket, struct sockaddr *indirizzo, int dimensione);
```

Figura 10.21 Prototipo della system call *bind*.

Per poter usare la system call *bind* basterà quindi impostare un indirizzo di trasporto come secondo parametro. Se non si intende limitare la socket a essere raggiungibile solo tramite una data interfaccia di rete è possibile assegnare la costante `INADDR_ANY` al posto dell'indirizzo di rete; diversamente occorrerà effettuare la conversione dell'indirizzo locale di cui si vuole fare uso. Il terzo parametro della system call, invece, è un numero corrispondente alla dimensione in byte del secondo parametro ed è utile al sistema per interpretare correttamente quest'ultimo.

I metodi possibili per ricavare l'indirizzo delle schede di rete locali sono estremamente dipendenti dal sistema operativo adottato e non verranno trattati in questo testo.



Nella Figura 10.22 viene proposto il codice necessario per allocare una socket e associarla alla porta 7000. Le righe 10 e 12 si occupano della creazione della socket, le righe dalla 14 alla 17 stabiliscono l'indirizzo di trasporto a cui fare l'associazione e la riga 19 effettua il bind. È importante notare come, per un corretto funzionamento, sia necessario usare lo stesso sistema di indirizzamento, AF_INET per il bind e per la socket alle righe 12 e 15. La riga 25 mette in attesa il programma per 120 secondi; durante tale intervallo di tempo è possibile usare netstat per controllare che la porta sia stata effettivamente allocata. Il risultato sarà simile al seguente:

```
Active Internet connections (including servers)
Proto Recv-Q Send-Q Local Address Foreign Address (state)
tcp4 0 0 *.*.7000 *.* CLOSED
```

Lo stato della socket (CLOSED) non va letto come “chiuso” ma bensì come “disconnesso”. La socket è stata creata ma non è ancora stata associata a una controparte. Nel momento in cui la socket verrà associata lo stato passerà da CLOSED a CONNECTED.

```
1 #include <sys/types.h>
2 #include <sys/socket.h>
3 #include <net/if.h>
4 #include <sys/types.h> // per il tipo di struct_if
5 #include <sys/conf.h> // per protos
6 #include <sys/types.h> // per sleep
7
8 int main(int argc, char **argv, char **envp) {
9     struct protoent *pe = getprotobyname("tcp");
10    struct sockaddr_in bindaddr;
11    int s = socket(AF_INET, SOCK_STREAM, pe->p_proto);
12    bindaddr.sin_family = AF_INET;
13    bindaddr.sin_port = htons(7000);
14    bindaddr.sin_addr.s_addr = htonl(INADDR_ANY);
15    if (bind(s, (struct sockaddr *) &bindaddr, sizeof(bindaddr)) != 0)
16        perror("bind error");
17    sleep(120);
18    return 0;
19 }
20
21 sleep(120);
22 return 0;
23
24
25
26
27 }
```

Figura 10.22 Associazione di una socket alla porta 7000.

Per quanto riguarda invece la possibilità di effettuare un binding alla prima porta disponibile sarà sufficiente porre a zero il numero della porta, tuttavia ci si troverà di fronte a un nuovo problema: quello di capire a quale indirizzo di trasporto la socket è stata associata.

Per rispondere a questa domanda è possibile fare uso della system call *getsockname*. Questa system call prende come parametro una socket e riempie una struttura *sockaddr_in* con le informazioni relative all'indirizzo a essa associato.

```
1 #include <sys/socket.h>
2 int getsockname(int s, struct sockaddr *indirizzo, int *dimensione);
```

Figura 10.23 Prototipo della system call *getsockname*.

È semplice modificare il programma nella Figura 10.22 in modo da fargli allocare una porta dal sistema e poi stamparne il numero; si veda a questo riguardo la Figura 10.24. Vale la pena notare che il terzo parametro di questa funzione, diversamente dal caso di *bind*, è un puntatore a un intero e non un intero; questo perché tale parametro rappresenta un valore in uscita per la funzione e al termine della chiamata conterrà la dimensione in byte dell'indirizzo della socket.

```
17     indirizzo.sin_port = htons(0);
18
19     bind(s, (struct sockaddr *) &indirizzo, sizeof(indirizzo));
20
21     struct sockaddr_in indirizzo;
22     int dimensione;
23
24     if ((bind(s, (struct sockaddr *) &indirizzo, sizeof(indirizzo)) < 0) ||
25         (getsockname(s, (struct sockaddr *) &indirizzo, &dimensione) < 0) ||
26         (dimensione != 16)) {
27         perror("getsockname");
28         exit(-2);
29     }
30
31     printf("Porta allocata: %d\n", ntohs(indirizzo.sin_port));
32
33     sleep(120);
34
35     return 0;
36 }
```

Figura 10.24 Associazione di una socket a una porta assegnata dal sistema.

10.6 Porte e servizi standard

Nel momento in cui è necessario contattare od erogare un servizio standard è preferibile utilizzare come riferimento nel codice il nome e non il numero di porta. Ciò permette di dare più flessibilità al codice in caso di variazione di uno standard; se il servizio HTTP dovesse passare dalla porta 80 alla 81 sarà sufficiente variare un file di configurazione del sistema operativo ospite e non si renderà necessario ricompilare il software.

Per risalire al numero di porta avendo il nome di un servizio è possibile usare la funzione *getservbyname* (*get service by name*).

```
#include <netdb.h>
struct servent * getservbyname(char * nome, char * protocollo);
```

Figura 10.25 Prototipo della funzione *getservbyname*.

```
struct servent {
    char * s_name;           /* nome dell'elenco */
    char * s_aliases;        /* elenchi alternativi */
    int s_port;              /* numero di porta */
    char * s_proto;          /* protocollo */
};
```

Figura 10.26 Definizione della struttura *servent*.

La funzione *getservbyname* prende come parametri i nomi di un servizio e di un protocollo e restituisce un puntatore a una struttura *servent* già allocata e contenente tutte le informazioni relative al servizio stesso. Il reperimento delle informazioni non

tiene conto di maiuscole e minuscole, quindi non c'è differenza di risultato nell'usare "http" o "HTTP".

Ora quindi, rifacendosi anche alla Figura 10.20, è possibile estendere il programma come riportato nella Figura 10.27.

```
1 *include <sys/socket.h>
2 *include <stroq.h>
3 *include <stroq.h>
4
5 int main(int argc, char ** argv, char ** envp) {
6
7     char * nome_host = "www.google.it"; // nome del host
8     char * nome_servizio = "http"; // nome del servizio
9     char * nome_protocollo = "http";
10
11    int socket_desc = AF_INET; // tipo di indirizzo trasporto
12    int sin_port = htons(80); // da ricevere come numero
13    int porta; // da ricevere da nome_servizio
14
15    struct hostent * ho = gethostbyname(nome_host);
16    struct servent * se = getservbyname(nome_servizio, nome_protocollo);
17
18    if (ho == NULL) // se non trovato nel database
19        if (se == NULL) // se non trovato nel database
20            perror("gethostbyname");
21        else
22            perror("getservbyname");
23
24    porta = ntohs(se-> s_port);
25
26    struct sockaddr_in socckt_desc; // da trasportare
27    lsd.sin_port.sin_port = htons(porta);
28    lsd.sin_port.sin_family = AF_INET;
29    lsd.sin_port.sin_addr.s_addr = htonl(INADDR_ANY);
30    lsd.sin_port.sin_port = htons(porta);
31
32    return 0;
33 }
```

Figura 10.27 Impostazione dell'indirizzo di trasporto che identifica il servizio HTTP di www.google.it.

È importante notare come il valore del campo *s_port* sia già rappresentato in formato di rete; quindi, per uniformità di gestione è opportuno convertirlo facendo uso di *ntohs* per assegnarlo alla variabile *porta* nella riga 23. In alternativa, sarebbe anche stato possibile alla riga 28 assegnare al campo *sin_port* direttamente il valore contenuto nel campo *s_port*, senza fare uso di un intermediario.

```
ind_trasporto.sin_port = se -> s_port;
```



È una pessima idea mantenere variabili con dati facenti uso della rappresentazione di rete: questo porta a errori molto difficili da rintracciare o a codice non riutilizzabile. Se si è costretti a gestire variabili con il formato di rete è quantomeno consigliabile fare uso di una nomenclatura chiara, ad esempio *net_port*.

10.7 Associazione di due socket

10.7.1 Operazioni lato ricevente

Se la socket è di tipo datagram (SOCK_DGRAM) non sono necessarie ulteriori operazioni per poterla utilizzare, a meno che non la si voglia associare a uno specifico mittente. Questa operazione è analoga a quanto fatto dal chiamante, per cui il lettore può fare riferimento al paragrafo successivo.

Se, invece, la socket è di tipo byte stream (SOCK_STREAM), allora il programma deve effettuare due operazioni: la prima è quella di abilitare la socket a essere associata, tramite la system call *listen* e la seconda è quella di mettersi in attesa di richiesta di associazione tramite la system call *accept* (Figura 10.29).

```
#include <sys/socket.h>
#include <sys/types.h>
```

Figura 10.28 Prototipo della system call *listen*.

La sistem call *listen*, nell'abilitare lo stato di attesa della socket, impone anche una coda di attesa massima, al raggiungimento della quale un nuovo chiamante si vedrà rifiutare la richiesta di associazione. Il valore massimo accettabile per la lunghezza della coda varia da sistema a sistema, tuttavia risulta essere determinante solo nel caso in cui si voglia usare la socket per implementare un modello di comportamento iterativo.

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int socket, struct sockaddr *address, int *addresslen);
```

Figura 10.29 Prototipo della system call *accept*.

La system call *accept* è una system call bloccante, arresta cioè l'esecuzione del programma fino al verificarsi di un evento, che sarà la richiesta di associazione da parte di un'altra socket.

I suoi parametri seguono la stessa logica della funzione *getsockname*, solo che conterranno dati riguardanti l'indirizzo della socket chiamante. Il valore restituito, invece, sarà una nuova socket da utilizzarsi per scambiare dati.

È possibile in questo modo estendere il programma proposto nella Figura 10.24, aggiungendo in coda le chiamate a *listen* e *accept* e ottenendo quanto riportato nella Figura 10.30.

Il programma terminerà nel momento in cui un chiamante effettuerà la connessione.

Per effettuare una prova pratica è possibile compilare e lanciare il programma scritto fin qui; esso si bloccherà subito dopo aver prodotto un output simile al seguente:

```
Porta allocata: 50226
```

dove però il numero sarà diverso.

```
1 29 printf("Porta allocata %d\n", ntohs(bind.local.sin_port));
2 30 if (listen(s, 128) < 0) {
3 31     perror("bind error");
3 32     exit(1);
3 33 }
3 34
3 35 struct sockaddr_in chiamante;
3 36 int ncln_socset = accept(s, (struct sockaddr *) &chiamante, &nlen);
3 37 if (ncln_socset < 0)
3 38     perror("accept error");
3 39
3 40 if (ncln_socset > 0)
3 41     printf("connesso\n");
3 42
3 43 close(s);
3 44 return 0;
3 45 }
```

Figura 10.30 Messa in attesa di una socket byte-stream.

Usando nuovamente netstat, comparirà una riga con la seguente informazione:

```
Active Internet connections (including servers)
Proto Recv-Q Send-Q Local Address Foreign Address (state)
tcp4 0 0 * .50226 *.* LISTEN
```

La colonna relativa allo stato della socket conferma il fatto che il programma sia effettivamente in attesa di una chiamata.

Successivamente, tramite una seconda finestra con un prompt di comandi, si può usare il comando telnet per contattare il programma in attesa, avendo l'accortezza di usare il giusto numero di porta.

```
telnet localhost 50226
```

Nel momento in cui il comando telnet riuscirà a collegarsi si osserverà la terminazione del programma in attesa.

Esercizio 10.5



Si estenda il programma proposto nella Figura 10.30 in maniera tale da stampare le informazioni relative all'indirizzo di trasporto del chiamante.

10.7.2 Operazioni lato chiamante

Il chiamante, da parte sua, non deve fare altro che effettuare una richiesta di associazione a una socket remota. Questa funzione viene svolta dalla system call *connect*.

```
#include <sys/socket.h>
int connect(int socket, struct sockaddr *indirizzo, int dimensione);
```

Figura 10.31 Prototipo della system call *connect*.

I parametri della system call *connect* seguono la stessa logica di quelli della system call *bind*, con la differenza che l'indirizzo del secondo parametro deve essere quello della socket alla quale associarsi.

Prendendo come punto di partenza il programma rappresentato nella Figura 10.27 è possibile aggiungere una chiamata a *connect* per effettuare la connessione a www.google.it.

```
1 #include <sys/types.h>
2 #include <sys/socket.h>
3 #include <netdb.h>
4 #include <errno.h>
5 #include <sys/types.h>
6 #include <sys/socket.h>
7 #include <sys/un.h>
8 #include <sys/conf.h>
9 #include <sys/conf.h>
10 #include <sys/conf.h>
11 #include <sys/conf.h>
12 #include <sys/conf.h>
13 #include <sys/conf.h>
14 #include <sys/conf.h>
15 #include <sys/conf.h>
16 #include <sys/conf.h>
17 #include <sys/conf.h>
18 #include <sys/conf.h>
19 #include <sys/conf.h>
20 #include <sys/conf.h>
21 #include <sys/conf.h>
22 #include <sys/conf.h>
23 #include <sys/conf.h>
24 #include <sys/conf.h>
25 #include <sys/conf.h>
26 #include <sys/conf.h>
27 #include <sys/conf.h>
28 #include <sys/conf.h>
29 #include <sys/conf.h>
```

Figura 10.32 Collegamento al servizio HTTP di www.google.it.

Esercizio 10.6



Si modifichi il programma proposto nella Figura 10.32 in maniera tale da poterlo utilizzare per sbloccare il programma in attesa realizzato per l'Esercizio 10.5.

10.8 Trasferimento dati

Esistono in C due coppie di system call per leggere e scrivere dati su una socket, ognuna declinata nel caso di socket connesse o non connesse.

10.8.1 Socket connesse

Le due system call usate per inviare e ricevere dati tramite socket connesse tra loro sono *send* e *recv*.

```
include <sys/socket.h>
int send(int socket, void *buffer, int dim_buffer, int options);
int recv(int socket, void *buffer, int dim_buffer, int options);
```

Figura 10.33 Prototipi delle system call *send* e *recv*.

Come è facile osservare nella Figura 10.33, le due system call fanno uso dello stesso insieme di parametri; in particolare, *buffer* risulta essere un puntatore a una zona di memoria usata per lo scambio dei dati e *dim_buffer* rappresenta il quantitativo di byte da inviare o ricevere.

La system call *send* invia sulla rete i dati contenuti nel buffer per la dimensione richiesta e restituisce il numero di byte inviati sulla rete, cioè *dim_buffer*, o un valore inferiore a zero in caso di errore.

La system call *recv*, invece, sospende l'esecuzione del processo fino all'arrivo del messaggio successivo dalla rete. Alla disponibilità di dati viene restituito il numero di byte letti dalla socket. Il numero di byte ricevuti potrebbe essere minore di quanto richiesto con il parametro *dim_buffer*; questo, però, non indica un errore, semplicemente la dimensione del messaggio in arrivo è inferiore a quella del buffer. In questo caso *dim_buffer*

è da interpretare con un *numero massimo accettabile* di byte in ingresso. In generale, la politica migliore è quella di impostare il valore di *dim_buffer* alla dimensione del buffer e successivamente usare il valore restituito da *recv* per delimitare la porzione di memoria utilizzata.

È possibile, quindi, estendere ulteriormente il programma rappresentato nella Figura 10.30 come illustrato nella Figura 10.34. Per provare il programma è possibile operare in maniera simile a quanto fatto nel Paragrafo 10.7.1, ma questa volta il comando telnet non terminerà appena collegato al programma in attesa, bensì entrerà in uno stato di apparente blocco. Sarà possibile far evolvere la situazione digitando una qualunque stringa di caratteri e terminando con un ritorno a capo; il programma realizzato nell'esempio stamperà un messaggio relativo alla stringa e terminerà, provocando anche la terminazione di telnet.

```
Porta allocata: 61877
Ricevuta stringa 'messaggio di prova', usando 20 byte
```

Il significato della riga 52 può sembrare un po' oscuro; tuttavia si tratta di un'operazione necessaria in quanto il messaggio inviato dal chiamante non comprende il terminatore di stringa ("\\0"), tale informazione è necessaria per poter usare *buffer* come parametro della funzione *printf*. Non imponendo il terminatore di stringa, *printf* stamperebbe la stringa letta dalla rete seguita da qualunque cosa presente nel buffer fino a incontrare un byte che, per caso, assume il valore del terminatore. Secondariamente, comprende anche il carattere di

ritorno a capo, che in questo caso è dannoso perché rovina il formato dell'output.

```
41
42
43     int dimBuffer = 1024;
44     char buffer[1024];
45
46     t = recv(data->socket, buffer, dimBuffer, 0);
47
48     if (t < 0)
49         perror("recv");
50
51     return t;
52 }
53
54 buffer[t - 2] = '\0'; // terminatore di stringa
55 // al posto di \000 come in C
56
57 print("Ricevuta stringa \"%s\", lungo %d byte(s), buffer, %s");
58
59 }
```

Figura 10.34 Ricezione di una stringa su una socket connessa.



Esercizio 10.7

Modificare il programma proposto nella Figura 10.34 in maniera tale da far visualizzare tramite il comando telnet un *prompt*, cioè una stringa che dia informazioni su come procedere.

Una nota a parte merita ora il parametro *opzioni*. Lo scopo di questo parametro varia a seconda che si stia effettuando un invio o una ricezione; in generale serve a istruire la system call di operare sul messaggio in maniera non standard. Per assegnare un valore a questo parametro si fa uso di costanti definite nell'header “sys/socket.h”.

Per quanto riguarda l'invio dei dati le opzioni possibili sono:

MSG_OOB tratta il messaggio come out-of-bound
MSG_DONTROUTE ignora le indicazioni di instradamento e fa

I messaggi *out-of-bound* sono detti *fuori banda*, sono cioè messaggi che dovrebbero essere trattati con una priorità più alta di quelli ordinari. Una socket è in grado di richiedere al livello di rete che un messaggio sia trattato come fuori banda; il supporto di tale funzionalità è però delegato all'infrastruttura di rete. Disgraziatamente, IP non implementa politiche per la priorità dei pacchetti, per cui questa modalità non viene di fatto mai utilizzata.

La costante `MSG_DONTROUTE`, invece, è utile solo nel caso si intenda fare della diagnostica di rete, permette infatti di mandare un pacchetto sulla rete locale, e solo su quella, indipendentemente dalla sua destinazione finale.

Per quanto riguarda la ricezione è invece possibile fare uso delle seguenti costanti:

`MSG_OOB` gestisce un messaggio in ingresso di tipo *out-of-bound*
`MSG_WAITALL` sospende l'esecuzione fino all'arrivo di `dim_buf`

Di questi, gli ultimi due possono avere un'utilità pratica. A volte `MSG_PEEK` è utile per verificare i primi byte del messaggio inviato e poi, in base a questi, decidere quanta memoria allocare al buffer per leggere dalla rete. La costante `MSG_WAITALL` è invece molto utile nel caso in cui si sappia a priori la dimensione del messaggio in arrivo in quanto può evitare l'uso iterato di `recv` fino allo riempimento del buffer.



Anche con una socket di tipo byte-stream non è per nulla garantito che a fronte di un invio di N byte ci si trovi a osservare una ricezione di esattamente N byte all'interno di un solo messaggio. Questo comportamento, apparentemente anomalo, può dipendere dalla frammentazione effettuata a livello di rete oppure dalla disponibilità dei buffer dei sistemi operativi dei nodi ricevente o trasmettente.

Come caso particolare, se si usano socket connesse di tipo byte-stream, è possibile utilizzare le system call *write* e *read*; esse hanno un comportamento identico a *send* e *receive*, a parte il fatto di non utilizzare il parametro per le opzioni.

```
#include <sys/types.h>
int write(int socket, void *buffer, int len,buffer);
int read(int socket, void *buffer, int len,buffer);
```

Figura 10.35 Prototipi delle system call *write* e *read*.



Esercizio 10.8

Implementare, facendo uso della system call *read*, la stessa funzionalità svolta da *recv* invocata con l'opzione MSG_WAITALL.

10.8.2 Socket non connesse

Le due system call usate per mandare e ricevere dati tramite socket non connesse sono *sendto* e *recvfrom*.

Come si può facilmente notare nella Figura 10.36, i primi quattro parametri sono in comune con le due system call viste per il caso delle socket connesse.

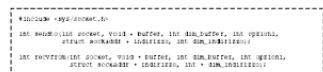


Figura 10.36 Prototipi delle system call *sendto* e *receivefrom*.

I parametri *indirizzo* e *dim_indirizzo* vengono utilizzati per memorizzare l'indirizzo a livello di trasporto della socket con cui si stanno scambiando informazioni e la sua dimensione; secondo la stessa logica vista nel caso di *connect* (10.7.2) e *accept* (10.7.1).

La system call *sendto* invia sulla rete i dati contenuti nel buffer all'indirizzo specificato come quinto parametro e restituisce il numero di byte inviati sulla rete, cioè *dim_buffer*, o un valore inferiore a zero in caso di errore.

Anche nel caso della ricezione il comportamento è del tutto simile a quanto visto prima, se non per il dettaglio che la system call *recvfrom*, a fronte della disponibilità di dati via rete, riempie

la struttura passata come quinto parametro con le informazioni relative al mittente. Tali informazioni possono essere usate come parametro per una chiamata alla system call *sendto* al fine di mandare la risposta.

Per cui, è possibile scrivere una versione non orientata alla connessione del programma appena visto per la ricezione di una stringa. Nel programma rappresentato nella Figura 10.37 sono state tolte le chiamate alle system call *listen* e *accept*. Inoltre, facendo ora uso di una modalità non connessa non è più possibile usare il comando telnet per verificarne il funzionamento.

```
12 int s = socket(AF_INET, SOCK_DGRAM, 0); // p->p_gentozz
13 //printf("Porta allocata %d\n", atoi(argv[1]));
14 int dim_buffer = 1024;
15 char buffer[dim_buffer];
16 struct sockaddr_in chiamante;
17 unsigned int dim_chiamante;
18
19 t = recvfrom(s, buffer, dim_buffer, 0, &chiamante, &dim_chiamante);
20 if (t < 0) {
21     perror("recvfrom");
22 }
23 t = strlen(buffer);
24 buffer[t] = '\0'; // terminazione di stringa
25 // al punto di tira finito
26 printf("Ricevuta stringa '%s', ormai si bytestr", buffer);
27
28 return 0;
29 }
```

Figura 10.37 Ricezione di una stringa su una socket non connessa.

Esercizio 10.9



Implementare, facendo uso di una socket in modalità non connessa, un programma che mandi una stringa

all'esempio della Figura 10.37 e se ne verifichi il buon funzionamento.

10.9 Chiusura del canale

La chiusura esplicita del canale viene normalmente effettuata facendo uso della system call *close*. Da quel momento in poi la socket non potrà più essere utilizzata per trasferire dati.



Figura 10.38 Prototipo della system call *close*.

In alternativa, è possibile effettuare la terminazione del processo per ottenere una chiusura implicita del canale. La terminazione di un processo si ottiene tramite il raggiungimento del termine della funzione *main* oppure con una chiamata alla funzione *exit*.



Figura 10.39 Prototipo della system call *exit*.

Occorre fare molta attenzione se si sta implementando un server multiprocesso; quando un processo genera un figlio,

tramite la system call *fork* il nuovo processo eredita tutte le informazioni del padre, compresi i descrittori dei canali di I/O aperti, tra cui si trovano anche le socket. In queste condizioni, fare uso della system call *close* in uno dei due processi non implica che il kernel liberi le strutture dati relative al canale. Infatti, dopo l'uso di una *fork*, continuerà a esistere un solo canale ma con due descrittori diversi nei due processi; alla chiusura di un descrittore il kernel mantiene le risorse occupate per via del secondo descrittore, ancora esistente e che potrebbe essere utilizzato.

Capitolo 11

Implementazione di sistemi client-server

In questo capitolo si affrontano le problematiche proposte nel Capitolo 5 utilizzando gli strumenti che il linguaggio C mette a disposizione.

Si richiama l'attenzione del lettore sul fatto che, per motivi di spazio e comprensibilità, anche in questo capitolo i controlli riguardanti eventuali condizioni di errore saranno presenti solo all'introduzione di nuove funzionalità o quando importanti per la discussione del codice. Ciononostante, si ricorda che il controllo e la gestione degli errori è fondamentale per il buon funzionamento di ogni programma, per cui è buona norma inserire sempre tali controlli durante la fase di sviluppo.

11.1 Formato dei messaggi inviati

Il linguaggio C dà al programmatore libero accesso alla memoria, che viene vista come un array di byte, tramite l'uso dei puntatori; di fatto, un buffer non è altro che una zona contigua di memoria destinata al transito di dati. La manipolazione di un array di byte può avvenire per accesso diretto, con indici o puntatori. Tipi di dati estesi, come *struct* e *union*, possono essere utili, se costruiti con i giusti criteri per accedere tramite operazioni di casting a zone specifiche di un buffer associandovi un riferimento mnemonico. Fare casting del puntatore a un array di byte a una struttura è un metodo molto usato per interpretare velocemente gli header dei pacchetti di un protocollo.

Esistono, inoltre, una serie di funzioni accessorie di cui le principali sono *memcpy*, *memset* e *sprintf*. Le prime due manipolano zone di memoria facendone una copia o impostando tutti i byte a un determinato valore, l'ultima permette di depositare una stringa formattata secondo le stesse convenzioni di *printf* all'interno di un buffer.

11.1.1 Caratteri

Viene definito il tipo nativo *char* della dimensione di 8 bit e con cui si gestiscono i caratteri. La codifica è ASCII [42], per cui non sono necessarie conversioni.

11.1.2 Stringhe

Il linguaggio C mette a disposizione un'astrazione dell'operazione di *ritorno a capo* tramite il carattere “\n”. Secondo lo standard, quando tale carattere è utilizzato su di un canale di I/O dichiarato come testuale, viene trasformato nella corretta sequenza di byte relativa alla piattaforma sottostante (che potrebbe non essere più formata da un singolo carattere); questo vuol dire che su sistemi UNIX verrà tradotto in LF e su altri in CR+LF.

L'operazione di *Carriage Return* (CR) viene invece sempre rappresentata tramite il carattere “\r”.

La conversione delle sequenze di caratteri che rappresentano il ritorno a capo provenienti dalla rete nella sequenza standard (“\n”), utilizzabile nelle operazioni di I/O è totalmente a carico dell'applicazione, tuttavia, si tratta solo di sostituzioni all'internodi una stringa come negli esempi delle Figure 11.1 e 11.2.

```
1 void convertInCR(char *strng)
2 {
3     if (*strng == '\n') -- *strng;
4     if (*strng == '\r') -- *strng;
5 }
```

Figura 11.1 Conversione del ritorno a capoin CR+LF.

Nel codice proposto dalla Figura 11.1 alla riga 2 si controlla se la sequenza attesa sia già presente, in caso negativo viene rintracciato il punto della stringa dove si trova il ritorno a capo da sostituire (riga 3) e lo si sovrascrive con quello richiesto (riga 4).

```
1 void convertInLF(char * stringa)
2 {char * s1 = strstr(stringa, "\r\n");
3  if (s1 != NULL)
4  {strcpy(s1, "\n", s1 + 1);
5  }
6 }
```

Figura 11.2 Conversione del ritorno a capo in LF.

Nella Figura 11.2, invece, alla riga 2 si cerca la posizione del ritorno a capo da sostituire, se presente (riga 3) si sovrascrive la stringa da quel punto in poi con la porzione di sé stessa più avanti di una posizione; ovvero, vengono traslati indietro di una posizione tutti i caratteri successivi a “\r”.

Anche se non sono necessarie conversioni esplicite per ottenere una stringa a partire da un array di byte, occorre comunque inserire il carattere terminazione (“\0”), per cui è sempre consigliato, a fronte di un buffer di dimensione *dim buffer*, effettuare una ricezione di al più *dim buffer* – 1 byte. Questo, a meno di casi in cui si è sicuri di dover eliminare un ritorno a capo al termine della stringa per cui viene ugualmente garantito lo spazio necessario per il terminatore, come negli esempi proposti in questo capitolo.

11.1.3 Valori numerici

Per ordinare in un formato standard i byte di un'informazione numerica possono tornare utili le funzioni *htonl*, *htons*, *ntohl* e *ntohs* già viste nel Paragrafo 10.4. Le funzioni appena menzionate non sono però in grado di gestire numeri in virgola mobile, per cui lo sviluppatore è costretto a implementare comunque un proprio formato, e le corrispondenti funzioni per la conversione.

La conversione di un valore numerico in una stringa che lo rappresenta si ottiene facilmente facendo uso di *sprintf*.

Per convertire una stringa in un valore numerico esistono le funzioni *strtol* e *strtod*, entrambe definite nell'header “*stdlib.h*” e che restituiscono rispettivamente valori di tipo intero e in virgola mobile.

```
#include <stdlib.h>
long strtol(char * buffer, char ** terminatore, int base);
double strtod(char * buffer, char ** terminatore);
```

Figura 11.3 Prototipi delle funzioni *strtol* e *strtod*.

Nel caso specifico in cui le cifre siano decimali è possibile usare, al posto di *strtol*, e *strtod*, le funzioni *atol* e *atof*.



11.1.4 Dati strutturati

La conversione di una struttura (*struct*) in una sequenza di byte richiede che il programmatore implementi una funzione specifica per ogni tipo di struttura. La manipolazione dei singoli campi dipenderà poi dal loro tipo specifico.

Se il risultato della conversione è un stringa valida, l'operazione di riconversione può essere agevolata facendo uso della funzione *strtok*. La funzione *strtok* è definita nell'header "string.h" e si occupa di *spezzare* una stringa in una serie di token intervallati da delimitatori. Si ricordi, a questo proposito, che una stringa *valida* non necessariamente è una stringa *stampabile* a video ma è sufficiente che non contenga il carattere terminatore di stringa "\0" al suo interno.

11.2 Gestione dei messaggi

Come già discusso, i problemi di gestione dei messaggi si presentano in modo più marcato con socket di tipo byte-stream e per poterli affrontare in maniera agevole è opportuno che il lessico del protocollo sia stato ben progettato.

Se il messaggio da ricevere è di lunghezza fissata, o possiede quantomeno una parte iniziale di lunghezza nota (*header*), una buona strategia è quella di effettuare una prima ricezione

utilizzando il parametro `MSG_WAITALL`, dai dati ricevuti ricavare l'informazione relativa alla dimensione totale del messaggio e poi effettuare una successiva lettura dalla socket del numero di byte restanti nuovamente con `MSG_WAITALL`.

Se invece il messaggio ha una lunghezza variabile e fa uso di un terminatore (come, ad esempio, una sequenza di stringhe testuali in arrivo da un web server), la cosa più semplice è leggere un carattere alla volta e depositarlo nei singoli elementi del buffer fino alla ricezione del terminatore.

11.3 Implementazione di un client

In base a quanto già visto nel capitolo precedente è possibile sostituire le diciture nella Figura 5.1 con i nomi delle opportune system call e ottenere la Figura 11.4, dove non sono state riportate tutte le chiamate coinvolte, ma solo quelle più significative.

Durante la discussione del codice, per favorire la comprensione e semplificare la struttura dei programmi, la fase di scambio dati verrà collocata all'interno di una funzione apposita (*fruizione_servizio*), mentre il codice relativo alla gestione della connessione con il server risiederà all'interno del corpo principale del programma.

Si esamineranno ora separatamente i due casi in cui il client fa uso di socket connesse o non connesse.

11.3.1 Client con socket connesse

Facendo riferimento alla Figura 11.4(a), e ricordando quanto già appreso, è possibile cominciare a scrivere il codice per la prima fase, presentato nella Figura 11.5.

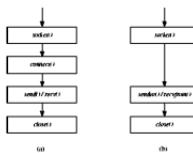


Figura 11.4 Diagrammi delle operazioni di un client con socket connesse (a) e non connesse(b).

```
1 #include <sys/types.h>
2 #include <sys/socket.h>
3 #include <netdb.h>
4 #include <errno.h>
5 #include <string.h>
6 #include <stropts.h>
7
8 int main(int argc, char ** argv, char ** envp)
9 {
10     int fd;
11     // creazione socket
12     struct sockaddr * pa = gethostbyname(argv[1]);
13     int s = socket(AF_INET, SOCK_STREAM, 0);
14     bind(s, pa, sizeof(struct sockaddr));
15 }
```

Figura 11.5 Implementazione della prima fase di un client con socket connesse.

Con l'aggiunta della seconda fase (Figura 11.6) si è costretti a inserire alcune variabili dove memorizzare il nome del nodo a cui connettersi e la porta dove è in attesa il server (righe 17 e 18).

```
25 // giacimenti
26 char *host_list = "localhost";
27 int host_index = 0;
28
29 // connessione
30 struct sockaddr_in *sa = (struct sockaddr_in *)(&host_list[1]);
31 int len = strlen(host_list) + 1; /* (sa->.sa_len) + 1 */
32 sa->.sa_family = AF_INET;
33 sa->.sa_port = htons(22);
34 sa->.sin_port = htons(22);
35 sa->.sin_addr.s_addr = htonl(0xffffffff);
```

Figura 11.6 Implementazione della seconda fase di un client con socket connesse.

Seguendo l'approccio adottato già a partire dal Paragrafo 7.4, sono stati assegnati a queste variabili valori codificati all'interno del programma; purtroppo però, e questo dovrebbe essere emerso anche durante lo svolgimento dell'Esercizio 10.6, il *cablare* valori all'interno del codice non è mai una buona scelta. Se valori dai quali dipende il funzionamento di un programma sono fissati in maniera statica all'interno del suo codice, si osserverà una riduzione di flessibilità e, molto probabilmente, al variare di qualche condizione esterna, quale l'indirizzo del server da contattare, sarà necessaria una nuova compilazione dei sorgenti. Una soluzione molto più efficace consiste nel richiedere all'utente di fornire i parametri in linea di comando, dando al più dei valori di default; in situazioni molto complesse o in presenza di parametri facoltativi è anche possibile ricorrere alla funzione *getopt*.

Un esempio di come sia possibile gestire parametri in linea di comando è presentato nella Figura 11.7.

```

3 #include <sys/types.h>
4 #include <sys/socket.h>
5 #include <netdb.h>
6
7 // parametri
8 char * nome_host = "localhost";
9 int porta = 7000;
10
11 // controllo dei parametri
12 if (argc != 2) {
13     fprintf(stderr, "non ho host portato", argc[1]);
14     exit(1);
15 }
16 nome_host = argv[1];
17 if (porta <= 1024 || porta >= 65535) {
18     fprintf(stderr, "valore della porta non valido %d", porta);
19     exit(1);
20 }
21
22 // connessione
23

```

Figura 11.7 Esempio di gestione di parametri in linea di comando.



La Figura 11.7, nonostante sia funzionale alla spiegazione, non è un esempio di buona strutturazione del codice.

Infatti, nel caso in cui i parametri non fossero conformi alle richieste, l'esecuzione del programma verrebbe interrotta pur avendo già allocato delle risorse (la creazione della socket). Dal punto di vista stilistico la cosa migliore è effettuare per prima cosa il controllo dei parametri terminando eventualmente l'esecuzione e in un secondo momento allocare le risorse di rete.

Si noti, in particolare, l'uso di *fprintf* anziché di *perror* per la segnalazione della condizione di errore alle righe 22 e 29. Tecnicamente, il sistema operativo non ha riscontro di una situazione anomala: la non conformità dei parametri è unicamente un problema dell'applicazione. Inoltre, è consigliabile usare *stderr* come primo parametro di *fprintf* per

indirizzare tutti i messaggi attraverso il canale destinato alla visualizzazione degli errori per evitare che vengano visualizzati fuori ordine in presenza di altre chiamate a *perror*.

Tornando all'implementazione del client, rimangono la terza e quarta fase. Della terza fase, come già anticipato, ci si occuperà separatamente; mentre la quarta, la chiusura del canale di comunicazione, risulta essere abbastanza banale. La parte conclusiva del codice del client è presentata nella Figura 11.8.

```
5 #include <canal.h>
6
7 int fruizione_servizio(int) // prototipo
8 {
9     int socket, request, response; // AT&T_SOCKTYPE, integer(16bit)
10    char buffer[100];
11
12    // apertura del socket
13    if ((socket = socket_create(1, 0)) < 0)
14        return -1;
15
16    // chiusura canale
17    close(socket);
18
19    return 0;
20 }
```

Figura 11.8 Implementazione della terza e quarta fase di un client con socket connesse.

La funzione *fruizione_servizio* accetta come parametro una socket già connessa al server e restituisce un valore numerico che segue la stessa logica delle funzioni di sistema. In tal modo risulterà più semplice l'intercettazione di errori avvenuti durante la fase di scambio dati.

A questo punto il codice che permette al client di collegarsi al server è completo ed è possibile concentrarsi sulla parte di scambio dei dati implementando il corpo della funzione *fruizione_servizio*. Come primo passo si andrà a implementare un client per interagire con il programma realizzato nella Figura 10.34. In questo caso il protocollo a livello applicativo

è estremamente semplice: il client manda una stringa al server e termina. Operativamente, il programma dovrà aspettare che l'utente inserisca una stringa, mandarla al server e terminare. Una possibile implementazione viene proposta nella Figura 11.9.

Come si può notare, la maggior parte del codice si occupa di operazioni che non hanno a che vedere con la rete: le righe dalla 56 alla 59 servono ad acquisire quanto digitato dall'utente, mentre dalla riga 61 alla 64 viene operata una manipolazione dei dati per renderli compatibili con quanto atteso dal server.

Esercizio 11.1



Modificare il programma proposto nella Figura 11.9 in maniera tale da usarlo come client per il server sviluppato per l'Esercizio 10.7. Il client deve prima ricevere dal server una stringa e visualizzarla, poi attende che l'utente digiti una stringa e la invia al server.

```
1 //include <string.h>
2
3 #include <sys/types.h>
4 #include <sys/socket.h>
5 #include <sys/un.h>
6 #include <stropts.h>
7 #include <stropt.h>
8
9 #define PORT 5000
10
11 int main()
12 {
13     int socket_desc;
14     char buffer[1024];
15     char str[1024];
16
17     socket_desc = socket(AF_UNIX, SOCK_STREAM, 0);
18     if (socket_desc == -1)
19     {
20         perror("socket error");
21         exit(1);
22     }
23
24     if (connect(socket_desc, (struct sockaddr *) &addr, sizeof(addr)) == -1)
25     {
26         perror("connection error");
27         exit(1);
28     }
29
30     write(socket_desc, "Hello", strlen("Hello"));
31     read(socket_desc, str, 1024);
32     printf("%s", str);
33
34     close(socket_desc);
35
36     return 0;
37 }
```

Figura 11.9 Fruizione del servizio di spedizione di una stringa con socket connesse.

Ovviamente, un client in grado di inviare una singola stringa non risulta molto utile ai fini pratici. Per renderlo più flessibile è opportuno estenderne il funzionamento inserendo un ciclo infinito di lettura e scrittura: il client attende una stringa in input dall'utente, la invia al server e poi si mette in attesa della risposta; al suo arrivo la risposta viene visualizzata e il client si mette nuovamente in attesa di input da parte dell'utente. Per ottenere questo comportamento, è sufficiente inserire le righe dalla 56 alla 67 in un ciclo infinito avendo cura di spostare la definizione delle variabili *nl* e *r* al di fuori di quest'ultimo e aggiungere le istruzioni per ricevere e visualizzare il messaggio di risposta. Il codice risultante è proposto nella Figura 11.10.

Nonostante il client appena proposto funzioni, esso presenta un errore di progettazione non indifferente: non termina mai. Sotto un aspetto formale sarebbe come aver progettato un protocollo il cui automa non presenta stati finali. Sebbene da un punto di vista teorico questo non sia un problema, all'atto pratico non è applicabile: il client dovrà terminare, se non al ricevimento di un messaggio di chiusura quantomeno di fronte alla chiusura della socket. Nella Figura 11.11 viene estesa la funzione di fruizione del servizio in modo da terminarne l'esecuzione quando viene rilevata la chiusura della socket; per fare questo si sfrutta il fatto che la sistem call *recv* restituisce un numero di byte letti pari a zero se la socket è stata chiusa dalla controparte. Leggendo attentamente il codice, però, si può notare anche che si è ipotizzato che la socket possa essere chiusa

solo durante la fase di lettura, ovvero, che il server chiuda la connessione a fronte di una richiesta del client.

```
53 int fruizione_servizio(int socket) {
54     lse dimBuffer = 1024;
55     lse maxDimBuffer=1024;
56     char * coda;
57     lse ok;
58
59     while(1) {
60         if (input_coda(coda, dimBuffer, ok)) {
61             if (ok == 0) {
62                 return -1;
63             }
64             if (strcmp(coda, "ciao") == 0) {
65                 ok = strinca(coda, "\n");
66                 apri(coda, "txt");
67             }
68             // invio del messaggio al server
69             if (ok == 1) {
70                 sendsocket(coda, ok);
71             }
72             if (ok == 0) {
73                 coda = receivesocket(coda, dimBuffer, ok);
74                 dimBuffer = 21 + strlen(coda);
75                 print("ok", coda);
76             }
77     }
78 }
```

Figura 11.10 Fruizione del servizio di scambio di stringhe con socket connesse.

```
1 // visualizzazione stringhe
2 s = receivesocket(coda, dimBuffer, ok);
3 if (ok == 0) return 0;
4 else {
5     print("ok", coda);
6 }
76
```

Figura 11.11 Rilevamento del fatto che il server ha chiuso la connessione.

Esercizio 11.2



Modificare il programma proposto fino alla Figura 11.11 in maniera tale da usarlo come client per il server sviluppato per l'Esercizio 10.7. Il client deve prima ricevere dal server una stringa e visualizzarla, poi effettuare il ciclo per inviare stringhe al server e attendere le relative risposte.

Esercizio 11.3



Modificare il programma proposto fino alla Figura 11.11 in maniera tale fare terminare il client nel momento in cui l'utente digita una stringa composta unicamente dal carattere “.”.

11.3.2 Client con socket non connesse

Confrontando tra loro i diagrammi nella Figura 11.4 è piuttosto semplice, partendo dal codice proposto nel paragrafo precedente, ricavare un client che faccia uso di socket non connesse e possa usufruire del server per la ricezione di una stringa proposta nella Figura 10.37.

Innanzitutto, rispetto alla Figura 11.5, occorre fare uso di socket UDP in quanto le socket TCP utilizzate negli esempi precedenti non supportano la modalità non connessa. Nella Figura 11.6 la chiamata a *connect* non è più necessaria; vanno però mantenute tutte le operazioni relative ai campi della struttura *ind_server*, perché questa verrà comunque utilizzata come parametro per l'invio del messaggio. Per quanto riguarda, invece, la funzione di fruizione del servizio nella Figura 11.9 è necessario usare *sendto* anziché *send*.

```

20 int main(int argc, char ** argv, char ** envp) {
21
22     // creazione socket
23     int s = socket(AF_INET, SOCK_STREAM, 0);
24     int a = bind(s, (struct sockaddr_in *) &server, sizeof(server));
25
26     // parametri
27     if (argc < 2) {
28         perror("usage: ./socket \"localhost\"");
29     }
30
31     // acquisizione parametri
32     if (argc != 2) {
33         perror("usage: ./socket \"name da host portale\", arg[0]");
34     }
35
36     // nome host
37     struct hostent * he = gethostbyname(argv[1]);
38     int addrlen = sizeof(struct hostent) + ((he->h_addr_list[0]))[0];
39
40     // creazione servizio
41     int lnserv = socket(AF_INET, SOCK_STREAM, 0);
42     if (lnserv == -1) {
43         perror("socket error");
44     }
45
46     // impostazione indirizzo
47     lnserv = bind(lnserv, (struct sockaddr_in *) &server, addrlen);
48
49     // ascolto sul
50     if (listen(lnserv, 5) < 0) {
51         perror("listen error");
52     }
53
54     // chiusura canale
55     close(lnserv);
56
57     return 0;
58 }

```

Figura 11.12 Client con socket non connesse.

Le modifiche appena descritte, per la parte di gestione della socket e per la fruizione del servizio sono presentate rispettivamente nelle Figure 11.12 e 11.13.

In particolare, per quanto riguarda la funzione di fruizione del servizio, è stato necessario estendere il numero di parametri: se la socket non è connessa occorre rendere noto anche l'indirizzo di trasporto a cui inviare il messaggio. Alternativamente, si sarebbe potuta rendere globale la variabile *ind_server*, ma sarebbe stata una pessima scelta dal punto di vista della strutturazione del codice.

```

9 int fruizione_servizio(int, struct sockaddr_in *)
10
11 int fruizione_servizio(int socket, struct sockaddr_in * indirizzo {
12     char buffer[1024];
13
14     if (getpeername(socket, &buffer, &stain) == -1) {
15         perror("getpeername");
16     }
17
18     if (stain.s_port == 0) {
19         close(socket);
20     }
21
22     if (stain.s_port == 0) {
23         strcpy(buffer, "0.0.0.0");
24     }
25
26     if (stain.s_port == 1) {
27         strcpy(buffer, "127.0.0.1");
28     }
29
30     if (stain.s_port == 2) {
31         strcpy(buffer, "192.168.1.1");
32     }
33
34     if (stain.s_port == 3) {
35         strcpy(buffer, "192.168.1.2");
36     }
37
38     if (stain.s_port == 4) {
39         strcpy(buffer, "192.168.1.3");
40     }
41
42     if (stain.s_port == 5) {
43         strcpy(buffer, "192.168.1.4");
44     }
45
46     if (stain.s_port == 6) {
47         strcpy(buffer, "192.168.1.5");
48     }
49
50     if (stain.s_port == 7) {
51         strcpy(buffer, "192.168.1.6");
52     }
53
54     if (stain.s_port == 8) {
55         strcpy(buffer, "192.168.1.7");
56     }
57
58     if (stain.s_port == 9) {
59         strcpy(buffer, "192.168.1.8");
60     }
61
62     if (stain.s_port == 10) {
63         strcpy(buffer, "192.168.1.9");
64     }
65
66     if (stain.s_port == 11) {
67         strcpy(buffer, "192.168.1.10");
68     }
69
70     if (stain.s_port == 12) {
71         strcpy(buffer, "192.168.1.11");
72     }
73
74     if (stain.s_port == 13) {
75         strcpy(buffer, "192.168.1.12");
76     }
77
78     if (stain.s_port == 14) {
79         strcpy(buffer, "192.168.1.13");
80     }
81
82     if (stain.s_port == 15) {
83         strcpy(buffer, "192.168.1.14");
84     }
85
86     if (stain.s_port == 16) {
87         strcpy(buffer, "192.168.1.15");
88     }
89
90     if (stain.s_port == 17) {
91         strcpy(buffer, "192.168.1.16");
92     }
93
94     if (stain.s_port == 18) {
95         strcpy(buffer, "192.168.1.17");
96     }
97
98     if (stain.s_port == 19) {
99         strcpy(buffer, "192.168.1.18");
100    }
101
102    if (stain.s_port == 20) {
103        strcpy(buffer, "192.168.1.19");
104    }
105
106    if (stain.s_port == 21) {
107        strcpy(buffer, "192.168.1.20");
108    }
109
110    if (stain.s_port == 22) {
111        strcpy(buffer, "192.168.1.21");
112    }
113
114    if (stain.s_port == 23) {
115        strcpy(buffer, "192.168.1.22");
116    }
117
118    if (stain.s_port == 24) {
119        strcpy(buffer, "192.168.1.23");
120    }
121
122    if (stain.s_port == 25) {
123        strcpy(buffer, "192.168.1.24");
124    }
125
126    if (stain.s_port == 26) {
127        strcpy(buffer, "192.168.1.25");
128    }
129
130    if (stain.s_port == 27) {
131        strcpy(buffer, "192.168.1.26");
132    }
133
134    if (stain.s_port == 28) {
135        strcpy(buffer, "192.168.1.27");
136    }
137
138    if (stain.s_port == 29) {
139        strcpy(buffer, "192.168.1.28");
140    }
141
142    if (stain.s_port == 30) {
143        strcpy(buffer, "192.168.1.29");
144    }
145
146    if (stain.s_port == 31) {
147        strcpy(buffer, "192.168.1.30");
148    }
149
150    if (stain.s_port == 32) {
151        strcpy(buffer, "192.168.1.31");
152    }
153
154    if (stain.s_port == 33) {
155        strcpy(buffer, "192.168.1.32");
156    }
157
158    if (stain.s_port == 34) {
159        strcpy(buffer, "192.168.1.33");
160    }
161
162    if (stain.s_port == 35) {
163        strcpy(buffer, "192.168.1.34");
164    }
165
166    if (stain.s_port == 36) {
167        strcpy(buffer, "192.168.1.35");
168    }
169
170    if (stain.s_port == 37) {
171        strcpy(buffer, "192.168.1.36");
172    }
173
174    if (stain.s_port == 38) {
175        strcpy(buffer, "192.168.1.37");
176    }
177
178    if (stain.s_port == 39) {
179        strcpy(buffer, "192.168.1.38");
180    }
181
182    if (stain.s_port == 40) {
183        strcpy(buffer, "192.168.1.39");
184    }
185
186    if (stain.s_port == 41) {
187        strcpy(buffer, "192.168.1.40");
188    }
189
190    if (stain.s_port == 42) {
191        strcpy(buffer, "192.168.1.41");
192    }
193
194    if (stain.s_port == 43) {
195        strcpy(buffer, "192.168.1.42");
196    }
197
198    if (stain.s_port == 44) {
199        strcpy(buffer, "192.168.1.43");
200    }
201
202    if (stain.s_port == 45) {
203        strcpy(buffer, "192.168.1.44");
204    }
205
206    if (stain.s_port == 46) {
207        strcpy(buffer, "192.168.1.45");
208    }
209
210    if (stain.s_port == 47) {
211        strcpy(buffer, "192.168.1.46");
212    }
213
214    if (stain.s_port == 48) {
215        strcpy(buffer, "192.168.1.47");
216    }
217
218    if (stain.s_port == 49) {
219        strcpy(buffer, "192.168.1.48");
220    }
221
222    if (stain.s_port == 50) {
223        strcpy(buffer, "192.168.1.49");
224    }
225
226    if (stain.s_port == 51) {
227        strcpy(buffer, "192.168.1.50");
228    }
229
230    if (stain.s_port == 52) {
231        strcpy(buffer, "192.168.1.51");
232    }
233
234    if (stain.s_port == 53) {
235        strcpy(buffer, "192.168.1.52");
236    }
237
238    if (stain.s_port == 54) {
239        strcpy(buffer, "192.168.1.53");
240    }
241
242    if (stain.s_port == 55) {
243        strcpy(buffer, "192.168.1.54");
244    }
245
246    if (stain.s_port == 56) {
247        strcpy(buffer, "192.168.1.55");
248    }
249
250    if (stain.s_port == 57) {
251        strcpy(buffer, "192.168.1.56");
252    }
253
254    if (stain.s_port == 58) {
255        strcpy(buffer, "192.168.1.57");
256    }
257
258    if (stain.s_port == 59) {
259        strcpy(buffer, "192.168.1.58");
260    }
261
262    if (stain.s_port == 60) {
263        strcpy(buffer, "192.168.1.59");
264    }
265
266    if (stain.s_port == 61) {
267        strcpy(buffer, "192.168.1.60");
268    }
269
270    if (stain.s_port == 62) {
271        strcpy(buffer, "192.168.1.61");
272    }
273
274    if (stain.s_port == 63) {
275        strcpy(buffer, "192.168.1.62");
276    }
277
278    if (stain.s_port == 64) {
279        strcpy(buffer, "192.168.1.63");
280    }
281
282    if (stain.s_port == 65) {
283        strcpy(buffer, "192.168.1.64");
284    }
285
286    if (stain.s_port == 66) {
287        strcpy(buffer, "192.168.1.65");
288    }
289
290    if (stain.s_port == 67) {
291        strcpy(buffer, "192.168.1.66");
292    }
293
294    if (stain.s_port == 68) {
295        strcpy(buffer, "192.168.1.67");
296    }
297
298    if (stain.s_port == 69) {
299        strcpy(buffer, "192.168.1.68");
300    }
301
302    if (stain.s_port == 70) {
303        strcpy(buffer, "192.168.1.69");
304    }
305
306    if (stain.s_port == 71) {
307        strcpy(buffer, "192.168.1.70");
308    }
309
310    if (stain.s_port == 72) {
311        strcpy(buffer, "192.168.1.71");
312    }
313
314    if (stain.s_port == 73) {
315        strcpy(buffer, "192.168.1.72");
316    }
317
318    if (stain.s_port == 74) {
319        strcpy(buffer, "192.168.1.73");
320    }
321
322    if (stain.s_port == 75) {
323        strcpy(buffer, "192.168.1.74");
324    }
325
326    if (stain.s_port == 76) {
327        strcpy(buffer, "192.168.1.75");
328    }
329
330    if (stain.s_port == 77) {
331        strcpy(buffer, "192.168.1.76");
332    }
333
334    if (stain.s_port == 78) {
335        strcpy(buffer, "192.168.1.77");
336    }
337
338    if (stain.s_port == 79) {
339        strcpy(buffer, "192.168.1.78");
340    }
341
342    if (stain.s_port == 80) {
343        strcpy(buffer, "192.168.1.79");
344    }
345
346    if (stain.s_port == 81) {
347        strcpy(buffer, "192.168.1.80");
348    }
349
350    if (stain.s_port == 82) {
351        strcpy(buffer, "192.168.1.81");
352    }
353
354    if (stain.s_port == 83) {
355        strcpy(buffer, "192.168.1.82");
356    }
357
358    if (stain.s_port == 84) {
359        strcpy(buffer, "192.168.1.83");
360    }
361
362    if (stain.s_port == 85) {
363        strcpy(buffer, "192.168.1.84");
364    }
365
366    if (stain.s_port == 86) {
367        strcpy(buffer, "192.168.1.85");
368    }
369
370    if (stain.s_port == 87) {
371        strcpy(buffer, "192.168.1.86");
372    }
373
374    if (stain.s_port == 88) {
375        strcpy(buffer, "192.168.1.87");
376    }
377
378    if (stain.s_port == 89) {
379        strcpy(buffer, "192.168.1.88");
380    }
381
382    if (stain.s_port == 90) {
383        strcpy(buffer, "192.168.1.89");
384    }
385
386    if (stain.s_port == 91) {
387        strcpy(buffer, "192.168.1.90");
388    }
389
390    if (stain.s_port == 92) {
391        strcpy(buffer, "192.168.1.91");
392    }
393
394    if (stain.s_port == 93) {
395        strcpy(buffer, "192.168.1.92");
396    }
397
398    if (stain.s_port == 94) {
399        strcpy(buffer, "192.168.1.93");
400    }
401
402    if (stain.s_port == 95) {
403        strcpy(buffer, "192.168.1.94");
404    }
405
406    if (stain.s_port == 96) {
407        strcpy(buffer, "192.168.1.95");
408    }
409
410    if (stain.s_port == 97) {
411        strcpy(buffer, "192.168.1.96");
412    }
413
414    if (stain.s_port == 98) {
415        strcpy(buffer, "192.168.1.97");
416    }
417
418    if (stain.s_port == 99) {
419        strcpy(buffer, "192.168.1.98");
420    }
421
422    if (stain.s_port == 100) {
423        strcpy(buffer, "192.168.1.99");
424    }
425
426    if (stain.s_port == 101) {
427        strcpy(buffer, "192.168.1.100");
428    }
429
430    if (stain.s_port == 102) {
431        strcpy(buffer, "192.168.1.101");
432    }
433
434    if (stain.s_port == 103) {
435        strcpy(buffer, "192.168.1.102");
436    }
437
438    if (stain.s_port == 104) {
439        strcpy(buffer, "192.168.1.103");
440    }
441
442    if (stain.s_port == 105) {
443        strcpy(buffer, "192.168.1.104");
444    }
445
446    if (stain.s_port == 106) {
447        strcpy(buffer, "192.168.1.105");
448    }
449
450    if (stain.s_port == 107) {
451        strcpy(buffer, "192.168.1.106");
452    }
453
454    if (stain.s_port == 108) {
455        strcpy(buffer, "192.168.1.107");
456    }
457
458    if (stain.s_port == 109) {
459        strcpy(buffer, "192.168.1.108");
460    }
461
462    if (stain.s_port == 110) {
463        strcpy(buffer, "192.168.1.109");
464    }
465
466    if (stain.s_port == 111) {
467        strcpy(buffer, "192.168.1.110");
468    }
469
470    if (stain.s_port == 112) {
471        strcpy(buffer, "192.168.1.111");
472    }
473
474    if (stain.s_port == 113) {
475        strcpy(buffer, "192.168.1.112");
476    }
477
478    if (stain.s_port == 114) {
479        strcpy(buffer, "192.168.1.113");
480    }
481
482    if (stain.s_port == 115) {
483        strcpy(buffer, "192.168.1.114");
484    }
485
486    if (stain.s_port == 116) {
487        strcpy(buffer, "192.168.1.115");
488    }
489
490    if (stain.s_port == 117) {
491        strcpy(buffer, "192.168.1.116");
492    }
493
494    if (stain.s_port == 118) {
495        strcpy(buffer, "192.168.1.117");
496    }
497
498    if (stain.s_port == 119) {
499        strcpy(buffer, "192.168.1.118");
500    }
501
502    if (stain.s_port == 120) {
503        strcpy(buffer, "192.168.1.119");
504    }
505
506    if (stain.s_port == 121) {
507        strcpy(buffer, "192.168.1.120");
508    }
509
510    if (stain.s_port == 122) {
511        strcpy(buffer, "192.168.1.121");
512    }
513
514    if (stain.s_port == 123) {
515        strcpy(buffer, "192.168.1.122");
516    }
517
518    if (stain.s_port == 124) {
519        strcpy(buffer, "192.168.1.123");
520    }
521
522    if (stain.s_port == 125) {
523        strcpy(buffer, "192.168.1.124");
524    }
525
526    if (stain.s_port == 126) {
527        strcpy(buffer, "192.168.1.125");
528    }
529
530    if (stain.s_port == 127) {
531        strcpy(buffer, "192.168.1.126");
532    }
533
534    if (stain.s_port == 128) {
535        strcpy(buffer, "192.168.1.127");
536    }
537
538    if (stain.s_port == 129) {
539        strcpy(buffer, "192.168.1.128");
540    }
541
542    if (stain.s_port == 130) {
543        strcpy(buffer, "192.168.1.129");
544    }
545
546    if (stain.s_port == 131) {
547        strcpy(buffer, "192.168.1.130");
548    }
549
550    if (stain.s_port == 132) {
551        strcpy(buffer, "192.168.1.131");
552    }
553
554    if (stain.s_port == 133) {
555        strcpy(buffer, "192.168.1.132");
556    }
557
558    if (stain.s_port == 134) {
559        strcpy(buffer, "192.168.1.133");
560    }
561
562    if (stain.s_port == 135) {
563        strcpy(buffer, "192.168.1.134");
564    }
565
566    if (stain.s_port == 136) {
567        strcpy(buffer, "192.168.1.135");
568    }
569
570    if (stain.s_port == 137) {
571        strcpy(buffer, "192.168.1.136");
572    }
573
574    if (stain.s_port == 138) {
575        strcpy(buffer, "192.168.1.137");
576    }
577
578    if (stain.s_port == 139) {
579        strcpy(buffer, "192.168.1.138");
580    }
581
582    if (stain.s_port == 140) {
583        strcpy(buffer, "192.168.1.139");
584    }
585
586    if (stain.s_port == 141) {
587        strcpy(buffer, "192.168.1.140");
588    }
589
590    if (stain.s_port == 142) {
591        strcpy(buffer, "192.168.1.141");
592    }
593
594    if (stain.s_port == 143) {
595        strcpy(buffer, "192.168.1.142");
596    }
597
598    if (stain.s_port == 144) {
599        strcpy(buffer, "192.168.1.143");
600    }
601
602    if (stain.s_port == 145) {
603        strcpy(buffer, "192.168.1.144");
604    }
605
606    if (stain.s_port == 146) {
607        strcpy(buffer, "192.168.1.145");
608    }
609
610    if (stain.s_port == 147) {
611        strcpy(buffer, "192.168.1.146");
612    }
613
614    if (stain.s_port == 148) {
615        strcpy(buffer, "192.168.1.147");
616    }
617
618    if (stain.s_port == 149) {
619        strcpy(buffer, "192.168.1.148");
620    }
621
622    if (stain.s_port == 150) {
623        strcpy(buffer, "192.168.1.149");
624    }
625
626    if (stain.s_port == 151) {
627        strcpy(buffer, "192.168.1.150");
628    }
629
630    if (stain.s_port == 152) {
631        strcpy(buffer, "192.168.1.151");
632    }
633
634    if (stain.s_port == 153) {
635        strcpy(buffer, "192.168.1.152");
636    }
637
638    if (stain.s_port == 154) {
639        strcpy(buffer, "192.168.1.153");
640    }
641
642    if (stain.s_port == 155) {
643        strcpy(buffer, "192.168.1.154");
644    }
645
646    if (stain.s_port == 156) {
647        strcpy(buffer, "192.168.1.155");
648    }
649
650    if (stain.s_port == 157) {
651        strcpy(buffer, "192.168.1.156");
652    }
653
654    if (stain.s_port == 158) {
655        strcpy(buffer, "192.168.1.157");
656    }
657
658    if (stain.s_port == 159) {
659        strcpy(buffer, "192.168.1.158");
660    }
661
662    if (stain.s_port == 160) {
663        strcpy(buffer, "192.168.1.159");
664    }
665
666    if (stain.s_port == 161) {
667        strcpy(buffer, "192.168.1.160");
668    }
669
670    if (stain.s_port == 162) {
671        strcpy(buffer, "192.168.1.161");
672    }
673
674    if (stain.s_port == 163) {
675        strcpy(buffer, "192.168.1.162");
676    }
677
678    if (stain.s_port == 164) {
679        strcpy(buffer, "192.168.1.163");
680    }
681
682    if (stain.s_port == 165) {
683        strcpy(buffer, "192.168.1.164");
684    }
685
686    if (stain.s_port == 166) {
687        strcpy(buffer, "192.168.1.165");
688    }
689
690    if (stain.s_port == 167) {
691        strcpy(buffer, "192.168.1.166");
692    }
693
694    if (stain.s_port == 168) {
695        strcpy(buffer, "192.168.1.167");
696    }
697
698    if (stain.s_port == 169) {
699        strcpy(buffer, "192.168.1.168");
700    }
701
702    if (stain.s_port == 170) {
703        strcpy(buffer, "192.168.1.169");
704    }
705
706    if (stain.s_port == 171) {
707        strcpy(buffer, "192.168.1.170");
708    }
709
710    if (stain.s_port == 172) {
711        strcpy(buffer, "192.168.1.171");
712    }
713
714    if (stain.s_port == 173) {
715        strcpy(buffer, "192.168.1.172");
716    }
717
718    if (stain.s_port == 174) {
719        strcpy(buffer, "192.168.1.173");
720    }
721
722    if (stain.s_port == 175) {
723        strcpy(buffer, "192.168.1.174");
724    }
725
726    if (stain.s_port == 176) {
727        strcpy(buffer, "192.168.1.175");
728    }
729
730    if (stain.s_port == 177) {
731        strcpy(buffer, "192.168.1.176");
732    }
733
734    if (stain.s_port == 178) {
735        strcpy(buffer, "192.168.1.177");
736    }
737
738    if (stain.s_port == 179) {
739        strcpy(buffer, "192.168.1.178");
740    }
741
742    if (stain.s_port == 180) {
743        strcpy(buffer, "192.168.1.179");
744    }
745
746    if (stain.s_port == 181) {
747        strcpy(buffer, "192.168.1.180");
748    }
749
750    if (stain.s_port == 182) {
751        strcpy(buffer, "192.168.1.181");
752    }
753
754    if (stain.s_port == 183) {
755        strcpy(buffer, "192.168.1.182");
756    }
757
758    if (stain.s_port == 184) {
759        strcpy(buffer, "192.168.1.183");
760    }
761
762    if (stain.s_port == 185) {
763        strcpy(buffer, "192.168.1.184");
764    }
765
766    if (stain.s_port == 186) {
767        strcpy(buffer, "192.168.1.185");
768    }
769
770    if (stain.s_port == 187) {
771        strcpy(buffer, "192.168.1.186");
772    }
773
774    if (stain.s_port == 188) {
775        strcpy(buffer, "192.168.1.187");
776    }
777
778    if (stain.s_port == 189) {
779        strcpy(buffer, "192.168.1.188");
780    }
781
782    if (stain.s_port == 190) {
783        strcpy(buffer, "192.168.1.189");
784    }
785
786    if (stain.s_port == 191) {
787        strcpy(buffer, "192.168.1.190");
788    }
789
790    if (stain.s_port == 192) {
791        strcpy(buffer, "192.168.1.191");
792    }
793
794    if (stain.s_port == 193) {
795        strcpy(buffer, "192.168.1.192");
796    }
797
798    if (stain.s_port == 194) {
799        strcpy(buffer, "192.168.1.193");
800    }
801
802    if (stain.s_port == 195) {
803        strcpy(buffer, "192.168.1.194");
804    }
805
806    if (stain.s_port == 196) {
807        strcpy(buffer, "192.168.1.195");
808    }
809
810    if (stain.s_port == 197) {
811        strcpy(buffer, "192.168.1.196");
812    }
813
814    if (stain.s_port == 198) {
815        strcpy(buffer, "192.168.1.197");
816    }
817
818    if (stain.s_port == 199) {
819        strcpy(buffer, "192.168.1.198");
820    }
821
822    if (stain.s_port == 200) {
823        strcpy(buffer, "192.168.1.199");
824    }
825
826    if (stain.s_port == 201) {
827        strcpy(buffer, "192.168.1.200");
828    }
829
830    if (stain.s_port == 202) {
831        strcpy(buffer, "192.168.1.201");
832    }
833
834    if (stain.s_port == 203) {
835        strcpy(buffer, "192.168.1.202");
836    }
837
838    if (stain.s_port == 204) {
839        strcpy(buffer, "192.168.1.203");
840    }
841
842    if (stain.s_port == 205) {
843        strcpy(buffer, "192.168.1.204");
844    }
845
846    if (stain.s_port == 206) {
847        strcpy(buffer, "192.168.1.205");
848    }
849
850    if (stain.s_port == 207) {
851        strcpy(buffer, "192.168.1.206");
852    }
853
854    if (stain.s_port == 208) {
855        strcpy(buffer, "192.168.1.207");
856    }
857
858    if (stain.s_port == 209) {
859        strcpy(buffer, "192.168.1.208");
860    }
861
862    if (stain.s_port == 210) {
863        strcpy(buffer, "192.168.1.209");
864    }
865
866    if (stain.s_port == 211) {
867        strcpy(buffer, "192.168.1.210");
868    }
869
870    if (stain.s_port == 212) {
871        strcpy(buffer, "192.168.1.211");
872    }
873
874    if (stain.s_port == 213) {
875        strcpy(buffer, "192.168.1.212");
876    }
877
878    if (stain.s_port == 214) {
879        strcpy(buffer, "192.168.1.213");
880    }
881
882    if (stain.s_port == 215) {
883        strcpy(buffer, "192.168.1.214");
884    }
885
886    if (stain.s_port == 216) {
887        strcpy(buffer, "192.168.1.215");
888    }
889
890    if (stain.s_port == 217) {
891        strcpy(buffer, "192.168.1.216");
892    }
893
894    if (stain.s_port == 218) {
895        strcpy(buffer, "192.168.1.217");
896    }
897
898    if (stain.s_port == 219) {
899        strcpy(buffer, "192.168.1.218");
900    }
901
902    if (stain.s_port == 220) {
903        strcpy(buffer, "192.168.1.219");
904    }
905
906    if (stain.s_port == 221) {
907        strcpy(buffer, "192.168.1.220");
908    }
909
910    if (stain.s_port == 222) {
911        strcpy(buffer, "192.168.1.221");
912    }
913
914    if (stain.s_port == 223) {
915        strcpy(buffer, "192.168.1.222");
916    }
917
918    if (stain.s_port == 224) {
919        strcpy(buffer, "192.168.1.223");
920    }
921
922    if (stain.s_port == 225) {
923        strcpy(buffer, "192.168.1.224");
924    }
925
926    if (stain.s_port == 226) {
927        strcpy(buffer, "192.168.1.225");
928    }
929
930    if (stain.s_port == 227) {
931        strcpy(buffer, "192.168.1.226");
932    }
933
934    if (stain.s_port == 228) {
935        strcpy(buffer, "192.168.1.227");
936    }
937
938    if (stain.s_port == 229) {
939        strcpy(buffer, "192.168.1.228");
940    }
941
942    if (stain.s_port == 230) {
943        strcpy(buffer, "192.168.1.229");
944    }
945
946    if (stain.s_port == 231) {
947        strcpy(buffer, "192.168.1.230");
948    }
949
950    if (stain.s_port == 232) {
951        strcpy(buffer, "192.168.1.231");
952    }
953
954    if (stain.s_port == 233) {
955        strcpy(buffer, "192.168.1.232");
956    }
957
958    if (stain.s_port == 234) {
959        strcpy(buffer, "192.168.1.233");
960    }
961
962    if (stain.s_port == 235) {
963        strcpy(buffer, "192.168.1.234");
964    }
965
966    if (stain.s_port == 236) {
967        strcpy(buffer, "192.168.1.235");
968    }
969
970    if (stain.s_port == 237) {
971        strcpy(buffer, "192.168.1.236");
972    }
973
974    if (stain.s_port == 238) {
975        strcpy(buffer, "192.168.1.237");
976    }
977
978    if (stain.s_port == 239) {
979        strcpy(buffer, "192.168.1.238");
980    }
981
982    if (stain.s_port == 240) {
983        strcpy(buffer, "192.168.1.239");
984    }
985
986    if (stain.s_port == 241) {
987        strcpy(buffer, "192.168.1.240");
988    }
989
990    if (stain.s_port == 242) {
991        strcpy(buffer, "192.168.1.241");
992    }
993
994    if (stain.s_port == 243) {
995        strcpy(buffer, "192.168.1.242");
996    }
997
998    if (stain.s_port == 244) {
999        strcpy(buffer, "192.168.1.243");
1000    }
1001
1002    if (stain.s_port == 245) {
1003        strcpy(buffer, "192.168.1.244");
1004    }
1005
1006    if (stain.s_port == 246) {
1007        strcpy(buffer, "192.168.1.245");
1008    }
1009
1
```

Figura 11.13 Fruizione del servizio di spedizione di una stringa con socket non connesse.

Come si può notare, le modifiche apportate rispetto al caso precedente sono minime e si limitano alle righe 8, 13, 14, 40, 42, 51 e 66.

Un problema, in questo caso, è dato dal fatto che non è possibile implementare un equivalente del programma richiesto per l'Esercizio 11.1 in quanto, mancando la fase di connessione, il server non è in grado di *prendere l'iniziativa* e mandare un *prompt* al client.

Viceversa, l'aggiunta di un ciclo di lettura e scrittura (Figura 11.14) viene semplificata: essendo la socket non connessa, non è più necessario (possibile) controllare se il server ha chiuso il canale di connessione però, non avendo un riscontro dell'effettiva consegna dei pacchetti a livello di trasporto, tutti i messaggi da un certo momento in avanti potrebbero essere persi perché inviati a un indirizzo non più associato a una socket.

La soluzione appena proposta e rappresentata nella Figura 11.14 presenta però due gravi problemi.

Il primo riguarda la sicurezza: per essere assolutamente certi dell'autenticità della risposta è necessario confrontare il valore dei campi della variabile *risposta* con quelli della variabile *indirizzo*. Diversamente, un altro programma malintenzionato potrebbe intercettare la richiesta e rispondere al posto del

server, influenzando in maniera anomala il comportamento del client.

```
71 100 #include <sys/types.h> // socket, struct sockaddr_in + in_addr.h
72 101 int dia_buffer = 1024;
73 102 #include <sys/socket.h> // socket, close, bind, listen, accept
74 103 #include <sys/types.h> // per la gestione della risposta
75 104 unsigned int dia_server;
76 105
77 106 while(1)
78 107 {
79 108     if ((ptr_diabuff, dia_buffer, atom) == NULL) {
80 109         perror("Error");
81 110         exit(-1);
82 111     }
83 112     if (atom[0] == '\n') == NULL) {
84 113         char * s1 = strchr(atom, '\n');
85 114         *s1 = '\0';
86 115     }
87 116     atom[0] = '\0';
88 117     atom[1] = '\0';
89 118     atom[2] = '\0';
90 119     atom[3] = '\0';
91 120     atom[4] = '\0';
92 121     atom[5] = '\0';
93 122     atom[6] = '\0';
94 123     atom[7] = '\0';
95 124     atom[8] = '\0';
96 125     atom[9] = '\0';
97 126     atom[10] = '\0';
98 127     atom[11] = '\0';
99 128     atom[12] = '\0';
100 129     atom[13] = '\0';
101 130     atom[14] = '\0';
102 131     atom[15] = '\0';
103 132     atom[16] = '\0';
104 133     atom[17] = '\0';
105 134     atom[18] = '\0';
106 135     atom[19] = '\0';
107 136     atom[20] = '\0';
108 137     atom[21] = '\0';
109 138     atom[22] = '\0';
110 139     atom[23] = '\0';
111 140     atom[24] = '\0';
112 141     atom[25] = '\0';
113 142     atom[26] = '\0';
114 143     atom[27] = '\0';
115 144     atom[28] = '\0';
116 145     atom[29] = '\0';
117 146     atom[30] = '\0';
118 147     atom[31] = '\0';
119 148     atom[32] = '\0';
120 149     atom[33] = '\0';
121 150     atom[34] = '\0';
122 151     atom[35] = '\0';
123 152     atom[36] = '\0';
124 153     atom[37] = '\0';
125 154     atom[38] = '\0';
126 155     atom[39] = '\0';
127 156     atom[40] = '\0';
128 157     atom[41] = '\0';
129 158     atom[42] = '\0';
130 159     atom[43] = '\0';
131 160     atom[44] = '\0';
132 161     atom[45] = '\0';
133 162     atom[46] = '\0';
134 163     atom[47] = '\0';
135 164     atom[48] = '\0';
136 165     atom[49] = '\0';
137 166     atom[50] = '\0';
138 167     atom[51] = '\0';
139 168     atom[52] = '\0';
140 169     atom[53] = '\0';
141 170     atom[54] = '\0';
142 171     atom[55] = '\0';
143 172     atom[56] = '\0';
144 173     atom[57] = '\0';
145 174 }
```

Figura 11.14 Fruizione del servizio di scambio di stringhe con socket non connesse.

Esercizio 11.4



Modificare il programma proposto nella Figura 11.14 in maniera tale da prevenire attacchi di sicurezza.

Il secondo problema, invece, riguarda l'affidabilità: il client implementa un protocollo di tipo *stop-and-wait* (in cui cioè vi è un'alternanza stretta nello scambio di messaggi) su un canale inaffidabile; se un messaggio viene perso il sistema si blocca irrimediabilmente. Purtroppo, con gli strumenti presi in esame fin qui non è ancora possibile risolvere questo problema.

Esercizio 11.5



Modificare il programma proposto nella Figura 11.14 in maniera tale fare terminare il client nel momento in cui l'utente digita una stringa composta unicamente dal carattere “.”.

Esercizio 11.6



Si modifichi il codice riportato nella Figura 11.12 in maniera tale da aggiungere un terzo parametro alla linea di comando per permettere all'utente di scegliere se il client debba utilizzare socket conconnessione o senza connessione.

11.4 Implementazione di un server iterativo

Come nel caso del client, per semplificare la comprensione del codice la fase di scambio dati verrà collocata all'interno di una funzione apposita (*erogazione_servizio*).

11.4.1 Server iterativo con socket connesse

Combinando tra loro i diagrammi di client e server iterativi e sostituendo le operazioni con le opportune chiamate a system call, si ottiene quanto rappresentato nella Figura 11.15.

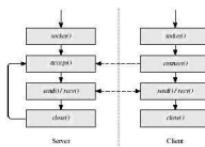


Figura 11.15 Diagramma delle operazioni congruenti di client e server iterativo con socket connesse.

Similmente a quanto già visto analizzando la struttura del client, sono quattro le operazioni che è possibile tradurre in codice. La prima, la creazione della socket, è identica a quanto visto precedenza (Figura 11.5), quindi non verrà riportata ulteriormente. Il programma prosegue poi con l'analisi dei parametri forniti dall'utente; in questo caso ci si trova di fronte a necessità diverse e il codice cambia, come si può osservare nella Figura 11.16.

In manzitutto non è più necessaria una variabile con il nome di un nodo a cui collegarsi; secondariamente, quello della porta diviene un parametro facoltativo (e per convenzione nel messaggio di errore viene indicato tra parentesi quadre), per cui l'unico vincolo è che il numero di parametri non sia superiore

a 1 (riga 20). Il valore di default per la porta è 0, quindi, in assenza di parametri si lascia che sia il sistema operativo a selezionarne una. Se è specificata una porta sulla quale mettersi in ascolto (riga 25), allora viene interpretato il primo parametro (riga 26); in questo caso anche il valore 0 è accettabile (riga 27) perché l'utente potrebbe voler esplicitamente lasciare la scelta al sistema.

```
3 #include <errno.h>
4 #include <sys/types.h>
5
6 // parametri
7 // se non viene specificata la porta si dovrà
8 // assegnare un valore
9 // se viene specificata la porta si dovrà
10 // assegnare un valore
11 // se non viene specificata la porta si dovrà
12 // assegnare un valore
13 // se viene specificata la porta si dovrà
14 // assegnare un valore
15 // se non viene specificata la porta si dovrà
16 // assegnare un valore
17 // se viene specificata la porta si dovrà
18 // assegnare un valore
19 // se non viene specificata la porta si dovrà
20 // assegnare un valore
21 // se viene specificata la porta si dovrà
22 // assegnare un valore
23 // se non viene specificata la porta si dovrà
24 // assegnare un valore
25 // se viene specificata la porta si dovrà
26 // assegnare un valore
27 // se non viene specificata la porta si dovrà
28 // assegnare un valore
29 // se viene specificata la porta si dovrà
30 // assegnare un valore
31 // se non viene specificata la porta si dovrà
32 // assegnare un valore
33 }
```

Figura 11.16 Gestione dei parametri per un server iterativo.

Successivamente, si procede ad associare la socket a un indirizzo di trasporto e ad abilitarla alla ricezione di chiamate facendo uso di *bind* e *listen* (Figura 11.17).

```
31 // associazione della socket all'indirizzo
32 struct sockaddr_in indirizzo;
33
34 // impostazione delle proprietà della socket
35 // impostazione dell'indirizzo - IP4
36 // impostazione della porta - 10000 - TIPOSOGLIOLO
37 // impostazione della durata della connessione
38 // (dovuta alla funzione bind) a 0
39 // perché bind() non ha un timeout
40 // impostazione della durata della connessione
41 // (dovuta alla funzione accept()) a 0
42 // impostazione della durata della connessione
43 // (dovuta alla funzione accept()) a 0
44 // impostazione della durata della connessione
45 // (dovuta alla funzione accept()) a 0
46 // impostazione della durata della connessione
47 // (dovuta alla funzione accept()) a 0
48 // impostazione della durata della connessione
49 // (dovuta alla funzione accept()) a 0
50 }
```

Figura 11.17 Associazione e abilitazione alla ricezione della socket.

È importante in questo punto effettuare un controllo sul valore di ritorno della system call *bind* (riga 38); la porta selezionata

dall'utente potrebbe essere già occupata e il programma deve essere interrotto.

Le righe dalla 44 alla 49 vengono eseguite solo nel caso un cui il valore della porta è uguale a 0 e richiedono al sistema operativo i dati relativi alla socket, per comunicare all'utente la porta a cui questa è stata associata.

Fino a questo punto il programma è nella fase iniziale: sono state infatti solo allocate le risorse per permettere l'erogazione del servizio.

Il codice relativo al il ciclo di gestione dei client viene proposto nella Figura 11.18.

```
3 *include <sys/types.h>
4
5 #include <sys/socket.h>
6
7 int eseguzione_servizio(int); // prototipo
8
9 /**
10  * struct sockaddr_in chiamante
11  * contiene int din_MiAscolta = bind(din, chiamante);
12  * int socket_dati = accept(din, din_MiAscolta, &din_chiamante);
13  */
14
15 // ciclo di gestione dei client
16 while (1) {
17     // attesa di connessione
18     socket_dati = accept(din, din_chiamante, &din_chiamante);
19
20     // ricezione dati
21     if (read(socket_dati, servizio, socket_dati) < 0)
22         return -1;
23
24     // chiusura canale dati
25     close(socket_dati);
26 }
27 }
```

Figura 11.18 Il ciclo di gestione dei client di un server iterativo con socket connesse.

Come si può notare, la socket principale non viene mai chiusa e il programma non termina; all'interno del ciclo (righe dalla 56 alla 66) viene per prima cosa effettuata una chiamata alla system call *accept* (riga 58). Tale chiamata blocca l'esecuzione fino a che un client non effettuerà una *connect*, nel momento in cui l'esecuzione riprende viene erogato il servizio tramite la

socket dati appena creata (riga 61) e successivamente vengono rilasciate le risorse (riga 65) per tornare in attesa del prossimo client.

Terminati tutti i passaggi per la gestione della socket, la funzione di erogazione del servizio diventa a questo punto l'unica preoccupazione; si tratta di usare le system call *send* e *recv* in maniera opportuna per scambiare dati, secondo un protocollo stabilito, con la funzione di fruizione del servizio implementata dal client. Nella Figura 11.19 viene presentato il codice in grado di dialogare con il client proposto nella Figura 11.10: il client manda un messaggio al server e che lo rimanda al client.

Il servizio appena descritto prende il nome di “echo” [63], standardizzato dalla IANA sulla porta 7 sia per UDP che per TCP.

Nonostante la funzione di erogazione del servizio sia molto breve, vale la pena fare qualche commento. Il servizio non entra nel merito dei dati: legge dalla socket una sequenza di byte e ve li riscrive, per cui non è necessario adattare la sequenza per il ritorno a capo.

```
11 int erogazione_servizio(int socket) {
12     char buffer[100];
13     char bufferIn,bufferOut;
14     int n;
15
16     while(1) {
17         if ((n = recv(socket, buffer, 100, 0)) < 0)
18             send(socket, buffer, n, 0);
19         else
20             return n;
21     }
22 }
```

Figura 11.19 Funzione di erogazione del servizio echo con socket connesse.

Inoltre, il valore di ritorno della system call *recv* viene usato come discriminante: se sono stati letti dei dati e l'espressione alla riga 78 risulta vera il buffer viene rispedito al mittente; diversamente, l'erogazione del servizio termina restituendo un valore negativo in caso di errore o zero altrimenti (riga 81).

È assai improbabile trovare oggi dei nodi su Internet con il servizio echo disponibile. Il servizio echo, infatti, può facilmente essere usato per un attacco di tipo DOS: un client malintenzionato manda messaggi indiscriminatamente in maniera tale da utilizzare tutta la banda disponibile e prevenire l'erogazione di altri servizi.

Esercizio 11.7

Modificare il programma proposto nella Figura 11.19 in maniera tale da stampare informazioni sulle stringhe in transito.

Esercizio 11.8

Modificare il programma proposto nella Figura 11.19 in maniera tale per cui nel momento in cui il client invia un messaggio composto unicamente dal carattere “.” il server eroga il servizio e poi termina.

Esercizio 11.9



Implementare un server iterativo per l'erogazione del servizio *chargen* [64]. Se ne verifichi il buon funzionamento facendo uso del comando telnet.

Esercizio 11.10



Implementare un server iterativo per l'erogazione di un servizio di calcolo secondo la notazione polacca inversa (RPN). Un'operazione si compone di tre messaggi in sequenza: due numeri che possono essere interi o in virgola mobile e l'operazione da effettuare. Quando riceve il

messaggio relativo all'operazione il server calcola il risultato, lo manda al client e chiude la connessione. Se ne verifichi il buon funzionamento utilizzando il comando telnet.

11.4.2 Server iterativo con socket non connesse

Combinando tra loro i diagrammi di client e server iterativi facendo uso di socket non connesse e operando le sostituzioni alle diciture dei blocchi si ottiene il diagramma congiunto rappresentato nella Figura 11.20.

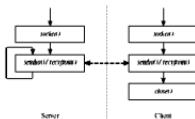


Figura 11.20 Diagramma delle operazioni congruenti di client e server iterativo con socket non connesse.

Un server di questo tipo può essere ricavato facilmente modificando i sorgenti proposti nelle Figure 11.16, 11.17 e 11.18 e facendo una serie di considerazioni simili a quelle viste per un client con socket non connesse. La socket deve essere di tipo datagram e utilizzare il protocollo UDP, l'operazione di ricezione della connessione non è più necessaria come pure non

è più necessaria una socket separata per i dati; il ciclo di gestione dei client, infine, viene effettuato sulla funzione di erogazione del servizio.

Il codice del server risultante è proposto nella Figura 11.21 che mostra la differenza con il codice visto nel caso di socket connesse.

```
29 struct protocol { p = getprotobyname("tcp");}
30 int s = socket(AF_INET, SOCK_STREAM, p->p_proto);
31 ...
32     // la gestione del client
33     while (1) {
34         readable(s);
35         if (read(s, &servizio, 1) < 0)
36             return -1;
37     }
38 }
```

Figura 11.21 Server iterativo con socket non connesse.

Relativamente alla funzione di erogazione del servizio ci si trova di fronte a un cambiamento: mancando la fase di ricezione della connessione, ogni messaggio deve essere gestito come una informazione a sé stante in quanto non è detto che due messaggi successivi provengano dallo stesso client. Le alternative possibili sono due: ricevere il messaggio all'interno del ciclo di gestione dei client e demandare solo la risposta alla funzione di erogazione del servizio oppure svolgere tutte e le operazioni all'interno di quest'ultima. La prima soluzione è sconsigliata in quanto complica la struttura del programma e operazioni di lettura e scrittura correlate tra loro sarebbero collocate in punti diversi del programma. La seconda è di più facile implementazione e permette di mantenere inalterati i parametri della funzione, per cui negli esempi si è scelto di seguire questa strada.

Il codice della funzione di erogazione del servizio è quasi identico alla versione con socket connesse se non per il fatto che gestisce un solo messaggio e le variabili necessarie a memorizzare l'indirizzo del mittente vengono ora definite al suo interno (Figura 11.22).

```
11.22 erogazione_servizio(int socket) {
12     struct sockaddr_in chiamante;
13     int dim_chiamante = sizeof(struct sockaddr);
14     int dim_buffer = 1024;
15     char buffer[dim_buffer];
16     int t;
17
18     t = recvfrom(socket, buffer, dim_buffer, 0,
19                  &chiamante, &dim_chiamante);
20
21     if (t > 0)
22         sendto(socket, buffer, t, 0, chiamante, dim_chiamante);
23
24 }
```

Figura 11.22 Funzione di erogazione del servizio echo con socket non connesse.

Esercizio 11.11



Si modifichi il codice riportato nella Figura 11.21 in maniera tale da aggiungere un secondo parametro alla linea di comando per permettere all'utente di scegliere se il server debba utilizzare socket con connessione o senza connessione.

Da questo momento e per il resto del capitolo la funzione di erogazione del servizio rivestirà un interesse secondario: le differenze tra i vari tipi di server non risiedono nel modo in cui questi scambiano messaggi con il client, bensì nelle diverse

modalità e strategie in cui questa funzione viene invocata nei vari contesti.

11.5 Implementazione di un server concorrente

I sistemi operativi basati su UNIX mettono a disposizione del programmatore la system call *select* definita nell'header “sys/select.h” (oppure in “sys/unistd” su sistemi meno recenti). Questa system call può essere usata per bloccare l'esecuzione di un programma fino al verificarsi di un evento relativo a uno o più canali di I/O.

```
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/conf.h>
#include <sys/conf.h>
#include <sys/conf.h>
```

Figura 11.23 Prototipo della system call *select*.

Il primo parametro, *nfds*, istruisce la *select* fino a quale descrittore di file effettuare il controllo.

nfds non rappresenta il numero di descrittori di file, ma un valore massimo. La chiamata a *select* sospenderà l'esecuzione fino al sopraggiungere di un evento in un descrittore di file da 0 a *nfds* – 1. Se, ad esempio, si è



interessati a monitorare i descrittori di file 4 e 17, il valore di *nfds* non dovrà essere 2, bensì 18 (17+1).

I tre parametri successivi risultano essere di tipo *fd_set*; tale tipo è in realtà un array di bit dove ogni valore rappresenta il descrittore di file il cui numero corrisponde all'indice dell'elemento. Questi array di bit non possono essere manipolati direttamente con degli indici ma occorre fare uso di una serie di macro apposite e riportate nella Tabella 11.1. All'interno dei tre array dovranno essere impostati a 1 i bit relativi ai descrittori di file su cui si è interessati a sospendere l'esecuzione. Il primo array farà terminare le *select* a fronte di dati disponibili per la lettura (il client ha inviato un messaggio); il secondo a fronte di una richiesta di dati (il server dovrà inviare un messaggio); l'ultimo array farà invece terminare la chiamata a fronte di una condizione di errore. Se uno di questi parametri avrà un valore NULL, allora verrà considerato come se composto di soli zeri.

Macro	Effetto
FD_SET(fd, fd_set * array)	Assegna tutti i bit nell'array
FD_SET(fd, fd_set * array)	Imposta a 1 il bit <i>fd</i> dell'array
FD_CLR(fd, fd_set * array)	Imposta a 0 il bit <i>fd</i> dell'array
FD_ISSET(fd, fd_set * array)	Restituisce un valore positivo se il bit <i>fd</i> è impostato a 1 nell'array

Tabella 11.1 Macro definite per la gestione di variabili di tipo *fd_set*.

L'ultimo parametro è un puntatore a una struttura di tipo *timeval* che specifica un intervallo di tempo massimo per cui

sospendere l'esecuzione. Se il valore del parametro è NULL, l'intervallo di tempo si considera infinito.

Nel momento in cui l'esecuzione del programma riprenderà, il valore di ritorno della chiamata a *select* sarà il numero di descrittori di file che ne hanno determinato la terminazione, oppure zero se terminata per via del parametro *timeout*. Inoltre, i valori dei tre array dopo la chiamata saranno diversi: solo i bit relativi ai descrittori di file di interesse saranno impostati a 1. Questo vuol dire anche che è necessario mantenere sempre due copie di tutte le variabili: una su cui operare e una da usare come parametro.

Il diagramma congiunto che descrive l'interazione tra server concorrente e client è riportato nella Figura 11.24.

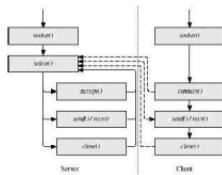


Figura 11.24 Diagramma delle operazioni congiunte di client e server concorrente.

È possibile fare uso della system call *select* all'interno del ciclo di gestione dei client per trasformare un server iterativo in un server concorrente. Nell'esempio che si discuterà ora si è scelto di implementare un server per il servizio echo secondo il modello concorrente a partire dalla Figura 11.8.

Per prima cosa, visibile nella Figura 11.25, occorre definire una coppia di variabili che saranno usate per gestire l'array di bit in lettura (l'unico a cui si è interessati) e abilitare solo l'elemento relativo alla socket utilizzata per la ricezione di nuovi client (righe 57 e 58); in questo modo la *select* potrà terminare anche in corrispondenza di una *connect*.

```
51 struct sockaddr_in chiamante;
52 unsigned int dim_chiamante = sizeof(chiamante);
53
54 fd_set lettura, parametri;
55
56 FD_ZERO(&parametri);
57 FD_SET(fd, &lettura);
58 ... ciclo di gestione dei client.
```

Figura 11.25 Gestione delle variabili necessarie alla system call *select*.

All'interno del ciclo di gestione dei client la prima operazione da fare, come riportato nella Figura 11.26, è duplicare il contenuto dell'array di bit nella variabile che verrà usata come parametro (riga 62) ed effettuare la chiamata alla system call *select* (riga 63).

```
60 ... ciclo di gestione dei client
61 while (1)
62     FD_SET(fd, &parametri);
63     if (select(FOPEN_MAX, &parametri, NULL, NULL, NULL) < 0)
64         perror("select");
65 }
```

Figura 11.26 Chiamata della system call *select*.

Per comodità, è possibile usare la costante `FOPEN_MAX` come primo parametro; questa costante è definita a livello di sistema operativo e assume il valore del numero massimo di descrittori di file allocabili da un singolo processo.

Nel momento in cui la *select* termina occorre discriminare l'evento; per prima cosa si può controllare se sia il caso di un nuovo client in arrivo, come nella Figura 11.27. In caso positivo, rilevato da *FD_ISSET* alla riga 68, viene effettuata una chiamata alla system call *accept* (riga 70) e la socket per lo scambio dei dati viene abilitata all'interno dell'array in lettura; da questo momento in poi anche la socket appena creata potrà causare la terminazione della *select*.

L'operazione alla riga 72 può sembrare un po' oscura, tuttavia ha un significato ben preciso. A fronte della connessione di un nuovo client non è possibile escludere che ci siano anche dati disponibili tramite altre socket, per cui è necessario effettuare un ciclo su tutti i descrittori di file per controllare quali altri bit risultano abilitati all'interno dell'array, come proposto nella Figura 11.28. Siccome non ha senso erogare il servizio sulla socket abilitata per la ricezione, è opportuno disabilitare forzatamente il relativo bit; in alternativa si sarebbe potuto mettere un controllo alla riga 76 escludendo in caso in cui *fd* assume il valore *s*.

```

70     if (FD_ISSET(fd, &readset)) {
71         /* C'è qualcosa da leggere */
72         socket_desc = accept(fd, (struct sockaddr *)NULL,
73                               &socket_desc_size);
74         FD_SET(socket_desc, &readset);
    }

```

Figura 11.27 Controllo di un nuovo client in arrivo.

```

74     for (fd=0; fd < PUFFIN_MAX_FD + 1; fd++) {
75         if (FD_ISSET(fd, &readset)) {
76             /* C'è qualcosa da leggere */
77             /* E' un'applicazione_servizio(fd); */
78             if (fd == socket_desc) {
79                 /* Chiama la funzione del client */
80                 /* Se tutto va bene, si cancella dal client */
81                 close(fd);
82             }
83             else
84                 /* E' un'altra socket */
85                 FD_CLR(fd, &readset);
86         }
87     }
88 }

```

Figura 11.28 Gestione dei descrittori di file con dati in arrivo.

Se un certo descrittore di file viene trovato attivo, alla riga 76 viene richiamata la funzione di erogazione del servizio passando come parametro la socket (riga 78). In questo caso si è ipotizzato che la funzione di erogazione gestisca un solo messaggio; il codice è quasi identico a quello presentato nella Figura 11.22, ma con di socket connesse. Se il numero di byte letti dal canale è zero o meno, un client non è più collegato e la relativa socket deve essere chiusa e rimossa dall'array.

Esercizio 11.12



In realtà, la *select* non è strettamente legata al concetto di socket ma può essere utilizzata per mettere sotto osservazione un generico canale di I/O.

Si implementi un client per il servizio echo che non faccia uso di alternanza stretta: il client deve essere in grado di attendere dati parallelamente dalla socket con cui è collegato al server e dall'utente. Si ricordi, a tal proposito, che il file descriptor associato all'input dell'utente è il numero zero.

11.6 Implementazione di un server multiprocesso/multithread

Il linguaggio C mette a disposizione un meccanismo molto semplice per la gestione dei processi: la system call *fork*.



Figura 11.29 Prototipo della system call *fork*.

Al momento della chiamata di *fork* il sistema operativo creerà una copia identica del processo chiamante e che potrà evolversi in maniera indipendente. Il processo generato (*figlio*) si distinguerà dal processo principale (*padre*) per il valore restituito dalla system call, che per il figlio sarà zero e per il padre il numero con cui sistema operativo identifica il processo appena creato.

Le variabili dei due processi hanno gli stessi valori ma *non sono le stesse*: dopo l'operazione di *fork* le modifiche effettuate dal figlio non saranno visibili al padre e viceversa. I canali di I/O, invece, essendo definiti a livello di sistema operativo sono effettivamente condivisi; in ognuno



dei due processi è sempre consigliabile chiudere tutti i canali non necessari e che verranno utilizzati dall’altro.

La parte critica della gestione dei processi, comunque, non è la loro creazione ma la terminazione: nel momento in cui un processo figlio termina il sistema operativo lo pone in stato di attesa (processo *zombie*) e notifica la terminazione al padre. Fintanto che il processo padre non si occuperà di gestire il segnale proveniente dal sistema operativo il processo figlio rimarrà in attesa e le risorse non saranno rese nuovamente disponibili.

Per poter gestire la morte del figlio il padre deve fare uso di due system call: *signal* e *wait*. La prima richiede che l’esecuzione del padre sia interrotta in maniera asincrona all’arrivo di un certo segnale per eseguire una data funzione, la seconda si occupa di gestire i messaggi riguardante la morte di processi figli. I rispettivi prototipi sono presentati nella Figura 11.30.

```
#include <sys/types.h>
void * signal(int signale, void (*funzione)(int));
#include <sys/wait.h>
int waitpid * statut;
```

Figura 11.30 Prototipi delle system call *signal* e *wait*.

A prima vista il prototipo di *signal* può mettere in difficoltà; si tratta di una funzione che accetta due parametri: un intero e un

puntatore a una funzione che vuole a sua volta un parametro intero e non restituisce nessun valore. Per semplificare le cose è possibile definire il seguente tipo *manager*:

```
typedef void (* manager) (int);
```

per cui il prototipo diventa:

```
int signal(int segnale, manager funzione);
```

All'arrivo del segnale indicato come primo parametro il programma verrà interrotto ed eseguita la funzione specificata come secondo parametro; al termine di tale funzione l'esecuzione riprenderà dal punto in cui era stata interrotta.

L'effetto della chiamata di *signal* è valido solo per un segnale: deve essere ripetuta ogni volta per garantire la ricezione del segnale successivo.

La funzione che si occupa di gestire la morte dei processi figli dovrà fare uso al suo interno di *wait* per gestire i segnali e permettere al sistema operativo di liberare le risorse.

In realtà, nei sistemi operativi moderni è possibile utilizzare il linguaggio C per gestire anche thread, tuttavia esistono varie implementazioni e non tutte le piattaforme li supportano secondo le stesse modalità; per questo motivo all'interno del testo si è scelto di fare riferimento solo a processi.

Il diagramma congiunto che descrive l'interazione tra server multiprocesso e client è rappresentato nella Figura 11.31.

Il codice, anche in questo caso, può essere ricavato a partire da quello di un server iterativo. Come si può notare nella Figura 11.32, al termine della chiamata di *accept* (riga 64) invocata la system call *fork* (riga 67); per quanto discusso prima, solo il processo figlio esegue il codice alle righe 69 e 70, erogando il servizio e poi terminando. Il padre, invece, chiude la socket per lo scambio dei dati perché gli è inutile e torna in attesa del prossimo client.

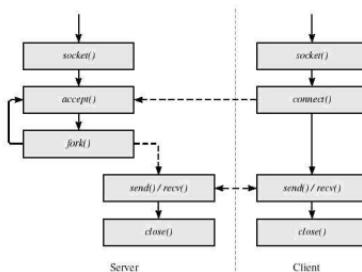


Figura 11.31 Diagramma delle operazioni congiunte di client e server multiprocesso.

```
1 #include <signal.h>
2 #include <sys/types.h>
3 #include <sys/conf.h>
4
5 typedef void (*manager) (int);
6
7 int erogazione_servizio(int);
8 void gestione_processo();
9
10 int main(int argc, char ** argv, char ** envp) {
11     signal(SIGCHLD, (manager) gestione_processo);
12     ...
13
14     /* ciclo di gestione del client */
15     while (1) {
16         /* ricezione connessione */
17         socket_dati = accept(4, chiamante, &dim_chiamante);
18
19         /* creazione processo */
20         if (fork() == 0) {
21             /* esecuzione del figlio */
22             /* erogazione servizio(dati);
23             erogazione_servizio(socket_dati);
24             return 0;
25         }
26
27         /* chiudere canale dati */
28         close(socket_dati);
29     }
30 }
```

Figura 11.32 Codice di un server multiprocesso.

La funzione di erogazione del servizio è identica a quella vista nel caso di un server iterativo e viene presentata nella Figura 11.19, mentre il codice per la gestione della morte del processo figlio è rappresentato in Figura 11.33. In particolare, si faccia caso alla chiamata di *signal* alla riga 95 identica a quella della riga 17, dove la costante **SIGCHLD** identifica l'evento della terminazione di un processo figlio. Sempre all'interno della funzione *gestione processo* è possibile osservare che la system call *wait* è stata collocata all'interno di un ciclo *while*; questo è necessario perché se più processi figli dovessero terminare in rapida sequenza il sistema operativo genererebbe un solo segnale, per cui la chiamata deve essere iterata fintanto che esistono segnali in attesa di essere gestiti.

```
1 void gestione_processo(int segnale) {
2     int status;
3     while(wait(&status) == 0) {
4         signal(SIGCHLD, manager gestione_processo);
5     }
6 }
```

Figura 11.33 Funzione di gestione della morte dei processi figli.



Esercizio 11.13

Si modifichi il programma proposto nelle Figure 11.32 e 11.33 in maniera tale da stampare le informazioni relative ai processi creati e distrutti e da non permettere mai l'esistenza di più di tre processi figli.

Volendo, sarebbe anche possibile per il processo figlio segnalare una condizione di errore, restituendo alla riga 70 un valore negativo che il padre sarebbe in grado di intercettare analizzando il valore della variabile *status* con la macro **WEXITSTATUS**.



Esercizio 11.14

Si modifichi il programma proposto nelle Figure 11.32 e 11.33 in maniera tale da permettere al processo padre di accorgersi di eventuali condizioni di errore rilevate dal figlio.

Capitolo 12

Accesso alle risorse web

L'avvento del linguaggio C risale a un periodo storico anteriore a quello del Web di almeno 20 anni, per cui le sue librerie di base non sono equipaggiate con strumenti per semplificare la fruizione di questo tipo di servizio. All'interno di questo capitolo si discuteranno gli accorgimenti più importanti da adottare per poter consultare un web server senza l'ausilio di librerie aggiuntive.

12.1 Servizio web

La fruizione del servizio avviene tramite una singola socket; viene richiesto l'accesso a una data risorsa e il server invia in risposta il contenuto a essa associato; tale contenuto potrebbe essere di natura testuale ma anche binario. Il client deve gestire la socket per la comunicazione e più buffer per ospitare i dati

in arrivo dal server; tali dati potrebbero essere utilizzati internamente al client stesso o forniti in input a un'altra applicazione.

Un client per il servizio web non implementa necessariamente un'interfaccia grafica e non sempre ha come obiettivo la navigazione dei contenuti, tant'è vero che esistono web browser puramente testuali, come lynx [65] e w3m [66], come pure client per il solo ascolto di musica via Web, come winamp [67].



12.2 Protocollo HTTP

I messaggi gestiti dal protocollo HTTP sono composti da stringhe di lunghezza variabile che terminano con “\r\n”. Purtroppo, ne consegue che non è possibile fare ipotesi sul numero di byte che saranno letti dalla socket di volta in volta.

Sono quindi disponibili due approcci: utilizzare un buffer su cui fare manipolazioni oppure leggere e scrivere un byte alla volta sul canale. Tutti e due gli approcci presentano pro e contro: l'uso di un buffer può essere problematico nel caso in cui un messaggio non arrivi completamente (o il buffer non sia in grado di contenerlo), mentre trasferire un byte alla volta penalizza eccessivamente le prestazioni.

Una soluzione ragionevole al problema appena esposto è quella di dividere la comunicazione in due fasi: la prima, quella riguardante linea iniziale e header, perché le dimensioni abbastanza contenute e può essere gestita un byte alla volta o con un buffer sovradimensionato; la seconda fase, quella relativa al contenuto, può essere gestita con un buffer e facendo molteplici accessi al canale, in quanto non necessita di manipolazioni.

12.2.1 Messaggio di richiesta

Per comporre tramite un buffer un messaggio di richiesta è possibile fare uso di *sprintf* per scrivere una successione di stringhe in una zona di memoria, come nell'esempio riportato nella Figura 12.1.

```
10 int trascinare_servizio(int socket) {
11
12     int numBuffer = 2048;
13     char *buffer = (char*)malloc(numBuffer);
14     char *c = buffer;
15
16     char *metodo = "GET/HTTP/1.1";
17     char *uri = "/index.html";
18     char *versione = "HTTP/1.1\r\n";
19
20     // costruzione messaggio di richiesta
21     c += sprintf(c, "%s %s %s", metodo, uri, versione);
22     c += sprintf(c, "Host: www.google.com\r\n");
23     c += sprintf(c, "User-Agent: Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/48.0.2564.103 Safari/537.36\r\n");
24
25     // invio messaggio di richiesta
26     send(socket, buffer, c - buffer, 0);
}
```

Figura 12.1 Invio di un messaggio di richiesta tramite un buffer.

Nell'esempio si è scelto di definire una serie di variabili per memorizzare gli elementi con cui comporre l'header (righe dalla 16 alla 18) e una variabile *c*, di tipo puntatore a carattere, per scorrere il buffer (riga 14). Inizialmente *c* coincide con l'inizio del buffer e all'aggiunta di ogni riga viene incrementata

del numero di byte utilizzati da *sprintf*, in questo modo si è sicuri che le stringhe siano collocate nel buffer in maniera sequenziale (righe dalla 21 alla 23). La variabile *c* identifica di volta in volta il primo byte non ancora utilizzato all'interno del buffer. Dal buffer, vengono poi inviati sulla socket (riga 26) solo i byte compresi tra il primo e quello puntato da *c*, tramite l'espressione “*c - buffer*”.

Se invece si intende procedere mandando separatamente ogni stringa è possibile operare come indicato nella Figura 12.2.

```
19 int trasmette_messaggio(int socket) {
20     int da_inviare = 2040;
21     char buffer[2040];
22
23     (char *)buffer[0] = '\0';
24
25     buffer[0] = '\0';
26
27     da_inviare = strlen(da_inviare);
28
29     sendto(socket, da_inviare, strlen(da_inviare), 0);
30     sendto(socket, da_inviare, strlen(da_inviare), 0);
31     sendto(socket, da_inviare, strlen(da_inviare), 0);
32     sendto(socket, da_inviare, strlen(da_inviare), 0);
33
34     return da_inviare;
35 }
```

Figura 12.2 Invio di un messaggio di richiesta come una sequenza di stringhe.

Diversamente dal caso precedente sono stati usati una serie di literal; ogni stringa componente il messaggio è stata assegnata alla variabile *da_inviare* e poi subito trasmessa al server. La definizione di *da_inviare* è necessaria per poter calcolare facilmente il numero di byte da inviare tramite *strlen*, inoltre non si è usato un buffer all'interno del quale collocare ogni singola riga con *sprintf*, perché in quel caso non ci sarebbe stato un vero vantaggio rispetto al metodo precedente.

12.2.2 Messaggio di risposta

La ricezione di un messaggio di risposta tramite un buffer risulta a volte scomoda da attuare: non ci sono garanzie che all'interno del buffer non sia presente anche la prima parte del contenuto in arrivo; tuttavia, una volta isolata e interpretata la parte relativa all'intestazione, si può procedere avendo cura di non sovrascrivere idati già ricevuti. Un esempio è proposto nella Figura 12.3.

Nell'esempio il buffer viene riempito inizialmente con i dati in arrivo dalla rete (riga 29) e successivamente viene aggiunto un terminatore di stringa dopo i byte letti (riga 30). I vari elementi vengono separati tramite la funzione *strtok* che, data una stringa, ne isola via via alcuni segmenti (*token*) delimitati dai caratteri usati come secondo parametro; se si considera il buffer come una stringa, allora i vari elementi possono essere facilmente isolati usando i giusti delimitatori.

```
28 // ricezione della risposta
29 int *recv(socket, buffer, size_buffer, 0);
30 if(recv(&socket, &buffer, size_buffer, 0) < 0)
31 {
32     char *tokens;
33
34     // interpretazione della linea iniziale
35     tokens = strtok(buffer, "\r\n");
36     print("Richiesta HTTP del server Aaaa", tokens);
37     tokens = strtok(NULL, "\r\n");
38     print("Codice di risposta: %s", tokens);
39     token = strtok(tokens, " ");
40     print("Commento: %s", token);
41
42     print("Header(s):\n");
43     char *value;
44
45     // interpretazione dell'header
46     while ((tokens = strtok(tokens, "\r\n")) != NULL) {
47         tokens = strtok(NULL, "\r\n");
48         value = tokens;
49         tokens = strtok(tokens, " ");
50         tokens = strtok(tokens, " ");
51         tokens = strtok(tokens, " ");
52         tokens = strtok(tokens, " ");
53         print("%s - %s", tokens, value);
54     }
55 }
```

Figura 12.3 Ricezione di un messaggio di risposta tramite un buffer.

La stringa analizzata non è solo la linea iniziale ma comprende il messaggio di risposta nella sua totalità: linea iniziale, header e potenzialmente parte del contenuto; sono compresi anche CR e LF.



La versione di HTTP del server e il codice di risposta sono entrambi seguiti da uno spazio, per cui alle righe 34 e 36 si estraggono i primi due token usando uno spazio come delimitatore; il commento è terminato da un ritorno a capo, per cui alla riga 38 si usa invece il carattere “\n”; il delimitatore non compare nel token, mail carattere “\r” sì, quindi è bene eliminarlo sostituendolo con un terminatore di stringa (riga 39). I token successivi sono i vari elementi dell’header e vengono estratti in sequenza, una riga dopo l’altra, fino alla ricezione di una stringa vuota; infatti, la condizione di uscita dal ciclo è che il token inizi con il carattere “\r”. All’interno del ciclo ogni stringa relativa a un’opzione viene spezzata in due parti in corrispondenza della stringa “:”, la prima è assegnata alla variabile *campo* e la seconda a *valore*.

Se invece l’intenzione è quella di gestire il messaggio di risposta leggendo un byte alla volta dalla socket, la cosa più agevole è definire una funzione apposita per la ricezione, come quella proposta nella Figura 12.4.

```
int lettura_con_delimitatore(int socket, char *buffer, char delimitatore)
{
    char *posizione = buffer;
    while (recv(socket, posizione, 1, 0) >= 0) {
        if (*posizione == delimitatore) break;
        posizione++;
    }
    *posizione = '\0';
    return posizione - buffer;
}
```

Figura 12.4 Funzione per la ricezione di una sequenza di byte fino all'arrivo di un dato carattere.

Esercizio 12.1



Si estenda la funzione proposta nella Figura 12.4 in modo da usare una stringa come guardia e non un singolo carattere e si modifichi il client in maniera tale da poterla usare con i delimitatori “ ” (carattere di spazio) e “\r\n”.

La funzione *lettura_con_delimitatore* legge un byte alla volta da una socket e deposita il risultato in un buffer fino a che non viene letto il carattere specificato come terzo parametro o il canale viene chiuso dalla controparte. Per coerenza con la funzione *strtok* il carattere di delimitazione non è incluso nel buffer; il valore restituito dalla funzione è il numero di byte letti prima del delimitatore. Nella Figura 12.5 viene proposto codice che, facendo uso della funzione appena descritta, estrae un elemento alla volta dalla socket.

```

28 //interpretazione della linea iniziale
29 lettura_m,delimitatore+<socket,buffer>,1,12
30 lettura_m,delimitatore+<socket,buffer>,13,16
31 lettura_m,delimitatore+<socket,buffer>,17,19
32 lettura_m,delimitatore+<socket,buffer>,20,22
33 lettura_m,delimitatore+<socket,buffer>,23,25
34 lettura_m,delimitatore+<socket,buffer>,26,28
35 print("Commerce View: Buffer");
36
37 print("Response:");
38 char opinion[10];
39 int campi;
40 int valori;
41
42 //interpretazione dell'header
43 while((campi = opinion[0]) > 0) {
44     opinion[0] = '\0';
45     campi = opinion;
46     opinion = opinion + 1;
47     valori = 2 * "0";
48     printf("%s - %s", campi, valori);
49 }

```

Figura 12.5 Ricezione di un messaggio di risposta come una sequenza di byte.

Osservando con attenzione si può notare che la struttura del codice è essenzialmente identica a quella della Figura 12.3 se non per il fatto che i token vengono estratti dalla socket anziché da un buffer in memoria.

12.3 Accesso a una risorsa

Per poter accedere a una risorsa sarà a questo punto sufficiente assegnare i giusti valori agli elementi della linea iniziale e combinare tra loro il codice per l'invio di una richiesta (Figura 12.1 o 12.2) con quello relativo alla gestione della risposta (Figura 12.3 o 12.5). Negli esempi che seguiranno si farà uso di un buffer per inviare la richiesta e la risposta sarà interpretata un carattere alla volta; si farà cioè un uso del codice riportato nelle Figure 12.1, 12.4 e 12.5.

12.3.1 HEAD

Per usare il metodo HEAD è sufficiente il codice scritto fin qui, una possibile combinazione di valori per comporre il messaggio di richiesta può essere quella proposta nella Figura 12.6.

```
16 char * method = "HEAD";
17 char * uri = "HTTP://www.lila.it/";
18 char * version = "HTTP/1.1";
```

Figura 12.6 Possibile impostazione per un test del metodo HEAD.

Si sta implicitamente supponendo di aver aperto il canale di comunicazione con il nodo www.lila.it sulla porta 80.



Eseguendo il programma così ottenuto si otterrà un output simile a quello che segue; per chiarezza, sono state riportate solo le parti più significative dell'header.

```
Versione HTTP del server: HTTP/1.1
Codice di risposta: 200
Commento: OK
Header:
Server -> Microsoft-IIS/6.0
Last-Modified -> Fri, 10 Oct 2008 12:45:53 GMT
Content-Length -> 22559
```

Content-Type -> text/html

Connection -> close

Esercizio 12.2



Si modifichi il client in maniera tale da utilizzare l'opzione *Accept* per richiedere solo contenuti di tipo *text/html* e se ne verifichi il buon funzionamento.

12.3.2 GET

Per inviare una richiesta di tipo GET occorre innanzitutto cambiare il metodo di accesso alla riga 16 e successivamente estendere il codice per effettuare la ricezione del contenuto. Le variazioni necessarie sono proposte nella Figura 12.7.

```
7 #include <sys/types.h>
8 #include <sys/socket.h>
9 #include <sys/conf.h>
10 #include <sys/malloc.h>
11 #include <sys/param.h>
12 #include <sys/kernel.h>
13 #include <sys/time.h>
14 #include <sys/conf.h>
15 #include <sys/socket.h>
16 #include <sys/malloc.h>
17 #include <sys/conf.h>
18 #include <sys/time.h>
19 #include <sys/socket.h>
20 #include <sys/malloc.h>
21 #include <sys/conf.h>
22 #include <sys/time.h>
23 #include <sys/socket.h>
24 #include <sys/malloc.h>
25 #include <sys/conf.h>
26 #include <sys/time.h>
27 #include <sys/socket.h>
28 #include <sys/malloc.h>
29 #include <sys/conf.h>
30 #include <sys/time.h>
31 #include <sys/socket.h>
32 #include <sys/malloc.h>
33 #include <sys/conf.h>
34 #include <sys/time.h>
35 #include <sys/socket.h>
36 #include <sys/malloc.h>
37 #include <sys/conf.h>
38 #include <sys/time.h>
39 #include <sys/socket.h>
40 #include <sys/malloc.h>
41 #include <sys/conf.h>
42 #include <sys/time.h>
43 #include <sys/socket.h>
44 #include <sys/malloc.h>
45 #include <sys/conf.h>
46 #include <sys/time.h>
47 #include <sys/socket.h>
48 #include <sys/malloc.h>
49 #include <sys/conf.h>
50 #include <sys/time.h>
51 #include <sys/socket.h>
52 #include <sys/malloc.h>
53 #include <sys/conf.h>
54 printf("Content-type: %s\r\n", type);
55 lseek(fd, 0, SEEK_SET);
56 /* lettura del contenuto */
57 do {
58     if (read(fd, buffer, minbuffer) == -1)
59         error("Error reading from socket");
60     else {
61         minbuffer = 0;
62         if (read(fd, buffer, minbuffer) == -1)
63             error("Error reading from socket");
64         minbuffer = 0;
65     }
66 } while (minbuffer > 0);
67 printf("\r\n");
68 }
```

Figura 12.7 Modifiche al client per la ricezione del contenuto.

Alla riga 15 è stata introdotta una nuova variabile *dimensione* in cui memorizzare il quantitativo di byte da leggere dopo l'header; tale variabile viene assegnata alla riga 49, quando il campo dell'opzione assume il valore *Content-Length*. Si noti l'uso di *strcasecmp* anziché di *strcmp*: anche se nella stragrande maggioranza dei casi le due chiamate restituiscono lo stesso risultato, le specifiche non vincolano strettamente l'uso di maiuscole e minuscole, per cui è consigliabile essere il più elastic possibile. Il ciclo nelle righe dalla 58 alla 64 legge il contenuto dalla socket e lo visualizza; la condizione di permanenza all'interno del ciclo è che ci siano ancora byte da ricevere, infatti, la variabile *dimensione* viene ogni volta decrementata del numero di byte letti (riga 63) e il ciclo termina quando questa arriva al valore 0. Il quantitativo di byte (variabile *da_leggere*) va calcolato di volta in volta alla riga 59: deve essere il quantitativo minimo tra la dimensione del buffer e il numero di byte ancora da ricevere; per agevolare questa operazione è stata definita un'apposita macro alla riga 7. Come ultima nota, l'uso di “*dim_buffer - 1*” serve a garantire che, anche nel caso in cui *recv* legga tutti i dati, il terminatore di stringa rimanga all'interno del buffer.

Eseguendo il programma così composto si ottiene in output il testo della pagina principale di <http://www.lila.it/>.

```
Versione HTTP del server: HTTP/1.1
Codice di risposta: 200
Commento: OK
Header:
Content-Length -> 22559
Contenuto:
```

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
...
```

Se però il server non dovesse fornire l'opzione *Content-Length* nell'header, la situazione si complicherebbe leggermente: il termine del contenuto corrisponderebbe alla chiusura del canale. Una soluzione facilmente praticabile è quella di assegnare inizialmente a *dimensione* il massimo valore possibile (`INT32_MAX` o $2^{32} - 1$) e di impostarlo diversamente solo quando specificato nell'header, in questo modo il client si predispone per leggere una sequenza virtualmente infinita di byte. A questo punto però il problema diventa la terminazione del ciclo di lettura, che deve dipendere, oltre che dal numero di byte letti anche dal fatto che il canale sia stato chiuso dal server. La Figura 12.8 mostra il codice relativo ai cambiamenti appena descritti.

```
29 long dimension = INT32_MAX;
30 ...
31 if (n_left > dimension, dim_buffer - 1);
32 E = read(socket, buffer, dim_buffer - 1, 0);
33 if (E < 0)
34     buffer[E] = '\0';
35     print("%s", buffer);
36     dimension = 0;
37 }
38
39 } while (dimension > 0 && E > 0);
40 print("%s", buffer);
```

Figura 12.8 Ricezione del contenuto in assenza di *Content-Length*.

Esercizio 12.3



Si faccia uso del metodo GET per ottenere il contenuto relativo a una immagine e anziché visualizzarlo a video lo si memorizzi sul disco. Se ne verifichi il buon funzionamento visualizzando l'immagine.

12.4 Contenuti statici e dinamici

L'accesso a contenuti dinamici con il metodo GET è piuttosto banale; basta aggiungere i vari parametri in coda alla variabile *url* alla riga 17, come descritto nella Figura 12.9.

```
16 char *method = "GET";
17 char *url = "http://www.google.it/search?hl=it&q=client+server";
18 char *page = "index.html";
19
```

Figura 12.9 Impostazioni per la consultazione di una pagina dinamica tramite il metodo GET.

12.4.1 POST

Per poter fare un esempio del metodo POST non è possibile utilizzare nuovamente Google, in quanto il server accetta solo richieste con GET. Yahoo, invece, può essere usato anche con il metodo POST; il codice per effettuare l'interrogazione è proposto nella Figura 12.10.

```
14 char *method = "POST";
15 char *url = "http://it.search.yahoo.com/web";
16 char *username = " ";
17 char *password = " ";
18
19 // costituzione header: si riporta
20 // l'header di default, ma nel verificando
21 // c e' possibile inserire altri header, se necessario
22 c = apnprintf("User-Agent: curl/7.19.0\r\n");
23 c = apnprintf("Content-Type: application/x-www-form-urlencoded\r\n");
24 c = apnprintf("Content-Length: %d\r\n", strlen(query));
25 c = apnprintf("Connection: close\r\n");
26
27 // invio messaggio di richiesta
28 // per la richiesta di ricerca, si
29 // spediscono, contenuto, stringa di contenuto, url
```

Figura 12.10 Richiesta con metodo POST al motore di ricerca di Yahoo.

Rispetto alla situazione precedente è stata inserita una variabile contenuto per memorizzare i parametri e sono stati inserite le due opzioni *Content-Type* e *Content-Length* nell'header (righe 24 e 25).

Nell'esempio si è scelto di inviare separatamente intestazione e *contenuto* (righe 29 e 30). Nel caso specifico si poteva pensare di inserire il contenuto nel buffer ed effettuare una sola spedizione; tuttavia, è buona pratica separare le due operazioni perché in generale non è detto che il contenuto possa essere ospitato nel buffer.



Esercizio 12.4



Verificare, e possibilmente anche gestire, l'errore restituito da Google nel momento in cui si usa il metodo POST.

12.5 Connessione persistente e non persistente

Per richiedere una connessione persistente è sufficiente inserire la stringa “*Connection: Keep-Alive*” nell’header. Di solito, questo può essere utile se la funzione per la fruizione del servizio è inserita all’interno di un ciclo e si vuole accedere a una serie di risorse senza dover ogni volta istituire un nuovo canale di comunicazione. Il codice presentato negli esempi fin qui, avendo cablato all’interno della funzione di fruizione l’URL a cui accedere, si presta poco a questo tipo di utilizzo; una soluzione più completa sarebbe quella di fornire URL e nome dell’host come parametri della funzione, coinvolgendo la porzione di codice che si occupa di gestire la connessione, come nell’esempio riportato nella Figura 12.11 dove la funzione *ricava_url* è esemplificativa di una porzione di codice che genera una sequenza di riferimenti a risorse da contattare.

```
function Avvi
while(1):
    var url = "http://www.google.it";
    if (fruizione.url != url) {
        fruizione.url = url;
        fruizione.timeout = 5000;
        fruizione.onreadystatechange = funzione;
        fruizione.open();
        fruizione.send();
    }
}
```

Figura 12.11 Esempio di ciclo in grado di sfruttare una connessione persistente.

12.6 Web proxy

La funzione di fruizione del servizio non subisce nessun tipo di variazione quando si usa un proxy: l'unico accorgimento necessario è quello di collegare la socket al server proxy piuttosto che al server dove è localizzata la risorsa, ma questo è solo un problema di indirizzo e non coinvolge la fruizione del servizio. L'output ottenuto in questo caso è identico ai precedenti a meno delle opzioni inserite dal proxy.

12.7 Cookie

La gestione dei cookie non complica la struttura del codice per quanto riguarda la gestione dei messaggi, a meno, ovviamente, di inserire le opportune opzioni nell'header.

La vera difficoltà da affrontare è la memorizzazione delle informazioni su un dispositivo permanente: la struttura con cui i cookie vengono ricevuti si presta molto poco a recuperare velocemente quelli relativi a un certo server, soprattutto a fronte di numeri significativi (centinaia). I programmati fanno generalmente uso di librerie esterne per memorizzare in maniera strutturata le informazioni e recuperarle velocemente. Una di queste librerie, di libera distribuzione, è *Berkeley DB* di Oracle [68].

12.8 Interrogazione di motori di ricerca

L'interrogazione di un motore di ricerca, come già visto, si riduce in realtà a simulare la compilazione di un form e raccogliere il risultato per estrarre dei dati. La vera difficoltà è invece rappresentata dall'estrazione delle informazioni: non essendoci una struttura prefissata non è possibile andare *a colpo sicuro*.

In realtà i motori di ricerca tendono a essere piuttosto fiscali sull'header del messaggi di richiesta, questo per verificare l'identità del client ed evitare attacchi di sicurezza o di essere usati da altri motori di ricerca senza ricevere credito. Potrebbe capitare che, nonostante il codice del risultato sia positivo, la pagina non contenga i dati attesi. Ciò può dipendere dalla conformazione dell'header della richiesta o dalla mancanza di cookie; purtroppo, è piuttosto difficile reperire informazioni ufficiali su questo argomento e i vincoli cambiano da un sito all'altro.



L'unico approccio in grado di funzionare quasi sempre è quello di identificare alcune stringhe sempre presenti all'interno della pagine, come titoli o tag in posizioni fisse, e da lì estrarre l'informazione che interessa isolandola con delimitatori o

adottando tecniche simili a *strtok*. Ad esempio, Google fa precedere a tutti i link relativi ai risultati la stringa “`<h3 class=r><a href=""`”, per cui il codice proposto nella Figura 12.12 può essere usato per estrarli dal testo della pagina.

Nel codice proposto si usa la variabile *c* per scorrere il buffer e localizzare i vari preamboli fino a che ci sono preamboli successivi alla posizione corrente di *c* (riga 7), il link inizia alla fine del preambolo (riga 8) e la nuova posizione di *c* sarà calcolata andando a rintracciare i doppi apici che identificano la fine del link (riga 9). Nelle righe 10, 11 si pone un terminatore di stringa in corrispondenza dei doppi apici per delimitare la stringa del risultato e si porta avanti di un byte la posizione identificata da *c* per poter iterare il ciclo.

```
1 void extrai_link(char * buffer) {
2     char * preambolo = "h3 class=r><a href=""";
3     char * c = buffer;
4     char * llink;
5
6     while (*preambolo >= 'A' & *preambolo <= 'Z') {
7         if (*c == '<' & *(c + 1) == '>' & *(c + 2) == 'h' & *(c + 3) == '3' & *(c + 4) == ' ' & *(c + 5) == 'c' & *(c + 6) == 'l' & *(c + 7) == 'a' & *(c + 8) == 's' & *(c + 9) == 's' & *(c + 10) == 's' & *(c + 11) == 's' & *(c + 12) == 's' & *(c + 13) == 's' & *(c + 14) == 's' & *(c + 15) == 's' & *(c + 16) == 's' & *(c + 17) == 's' & *(c + 18) == 's' & *(c + 19) == 's' & *(c + 20) == 's' & *(c + 21) == 's' & *(c + 22) == 's' & *(c + 23) == 's' & *(c + 24) == 's') {
8         llink = c + 1;
9         *llink = '"';
10        llink++;
11        *llink = '"';
12        printf("%s\n", llink);
13    }
14 }
```

Figura 12.12 Estrazione dei link relativi ai risultati da una pagina di Google.

Se Google dovesse cambiare la struttura delle sue pagine questo codice non funzionerà più.



Esercizio 12.5



I risultati di Google, se non diversamente specificato, vengono presentati a gruppi di 10. Si estenda il client sviluppato fin qui in maniera tale da effettuare una serie di richieste successive e ottenere tutte le pagine. Si utilizzino sia connessioni non persistenti che una connessione persistente.

Esercizio 12.6



Si inserisca nel codice la funzione proposta per l'estrazione dei link e la si usi in sequenza su tutte le pagine restituite da Google.

Esercizio 12.7



Si implementi un client in grado di collegarsi al sito di Wikipedia e di effettuare la ricerca di un termine fornito dall'utente; l'output del programma deve essere il contenuto della pagina HTML. Si faccia attenzione al fatto

che, in prima battuta, Wikipedia risponde con un messaggio che ridirige a un altro URL.

12.9 RSS

Il linguaggio C non dispone in maniera nativa di funzioni per il trattamento di XML, i programmatore fanno spesso uso di librerie esterne come *libxml* [69], che però fanno parte di progetti più ampi.

Nell'interpretazione di feed RSS si ripresentano quindi in forma ridotta gli stessi problemi discussi nel caso di pagine provenienti da motori di ricerca; fortunatamente, i tag possono essere usati come delimitatori e si presentano in una sequenza ben precisa, per cui, anche adottando un approccio molto semplice come individuare sottostringhe, porta a risultati accettabili.

Esercizio 12.8



Si selezioni un quotidiano dotato di feed RSS e si scriva un client in grado di estrarre i titoli delle singole notizie dal feed.

Bibliografia

- [1] The apache web server project.
<http://www.apache.org/>.
- [2] International Organization for Standardization (ISO).
<http://www.iso.org/>.
- [3] ISO Joint Technical Committee 1. “*Open Systems Interconnection (OSI) – Basic Reference Model: The Basic Model*”. Standard 35.100.01, ISO, giugno 2000.
- [4] Theodore J. Socolofsky e Claudia J. Kale. “*A TCP/IP Tutorial*”. RFC 1180, IETF, gennaio 1991.
<http://www.ietf.org/rfc/rfc1180.txt>.
- [5] Jon Postel. “*Internet Protocol, DARPA Internet Program Protocol Specification*”. RFC 791, IETF, settembre 1981. <http://www.ietf.org/rfc/rfc0791.txt>.
- [6] Stephen E. Deering e Robert M. Hinden. “*Internet Protocol, Version 6 (IPv6) Specification*”. RFC 2460, IETF, dicembre 1998. <http://www.ietf.org/rfc/rfc2460.txt>.
- [7] Jon Postel. “*Transmission Control Protocol*”. RFC793, IETF, settembre 1981. <http://www.ietf.org/rfc/rfc0793.txt>.
- [8] Mark Allman, Vern Paxson, e Richard W. Stevens. “*TCP Congestion Control*”. RFC 2581, IETF, aprile 1999.
<http://www.ietf.org/rfc/rfc2581.txt>.

- [9] Sally Floyd e Tom Henderson. “*The New Reno Modification to TCP’s Fast Recovery Algorithm*”. RFC 2582, IETF, aprile 1999. <http://www.ietf.org/rfc/rfc2582.txt>.
- [10] Jon Postel. “*User Datagram Protocol*”. RFC 768, IETF, agosto 1980. <http://www.ietf.org/rfc/rfc0768.txt>.
- [11] Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Henrik F. Nielsen, Larry Masinter, Paul J. Leach, e Tim Berners-Lee. “*Hypertext Transfer Protocol – HTTP/1.1*”. RFC 2616, IETF, giugno 1999. <http://www.ietf.org/rfc/rfc2616.txt>.
- [12] The Defense Advanced Research Projects Agency (DARPA). <http://www.darpa.org/>.
- [13] Internet Assigned Numbers Authority (IANA). <http://www.iana.org/>.
- [14] Kjeld B. Egevange Paul Francis. “*The IP Network Address Translator (NAT)*”. RFC 1631, IETF, maggio 1994. <http://www.ietf.org/rfc/rfc1631.txt>.
- [15] Novellinc. “*Advanced Netware v2.1 Internetwork Packet Exchange Protocol (IPX) with Asynchronous Event Scheduler*”. Technical report, Novell inc., luglio 1987.
- [16] Gursharan S. Sidhu, Richard F. Andrews, e Alan B. Oppenheimer. *Inside Apple Talk*. Addison-Wesley, seconda edizione, maggio 1990, ISBN 0-201-55021-0.
- [17] Xerox corp. “*Internet Transport Protocols*”. Xerox System Integration Standard XSIS 028112, Xerox corp., dicembre 1981.

- [18] Henning Schulzrinne, Stephen L. Casner, Ron Frederick, e Van Jacobson. “*RTP: A Transport Protocol for Real-Time Applications*”. RFC 3550, IETF, luglio 2003. <http://www.ietf.org/rfc/rfc3550.txt>.
- [19] Henning Schulzrinne, Anup Rao, e Robert Lanphier. “*Real Time Streaming Protocol (RTSP)*”. RFC 2326, IETF, aprile 1998. <http://www.ietf.org/rfc/rfc2326.txt>.
- [20] Paul Mockapetris. “*DNS Encoding of Network Names and Other Types*”. RFC 1101, IETF, aprile 1989. <http://www.ietf.org/rfc/rfc1101.txt>.
- [21] Carl Beame, Brent Callaghan, Mike Eisler, David Noveck, David Robinson, e Robert Thurlow. “*NFS version 4 Protocol*”. RFC 3010, IETF, dicembre 2000. <http://www.ietf.org/rfc/rfc3010.txt>.
- [22] John C. Klensin. “*Simple Mail Transfer Protocol*”. RFC2821, IETF, aprile 2001. <http://www.ietf.org/rfc/rfc2821.txt>.
- [23] John G. Myerse Marshall T. Rose. “*Post Office Protocol - Version 3*”. RFC 1939, IETF, maggio 1996. <http://www.ietf.org/rfc/rfc1939.txt>.
- [24] Mark R. Crispin. “*Internet Message Access Protocol - Version 4rev1*”. RFC 3501, IETF, marzo 2003. <http://www.ietf.org/rfc/rfc3501.txt>.
- [25] Jon Postel e Joyce Reynolds. “*File Transfer Protocol (FTP)*”. RFC 959, IETF, ottobre 1985. <http://www.ietf.org/rfc/rfc0959.txt>.

- [26] Karen R. Sollins. “*The TFTP Protocol (Revision2)*”. RFC 1350, IETF, luglio 1992. <http://www.ietf.org/rfc/rfc1350.txt>.
- [27] Defense Advanced Research Projects Agency, End-to-End Services Task Force, Internet Activities Board. “*Protocol Standard for a NetBIOS Service on a TCP/UDP Transport: Concepts and Methods*”. RFC 1001, IETF, marzo 1987. <http://www.ietf.org/rfc/rfc1001.txt>.
- [28] Jon Postel e Joyce Reynolds. “*Telnet Protocol Specification*”. RFC 854, IETF, maggio 1983. <http://www.ietf.org/rfc/rfc0854.txt>.
- [29] Chris Lonwick. “*SSH Protocol Architecture*”. Internet draft, IETF, dicembre 2004. <http://www.ietf.org/internet-drafts/draft-ietf-secsh-architecture-20.txt>.
- [30] Farhad Anklesaria, Mark McCahill, Paul Lindner, David Johnson, Daniel Torrey, e Bob Alberti. “*The Internet Gopher Protocol (a distributed document search and retrieval protocol)*”. RFC 1436, IETF, marzo 1993. <http://www.ietf.org/rfc/rfc1436.txt>.
- [31] Leslie Daigle. “*WHOIS Protocol Specification*”. RFC 3912, IETF, settembre 2004. <http://www.ietf.org/rfc/rfc3912.txt>.
- [32] Internet Engineering Task Force (IETF). <http://www.ietf.org/>.
- [33] T. Berners-Lee, R. Fielding, e L. Masinter. “*Uniform Resource Identifiers (URI): Generic Syntax*”. RFC 2396, IETF, agosto 1998. <http://www.ietf.org/rfc/rfc2396.txt>.

- [34] The Digital Object Identifier System. <http://www.doi.org/>.
- [35] John E. Hopcroft, Rajeev Motwani, e Jeffrey D. Ullman. *Automi, linguaggi e calcolabilità*. Addison-Wesley, prima edizione, febbraio 2003, ISBN 88-7192-154-2.
- [36] ITU Telecommunication Standardization Sector. “*X.25: Interface between Data Terminal Equipment (DTE) and Data Circuit-terminating Equipment (DCE)*”. Recommendation X.26, ITU, ottobre 1996.
- [37] ATM Forum. *ATM User-Network Interface Specification Version 3.0*. Prentice-Hall, 1993, ISBN 01-3225-863-3.
- [38] The linux foundation. <http://www.linux-foundation.org/>.
- [39] The FreeBSD project. <http://www.freebsd.org/>.
- [40] Chris Lonwick. “*The BSD syslog Protocol*”. RFC 3164, IETF, agosto 2001. <http://www.ietf.org/rfc/rfc3164.txt>.
- [41] Vic Abell’s Home Page, creator of lsof. <http://people.freebsd.org/abe/>.
- [42] “*American National Standard for Information Systems – Coded Character Sets – 7-Bit American National Standard Code for Information Interchange (7-Bit ASCII)*”. Technical Report ANSI X3.4-1986, American National Standards Institute, Inc., marzo 1986.
- [43] “*Information technology 8-bit single-byte coded graphic character sets*”. Technical Report ISO/IEC 8859, ISO/IEC.
- [44] Unicode Consortium. <http://www.unicode.org/>.

- [45] Julie D. Allen, editor. *The Unicode Standard 5.0*. Addison-Wesley, 2006, ISBN 0-321-48091-0.
- [46] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, e François Yergeau. “*Extensible Markup Language (XML) 1.0 (Fourth Edition)*”. Technical Report 862, W3C, agosto 2006. <http://www.w3.org/TR/xml/>.
- [47] My SQL Database Management System. <http://www.mysql.com/>.
- [48] PostgreSQL Database Management System. <http://www.postgresql.org/>.
- [49] Ned Freed e Nathaniel S. Borenstein. “*Multipurpose Internet Mail Extensions (MIME)*”. RFC 2045, IETF, novembre 1996. <http://www.ietf.org/rfc/rfc2045.txt>.
- [50] PuTTY:A Free Telnet/SSH Client. <http://www.chiark.greenend.org.uk/sgtatham/putty/>.
- [51] GNU Cygnus for Windows. <http://www.cygwin.com/>.
- [52] Squid: Optimising Web Delivery. <http://www.squid-cache.org/>.
- [53] David M. Kristol e Lou Montulli. “*HTTP State Management Mechanism*”. RFC 2109, IETF, febbraio 1997. <http://www.ietf.org/rfc/rfc2109.txt>.
- [54] Tim Berners-Lee e Daniel W. Connolly. “*Hypertext Markup Language – 2.0*”. RFC 1866, IETF, novembre 1995. <http://www.ietf.org/rfc/rfc1866.txt>.
- [55] Arnaud Le Horse Ian Jacobs. “*HTML4.01 Specification*”. Technical report, W3C, dicembre 1999. <http://www.w3.org/TR/html401/>.

- [56] Rael Dornfest, Paul Bausch, e Tara Calishain. *Google Hacks – Tips & Tools for Finding and Using the World’s Information*. O’Reilly Media, terza edizione, agosto 2006, ISBN 0-596-52706-3.
- [57] Arnaud Le Horse Ian Jacobs. “*XHTML™ 1.1 – Module-based XHTML – Second Edition*”. Technical report, W3C, febbraio 2007. <http://www.w3.org/TR/xhtml11/>.
- [58] RSS Advisory Board. “*RSS2.0 Specification*”. Technical report, RSS Advisory Board, ottobre 2007. <http://www.rssboard.org/rss-specification>.
- [59] Apple. “*Making a Podcast*”. Technical report. <http://www.apple.com/itunes/whatson/podcasts/specs.html>.
- [60] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, e David Orchard. “*Web Services Architecture*”. Technical report, W3C, febbraio 2004. <http://www.w3.org/TR/ws-arch/>.
- [61] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, e Yves Lafon. “*SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*”. Technical report, W3C, aprile 2007. <http://www.w3.org/TR/2007/REC-soap12-part1-20070427/>.
- [62] Michael Papazoglou. *Web Services: Principles and Technology*. Prentice Hall, prima edizione, settembre 2007, ISBN 0-321-15555-6.
- [63] Jon Postel. “*Echo Protocol*”. RFC 862, IETF, maggio 1983. <http://www.ietf.org/rfc/rfc862.txt>.

- [64] Jon Postel. “*Character Generator Protocol*”. RFC 864, IETF, maggio 1983. <http://www.ietf.org/rfc/rfc864.txt>.
- [65] Lynx source distribution and potpourri. <http://lynx.isc.org/>.
- [66] W3M Home Page. <http://w3m.sourceforge.net/>.
- [67] Winamp Media Player. <http://www.winamp.com/>.
- [68] Oracle corp. “*Oracle Berkeley DB*”. <http://www.oracle.com/technology/products/berkeley-db/db/>.
- [69] The XML Cparser and toolkit of Gnome. <http://xmlsoft.org/>.