Appunti del corso di
Sistemi di elaborazione: Reti II

PROF. G. BONGIOVANNI

Premessa

Questi appunti sono basati sui libri "Computer Networks" di A. Tanenbaum, terza edizione, ed. Prentice-Hall, e "Java Network Programming" di Merlin Hughes et al., Mannig Publications Co, Greenwich CT, adottati quali libri di testo del corso.

Essi rispecchiano piuttosto fedelmente il livello di dettaglio che viene seguito durante le lezioni, e costituiscono un ausilio didattico allo studio.

Tuttavia, è importante chiarire che gli appunti non vanno intesi come sostitutivi né del libro di testo né della frequenza alle lezioni, che rimangono fattori importanti per una buona preparazione dell'esame.

La realizzazione di questi appunti è stata resa possibile dalla collaborazione di alcuni studenti, che hanno avuto la pazienza di convertire in forma elettronica il contenuto testuale dei manoscritti da me preparati per il corso. La realizzazione delle figure, la formattazione e la rifinitura del testo sono opera mia.

1) Il World Wide Web

Il ***World Wide Web*** (detto anche ***Web***, ***WWW*** o ***W***[3]) è nato al Cern nel 1989 per consentire una agevole cooperazione fra i gruppi di ricerca di fisica sparsi nel mondo.

E' un'architettura software volta a fornire l'accesso e la navigazione a un enorme insieme di documenti collegati fra loro e sparsi su milioni di elaboratori.

Tale insieme di documenti forma un ***ipertesto*** (***hypertext***), cioè un testo che viene percorso in modo non lineare. Il concetto di ipertesto risale alla fine degli anni '40, e si deve a vari scienziati:

- Vannevar Bush (sistema Memex, basato su microfilm);
- Douglas Engelbart (sistema NLS/Augment, basato su elaboratori interconnessi);
- Ted Nelson (sistema Xanadu, con enfasi sulla tutela dei diritti d'autore: un documento poteva contenere un riferimento ad altri documenti, che venivano inclusi "al volo" in quello referente e mantenevano così la loro unicità e originalità).

Il Web ha diverse caratteristiche che hanno contribuito al suo enorme successo:

- architettura di tipo client-server:
  - ampia scalabilità;
  - adatta ad ambienti di rete;
- architettura distribuita:
  - perfettamente in linea con le esigenze di gestione di un ipertesto;
- architettura basata su standard di pubblico dominio:
  - possibilità per chiunque di proporre una implementazione;
  - uguali possibilità di accesso per tutte le piattaforme di calcolo;
- capacità di gestire informazioni di diverso tipo (testo, immagini, suoni, filmati, realtà virtuale, ecc.):
  - grande interesse da parte di tutti gli utenti.

I documenti che costituiscono l'ipertesto gestito dal Web sono detti ***pagine web***, e possono contenere, oltre a normale testo formattato, anche:

- rimandi (detti ***link*** o ***hyperlink***) ad altre pagine web;
- immagini fisse o in movimento;
- suoni;
- scenari tridimensionali interattivi;
- codice eseguibile localmente.

L'utilizzo del Web è semplicissimo:

- un utente legge il testo della pagina, vede le immagini, ascolta la musica, ecc.;

- se seleziona col mouse un link (che di solito appare come una parola sottolineata e di diverso colore) la pagina di partenza viene sostituita sullo schermo da quella relativa al link selezionato.

Si noti che la nuova pagina può provenire da qualunque parte del pianeta.

1.1) Architettura client-server del Web

Il Web è una architettura software di tipo ***client-server***, nella quale sono previste due tipologie di componenti software: il ***client*** e il ***server***, ciascuno avente compiti ben definiti.

1.1.1) Client

Il client (o ***user agent***) è lo strumento a disposizione dell'utente che gli permette l'accesso e la navigazione nell'ipertesto del Web.

Esso ha varie competenze:
- trasmettere all'opportuno server le richieste di reperimento dati che derivano dalle azioni dell'utente;
- ricevere dal server le informazioni richieste;
- visualizzare il contenuto della pagina Web richiesta dall'utente, gestendo in modo appropriato tutte le tipologie di informazioni in esse contenute;
- consentire operazioni locali sulle informazioni ricevute (ad esempio salvarle su disco, stamparle).

I client vengono comunemente chiamati ***browser*** (sfogliatori). Gli esempi più noti sono:
- NCSA Mosaic (il primo);
- Netscape Navigator;
- Microsoft Internet Explorer.

In generale è troppo complicato e costoso (sarebbero necessari aggiornamenti troppo frequenti) sviluppare un browser che sappia gestire direttamente tutti i tipi di informazioni presenti sul Web, poiché essi sono in continuo e rapido aumento.

Per questa ragione, di norma i browser gestiscono direttamente solo alcune tipologie di informazioni, quali:
- testo formattato;
- immagini fisse;
- codice eseguibile.

Viceversa, di norma gli altri tipi di informazioni vengono gestiti in uno (o entrambi) dei seguenti modi:
- consegnandoli a un programma esterno (***helper***) che provvederà alla corretta gestione (ad esempio, un file contenente un filmato verrà consegnato a un programma per il playback di filmati);
- se il browser ha un'architettura modulare le sue funzionalità possono essere estese per mezzo di ***plug-in***, ossia librerie di codice eseguibile specializzato che possono essere caricate in memoria secondo le necessità. In questa situazione, se il necessario plug-in è installato, il browser provvede a caricarlo e gli affida la gestione delle informazioni da trattare.

Una importante caratteristica di tutti i browser moderni è di essere ***multithreaded***, cioè di consentire che, quando la cpu è sotto il loro controllo, si alternino fra loro multipli ***thread di controllo***, cioè flussi di elaborazione concorrenti. Spesso si usa come sinonimo di thread il termine ***lightweight process***.

Ad esempio, nel caso di un sistema operativo (S.O.) che offre il multitasking, si può avere una situazione come quella seguente.

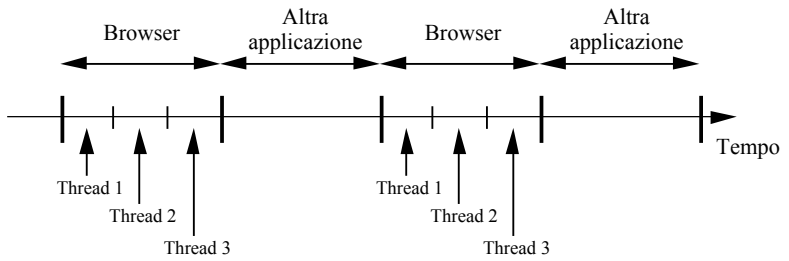


Figura 1-1: Uso della CPU in un browser multithreaded

Un thread, a differenza di un vero processo, è un contesto di esecuzione il cui spazio di indirizzamento viene ricavato all'interno di quello del processo che lo ha generato.

1.1.2) Server

Il server è tipicamente un processo in esecuzione su un elaboratore. Esso, di norma, è sempre in esecuzione (tranne che in situazioni eccezionali) ed ha delle incombenze molto semplici, almeno in linea di principio. Infatti deve:
- rimanere in ascolto di richieste da parte dei client;
- fare del suo meglio per soddisfare ogni richiesta che arriva:
    - se possibile, consegnare il documento richiesto;

- altrimenti, spedire un messaggio di notifica di errore (documento non esistente, documento protetto, ecc.).

Nonostante la apparente semplicità di tale compito, la realizzazione di un server non è banale, perché:
- deve fare il suo lavoro nel modo più efficiente possibile, dunque deve essere implementato con un occhio di riguardo alle prestazioni;
- deve essere in grado di gestire molte richieste contemporaneamente, e mentre fa questo deve continuare a rimanere in ascolto di nuove richieste.

Il secondo requisito in particolare implica una qualche forma di concorrenza nel lavoro del server. Essa si può ottenere in vari modi, anche in funzione delle caratteristiche del sistema operativo sottostante. Le due tecniche più diffuse sono descritte nel seguito.

Clonazione del server
L'idea è semplice:
- per ogni nuova richiesta che arriva, il server (che è sempre in ascolto):
  - crea una nuova copia di se stesso alla quale affida la gestione della richiesta;
  - si mette subito in attesa di nuove richieste;
- la copia clonata si occupa di soddisfare la richiesta e poi termina.

Le varie copie del server vivono in spazi di indirizzamento separati, e il loro avvicendamento nell'uso della CPU è governato dal sistema operativo.

Questo è un metodo tipico di S.O. multitasking quali UNIX, e si ottiene con l'uso della `fork()`.

Vantaggi:
- il codice del server rimane semplice, poiché la clonazione è demandata in toto al S.O.

Svantaggi
- poiché in genere la gestione di una richiesta è piuttosto rapida, il tempo di generazione del clone può non essere trascurabile rispetto al tempo di gestione della richiesta, introducendo così un overhead che può penalizzare l'efficienza del sistema.

Server multithreaded
Esiste una sola copia del server, che però è progettato per essere in grado di generare thread multipli:
- il thread principale (quello iniziale) rimane sempre in ascolto delle richieste;
- quando ne arriva una, esso genera un nuovo thread che prima la gestisce e poi termina.

Questo metodo richiede che il S.O. offra librerie di supporto al multithreading, che ormai sono presenti in tutti i S.O. moderni (UNIX, Windows 95 e NT, MacOS, Linux) per cui di fatto è universalmente applicabile.

Vantaggi:

- la creazione di un thread è molto più veloce di una `fork()` (anche 30 volte sotto UNIX), quindi in generale è più efficiente per gestire operazioni veloci come l'esaudire la richiesta del client.

Svantaggi:
- il codice del server diviene un pò più complesso, perché al suo interno si dovranno gestire la creazione dei thread ed il loro avanzamento, anche in termini di sincronizzazione.

1.2) Standard utilizzati nel Web

Ci sono tre standard principali che, nel loro insieme, costituiscono l'architettura software del Web:
- ***sistema di indirizzamento*** basato su ***Uniform Resource Locator*** (***URL***): è un meccanismo standard per fare riferimento alle entità indirizzabili (risorse) nel Web, che possono essere:
  - documenti (testo, immagini, suoni, ecc.);
  - programmi eseguibili (vedremo poi);
- ***linguaggio HTML*** (***HyperText Markup Language***): è il linguaggio per la definizione delle pagine Web;
- ***protocollo HTTP*** (***HyperText Transfer Protocol***): è il protocollo che i client e i server utilizzano per comunicare.

1.2.1) URL

Una URL costituisce un riferimento a una qualunque risorsa accessibile nel Web.

Tale risorsa ovviamente risiede da qualche parte, ed è in generale possibile accedervi in vari modi.

Dunque, una URL deve essere in grado di indicare:
- come si vuole accedere alla risorsa;
- dove è fisicamente localizzata la risorsa;
- come è identificata la risorsa.

Per queste ragioni, una URL è fatta di 3 parti, che specificano:
- il ***metodo di accesso***;
- l'***host*** che detiene la risorsa;
- l'***identità*** della risorsa.

Un tipico esempio di una URL è:

```
http://somewhere.net/products/index.html
```

nella quale:

`http://`	è il metodo di accesso
`somewhere.net`	è il nome dell'host
`/products/index.html`	è l'identità della risorsa

Metodo di accesso
Indica il modo di accedere alla risorsa, cioè che tipo di protocollo bisogna usare per colloquiare col server che controlla la risorsa.

I metodi di accesso più comuni sono:

http	protocollo nativo del Web
ftp	file transfer protocol
news	protocollo per l'accesso ai gruppi di discussione
gopher	vecchio protocollo per il reperimento di informazioni; concettualmente simile al Web, gestisce solo testo
mailto	usato per spedire posta
telnet	protocollo di terminale virtuale, per effettuare login remoti
file	accesso a documenti locali

Il Web nasce con l'idea di inglobare gli altri protocolli di accesso alle informazioni, per costituire un ambiente unificato che soddisfa tutte le esigenze.

Quando il client effettua la richiesta di una risorsa, usa nel dialogo col server il protocollo specificato dal metodo d'accesso. Se non è in grado di farlo, affida il compito a una ***applicazione helper*** esterna (questo è tipicamente il caso del protocollo telnet: il client lancia un emulatore di terminale passandogli il nome dell'host).

Dall'altra parte risponde il server di competenza, che può essere:
- un server Web in grado di gestire anche altri protocolli;
- un server preesistente per lo specifico protocollo (ftp, gopher, ecc.).

Nome dell'host
Può essere l'***indirizzo IP*** numerico o, più comunemente, il ***nome DNS*** dell'host a cui si vuole chiedere la risorsa.

Dopo il nome dell'host può essere incluso anche un ***numero di port***. Se non c'è, si intende il port 80 (che è il default). Ad esempio:

```
http://somewhere.net:8000/products/index.html
```

In questo modo si possono avere, sullo stesso host, diversi server Web in ascolto su diverse porte.

Identità della risorsa
Consiste, nella sua forma più completa, della specifica del nome di un file e del cammino che porta al direttorio in cui si trova.

Ad esempio, la URL:

```
http://somewhere.net/products/toasters/index.html
```

specifica il file `index.html` contenuto nel direttorio `toasters`, a sua volta contenuto nel direttorio `products` il quale si trova nel direttorio radice dell'host `somewhere.net`.

Si noti che:
- la sintassi è quella di Unix;
- il direttorio radice è relativo all'albero dei documenti Web, e non è necessariamente la radice dell'intero file system dell'elaboratore;
- ciò fa sì che sia di fatto impossibile accedere per mezzo del Web al di fuori di tale parte del file system: il server, di norma, non consente di accedere a nulla che non sia nell'albero dei documenti Web.

Esistono alcune regole per il completamento di URL non interamente specificate:
- se manca il nome del direttorio, si assume quello della pagina precedente;
- se manca il nome del file (ma c'è quello del direttorio), a seconda del server:
  - si restituisce un file prefissato del direttorio specificato (index.html, default.html oppure welcome.html);
  - se tale file non esiste, talvolta si restituisce un elenco dei file nel direttorio.

Infine, una convenzione usata spesso è la seguente. A fronte di una URL del tipo:

```
http://somewhere.net/~username/
```

il server restituisce il file `welcome.html` situato nel direttorio `public_html` situato nel direttorio principale (***home directory***) dell'utente `username`.

Questo meccanismo consente agli utenti, che di norma hanno libero accesso al proprio home directory, di mantenere facilmente proprie pagine Web.

1.2.2) Linguaggio HTML

Il linguaggio per la formattazione di testo HTML è una specializzazione del linguaggio ***SGML*** (***Standard Generalized Markup Language***) definito nello standard ISO 8879.

HTML è specializzato nel senso che è stato progettato appositamente per un utilizzo nell'ambito del Web.

Un ***markup language*** si chiama così perché i comandi (***tag***) per la formattazione sono inseriti in modo esplicito nel testo, a differenza di quanto avviene in un word processor ***WYSIWYG*** (***What You See Is What You Get***), nel quale il testo appare visivamente dotato dei suoi formati, come fosse stampato. TROFF e TeX sono altri markup language, mentre ad esempio Microsoft Word è WYSIWYG.

Per esempio in HTML il testo:

```
...questo è <B>grassetto</B> e questo no...
```

indica che la parola `grassetto` deve essere visualizzata in ***grassetto*** (***bold***). Quindi il testo in questione dovrà apparire come segue:

...questo è **grassetto** e questo no...

Il ruolo di HTML è quindi quello di definire il modo in cui deve essere visualizzata una pagina Web (detta anche ***pagina HTML***), che tipicamente è un documento di tipo testuale contenente opportuni tag di HTML.

Il client, quando riceve una pagina compie le seguenti operazioni:
- interpreta i tag presenti nella pagina;
- formatta la pagina di conseguenza, provvedendo automaticamente ad adattarla alle condizioni locali (risoluzione dello schermo, dimensione della finestra, profondità di colore, ecc.);
- mostra la pagina formattata sullo schermo.

Nella formattazione si ignorano:
- sequenze multiple di spazi;
- caratteri di fine riga, tabulazioni, ecc.

I tag HTML possono essere divisi in due categorie:
- tag per la formattazione di testo;
- tag per altre finalità (inclusione di immagini, interazione con l'utente, elaborazione locale, ecc.).

Il linguaggio HTML è in costante evoluzione, si è passati dalla versione 1.0 alla 2.0 (rfc 1866), poi alla 3.0 e ora alla 3.2.

E' in corso una attività di standardizzazione della versione 3, che cerca di mediare le proposte, spesso incompatibili, che sono portate avanti da diverse organizzazioni (quali Netscape e Microsoft) le quali spingono perché proprie estensioni (ad esempio i ***frame*** di Netscape e gli ***style sheet*** di Microsoft) divengano parte dello standard.

In genere i tag hanno la forma:

```
<direttiva> ... </direttiva>
```

e possono contenere parametri:

```
<direttiva parametro1="valore"...> ... </direttiva>
```

Struttura di un documento HTML
Una pagina HTML ha questa struttura:

```
<HTML>
<HEAD>
...
<TITLE>...</TITLE>
...
</HEAD >
<BODY>
...
</BODY>
</HTML>
```

Il ruolo di questi marcatori è il seguente:

HTML	Inizio e fine del documento
HEAD	Questa parte non viene mostrata e contiene metainformazioni sul documento (creatore, data di "scadenza", e se c'è, il titolo)
TITLE	Il titolo del documento: appare come titolo della finestra che lo contiene
BODY	Il suo contenuto viene visualizzato nella finestra

Tag per la formattazione
Alcuni dei tag esistenti per la formattazione del testo sono i seguenti:

`<B>...</B>`	Grassetto (***bold***)
`<I>...</I>`	Corsivo (***italic***)
`<Hx>...</Hx>`	Intestazione (***heading***) di livello x (da 1 a 6)
`<PRE>...</PRE>`	Testo visualizzato esattamente come è scritto (***preformatted***), con spazi multipli, caratteri di fine linea, ecc.

Ci sono moltissimi altri tag per la formattazione, coi quali si possono specificare:
- dimensione, colore, tipo dei caratteri;
- centratura del testo;
- liste di elementi;
- tabelle di testo in forma grafica (`<TABLE>`);
- divisori (`<HR>`,`<BR>`,`<P>`);
- colore di sfondo della pagina;
- suddivisione della finestra fra più pagine (`<FRAMESET>`, `<FRAME>`).

Tag per altre finalità
Questi sono i tag che forniscono al Web la sua grande versatilità. Anch'essi sono in continua evoluzione, permettendo di includere sempre nuove funzionalità.

I tag di questo tipo più usati sono quelli per la inclusione di ***immagini in-line*** (visualizzate direttamente all'interno della pagina) e per la gestione degli hyperlink.

Il tag per la inclusione di immagini ha la seguente forma:

```
<IMG SRC="url"> oppure <IMG SRC="url" ALT="testo...">
```

Questo tag fa apparire l'immagine di cui alla URL. L'immagine (se il client è configurato per farlo) viene richiesta automaticamente e quando è disponibile viene mostrata. Altrimenti, al suo posto appare una piccola icona, sulla quale bisogna fare click se si vuole vedere la relativa immagine (che solo allora verrà richiesta), seguita dal testo specificato nel parametro `ALT`.

Altri parametri del tag `<IMG>` servono a:
- specificare le dimensioni dell'immagine (`WIDTH, HEIGHT`);
- specificare l'allineamento dell'immagine e del testo circostante (`ALIGN`);
- specificare le aree dell'immagine sensibili ai click del mouse (`ISMAP`).

Tag per la gestione degli hyperlink
Costituiscono il fondamento funzionale su cui è basato il Web, perché è per mezzo di questi che si realizzano le funzioni ipertestuali.

Il tag è uno solo (con alcune varianti) e viene chiamato ***anchor***:

```
<A> .....</A>
```

La sua forma standard è:

```
...<A HREF="url">testo visibile</A>...
```

Nella pagina la stringa `testo visibile` appare sottolineata e, di norma, di colore blu:

...testo visibile...

Quando l'utente fa click su un'ancora (ossia sul testo visibile della stessa) il client provvede a richiedere il documento di cui alla URL, lo riceve, lo formatta e lo mostra nella finestra al posto di quello precedente.

1.2.3) Il protocollo HTTP

Il protocollo HTTP sovraintende al dialogo fra un client e un server web, ed è il linguaggio nativo del Web.

HTTP non è ancora uno standard ufficiale. Infatti, HTTP 1.0 (rfc 1945) è ***informational***, mentre HTTP 1.1 (rfc 2068) è ancora in fase di proposta; parleremo di quest'ultimo più avanti.

HTTP è un ***protocollo ASCII***, cioè i messaggi scambiati fra client e server sono costituiti da sequenze di caratteri ASCII (e questo, come vedremo, è un problema se è necessaria la riservatezza delle comunicazioni).

In questo contesto per ***messaggio*** si intende la richiesta del cliente oppure la risposta del server, intesa come informazione di controllo; viceversa, i dati della URL richiesta che vengono restituiti dal server non sono necessariamente ASCII (esempi di dati binari: immagini, filmati, suoni, codice eseguibile).

Il protocollo prevede che ogni singola interazione fra client e server si svolga secondo il seguente schema:
- apertura di una connessione di livello transport fra client e server (TCP è lo standard di fatto, ma qualunque altro può essere usato);
- invio di una singola richiesta da parte del client, che specifica la URL desiderata;
- invio di una risposta da parte del server e dei dati di cui alla URL richiesta;
- chiusura della connessione di livello transport.

Dunque, il protocollo è di tipo ***stateless***, cioè non è previsto il concetto di ***sessione*** all'interno della quale ci si ricorda dello stato dell'interazione fra client e server. Ogni singola interazione è storia a se ed è del tutto indipendente dalle altre.

La richiesta del client
Quando un client effettua una richiesta invia diverse informazioni:
- il metodo (cioè il comando) che si chiede al server di eseguire;
- il numero di versione del protocollo HTTP in uso;
- l'indicazione dell'oggetto al quale applicare il comando;
- varie altre informazioni, fra cui:
  - il tipo di client;
  - i tipi di dati che il client può accettare.

I metodi definiti in HTTP sono:

GET	Richiesta di ricevere un oggetto dal server
HEAD	Richiesta di ricevere la sola parte head di una pagina html
PUT	Richiesta di mandare un oggetto al server
POST	Richiesta di appendere sul server un oggetto a un altro (vedremo che si usa molto)
DELETE	Richiesta di cancellare sul server un oggetto
LINK e UNLINK	Richieste di stabilire o eliminare collegamenti fra oggetti del server

In proposito, si noti che:
- il metodo che si usa quasi sempre è GET;
- POST ha il suo più significativo utilizzo in relazione all'invio di dati tramite form, come vedremo in seguito;
- HEAD si usa quando il client vuole avere delle informazioni per decidere se richiedere o no la pagina;
- PUT, DELETE, LINK, UNLINK non sono di norma disponibili per un client, tranne che in quei casi in cui l'utente sia abilitato alla configurazione remota (via Web) del server Web.

Ad esempio, supponiamo che nel file HTML visualizzato sul client vi sia un'ancora:

```
<A HREF="http://somewhere.net/products/toasters/index.html"> ..... </A>
```

e che l'utente attivi tale link. A tal punto il client:
- chiede al DNS l'indirizzo IP di somewhere.net;
- apre una connessione TCP con somewhere.net, port 80;
- invia la sua richiesta.

Essa è costituita da un insieme di comandi (uno per ogni linea di testo) terminati con una linea vuota:

`GET /products/toasters/index.html HTTP/1.0`	Metodo, URL e versione protocollo
`User-agent: Mozilla/3.0`	Tipo del client
`Host: 160.10.5.43`	Indirizzo IP del client
`Accept: text/html`	Client accetta pagine HTML
`Accept: image/gif`	Client accetta immagini
`Accept: application/octet-stream`	Client accetta file binari qualunque
`If-modified-since: data e ora`	Inviare il documento solo se è più recente della data specificata

La risposta del server
La risposta del server è articolata in più parti, perché c'è un problema di fondo: come farà il client a sapere in che modo dovrà gestire le informazioni che gli arriveranno?

Ovviamente, non si può mostrare sotto forma di testo un'immagine o un file sonoro! Dunque, si deve informare il client sulla natura dei dati che gli arriveranno prima di iniziare a spedirglieli.

Per questo motivo la risposta consiste di 3 parti:
- una ***riga di stato***, che indica quale esito ha avuto la richiesta (tutto ok, errore, ecc.);
- delle ***metainformazioni*** che descrivono la natura delle informazioni che seguono;
- le ***informazioni*** vere e proprie (ossia, l'oggetto richiesto).

La riga di stato, a sua volta, consiste di tre parti:
- Versione del protocollo http;
- Codice numerico di stato;
- Specifica testuale dello stato.

Tipici codici di stato sono:

Esito	**Codice numerico**	**Specifica testuale**
Tutto ok	`200`	`OK`
Documento spostato	`301`	`Moved permanently`
Richiesta di autenticazione	`401`	`Unauthorized`
Richiesta di pagamento	`402`	`Payment required`
Accesso vietato	`403`	`Forbidden`
Documento non esistente	`404`	`Not found`
Errore nel server	`500`	`Server error`

Dunque, ad esempio, si potrà avere

```
HTTP/1.0 200 OK
```

Le metainformazioni dicono al client ciò che deve sapere per poter gestire correttamente i dati che riceverà.

Sono elencate in linee di testo successive alla riga di stato e terminano con una ***linea vuota***.

Tipiche metainformazioni sono:

`Server: ...`	Identifica il tipo di server
`Date: ...`	Data e ora della risposta
`Content-type: ...`	Tipo dell'oggetto inviato
`Content-length: ...`	Numero di byte dell'oggetto inviato
`Content-language: ...`	Linguaggio delle informazioni
`Last-modified: ...`	Data e ora di ultima modifica
`Content-encoding: ...`	Tipo di decodifica per ottenere il content

Il Content-type si specifica usando lo standard ***MIME*** (***Multipurpose Internet Mail Exchange***), nato originariamente per estendere la funzionalità della posta elettronica.

Un tipo MIME è specificato da una coppia

```
MIME type/MIME subtype
```

Vari tipi MIME sono definiti, e molti altri continuano ad aggiungersi. I più comuni sono:

Type/Subtype	**Estensione**	**Tipologia delle informazioni**
`text/plain`	`.txt, .java`	testo
`text/html`	`.html, .htm`	pagine html
`image/gif`	`.gif`	immagini gif
`image/jpeg`	`.jpeg, .jpg`	immagini jpeg
`audio/basic`	`.au`	suoni
`video/mpeg`	`.mpeg`	filmati
`application/octet-stream`	`.class, .cla, .exe`	programmi eseguibili
`application/postscript`	`.ps`	documenti Postscript
`x-world/x-vrml`	`.vrml, .wrl`	scenari 3D

Il server viene configurato associando alle varie estensioni i corrispondenti tipi MIME. Quando gli viene chiesto un file, deduce dall'estensione e dalla propria configurazione il tipo MIME che deve comunicare al client.

Se la corrispondenza non è nota, si usa quella di default (tipicamente text/html), il che può causare errori in fase di visualizzazione.

Anche la configurazione del client (in merito alle applicazioni helper) si fa sulla base dei tipi MIME.

Tornando al nostro esempio, una richiesta del client quale:

```
GET /products/toasters/index.html HTTP/1.0
User-agent: Mozilla/3.0
ecc.
```

riceverà come risposta dal server (supponendo che non ci siano errori) le metainformazioni, poi una riga vuota e quindi il contenuto del documento (in questo caso una pagina HTML costituita di 6528 byte):

```
HTTP/1.0 200 OK
Server: NCSA/1.4
Date: Tue, july 4, 1996 19:17:05 GMT
Content-type: text/html
Content-length: 6528
Content-language: en
Last-modified: Mon, july 3, 1996 15:05:35 GMT
                                                    <----- notare la riga vuota
<HTML>
<HEAD>
...
<TITLE>...</TITLE>
...
</HEAD >
<BODY>
...
</BODY>
</HTML>
```

Sulla base di quanto detto finora, si possono fare alcune osservazioni:

- il protocollo HTTP è molto semplice, essendo basato su interazioni che prevedono esclusivamente l'invio di una singola richiesta e la ricezione della relativa risposta;
- questa semplicità è insieme un punto di forza e di debolezza:
  - di forza perché è facilissimo, attraverso la definizione di nuovi tipi MIME e di corrispondenti funzioni sui client, estendere le tipologie di informazioni gestibili (il server non entra nel merito di ciò che contengono i file: si limita a consegnare i dati che gli vengono richiesti, senza preoccuparsi della loro semantica);
  - di debolezza perché queste estensioni di funzionalità talvolta mal si adattano alla concezione originaria (stateless) del protocollo, come ad esempio è il caso delle transazioni commerciali.

1.3) Estensioni del Web

Al crescente successo del Web si è accompagnato un continuo lavoro per ampliarne le possibilità di utilizzo e le funzionalità offerte agli utenti.

In particolare, si è presto sentita l'esigenza di fornire un supporto a una qualche forma di interattività superiore a quella offerta dalla navigazione attraverso gli hyperlink, ad esempio per consentire agli utenti di consultare una base di dati remota via Web.

Le principali estensioni del Web da questo punto di vista sono:
- introduzione delle ***form*** per l'invio di dati al server Web;
- introduzione del linguaggio ***LiveScript*** (poi chiamato ***JavaScript***) per la definizione di computazioni, eseguite dal client, direttamente associate a una pagina HTML;
- introduzione del linguaggio ***Java*** per la creazione di file eseguibili, che vengono trasmessi dal server al client e poi vengono eseguiti in totale autonomia dal client;

1.3.1) Estensione per mezzo delle form

Finora abbiamo sempre implicitamente considerato una URL come un riferimento ad un oggetto passivo, quale ad esempio una pagina HTML o un'immagine.

Ciò non è vero in generale, infatti una URL può fare riferimento a un qualunque tipo di oggetto, e in particolare anche a un file eseguibile.

Proprio grazie a queste generalità del meccanismo di indirizzamento URL si è definita una potente estensione Web, che lo mette in grado di integrare praticamente ogni tipo di applicazione esterna.

L'estensione di funzionalità di cui stiamo parlando è costituita da due componenti:
- sul client: la possibilità di offrire, nella pagina HTML, dei moduli, detti form, per immissione di dati e un meccanismo per l'invio di tali dati al server;
- sul server: un meccanismo funzionale per consegnare i dati provenienti dalla form ad una applicazione che sa gestirli, e per restituire al client gli eventuali risultati prodotti da tale applicazione.

Idealmente, la situazione è questa:

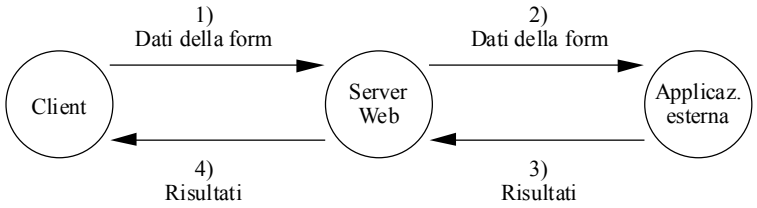


Figura 1-2: Dialogo fra cient, server Web e applicazione esterna

Ad esempio, supponiamo di avere questo scenario (che è anche il più tipico):
- esiste una base dati interessante, ad esempio quella di una biblioteca;
- si desidera consultarla via Web.

Immaginiamo che si voglia presentare all'utente un modulo di ricerca in biblioteca di questo tipo (in una pagina HTML):

Figura 1-3: Modulo per la ricerca in biblioteca

Esistono degli appositi tag di HTML destinati a creare simili moduli. è possibile includere:

- campi testo;
- radio button;
- checkbox;
- menù a tendina.

Nel nostro caso, il codice HTML che genera il modulo sopra visto è il seguente:

```
<HTML>
<HEAD><TITLE> ... </TITLE></HEAD>
</BODY>
<CENTER>
<H1>BIBLIOTECA DEI SOGNI</H1>
<H3>Modulo ricerca dati</H3>
<HR>
<FORM ACTION="http://somewhere.com/scripts/biblio.cgi">
Autore:<INPUT TYPE="text" NAME="autore" MAXLENGTH="64">
<P>
Titolo:<INPUT TYPE="text" NAME="titolo" MAXLENGTH="64">
<P>
<INPUT TYPE="submit"><INPUT TYPE="reset">
</FORM>
<HR>
</BODY>
</HTML>
```

Il tag `<FORM ... >` ha un parametro `ACTION` che specifica l'oggetto (ossia il programma) a cui il server dovrà consegnare i dati immessi dall'utente. In questo caso, vanno consegnati a un programma il cui nome `biblio.cgi` (vedremo poi il perché di questa estensione).

I tag `<INPUT ... >` definiscono le parti della form che gestiscono l'interazione con l'utente. In questo caso abbiamo:

- due campi per l'immissione di testo;
- due pulsanti, rispettivamente per l'invio dei dati e per l'azzeramento dei dati precedentemente immessi.

Quando l'utente preme il bottone "Submit Query" il client provvede a inviare i dati al server, assieme all'indicazione dell'URL alla quale consegnarli.

I dati vengono inviati come coppie:

`NomeDelCampo1=ValoreDelCampo1&NomeDelCampo2=ValoreDelCampo2` ecc.

dove:

- `NomeDelCampo` viene dal parametro `NAME`;
- `ValoreDelCampo` è la stringa immessa dall'utente in quel campo;
- le coppie sono separate dal carattere `&` e sono opportunamente codificate, se ci sono spazi e caratteri speciali.

La forma specifica del messaggio inviato dal client dipende anche da un altro parametro del tag `FORM`: oltre a `ACTION=...` c'è anche `METHOD=...` , che può essere:

- `GET`, assunto implicitamente (come nel nostro esempio);
- `POST`, che deve essere indicato in modo esplicito (questo è praticamente l'unico ambito nel quale è ammesso l'uso di tale comando).

Dunque, possiamo avere:

- `<FORM ACTION=".....">`
- `<FORM ACTION="....." METHOD="get">`
- `<FORM ACTION="....." METHOD="post">`

Nel primo e nel secondo caso, la richiesta inviata dal client consiste nella specifica della URL alla quale vengono "appesi" i dati richiesti (con un punto di domanda):

```
GET /scripts/biblio.cgi?titolo=congo&autore=crichton&submit=submit HTTP/1.0
User-agent: ...
Accept: ...
...
                                                    <----- riga vuota
```

Nel caso del metodo `POST`, che in questo contesto è molto diffuso, i dati vengono rinviati nel corpo del messaggio:

```
POST /scripts/biblio.cgi HTTP/1.0
User-agent: ...
Accept: ...
...
Content-type: application/x-www-form-urlencoded
Content-length: 42
                                                    <----- riga vuota
titolo=congo&autore=crichton&submit=submit     <----- dati della form
```

Il primo metodo è molto semplice, ma impone un limite al numero di caratteri che possono essere spediti. Il secondo invece non ha alcun limite, ed è da preferire perché probabilmente il primo verrà prima o poi abbandonato.

Quando il server riceve la richiesta, da questa è in grado di capire:

- i valori forniti dall'utente;
- a quale programma deve consegnarli.

Nel nostro esempio (che è un caso tipico) il programma `biblio.cgi` avrà l'incarico di:

- ricevere i dati dal server Web;
- trasformarli in una query da sottomettere al gestore della base dati;
- ricevere da quest'ultimo i risultati della query;
- provvedere a confezionarli sotto la forma di una pagina HTML generata al volo;
- consegnare tale pagina al server che la invia al client.

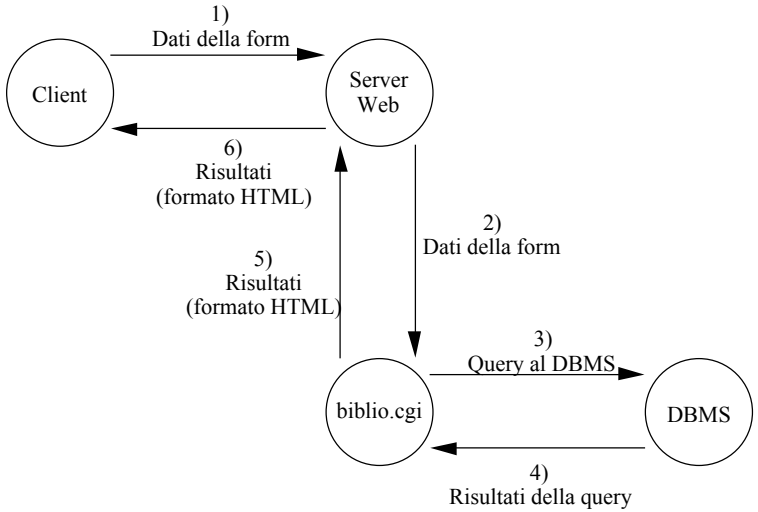


Figura 1-4: Dialogo fra cient, server Web, programma biblio.cgi e gestore della base dati (DBMS) della biblioteca

1.3.2) Common Gateway Interface

Il programma `biblio.cgi` del nostro esempio precedente fa da ***gateway*** fra il server Web e l'applicazione esterna che si vuole utilizzare, nel senso che oltre che metterli in comunicazione è anche incaricato di operare le opportune trasformazioni dei dati che gli giungono da entrambe le direzioni.

Da ciò deriva il nome ***CGI*** (***Common Gateway Interface***) per lo standard che definisce alcuni aspetti della comunicazione fra il server Web e il programma gateway, e l'uso dell'estensione .cgi per tali programmi.

Un programma CGI può essere scritto in qualunque linguaggio:
- script di shell (UNIX);
- Perl, TCL;
- C, Pascal, Fortran, Java;
- AppleScript (MacOS).

Si tenga presente che lo standard illustrato qui è di fatto relativo alla piattaforma UNIX. Per altre piattaforme i meccanismi di comunicazione previsti dalla relativa Common Gateway Interface possono essere differenti, fermo restando che le informazioni passate sono essenzialmente le stesse.

Per la comunicazione dal server Web al programma CGI si utilizzano:
- variabili d'ambiente;
- standard input del programma CGI (solo con il metodo `POST`: i dati della form vengono inviati mediante ***pipe***).

Molte variabili d'ambiente vengono impostate dal server prima di lanciare il programma CGI; le principali sono:

QUERY_STRING	La lista di coppie `campo=valore` (usata con il metodo `GET`)
PATH_INFO	Informazioni aggiuntive messe dopo l'URL (usate ad esempio con le ***clickable map***)
CONTENT_TYPE	Usata con il metodo `POST`, indica il tipo MIME delle informazioni che arriveranno (application/x-www-form-urlencoded);
CONTENT_LENGTH	Usata con il metodo `POST`, indica quanti byte arriveranno sullo standard input del programma CGI

In più ci sono altre variabili per sapere chi è il server, chi è il client, per la autenticazione, ecc.

Le comunicazioni dal programma CGI al server Web avvengono grazie al fatto che il programma CGI invia l'output sul suo standard output, che viene collegato tramite pipe allo standard input del server Web.

Ci si aspetta che il programma CGI possa fare solo due cose:
- generare al volo una pagina HTML, che poi sarà restituita dal server al client;
- indicare al server una pagina preesistente da restituire al client (***URL redirection***).

L'output del programma CGI deve iniziare con un breve header contenente delle metainformazioni, che istruisce il server su cosa fare. Poi una riga vuota, quindi l'eventuale pagina HTML.

Le metainformazioni che il server prende in considerazione (tutte le altre vengono passate al client) sono:

Metainformazione	**Valori**	**Significato**
Content-type:	Tipo/Sottotipo MIME	Tipo dei dati che seguono
Location:	URL	File da restituire al client
Status:	OK, Error	Esito dell'elaborazione

Ad esempio, supponendo di usare un linguaggio di programmazione nel quale sia definita una ipotetica procedura `println(String s)` che scrive una linea di testo sullo standard output, un programma CGI che genera al volo una semplice pagina HTML potrà contenere al suo interno una sequenza di istruzioni quale:

```
println("Status: 200 OK");
println("Content-type: text/html");
println(""); //linea vuota che delimita la fine delle metainformazioni
println("<HTML>");
println("<HEAD>");
...
println("</HEAD>");
println("<BODY>");
...
println("</BODY>");
println("</HTML>");
```

Viceversa, se il programma CGI vuole effettuare una URL redirection il codice eseguito potrà essere:

```
println("Status: 200 OK");
println("Location: /docs/index.html");
println(""); //linea vuota che delimita la fine delle metainformazioni
```

Per gli altri sistemi operativi i meccanismi di comunicazione possono essere diversi:

- su Windows si usano variabili d'ambiente e file temporanei;
- su MacOS si usa Applescript: le informazioni (inclusa la risposta del programma CGI) vengono passate come stringhe associate a variabili dal nome predefinito.

Dunque, la tipica catena di eventi è la seguente:

- il client invia una richiesta contenente:
  - identità del programma CGI;
  - dati della form;
- il server riceve i dati;
- il server lancia il programma CGI e gli passa i dati;
- il programma CGI estrae dai dati i valori immessi dall'utente;
- il programma CGI usa tali valori per portare avanti il suo compito, che può essere ad esempio:
  - aprire una connessione con un DBMS (anche remoto) ed effettuare una query;
  - lanciare a sua volta un'altra applicazione, chiedendole qualcosa;
- il DBMS (o l'applicazione) restituisce dei risultati al programma CGI;
- il programma CGI utilizza tali risultati per confezionare al volo una pagina HTML, che restituisce al server;
- il server invia tale pagina al client, che la visualizza sullo schermo.

Ulteriori usi del meccanismo CGI si presentano in relazione a:

- server side image map;
- motori di ricerca.

Server side image map

L'idea è di avere sul client un'immagine sensibile al click del mouse che possa fare da "ponte" verso nuove pagine HTML scelte sulla base del punto dell'immagine sul quale si preme il pulsante.

Sul server sono necessari:

- una immagine (ad esempio, image.gif) e una pagina HTML che la contiene;
- un file di testo detto ***mappa*** che definisce le parti dell'immagine sensibili al mouse (ad esempio, image.map);
- un programma CGI di servizio (ad esempio, image.cgi).

Il funzionamento è il seguente:

- l'utente fa click su un punto dell'immagine;
- il client invia le coordinate del punto di click al server;
- il server passa le coordinate al programma CGI;
- il programma GCI esamina la mappa e, sulla base delle coordinate ricevute, restituisce un opportuno documento.

Supponiamo che:

- l'immagine image.gif sia di 100x100 pixel;
- il file image.map contenga le linee:

```
rect /paginaA.html 5,5,95,45
rect /paginaB.html 5,55,95,95
```

Tali linee impongono la seguente struttura (invisibile) sull'immagine image.gif:

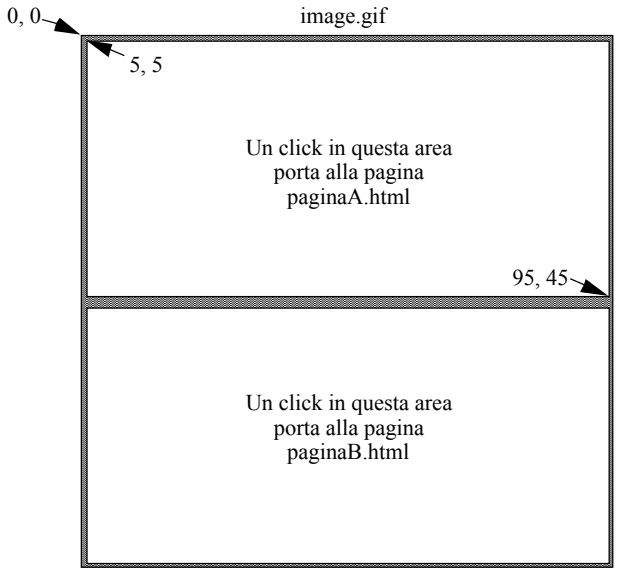


Figura 1-5: Aree sensibili definite da image.map

Nella pagina di partenza l'ancora che contiene l'immagine sensibile è del tipo:

```
<A HREF="image.cgi$image.map"><IMG SRC="image.gif" ISMAP></A>
```

Quando l'utente fa click, ad esempio, nel punto 50, 20 dell'immagine (l'origine è in alto a sinistra), il client invia la richiesta:

```
GET image.cgi$image.map?50,20 HTTP/1.0
```

Il server lancia il programma `image.cgi`, che riceve il nome del file di mappa e le coordinate, e quindi restituisce (tramite URL redirection) il file paginaA.html da mostrare al client.

Client Side Image Map
Recentemente è stato introdotto un nuovo meccanismo per ottenere un comportamento analogo senza bisogno di interagire col server.

Il client fa tutto da solo, poiché trova le informazioni necessarie direttamente nella pagina HTML.

In sostanza, si definisce la mappa in HTML e si dice al client di usare direttamente quella. Il caso precedente diventa:

```
<IMG SRC="image.gif" USEMAP="#localmap">
<MAP NAME="localmap">
<AREA COORDS="5,5,95,45" HREF="paginaA.html">
<AREA COORDS="5,55,95,95" HREF="paginaB.html">
</MAP>
```

Facendo click su una delle due aree, il client invia direttamente la richiesta della pagina A o B.

Per garantire la compatibilità con i client più vecchi, che non gestiscono il parametro `USEMAP`, si possono far coesistere le due possibilità:

```
<A HREF="image.cgi$image.map"><IMG SRC="image.gif" ISMAP USEMAP="#localmap"></A>
<MAP NAME="localmap">
<AREA COORDS="5,5,95,45" HREF="paginaA.html">
<AREA COORDS="5,55,95,95" HREF="paginaB.html">
</MAP>
```

Motori di ricerca
Esistono dei siti che offrono funzioni di ricerca di informazioni sull'intera rete Internet. In generale essi sono costituiti di 3 componenti:

- un ***robot di ricerca***, che è un programma che esplora automaticamente il Web e raccoglie informazioni sui documenti che trova (tipicamente, analizzando TITLE e HEAD). Con tali informazioni alimenta la base di dati di cui al punto successivo;
- una ***base di dati*** che raccoglie i riferimenti alle pagine collezionate, con opportune strutture di indici per velocizzare le ricerche;
- un ***CGI di interfaccia*** fra il Web e il programma di gestione della basi di dati, e una o più form per l'immissione dei criteri di ricerca da parte degli utenti.

1.3.3) Linguaggio JavaScript (già LiveScript)

E' una proposta di Netscape. Consiste nella definizione di un completo linguaggio a oggetti, con sintassi analoga a Java, con il quale si possono definire delle computazioni direttamente nel testo della pagina HTML. Tali computazioni vengono poi eseguite dal browser.

Ad esempio, si può usare JavaScript per controllare che i vari campi di una form siano riempiti in modo sintatticamente corretto prima di inviare i dati al server.

Nell'esempio che segue, il bottone "submit" chiama una routine JavaScript di controllo del campo cognome. Se esso è vuoto si avvisa l'utente dell'errore, altrimenti si invia la form.

```
<HTML>
```

```
<HEAD>
<SCRIPT LANGUAGE="LiveScript">
function checkForm (form){
       if (form.cognome.value == "") {
              alert("Errore: il campo \"Cognome\" non puo\' essere vuoto");
              form.cognome.focus();
              form.cognome.select();
       } else {
              form.submit();
       }
}
</SCRIPT>
</HEAD>

<BODY>
<CENTER>
<FORM>
Cognome:<INPUT NAME="cognome" VALUE="">
<P>
<INPUT TYPE="button" VALUE="Submit"
       onClick="checkForm (this.form);"
>
<INPUT TYPE="reset">
</FORM>
</CENTER>
</BODY>
</HTML>
```


L'effetto di un errore di immissione, dal punto di vista dell'utente, è il seguente:

Figura 1-6: Messaggio di errore di JavaScript

Si noti che all'utente il bottone di invio sembra di tipo Submit, ma in realtà è di tipo Button, altrimenti la pressione su di esso causerebbe comunque l'invio dei dati della form.

Il linguaggio consente:

- la definizione di funzioni;
- la definizioni di variabili (non tipate: il tipo si deduce dall'uso);
- la gestione degli eventi, fra cui:

    - onClick;
    - onFocus, onBlur;
    - onChange;
- l'accesso a tutti gli oggetti, automaticamente istanziati, relativi al contenuto della pagina;
- la comunicazione bidirezionale con gli applet Java presenti nella pagina.

Gli oggetti predefiniti sono organizzati in una gerarchia i cui principali elementi sono:
- Document
    - Anchor
    - Applet
    - Form
        - Button
        - Checkbox
        - Radio
        - Submit
    - Textarea
    - Image
    - Link

Ogni singolo oggetto contiene una istanza di ciascun tipo di oggetto in esso contenuto: ad esempio, un oggetto Form contiene le istanze di tutti i campi immissione dati, bottoni ecc. che sono inclusi nella form stessa.

1.3.4) Linguaggio Java

Anche disponendo delle form e dei programmi CGI, ciò che rimane impossibile è avere una rapida interazione con la pagina Web.

Inoltre, la vertiginosa apparizione di nuove tipologie di informazioni da gestire e nuovi tipi di servizi da offrire rende problematico l'aggiornamento dei client: troppo spesso si devono ampliare le loro funzionalità o si deve ricorrere a programmi helper e plug-in.

La prima risposta a entrambi questi problemi è venuta da una proposta della Sun che ha immediatamente suscitato un enorme interesse a livello mondiale: il linguaggio Java e il relativo interprete.

Originariamente (nei primi anni '90) il linguaggio si chiamava OAK ed era destinato alla produzione di software per elettrodomestici (orientati all'informazione) connessi in rete. Era progettato per:
- essere indipendente dalla piattaforma di calcolo;
- consentire una esecuzione senza rischi per l'attrezzatura di destinazione.

Successivamente il progetto fu riorientato verso il Web, il linguaggio ha preso il nome di Java ed ha conosciuto un enorme successo.

L'idea di fondo è molto semplice:
- una pagina Web contiene (fra le altre cose) un link a un programma scritto in Java, detto ***applet***;
- quando l'utente carica tale pagina, assieme ad essa viene recuperato il codice eseguibile (***bytecode***) dell'applet;
- dopo essere stato caricato, del tutto o in parte, l'applet viene mandato in esecuzione sul client "dentro" la pagina HTML.

In questo modo si ottiene di fatto la possibilità di estendere senza limiti la funzionalità del Web, infatti:
- se l'applet ha una finalità prettamente locale (gioco, applicazione specifica quale un word processor, tabellone elettronico, ecc.) si offre quella alta interattività che prima mancava al Web;
- viceversa l'applet può costituire una estensione delle funzionalità del browser. Ad esempio, l'inventore di un nuovo formato per la compressione dati (chiamiamolo MPEG-2000) scriverà un applet per la visualizzazione degli stessi, che sarà caricato dai client quando questi dovranno accedere a tali dati.

La piattaforma di calcolo basata su Java consiste di varie componenti:
- il linguaggio Java;
- un ***compilatore*** da Java a bytecode;
- un interprete di bytecode, ossia una ***macchina virtuale Java*** (***Java Virtual Machine***, ***JVM***).

Il linguaggio
E' principalmente caratterizzato da queste caratteristiche:
- general purpose;
- orientato a oggetti;
- fortemente tipato;
- concorrente (multithreading);
- privo di caratteristiche dipendenti dalla piattaforma;
- dotato di caratteristiche simili a SmallTalk;
- dotato di sintassi simile a C e C++.

Il compilatore
Traduce i file sorgenti scritti in Java (aventi l'estensione `.java`) in codice eseguibile, contenuto in uno o piu file caratterizzati dall'estensione `.class`.

Sono questi i file che viaggiano sulla rete e vengono caricati dai client Web per l'esecuzione.

I file `.class` hanno un formato indipendente dalla piattaforma:
- stream di byte (8-bit);

- quantità multibyte memorizzate in big-endian order.

La macchina virtuale
E' il cuore di tutto il meccanismo. Essa fornisce un dispositivo di calcolo astratto, capace di eseguire il codice contenuto nei file `.class` e dotato di:
- un set di istruzioni macchina;
- un insieme di registri, un program counter, ecc.

Si noti che attualmente i file `.class` si ottengono a partire da codice Java, ma questo non fa parte delle specifiche della macchina virtuale. Non è escluso che in futuro si possano usare anche altri linguaggi sorgente (con opportuni compilatori).

Per eseguire i bytecode su una macchina reale, bisogna realizzare un ***emulatore*** della macchina virtuale, il cui compito è tradurre i bytecode in istruzioni native della piattaforma HW/SW ospite.

Dunque, sono le varie implementazioni della macchina virtuale che offrono ai bytecode l'indipendenza dalla piattaforma.

Vari modi sono possibili per implementare emulatori della macchina virtuale Java:
- all'interno di un client Web: in questo caso l'emulatore della macchina virtuale risiede all'interno del codice eseguibile del client;
- applicazione a se stante: in questo caso il client viene configurato per usare la JVM come helper esterno (ad esempio, MS Internet Explorer può scegliere quale JVM usare);
- all'interno del S.O. (per ora con Linux, a breve anche con Windows e MacOS).

Nel secondo e terzo caso, la JVM è in grado di eseguire non soltanto gli applet, ma anche applicazioni vere e proprie scritte in Java, cioè programmi del tutto autonomi che possono risiedere permanentemente sulla macchina ospite e possono essere eseguiti senza utilizzare il client Web.

Tali programmi non sono soggetti alle restrizioni imposte (per ragioni di sicurezza, come vedremo poi) agli applet.

Sono in corso di sviluppo progetti volti a realizzare un chip che implementi la JVM direttamente in hardware, con ovvi benefici dal punto di vista delle prestazioni.

Funzionamento nel Web
In una pagina HTML si fa riferimento a un applet con uno specifico tag:

```
<APPLET CODE="game.class" WITDH=200 HEIGHT=200></APPLET>
```


E' possibile fornire all'applet anche ulteriori parametri, che vengono passati all'applet al momento del lancio, quali ad esempio:

```
<APPLET CODE="game.class" WITDH=200 HEIGHT=200>
<PARAM NAME="level" VALUE="5">
<PARAM NAME="weapons" VALUE="none">
</APPLET>
```

Quando il client incontra il tag Applet:
- riserva uno spazio di 200x200 pixel nella pagina;
- richiede con una GET il file `game.class`;
- quando riceve il file lo passa alla JVM (interna o esterna) che:
  - effettua gli opportuni controlli (come vedremo in seguito);
  - se è tutto OK, avvia l'esecuzione dell'applet;
- mostra nello spazio riservato nella pagina (200x200 pixel nell'esempio) il contesto di esecuzione dell'applet;
- provvede a richiedere (su indicazione della JVM) gli ulteriori file `.class` che sono necessari all'esecuzione dell'applet.

Spesso, a corredo di un emulatore di JVM è consegnato un ***AppletViewer***, che è un programma di supporto volto a fornire alla JVM la possibilità di reperire attraverso il protocollo HTTP i file `.class`. In sostanza, un AppletViewer può essere visto come un client Web che sa gestire solamente il tag APPLET e ignora tutto il resto.

1.4) I problemi del Web

Il protocollo HTTP è nato con una finalità (lo scambio di informazioni fra gruppi di ricercatori) che si è radicalmente modificata col passare del tempo.

Infatti, attualmente:
- il suo utilizzo si è ampliato enormemente;
- si cerca di sfruttarlo anche per transazioni di tipo commerciale.

Questi due aspetti hanno messo alla frusta il protocollo soprattutto da tre punti di vista:
- efficienza nell'uso delle risorse di rete;
- assenza del concetto di sessione;
- carenza di meccanismi per lo scambio di dati riservati.

Efficienza nell'uso della rete
Il problema di fondo è che il protocollo HTTP prevede che si apra e chiuda una connessione TCP per ogni singola coppia richiesta-risposta.

Si deve tenere presente che:
- l'apertura e la chiusura delle connessioni TCP non è banale (***three way handshake*** per l'apertura, idem più timeout per la chiusura);
- durante l'apertura e la chiusura non c'è controllo di flusso;
- la dimensione tipica dei dati trasmessi è piccola (pochi KByte);
- TCP usa la tecnica dello ***slow start*** per il controllo di flusso.

Di conseguenza, si ha che:
- il protocollo HTTP impone un grave overhead sulla rete (sono frequentissime le fasi di apertura / chiusura di connessioni TCP);
- gli utenti hanno la sensazione che le prestazioni siano scarse (per via dello slow start).

Assenza del concetto di sessione
In moltissime situazioni (ad esempio nelle transazioni di tipo commerciale) è necessario ricordare lo stato dell'interazione fra client e server, che si svolge per fasi successive.

Questo in HTTP è impossibile, per cui si devono trovare soluzioni ad hoc (ad esempio, restituire pagine HTML dinamiche contenenti un campo HIDDEN che identifica la transazione).

In proposito c'è una proposta (***State Management***, rfc 2109) basata sull'uso dei ***cookies***, cioè particolari file che vengono depositati presso il client e che servono a gestire una sorta di sessione.

Carenza di meccanismi per lo scambio di dati riservati
L'unico meccanismo esistente per il controllo degli accessi (***Basic Authentication***) fa viaggiare la password dell'utente in chiaro sulla rete, come vedremo fra breve.

HTTP versione 1.1
E' in corso di completamento la proposta, che diventerà uno standard ufficiale di Internet, della versione 1.1 di HTTP.

Essa propone soluzioni per tutti gli aspetti problematici che abbiamo citato ed introduce ulteriori novità. I suoi aspetti salienti sono:
- il supporto per connessioni persistenti;
- il supporto per l'autenticazione;
- l'introduzione dei metodi estensibili.

2) La sicurezza

In generale, la sicurezza ha a che fare con i seguenti aspetti:
- controllo del diritto di accesso alle informazioni;
- protezione delle risorse da danneggiamenti (volontari o involontari);
- protezione delle informazioni mentre esse sono in transito sulla rete;
- verifica che l'interlocutore sia veramente chi dice di essere.

La rete Internet è nata con la finalità originaria di offrire un efficace strumento per lo scambio di informazioni fra gruppi di ricercatori sparsi per il mondo. Di conseguenza le problematiche relative alla sicurezza non sono state prese in considerazione nel progetto dell'architettura TCP / IP, né tantomeno in quello dei protocolli di livello application.

L'interesse mostrato da chi sfrutta commercialmente Internet, però, sta cambiando la tipologia di utilizzo della rete, e i problemi legati alla scarsa sicurezza diventano sempre più pesanti, per cui ci sono molte attività in corso (compresa la riprogettazione di alcuni protocolli fondamentali quali IP) per incorporare nell'architettura meccanismi di sicurezza.

Nel caso specifico del Web, il fatto che esso sia nato come sistema aperto e disponibile a tutti lo rende particolarmente vulnerabile dal punto di vista della sicurezza (ovviamente, un sistema chiuso è piu facile da proteggere). Ciò nonostante, alcuni meccanismi sono disponibili e verranno brevemente descritti nel seguito.

2.1) Controllo dei diritti di accesso

2.1.1) Basic authentication in HTTP 1.0

Nel protocollo HTTP è presente un servizio detto ***Basic Authentication*** per fornire selettivamente l'accesso a informazioni private, sulla base di un meccanismo di gestione di password.

Sul server si mantengono, in opportuni file di configurazione:
- una lista di ***realm***, ossia di porzioni del file system gestito dal server Web per accedere alle quali ci vuole un permesso;
- per ogni realm, una lista degli utenti abilitati con le relative password.

Un realm è di fatto una stringa di testo. Tutti i documenti la cui URL contiene quella stringa fanno parte del realm.

Quando arriva una richiesta GET per un documento che appartiene a un realm, il server non restituisce il documento, ma un messaggio come questo:

```
HTTP/1.0 401 Unauthorized
WWW-Authenticate: Basic realm="NomeRealm"
Server: .....
Date: .....
Content-type: .....
Content-length: 0
```

Quando il client riceve questo messaggio, fa apparire automaticamente una finestra di dialogo predisposta per l'immissione di una ***username*** e di una ***password***.

L'utente immette i dati, e poi preme OK. A questo punto il client invia una nuova richiesta al server, simile alla seguente:

```
GET url(la stessa di prima) HTTP/1.0
Authorization: Basic ****************
User-agent: .....
Accept: .....
ecc.
```

dove il testo rappresentato con gli asterischi contiene la username e la password immesse dall'utente, codificate con il metodo ***base64 encoding*** (uno standard del mondo Unix, definito negli rfc 1341 e 1521, che non costituisce una cifratura ma serve solo a poter trasmettere caratteri ASCII estesi).

Quando il server riceve la richiesta, applica l'algoritmo di decodifica alla stringa di username-password, le confronta con quelle in suo possesso per il realm `NomeRealm` e:

- se è tutto OK restituisce il documento richiesto;
- altrimenti, restituisce un messaggio di errore (`403 forbidden`).

Il client di norma ricorda in una cache la coppia username-password immessa dall'utente e la utilizza anche per i successivi accessi a documenti dello stesso realm; il server deve comunque decodificare tale coppia ogni volta che arriva e verificarne la corrispondenza con quelle in suo possesso.

2.1.2) Digest authentication in HTTP 1.1

Il problema con questo approccio è che username e password di fatto viaggiano in chiaro sulla rete, dato che gli algoritmi usati per la codifica e la decodifica sono noti a tutti, e quindi può essere intercettata.

In proposito c'è una proposta (***Digest Authentication***, rfc 2069) per istituire un meccanismo di cifratura di username e password basato sul meccanismo di ***Message***

Digest, una sorta di funzione hash facile da calcolare ma difficile da invertire (la vedremo più avanti).

Questo protocollo è di tipo ***challenge-response*** dove:

- il challenge, inviato dal server al client, contiene un valore detto ***nonce*** che è ogni volta diverso;
- la response, inviata dal client al server, è il Message Digest (calcolato con l'algoritmo ***MD5***) di:
  - nonce ricevuto dal server;
  - username dell'utente;
  - password dell'utente.

In questo modo i dati riservati dell'utente (username e password) non viaggiano mai in chiaro sulla rete.

Quando il server riceve il Message Digest dal client, effettua anch'esso un identico calcolo e confronta i due valori. Se sono uguali è tutto OK, altrimenti no.

2.1.3) Firewall

In molte situazioni in cui esiste una rete aziendale connessa con una rete esterna (ad esempio Internet), può sorgere la necessità di:

- proteggere le informazioni riservate (ad esempio piani strategici, dati finanziari, ecc.) da accessi provenienti dall'esterno della rete aziendale, consentendo solo l'accesso a informazioni pubbliche (ad esempio listino prodotti);
- limitare l'accesso, da parte degli elaboratori posti sulla rete aziendale, alle informazioni presenti sulla rete esterna.

Questo si può ottenere per mezzo di un ***firewall*** (parete tagliafuoco), che è l'incarnazione moderna del fossato pieno d'acqua (e magari anche di coccodrilli) e del ponte levatoio che proteggevano gli antichi castelli.

Il principio è lo stesso:

- forzare il passaggio di tutto ciò che transita (esseri umani nell'antichità, traffico di rete oggi) attraverso un unico punto di ingresso e uscita, dove si provvede ad effettuare gli opportuni controlli.

Il firewall si inserisce fra la rete aziendale e quella esterna. In tal modo, tutto il traffico dall'una all'altra parte deve per forza transitare attraverso il firewall.

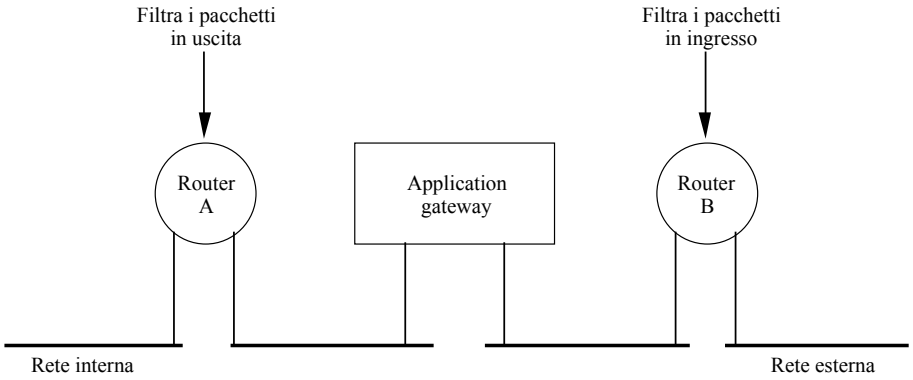


Figura 2-1: Tipica configurazione di un firewall

Esistono molte configurazioni di firewall, più o meno raffinate. In quella sopra illustrata si ricorre a due tipi di componenti:

- due ***router*** che filtrano i pacchetti (A filtra in uscita, B filtra in ingresso): ogni pacchetto in transito viene esaminato secondo criteri opportunamente impostati; se li soddisfa viene lasciato passare, altrimenti no;
- un ***application gateway***: questa componente opera a livello application, e quindi entra nel merito del contenuto dei dati in transito. Ad esempio, un ***mail gateway*** può decidere se lasciar passare un messaggio di posta elettronica sulla base di subject, o destinatario, o addirittura esaminando il contenuto del messaggio (ad esempio, se c'è la parola "bomb" lo ferma).

Criteri tipici di filtraggio dei pacchetti, che possono anche essere combinati fra loro, sono:

- indirizzo IP (o range di indirizzi) di partenza o di destinazione: questo può servire quando si vogliono connettere fra loro due reti aziendali remote attraverso una rete esterna, ottenendo una cosiddetta ***extranet***;
- numero di port di destinazione: questo può servire per abilitare certi servizi e altri no (ad esempio, si disabilita in ingresso il port 23: nessuno dalla rete esterna può fare login su un elaboratore della rete interna). Un problema in proposito è che alcuni servizi (come il Web) sono spesso offerti anche su porte non standard;
- tipo di connessione usata: è abbastanza comune bloccare tutto il traffico UDP, più difficile da analizzare.

Possono esistere vari application gateway, uno per ogni protocollo di livello application che si vuole controllare.

Spesso, per semplificare il controllo, si installa sulla rete interna un ***server proxy*** (che può anche coincidere col gateway), cioè un oggetto che agisce da intermediario fra i clienti della rete interna ed i serventi della rete esterna.

Ad esempio, nel caso di un server proxy per il protocollo HTTP (***HTTP proxy***) si ha tipicamente una situazione come questa:

- i client della rete interna vengono configurati in modo da fare riferimento all'HTTP proxy;
- il firewall viene configurato per lasciar transitare il traffico HTTP da e per l'HTTP proxy.

Quando un utente attiva un link che punta a un server Web della rete esterna succede questo:

1. il client apre una connessione col proxy e gli invia la richiesta;
2. il proxy (che può passare dal firewall) apre una connessione con il server Web esterno e gli invia la richiesta del client;
3. il server Web esterno invia la risposta al proxy;
4. il proxy "gira" la risposta al client.

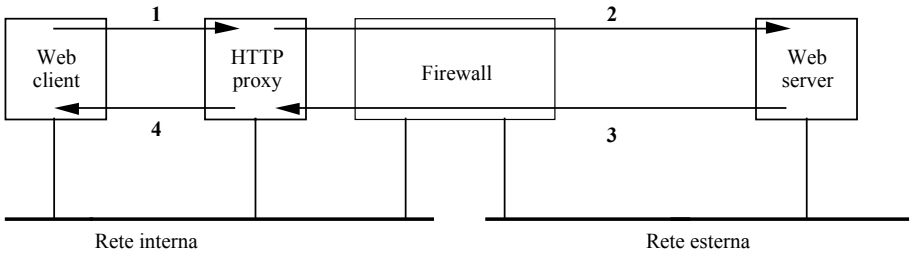


Figura 2-2: Uso di un HTTP proxy

I proxy server hanno anche una funzione di ***caching*** delle pagine più recenti, in modo da poterle offrire immediatamente se vengono richieste più di una volta, aumentando così l'efficienza e diminuendo l'uso di banda trasmissiva.

Infine, un'ultima nota: se l'azienda vuole pubblicare all'esterno sue informazioni (ad esempio con un server Web) basterà che collochi tale server all'esterno del firewall.

2.2) Protezione delle risorse da danneggiamento

Di norma i server Web non accettano altri metodi che GET (e POST in relazione alle form), quindi impediscono operazioni pericolose quali la scrittura o la cancellazione di file. Inoltre, di norma i server Web non considerano legali le URL che fanno riferimento a porzioni del file system esterne alla parte di competenza del server Web stesso.

Dunque, per lo meno quando si chiedono al server servizi standard (per il recupero di pagine Web) non ci sono grandi pericoli.

2.2.1) La sicurezza e le estensioni del Web

Il discorso però cambia completamente quando si allargano le funzionalità rese disponibili:
- sul server, con programmi CGI;
- sul client, con applicazioni helper.

In entrambi i casi, le opportunità per azioni che causano danneggiamenti travalicano le possibilità di controllo di client e server, e dipendono esclusivamente dalle caratteristiche dei programmi esterni. Si possono aprire delle voragini nella sicurezza!

Essenzialmente, si può dire questo:
- più è potente il programma (helper sul client e CGI sul server) e maggiori sono i pericoli ai quali si è esposti.

Lato client
Supponiamo che l'utente abbia configurato il suo client per lanciare un ***interprete PostScript*** quando riceve un file PostScript, che di norma contiene un insieme di comandi per la formattazione di testo e grafica indipendenti dalla piattaforma.

In questo scenario, l'interprete PostScript viene lanciato ed esegue uno a uno i comandi contenuti nel file, mostrando sul video (o stampando) il documento.

Ora, PostScript è un completo linguaggio di programmazione, e contiene anche comandi per operazioni sul file system. Se l'autore del documento PostScript ha sfruttato tali potenzialità per recare danni, l'utente ne sopporterà le conseguenze: ad esempio, il file PostScript potrebbe contenere delle istruzioni che cancellano tutti i file dal disco rigido dell'elaboratore.

Lato server
Uno scenario tipico, come abbiamo già detto, è questo:
- il programma CGI compone, in base ai dati immessi nei campi della form, un comando destinato ad un altro programma esterno;
- passa il comando a tale programma esterno e gli chiede di eseguirlo.

Abbiamo visto che nel caso di una interrogazione a una base dati:
- il comando è la formulazione di una query;
- il programma esterno è il gestore della base dati.

Ora, se invece:
- il programma esterno è molto potente (ad esempio: la ***shell***);
- il programma CGI non entra a sufficienza nel merito del comando che viene costruito;

- il server Web ha i privilegi di ***root***, e lancia con tali privilegi anche il programma CGI (e quindi, di riflesso, anche la shell);

allora si corrono enormi rischi: un utente remoto può inviare un comando per distruggere dei file, ricevere il file delle password, eccetera.

Alcune delle possibili precauzioni per evitare le situazioni sopra descritte sono le seguenti:
- il server in ascolto sulla porta 80 deve girare come root (altrimenti non può aprire un socket su nessuna well-known port), ma i suoi figli (o i thread) che gestiscono le singole richieste devono avere i minimi privilegi necessari per poter svolgere il loro compito (e questo vale anche per i programmi CGI ed i programmi esterni);
- i programmi CGI devono controllare la potenziale pericolosità di ogni comando che ricevono prima di consegnarlo al programma esterno;
- il programma esterno deve essere il meno potente possibile: ad esempio la shell è certamente da evitare, se possibile.

2.2.2) La sicurezza e Java

Una grande attenzione è stata posta, nel progetto del linguaggio e della JVM, ai problemi di sicurezza derivanti potenzialmente dal fatto di mandare in esecuzione sulla propria macchina un codice proveniente da una fonte ignota (e perciò non affidabile in linea di principio).

Ad esempio, si potrebbero ipotizzare questi scenari certamente indesiderabili:
- un applet cifra tutti i file del disco, e chi lo ha programmato chiede un riscatto per fornire la chiave di decifratura;
- un applet ruba informazioni riservate e le invia a qualcun altro;
- un applet cancella tutti i file del disco.

La prima linea di difesa è stata incorporata nel linguaggio, che è:
- fortemente tipato;
- con controlli sui limiti degli array;
- senza puntatori.

In tal modo è impossibile accedere a zone di memoria esterne a quelle allocate all'applet.

Tuttavia, ***Trudy*** (un personaggio che conosceremo di più in seguito) si diverte a modificare un compilatore C per produrre dei bytecode in modo da aggirare i controlli effettuati dal compilatore Java.

Per questa ragione, la JVM offre la seconda linea di difesa sotto forma di una componente, detta ***bytecode verifier***, che effettua numerosi controlli sui bytecode prima di mandarli in esecuzione, verificando ad esempio che non si cerchi di:

- costruire puntatori;
- chiamare metodi con parametri non validi;
- usare variabili non inizializzate.

La terza linea di difesa è rappresentata dal ***class loader***, il meccanismo di caricamento delle classi. Esso impedisce, ad esempio, che una classe dell'applet vada a sostituirsi a una delle classi di sistema in modo da aggirare i meccanismi di sicurezza di quest'ultima.

Infine, un'ulteriore linea di difesa è il ***security manager***, una classe che ha il compito di stabilire dei limiti a ciò che il programma (applet o application) può fare.

In particolare, di norma i client Web (e gli AppletViewer) caricano all'avvio un security manager che impedisce a tutti gli applet di:
- accedere al file system locale;
- aprire connessioni di rete con host diversi da quello di provenienza;
- lanciare altri programmi.

Viceversa, un'applicazione Java viene avviata di norma senza alcun security manager associato (a meno che non venga programmata diversamente), e quindi non ha alcuna delle limitazioni sopra citate.

2.3) Protezione delle informazioni durante il transito sulla rete

Esistono ulteriori problemi legati alla sicurezza, che non si possono risolvere semplicemente con meccanismi basati sull'uso di password.

Essi possono essere divisi nelle seguenti aree, collegate fra loro:
- ***segretezza***: si desidera inviare delle informazioni riservate, in modo che solo il destinatario sia in grado di leggerle;
- ***autenticazione del mittente***: si vuole essere sicuri che colui col quale si dialoga sia veramente chi dice di essere;
- ***integrità del messaggio***: si vuole esseri sicuri che il messaggio che arriva non sia stato manomesso durante il viaggio.

Nel campo dell'informatica e soprattutto delle reti, dove da un lato è possibile facilmente creare copie (e anche modificarle) di documenti e dall'altro non si può escludere che Trudy (ovvero un intruso) intercetti le informazioni in transito sulla rete, tutto è più difficile che nella vita quotidiana, dove esistono al proposito meccanismi consolidati (buste sigillate, documenti di identità, autenticazione dei documenti).

E ciò è soprattutto vero in una rete come Internet, dove:
- esiste la necessità di dialogare con entità precedentemente sconosciute;

- ci sono potenzialmente in ogni momento molte Trudy all'ascolto, pronte a rubare informazioni e a sfruttarle a proprio vantaggio.

I problemi precedenti possono essere risolti con un ***protocollo crittografico***, che consiste in una serie di passi nei quali si utilizzano le tecnologie seguenti:
- ***crittografia***;
- ***firme digitali***.

2.3.1) Crittografia

La crittografia (o scrittura nascosta) è la disciplina che si occupa di studiare le tecniche per scrivere un messaggio in modo tale che solo il legittimo destinatario sia in grado di leggerlo. Si occupa dunque del problema della segretezza.

I requisiti principali di tale tecnica sono:
- ragionevole efficienza nella creazione del messaggio;
- estrema difficoltà nella interpretazione del messaggio da parte di chi non è autorizzato;
- possibilità di cambiare con estrema rapidità il metodo usato.

Una prima possibilità è stabilire un metodo di trasformazione (***cifratura*** o ***encryption***) del messaggio originale e un corrispondente metodo di interpretazione (***decifratura*** o ***decryption***) che vengono tenuti gelosamente segreti, e sono noti solo alle persone autorizzate.

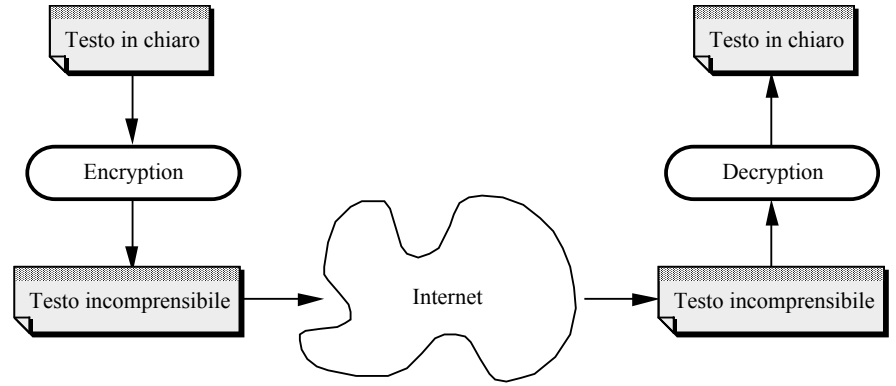


Figura 2-3: Cifratura e decifratura

Ad esempio, un banale metodo di trasformazione (segreto) può essere il seguente: sostituire ogni carattere con quello che, nell'alfabeto, lo segue immediatamente (con wrap-around).

Questo approccio però non soddisfa il terzo requisito, perché per cambiare metodo lo si deve riprogettare completamente. Per questo motivo, si ricorre invece a uno schema diverso, che consiste in:

- un ***metodo*** di cifratura e uno di decifratura che sono noti a tutti, ma sono parametrizzati da una chiave che deve essere data loro in input assieme al messaggio;
- una sequenza di bit, detta ***chiave***, che è nota solo alle persone autorizzate.

Di fatto il metodo di cifratura è una ***funzione*** E che accetta in ingresso il ***testo in chiaro*** (***plaintext***) P e una chiave k, producendo il ***testo cifrato*** (***ciphertext***) C:

$$C = E(P,k)$$

e che normalmente si indica come:

$$C = E_k(P)$$

Quindi, per ogni valore possibile della chiave si ottiene un diverso metodo di cifratura.

A titolo di sempio:

- metodo di cifratura (pubblico): sostituire ogni carattere con quello che lo segue a distanza k (con wrap-around);
- chiave (segreta): il valore k.

Il metodo di decifratura è un'altra funzione (ovviamente collegata alla prima) che accetta in ingresso il testo cifrato C, una chiave k e restituisce il testo in chiaro originale P:

$$P = D_k(C)$$

Ovviamente, si dovrà avere che:

$$D_k(E_k(P)) = P$$

Come vedremo, non è detto che si debba usare la stessa chiave nelle due fasi.
Il modello risultante è il seguente:

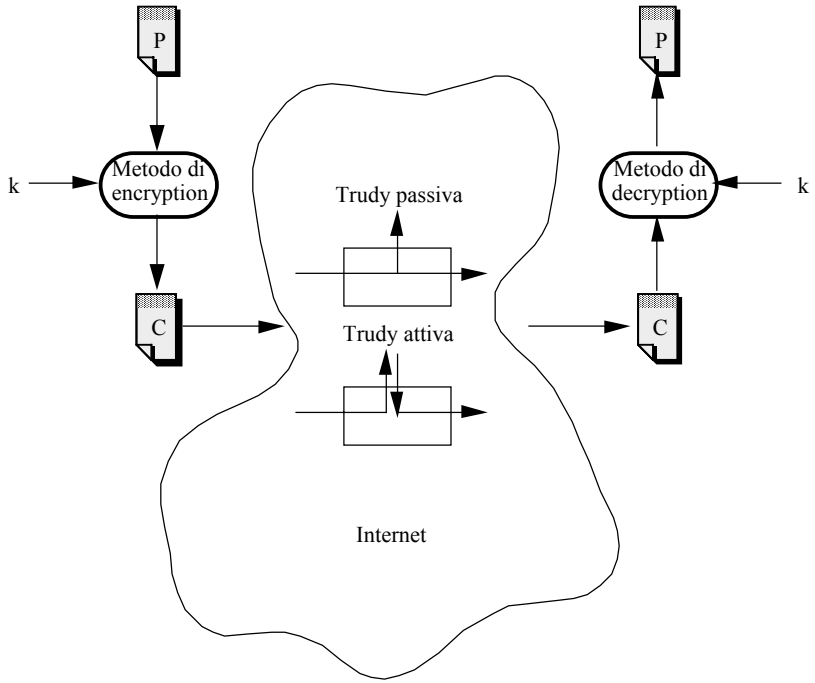


Figura 2-4: Cifratura e decifratura basate su chiave

Trudy può essere passiva (ascolta solamente) o attiva (ascolta ed altera i messaggi che riesce ad intercettare).

La crittografia, come abbiamo detto, si occupa di trovare buoni metodi per effettuare la cifratura e la decifratura. La ***cryptoanalisi*** (***cryptoanalysis***) invece si occupa di scoprire modi per decifrare i messaggi pur non conoscendo le regole note alle persone autorizzate (in particolare, la chiave).

Questa operazione può essere portata avanti in due modi:

- provare ad applicare al testo cifrato il metodo di decifratura con tutti i possibili valori della chiave. Questo approccio, detto di ***forza bruta***, produce certamente il testo in chiaro originale prima o poi, ma è ovviamente molto oneroso, ed anzi lo è tanto più quanto è lunga la chiave (ad esempio con 128 bit di chiave, ci sono 2^{128} prove da effettuare);
- scoprire le eventuali debolezze delle funzioni E e D, per ridurre lo spazio delle chiavi da esplorare.

Ci sono due principi fondamentali che ogni metodo di cifrature deve rispettare:

- i messaggi originali devono contenere della ridondanza: se così non fosse, ogni possibile messaggio cifrato sarebbe comunque valido e Trudy potrebbe quindi inviarne a piacere, dato che a destinazione essi verrebbero considerati autentici dopo essere stati decifrati;
- i messaggi originali devono contenere qualcosa (ad esempio un ***time stamp***) che impedisca a Trudy di rispedire nuovamente un messaggio valido. Altrimenti, ogni volta che Trudy intercetta un messaggio puo inviarlo nuovamente per i propri scopi.

2.3.1.1) Crittografia a chiave segreta (o simmetrica)

In questo tipo di crittografia, il mittente e il destinatario (Alice e Bob) si accordano, ad esempio incontrandosi di persona lontano da occhi indiscreti, su una singola chiave che verrà usata sia in fase di cifratura che di decifratura.

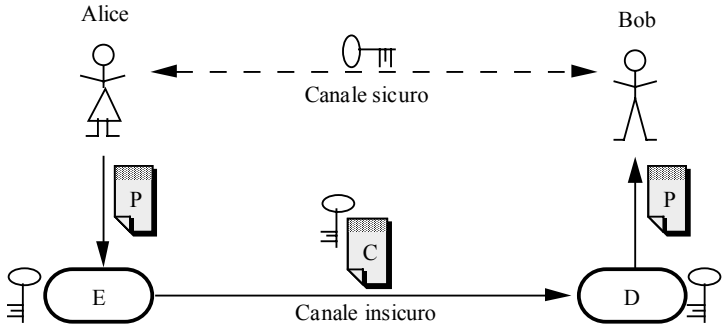


Figura 2-5: Crittografia a chiave segreta

L'algoritmo più diffuso in questa categoria è il ***DES*** (***Data Encryption Standard***), inventato dall'IBM e adottato come standard del governo U.S.A. nel 1977.

Il testo in chiaro è codificato in blocchi di 64 bit, che producono ciascuno 64 bit di testo cifrato (***cifratura a blocchi***).

L'algoritmo è parametrizzato da una chiave di 56 bit e consiste di ben 19 stadi, in ciascuno dei quali si opera una trasformazione dell'output dello stadio precedente.

Inoltre, in 16 dei 19 stadi la trasformazione effettuata è funzionalmente identica, ma è parametrizzata da opportune trasformazioni della chiave.

Il DES è stato al centro di controversie sin dal giorno in cui è nato:
- il progetto originale IBM prevedeva chiavi da 128 bit invece che da 56 bit, ma i militari U.S.A. "suggerirono" attraverso l'***NSA*** (***National Security Agency***, detta anche

malignamente ***No Such Agency***) tale riduzione. Secondo molti, la riduzione fu motivata dall'esigenza di mantenere la capacità (con opportune potenti macchine) di rompere il codice;
- oggi il DES non è più considerato sicuro, in quanto recenti tecniche di ***criptanalisi differenziale*** hanno ridotto lo spazio di ricerca a 2^{43} possibilità;

Una sua variante, il ***Triple DES***, è però a tutt'oggi considerato sicuro, in quanto non si conosce alcun modo di romperlo. Il meccanismo è il seguente.

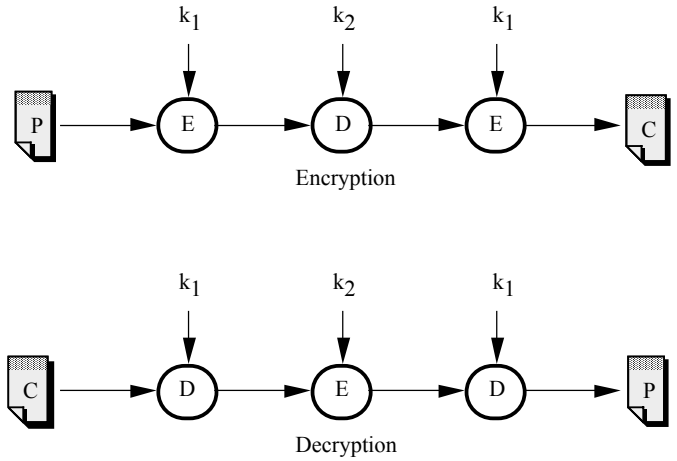


Figura 2-6: Triple DES

Questo schema, ponendo $k_1=k_2$, garantisce la compatibilità all'indietro col normale DES.

Effettivamente il Triple DES costituisce un codice per il quale l'approccio della forza bruta richiede 2^{112} tentativi: anche con un miliardo di chip che effettuano un miliardo di operazioni al secondo, ci vorrebbero 100 milioni di anni per la ricerca esaustiva.

Il DES funziona alla velocità di 2,5 MBps su un Pentium Pro a 200 MHz, e fino a 64 MBps su hardware specializzato.

Un altro importante algoritmo a chiave segreta è ***IDEA*** (***International Data Encryption Algorithm***).

Esso fu progettato nel '90 in Svizzera, e per questa ragione non è soggetto alle limitazioni sull'uso e sull'esportazione che esistono in U.S.A. (dove gli algoritmi di cifratura sono a tutti gli effetti di legge considerati armi da guerra).

Come il DES, IDEA effettua una cifratura a blocchi (di 64 bit), ma usa una chiave di 128 bit e consiste di otto stadi, nei quali ogni bit di output dipende da tutti i bit in input (il che non vale per il DES).

Non sono noti risultati di criptanalisi che lo indeboliscono.

IDEA funziona alla velocità di 2 MBps su un Pentium Pro a 200 MHz e a 22 MBps su hardware specializzato.

2.3.1.2) Crittografia a chiave pubblica

Un problema di fondo affligge la crittografia a chiave segreta quando aumenta il numero di persone che vogliono essere in grado di comunicare fra loro.

Poiché ogni coppia di persone deve essere in possesso di una corrispondente chiave, se N persone desiderano comunicare fra loro ci vogliono N(N-1)/2 chiavi, cioè una per ogni coppia.

Ciò rende estremamente difficile il problema della distribuzione delle chiavi, che resta il punto debole di tutto il sistema.

Nella seconda metà degli anni '70 fu introdotto (Diffie e Hellmann, Stanford University) un tipo di crittografia radicalmente nuovo, detto a ***chiave pubblica*** (o ***asimmetrica***).

L'idea è questa:
- ognuno possiede due chiavi, legate una all'altra:
  - una è la ***chiave privata***, nota solo al proprietario;
  - l'altra è la ***chiave pubblica***, nota a tutti;
- ciò che viene cifrato con la prima chiave può essere decifrato con l'altra (e di solito viceversa);
- è quasi impossibile, e comunque estremamente oneroso, derivare la prima chiave anche se si conosce la seconda.

Dunque, per un gruppo di N persone sono necessarie solo 2N chiavi.

Il funzionamento, per ottenere la ***segretezza***, è questo:
- Alice cifra il messaggio con la chiave pubblica di Bob (che è nota a tutti);
- Bob decifra il messaggio con la propria chiave privata (che è nota solo a lui).

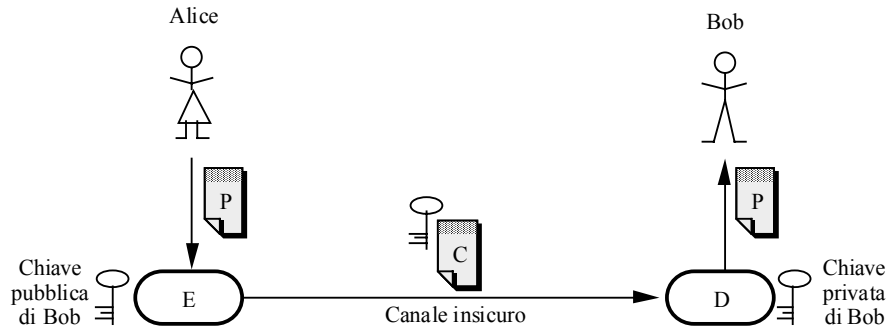


Figura 2-7: Riservatezza mediante crittografia a chiave pubblica

La crittografia a chiave pubblica fornisce anche un meccanismo per garantire l'***autenticazione del mittente***, cioè la garanzia che esso provenga veramente dall'autore e non da qualcun altro, e l'***integrità del messaggio***, cioè la garanzia che il messaggio non sia stato alterato.

In questo caso si opera alla rovescia:
- Alice cifra il messaggio con la propria chiave privata;
- Bob lo decifra con la chiave pubblica di Alice.

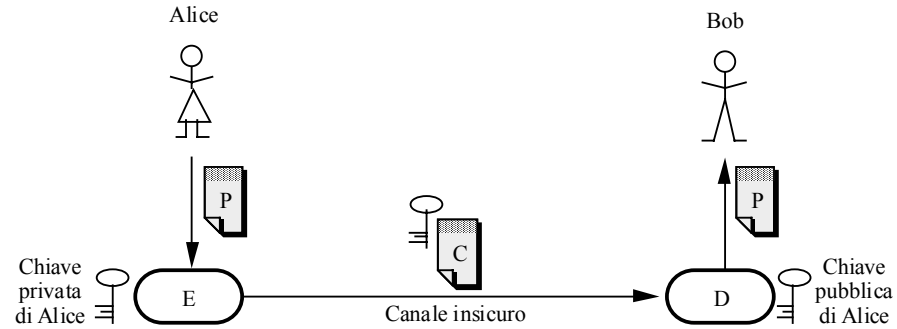


Figura 2-8: Autenticazione mediante crittografia a chiave pubblica

In questo caso non c'è segretezza, dato che chiunque può decifrare il messaggio, ma nessuno se non Alice avrebbe potuto costruirlo, ed inoltre nessuno può averlo alterato.

L'algoritmo a chiave pubblica piu noto ed usato è l'algoritmo ***RSA*** (dalle iniziali degli autori Rivest, Shamir e Adleman), nato nel 1978.

Esso trae la sua efficacia dalla enorme difficoltà di trovare la fattorizzazione di un grande numero (si stima che serva un miliardo di anni di tempo macchina per fattorizzare un numero di 200 cifre, e 10^{25} anni per un numero di 500 cifre).

Schematicamente, l'algoritmo funziona così:
1. scegliere due grandi numeri primi **p** e **q** (tipicamente maggiori di 10^{100});
2. calcolare **n** = **p*****q** e **z** = (**p** - 1)*(**q** - 1);
3. scegliere un numero **d** primo relativamente a **z**;
4. trovare il numero **e** tale che **e*****d** = 1 modulo **z**;

La base matematica di questo procedimento è il ***teorema di Eulero***, che in un particolare sottocaso afferma che, dati due numeri primi p e q, vale la relazione:

$$x^{(p-1)(q-1)} = 1 \text{ modulo } p*q$$

per tutti gli x tali che gcd(x, p*q) = 1.

A questo punto il procedimento prevede di porre:
- chiave pubblica = la coppia (**e, n**);
- chiave privata = la coppia (**d, n**).

Si noti che se non fosse difficile fattorizzare **n** (che è noto a tutti), Trudy potrebbe facilmente:
1. trovare **p** e **q**, e da questi **z**;
2. una volta determinati **z** ed **e** (anch'esso noto a tutti), trovare **d** con l'***algoritmo di Euclide***.

La cifratura e la decifratura vengono effettuate nel seguente modo:
- si divide il testo da cifrare in blocchi tali che, considerandoli come numeri binari, ogni blocco abbia un valore 0 <= P < **n** (basta cioè usare blocchi di k bit , con 2^k < **n**);
- per cifrare un blocco P, calcolare C = P^{e} modulo **n**;
- per decifrare un blocco C, calcolare P = C^{d} modulo **n**.

Può essere dimostrato che, per tutti i blocchi nel range specificato, le due funzioni sono ***inverse***. Quindi, si può anche cifrare con la chiave privata e decifrare con quella pubblica.

Nel 1994 RSA è stato rotto, in risposta ad una sfida degli autori pubblicata su Scientific American. Il procedimento si riferì a una chiave di 129 cifre (426 bit), e furono impiegati 1600 elaboratori su Internet per 8 mesi, per un totale di 5000 anni di calcolo a 1 MIPS (milione di istruzioni al secondo).

D'altronde, poiché RSA lavora con i numeri, la dimensione della chiave è variabile e può essere aumentata a piacere, per controbilanciare gli effetti derivanti dal miglioramento delle prestazioni degli elaboratori.

Infine, poiché gli algoritmi a chiave pubblica sono molto più onerosi computazionalmente di quelli a chiave segreta (di un fattore da 100 a 1000), essi sono usati soprattutto per negoziare in modo sicuro (come vedremo fra breve) una chiave segreta, detta ***chiave di sessione***, da usare nel corso della comunicazione vera e propria la cui riservatezza viene protetta con un algoritmo quale DES o IDEA.

2.3.2) Funzioni hash e firme digitali

Come abbiamo visto, la crittografia a chiave pubblica può essere usata per autenticare l'origine di un messaggio e per garantirne l'integrità, ossia di fatto per ***firmare*** un messaggio.

Però, a causa del costo computazionale, questo approccio sembra eccessivo: cifrare tutto il messaggio solo per firmarlo è poco efficiente. Inoltre, è scomodo rendere l'intero documento illeggibile ove solo una firma sia richiesta.

Ambedue i problemi si risolvono con una tecnica diversa e più efficiente, che però introduce un piccolo rischio in termini di sicurezza.

La tecnica in questione si basa sull'uso delle ***funzioni hash*** (dette anche ***funzioni digest***, cioè funzioni riassunto) che vengono applicate al messaggio e ne producono, appunto, un riassunto (***message digest***).

Tale riassunto è in generale di dimensioni ridotte, tipicamente fra i 10 e i 20 byte, indipendentemente dalla lunghezza del messaggio originario.

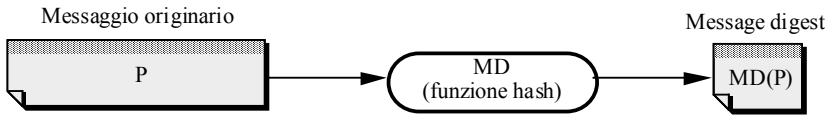


Figura 2-9: Calcolo del riassunto del messaggio

Per essere adatta allo scopo, la funzione hash deve possedere i seguenti requisiti:
- è computazionalmente poco oneroso calcolare MD(P);
- dato MD(P) è praticamente impossibile risalire a P;
- è praticamente impossibile trovare due documenti P_1 e P_2 tali per cui $MD(P_1) = MD(P_2)$. Si noti che questo requisito non discende dalla proprietà precedente.

Per soddisfare l'ultimo requisito il riassunto deve essere piuttosto lungo, almeno 128 bit. Ad ogni modo, è chiaro che dal punto di vista teorico non è possibile garantire che il requisito sia sempre soddisfatto, poiché in generale la cardinalità dello spazio dei messaggi è molto superiore a quella dello spazio dei riassunti.

L'algoritmo più diffuso per la generazione del message digest è ***MD5*** (***Message Digest 5***, Rivest 1992), il quinto di una serie. definito nell'rfc 1321. Produce digest di 128 bit, ognuno dei quali è funzione di tutti i bit del messaggio. Funziona a circa 7 MBps su un Pentium Pro a 200 MHz.

Un primo e semplice schema di utilizzo del message digest è il seguente, volto a garantire l'***integrità del messaggio***, ovvero a garantire che il messaggio che giunge a destinazione sia identico a quello che è stato inviato:

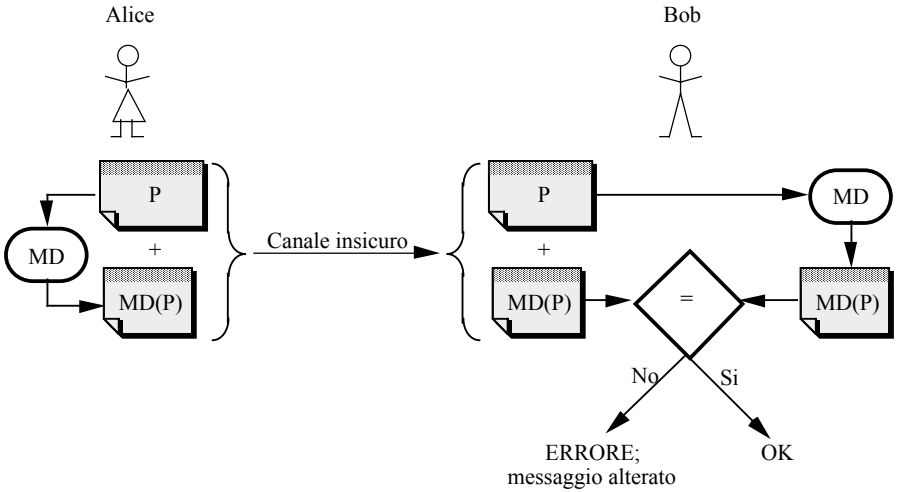


Figura 2-10: Uso del digest per il controllo di integrità del messaggio

Alice invia il messaggio corredato del riassunto; quando Bob riceve il tutto, ricalcola il riassunto e lo confronta con quello ricevuto.

Ovviamente, questa semplice modalità è esposta all'attacco di Trudy, che potrebbe intercettare il messaggio, sostituirlo con uno diverso correlato del relativo digest, e inviarlo a Bob come se fosse quello proveniente da Alice.

Per risolvere questo problema si ricorre a uno schema leggermente più complesso, che fa uso anche della crittografia a chiave pubblica: il riassunto, prima di essere spedito, viene cifrato dal mittente con la propria chiave privata e decifrato dal destinatario con la chiave pubblica del mittente. Un riassunto cifrato in questo modo si dice ***firma digitale*** (***digital signature***) del mittente, perché assicura sia l'integrità del messaggio che l'autenticità del mittente, proprio come una firma apposta (in originale) in calce a un documento cartaceo.

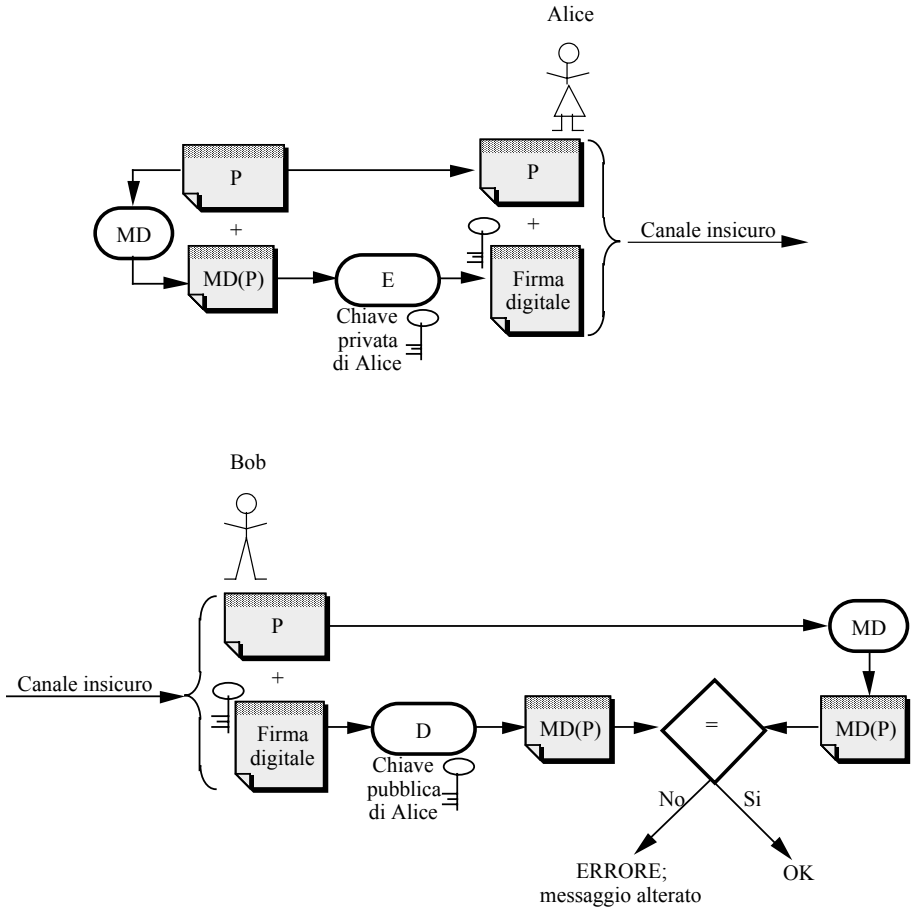


Figura 2-11: Firma digitale

2.3.3) Protocolli crittografici

Quello visto sopra è un esempio di ***protocollo crittografico***, cioè di una serie di regole che le parti debbono seguire per assicurarsi una conversazione conforme ai requisiti desiderati.

Un protocollo crittografico in generale non specifica gli algoritmi da usare nei vari passi, ma piuttosto:

- quali tecniche adottare (ad esempio: crittografia a chiave pubblica e/o privata, message digest, ecc.);
- quale successione di passi deve essere seguita, e quale tecnica va adottata in ogni passo.

Esistono vari protocolli crittografici, che si differenziano per:
- il contesto iniziale (ad esempio: i due partecipanti hanno una chiave segreta in comune o no? Conoscono le rispettive chiavi pubbliche o no?);
- gli scopi da raggiungere (ad esempio: autenticazione, segretezza, o entrambi?).

Vedremo ora alcuni protocolli che rivestono un particolare interesse in un contesto come quello del Web, dove:
- è possibile aver bisogno di autenticazione e segretezza nel dialogo con entità mai conosciute prima;
- i canali sono insicuri, e soggetti all'intrusione di Trudy (e magari anche di Gambadilegno!).

2.3.3.1) Chiave segreta di sessione

Un primo problema da affrontare e risolvere mediante un protocollo crittografico è il seguente: poiché la crittografia a chiave segreta è molto più efficiente di quella a chiave pubblica, si vuole usare la prima nel corso della comunicazione effettiva che deve essere portata avanti.

Però, in un contesto distribuito come il Web, è impensabile che ogni potenziale coppia di fruitori disponga di una chiave segreta. Dunque, bisogna trovare un protocollo per concordare, all'inizio della sessione, la chiave segreta da usare durante il resto della sessione, detta per questo chiave ***segreta di sessione***.

Un primo protocollo è di per se molto semplice, e sfrutta la crittografia a chiave pubblica:
1. Bob invia la sua chiave pubblica ad Alice;
2. Alice genera una nuova chiave segreta, la cifra con la chiave pubblica di Bob e la invia a Bob;
3. Bob riceve la chiave segeta (cifrata) e la decifra con la propria chiave privata;
4. Alice e Bob a questo punto condividono la chiave segreta di sessione per mezzo della quale possono comunicare in tutta sicurezza.

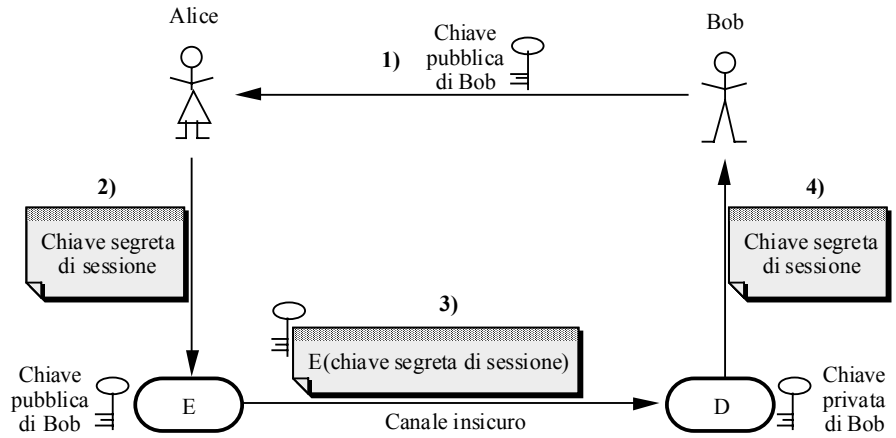


Figura 2-12: Determinazione della chiave segreta di sessione

Per evitare problemi derivanti dalla possibilità che Trudy esegua un ***replay attack*** (cioè invii duplicati di tutto ciò che intercetta) la chiave segreta di sessione dev'essere ogni volta diversa. Di norma la si calcola per mezzo di un generatore di numeri casuali, che deve essere progettato molto accuratamente (si veda in proposito il clamore suscitato da un bug contenuto in Netscape Navigator, riportato anche sul New York Times del 19/9/95).

2.3.3.2) Centro di distribuzione delle chiavi

Il protocollo precedente però ha un problema di fondo molto serio. Infatti, Trudy può riuscire a fare in modo che Alice riceva, al posto della chiave pubblica di Bob, quella di Trudy, e quindi interporsi nella successiva comunicazione e decifrare tutto (***man in the middle attack***).

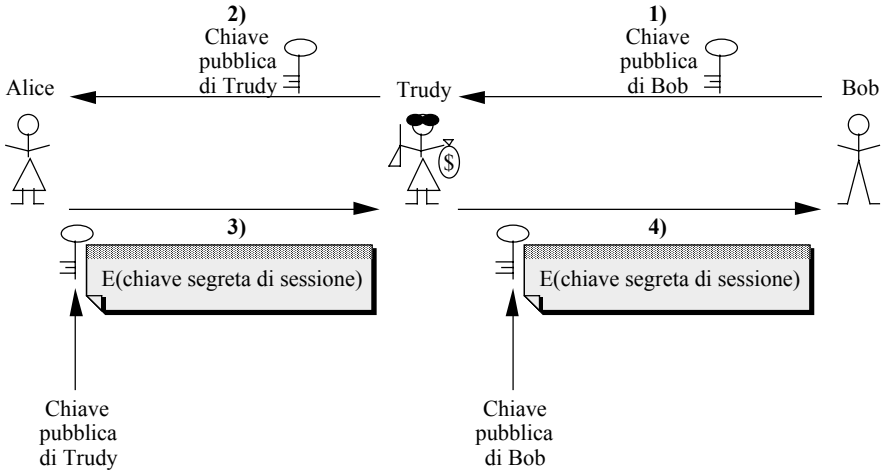


Figura 2-13: Trudy si interpone fra Alice e Bob

Per risolvere questo problema si introduce sulla scena una nuova entità, il ***centro di distribuzione delle chiavi***.

Esso è un ente, di norma governativo o comunque dotato di credibilità internazionale, che:

- possiede adeguati meccanismi di sicurezza (anche fisica) per garantire i dati in proprio possesso;
- possiede una coppia di chiavi (pubblica e privata), e provvede periodicamente a confermare ufficialmente la propria chiave pubblica, ad esempio con la pubblicazione sui principali quotidiani;
- offre a chiunque la richieda una coppia di chiavi (pubblica e privata), che poi provvede a mantenere con sicurezza;
- crea, per ciascuno dei clienti registrati (cioè coloro ai quali ha rilasciato una coppia di chiavi) un ***certificato digitale***, che in sostanza è un documento:
    - contenente la chiave pubblica del cliente;
    - contenente altre informazioni relative al cliente (nome, ecc.);
    - cifrato con la chiave privata del centro (ossia, ***firmato dal centro***).

Per questa ragione, il centro viene anche detto ***Certificate Authority*** (***CA***).

In generale il software usato da Alice ha cablata al suo interno la chiave pubblica della CA, per cui è in grado di verificare la firma dei certificati provenienti dalla CA, e quindi di essere sicuro della loro autenticità e integrità.

Il protocollo visto precedentemente per stabilire la chiave segreta di sessione viene quindi modificato nel senso che la chiave pubblica di Bob viene consegnata ad Alice sotto forma

di un certificato rilasciato a Bob da una CA; in questo modo Alice ha la garanzia che si tratta proprio della chiave di Bob e non di quella di Trudy.

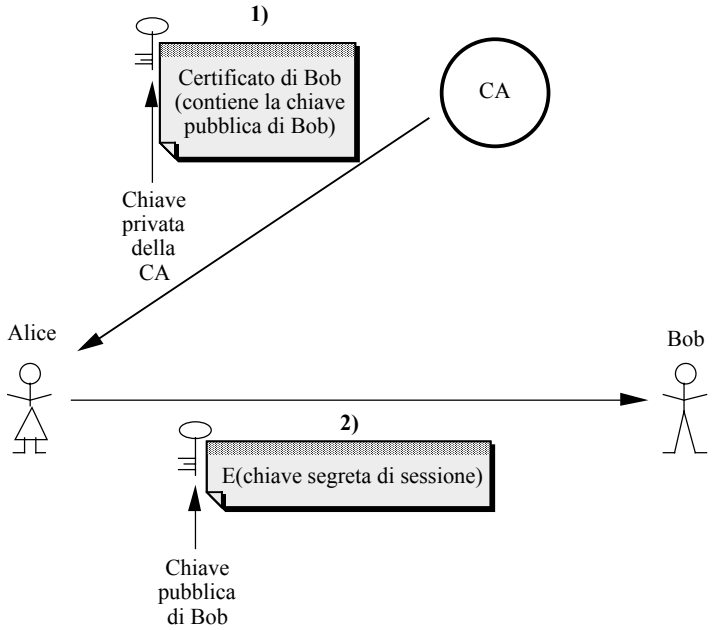


Figura 2-14: Ricorso ad una CA per avere la chiave pubblica di Bob

2.3.3.3) Secure Socket Layer e Secure-HTTP

I certificati prodotti da una CA possono essere conservati dove si desidera. Ad esempio, Bob può tenere una copia del proprio certificato, ed Alice può chiederlo direttamente a lui invece che alla CA.

Questo è precisamente quanto avviene sul Web quando si incontra un link gestito col metodo

```
https://
```

che viene gestito dal protocollo chiamato ***Secure Socket Layer*** (***SSL***), introdotto da Netscape.

Esso protegge l'intero stream di dati che fluisce sulla connessione TCP, per cui può essere usato sia con HTTP che con FTP o TELNET.

Ad esempio, una form HTML che consente l'ordine di beni da acquistare fornendo il numero della propria carta di credito potrà essere riferita col link:

```
https://www.server.com/order.html
```

Quando l'utente del client decide di seguire tale link, si sviluppa questa catena di eventi:

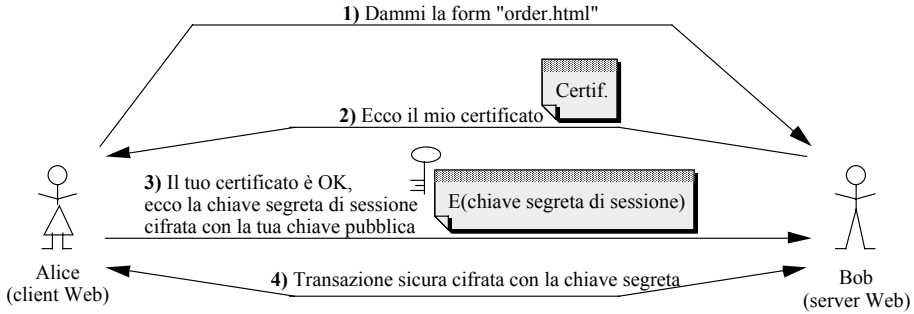


Figura 2-15: Funzionamento del protocollo SSL

Nell protocollo SSL è previsto l'utilizzo di:
- RSA come algoritmo a chiave pubblica;
- DES oppure RC4 a 128 bit come algoritmo a chiave segreta;
- MD5 oppure SHA per la creazione dei digest.

Per via delle leggi americane, le versioni internazionali del Navigator usano RC4 a soli 40 bit per la cifratura a chiave segreta. Peraltro, una recentissima legge ha diminuito le restrizioni esistenti relativamente all'uso del DES a 56 bit, che verrà quindi incorporato anche nei prodotti destinati all'esportazione.

Infine va citato un altro protocollo sicuro per il Web, ***Secure-HTTP*** (***S-HTTP***). Esso si basa su principi analoghi a SSL, ma si applica solo al traffico HTTP. Infatti, nella sostanza, si cifrano i singoli messaggi HTTP e non lo stream TCP sottostante.

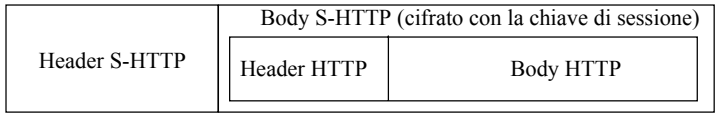


Figura 2-16: Formato di un messaggio S-HTTP

Nell'header S-HTTP, all'inizio, è contenuta la chiave di sessione cifrata con la chiave pubblica del server. Il certificato del server, invece, è incorporato dentro ogni pagina HTML sicura.

3) Utilizzo di Java per lo sviluppo di applicazioni di rete

In questa parte del corso si affronta il problema dello sviluppo di applicazioni di rete di tipo client-server.

Tale architettura software è praticamente una scelta obbligata al giorno d'oggi, in quanto si adatta perfettamente alle attuali reti di elaboratori (costituite da molteplici host indipendenti e non più da un singolo mainframe al quale sono connessi vari terminali).

In particolare, si vedrà:
- come sviluppare un client per i protocolli ASCII del livello application dell'architettura TCP / IP;
- come sviluppare un server multithreaded capace di gestire molti client contemporaneamente;
- quali vantaggi si possono ottenere impacchettando il flusso di dati da trasmettere in una serie di messaggi.

Figura 3-1: Formato di un messaggio

Il linguaggio che verrà usato è Java, in quanto dotato di molti vantaggi rispetto a possibili concorrenti:
- è totalmente a oggetti;
- possiede una interfaccia di programmazione molto pulita;
- è multipiattaforma (non ci si deve preoccupare di problemi di interoperabilità fra piattaforme diverse);
- è dotato di strumenti potenti e di alto livello per:
  - l'apertura di canali di comunicazione (classe `Socket` e `ServerSocket`);
  - la gestione degli errori (classe `Exception`);
  - la gestione della concorrenza (classe `Thread`).

Per le sue caratteristiche, questo linguaggio permette di ottenere con facilità le funzioni base richieste a un applicazione di rete, e di concentrarsi sugli aspetti più caratterizzanti dell'applicazione stessa.

Per poter affrontare lo sviluppo di applicazioni di rete, è necessario approfondire la conoscenza del linguaggio nei seguenti settori:

- gestione di Input/Output;
- gestione dei Socket;
- gestione dei Thread.

3.1) Ripasso di AWT

Vediamo ora un'***applicazione*** (cioè un programma che può girare da solo, appoggiandosi a una macchina virtuale, senza bisogno di una pagina HTML che lo richiama).

Le applicazioni non hanno le limitazioni degli applet, che tipicamente:

- non possono accedere al file system locale;
- non possono aprire connessioni di rete con host diversi da quello di provenienza.

Un'applicazione è una classe al cui interno esiste un metodo `main(String args[])` che viene eseguito all'avvio.

Esempio 1

Questa semplicissima applicazione presenta alcuni campi testo e alcuni bottoni, ed è predisposta per la gestione degli eventi.

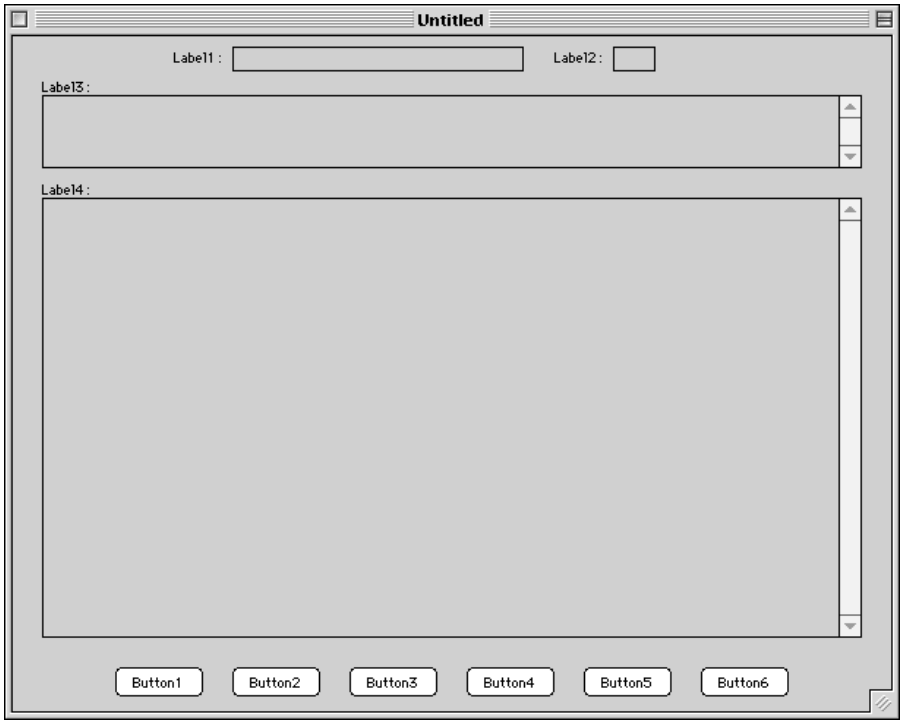


Figura 3-2: Interfaccia utente dell'esempio 1

Nel nostro caso, il `main()` crea una nuova istanza di un oggetto della classe `BaseAppE1`, che estende `Frame` (in pratica una finestra).

```
import java.awt.*;

public class BaseAppE1 extends Frame    {
    Label label1, label2, label3, label4;
    TextField textField1, textField2;
    TextArea textArea1, textArea2;
    Button button1, button2, button3, button4, button5, button6;
//-------------------------------------------------
public BaseAppE1() {
        this.setLayout(null);
        label1 = new Label("Label1:");
```

```
        label1.reshape(110, 5, 40, 15);
        this.add(label1);
        textField1 = new TextField();
        textField1.reshape(150, 10, 200, 15);
        this.add(textField1);
        label2 = new Label("Label2:");
        label2.reshape(370, 5, 40, 15);
        this.add(label2);
        textField2 = new TextField();
        textField2.reshape(410, 10, 30, 15);
        this.add(textField2);
        label3 = new Label("Label3:");
        label3.reshape(20, 25, 100, 15);
        this.add(label3);
        textArea1 = new TextArea();
        textArea1.reshape(20, 40, 560, 50);
        this.add(textArea1);
        label4 = new Label("Label4:");
        label4.reshape(20, 95, 100, 15);
        this.add(label4);
        textArea2 = new TextArea();
        textArea2.reshape(20, 110, 560, 300);
        this.add(textArea2);
        button1 = new Button("Button1");
        button1.reshape(70, 430, 60, 20);
        this.add(button1);
        button2 = new Button("Button2");
        button2.reshape(150, 430, 60, 20);
        this.add(button2);
        button3 = new Button("Button3");
        button3.reshape(230, 430, 60, 20);
        this.add(button3);
        button4 = new Button("Button4");
        button4.reshape(310, 430, 60, 20);
        this.add(button4);
        button5 = new Button("Button5");
        button5.reshape(390, 430, 60, 20);
        this.add(button5);
        button6 = new Button("Button6");
        button6.reshape(470, 430, 60, 20);
        this.add(button6);
        resize(600, 460);
        show();
    }
//-------------------------------------------------
    public static void main(String args[]) {
            new BaseAppE1();
    }
//-------------------------------------------------
    public boolean handleEvent(Event event) {
        if (event.id == Event.WINDOW_DESTROY) {
            hide();         // hide the Frame
            dispose();      // tell windowing system to free resources
            System.exit(0); // exit
            return true;
        }
        if (event.target == button1 && event.id == Event.ACTION_EVENT) {
            button1_Clicked(event);
        }
        if (event.target == button2 && event.id == Event.ACTION_EVENT) {
            button2_Clicked(event);
        }
        if (event.target == button3 && event.id == Event.ACTION_EVENT) {
            button3_Clicked(event);
        }
        if (event.target == button4 && event.id == Event.ACTION_EVENT) {
            button4_Clicked(event);
        }
        if (event.target == button5 && event.id == Event.ACTION_EVENT) {
```



```
            button5_Clicked(event);
        }
        if (event.target == button6 && event.id == Event.ACTION_EVENT) {
            button6_Clicked(event);
        }
        return super.handleEvent(event);
    }
//-------------------------------------------------
    void button1_Clicked(Event event) {
            textArea2.setText(textArea2.getText() + "Hai premuto bottone 1\n");
    }
//-------------------------------------------------
    void button2_Clicked(Event event) {
            textArea2.setText(textArea2.getText() + "Hai premuto bottone 2\n");
    }
//-------------------------------------------------
    void button3_Clicked(Event event) {
        textArea2.setText(textArea2.getText() + "Hai premuto bottone 3\n");
    }
//-------------------------------------------------
    void button4_Clicked(Event event) {
        textArea2.setText(textArea2.getText() + "Hai premuto bottone 4\n");
    }
//-------------------------------------------------
    void button5_Clicked(Event event) {
        textArea2.setText(textArea2.getText() + "Hai premuto bottone 5\n");
    }
//-------------------------------------------------
    void button6_Clicked(Event event) {
        textArea2.setText(textArea2.getText() + "Hai premuto bottone 6\n");
    }
}
```

3.2) Input/Output in Java

L'I/O in Java è definito in termini di ***stream*** (***flussi***). Gli stream sono un'astrazione di alto livello per rappresentare la connessione a un canale di comunicazione.

Il canale di comunicazione può essere costituito fra entità molto diverse, le più importanti delle quali sono:

- un file;
- una connessione di rete (ad esempio TCP/IP);
- un buffer in memoria.

Grazie all'astrazione rappresentata dagli stream, le operazioni di I/O dirette a (o provenienti da) uno qualunque degli oggetti di cui sopra sono realizzate con la stessa interfaccia.

Uno stream rappresenta un ***punto terminale*** di un canale di comunicazione unidirezionale, e può leggere dal canale (`InputStream`) o scrivervi (`OutputStream`):

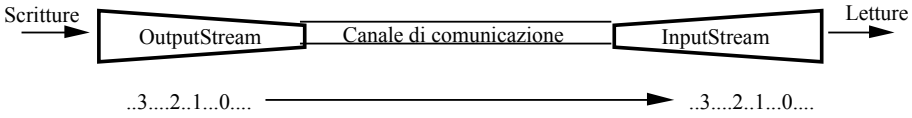


Figura 3-3: Stream di input e output

Tutto ciò che viene scritto sul canale tramite l'`OutputStream` viene letto dall'altra parte dal corrispondente `InputStream`.

Gli stream hanno diverse proprietà:

- sono ***FIFO***: ciò che viene scritto da un `OutputStream` viene letto nello stesso ordine dal corrispondente `InputStream`;
- sono ***ad accesso sequenziale***: non è fornito alcun supporto per l'accesso casuale (solo la classe `RandomAccessFile`, che però non è uno stream, offre tale tipo di accesso);
- sono ***read-only oppure write-only***: uno stream consente di leggere (`InputStream`) o scrivere (`OutputStream`) ma non entrambe le cose. Se ambedue le funzioni sono richieste, ci vogliono 2 distinti stream: questo è un caso tipico delle connessioni di rete, tant'è che da una connessione (`Socket`) si ottengono due stream, uno in scrittura e uno in lettura;
- sono ***bloccanti***: la lettura blocca il programma che l'ha richiesta finché i dati non sono disponibili. Analogamente, la scrittura blocca il richiedente finché non è completata;
- quasi tutti i loro metodi possono ***generare eccezioni***.

3.2.1) Classe InputStream

E' la classe astratta dalla quale derivano tutti gli stream finalizzati alla lettura da un canale di comunicazione.

Costruttore

```
public InputStream();
```

Vuoto.

Metodi più importanti

```
public abstract int read() throws IOException;
```

Legge e restituisce un byte (range 0-255) o si blocca se non c'è niente da leggere. Restituisce il valore -1 se incontra la fine dello stream.

```
public int read(byte[] buf) throws IOException;
```

Legge fino a riempire l'array `buf`, o si blocca se non ci sono abbastanza dati. Restituisce il numero di byte letti, o il valore -1 se incontra la fine dello stream. L'implementazione di default chiama tante volte la `read()`, che è astratta.

```
public int read(byte[] buf, int off, int len) throws IOException;
```

Legge fino a riempire l'array `buf`, a partire dalla posizione `off`, con `len` byte (o fino alla fine dell'array). Si blocca se non ci sono abbastanza dati. Restituisce il numero di byte letti, o il valore -1 se incontra la fine dello stream. L'implementazione di default chiama tante volte la `read()`, che è astratta.

```
public int available() throws IOException;
```

Restituisce il numero di byte che possono essere letti senza bloccarsi (cioè che sono disponibili):

- con un file: quelli che rimangono da leggere fino alla fine del file;
- con una connessione di rete: anche 0, se dall'altra parte non è ancora stato mandato nulla.

```
public void close() throws IOException;
```

Chiude lo stream e rilascia le risorse ad esso associate. Eventuali dati non ancora letti vengono perduti.

Questi metodi, e anche gli altri non elencati, sono supportati anche dalle classi derivate, e quindi sono disponibili sia che si legga da file, da un buffer di memoria o da una connessione di rete.

3.2.2) Classe OutputStream

E' la classe astratta dalla quale derivano tutti gli Stream finalizzati alla scrittura su un canale di comunicazione.

Costruttore

```
public OutputStream();
```

Vuoto.

Metodi più importanti

```
public abstract void write(int b) throws IOException;
```

Scrive un byte (8 bit) sul canale di comunicazione. Il parametro è `int` (32 bit) per convenienza (il risultato di espressioni su byte è intero), ma solo gli 8 bit meno significativi sono scritti.

```
public void write(byte[] buf) throws IOException;
```

Scrive un array di byte. Blocca il chiamante finché la scrittura non è completata..

```
public void write(byte[] buf, int off, int len) throws IOException;
```

Scrive la parte di array che inizia a posizione `off`, e consiste di `len` byte. Blocca il chiamante finché la scrittura non è completata..

```
public void flush() throws IOException;
```

Scarica lo stream, scrivendo effettivamente tutti i dati eventualmente mantenuti in un buffer locale per ragioni di efficienza. Ciò deriva dal fatto che in generale si introduce molto overhead scrivendo pochi dati alla volta sul canale di comunicazione:

- su file: molte chiamate di sistema;
- su connessione di rete: un TPDU per pochi byte.

```
public void close() throws IOException;
```

Chiama `flush()` e poi chiude lo stream, liberando le risorse associate. Eventuali dati scritti prima della `close()` vengono comunque trasmessi, grazie alla chiamata di `flush()`.

Esempio 2

Applicazione che copia ciò che viene immesso tramite lo ***standard input*** sullo `standard output`.

A tal fine si usano due stream accessibili sotto forma di ***field*** (cioè variabili) nella ***classe statica*** (e quindi sempre disponibile, anche senza instanziarla) `System`. Essi sono `System.in` e `System.out`.

```
import java.io.*;
public class InToOut    {
```



```
//------------------------------------------------
    public static void main(String args[]) {
        int charRead;
        try {
            while ((charRead = System.in.read()) != -1) {
                System.out.write(charRead);
                System.out.println("");//Ignorare questa istruzione (ma non toglierla)
            }
        } catch (IOException e) {
            //non facciamo nulla
        }
    }
}
```

Il programma si ferma immettendo un particolare carattere (***End Of File***, ***EOF***) che segnala fine del file:

- sotto UNIX si preme Ctrl-D;
- sotto Windows si preme Ctrl-Z;
- sotto MacOS si preme OK nella finestra di dialogo dello standard input.

3.2.3) Estensione della funzionalità degli Stream

Lavorare con singoli byte non è molto comodo. Per fornire funzionalità di I/O di livello più alto, in Java si usano estensioni degli stream dette ***stream di filtro***. Uno stream di filtro (che esiste sia per l'input che per l'output) è una estensione del corrispondente stream di base, si attacca ad uno stream esistente e fornisce ulteriori funzionalità.

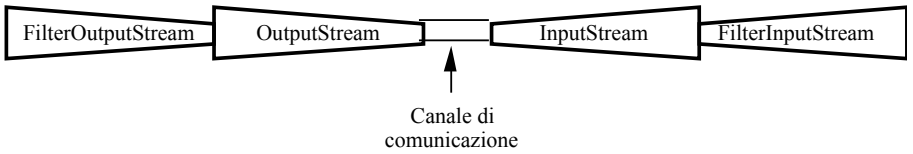


Figura 3-4: Stream di filtro

Aspetti importanti degli stream di filtro:

- sono classi derivate da `InputStream` e `OutputStream`, e quindi supportano tutti i loro metodi;
- si attaccano a un `InputStream` o `OutputStream` al quale passano di norma i metodi della superclasse: chiamare `write()` su uno stream di filtro significa chiamare `write()` sull'`OutputStream` attaccato;
- poiché sono anch'essi degli stream, è possibile attaccare vari stream di filtro in serie per ottenere potenti combinazioni.

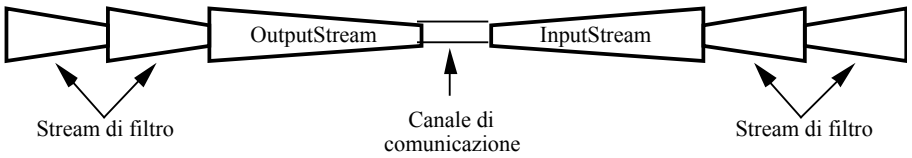


Figura 3-5: Concatenazione di stream di filtro

Il vantaggio principale di questa strategia, rispetto a estendere direttamente le funzionalità di uno stream di base, è che tali estensioni possono essere applicate a tutti i tipi di stream, in quanto sono sviluppate indipendentemente da essi.

3.2.3.1) Classe FilterInputStream

Estende `InputStream` ed è a sua volta la classe da cui derivano tutti gli stream di filtro per l'input. Si attacca a un `InputStream`.

Costruttore

```
protected FilterInputStream(InputStream in);
```

Crea uno stream di filtro attaccato a `in`.

Variabili

```
protected InputStream in;
```

Reference all'`InputStream` a cui è attaccato.

Metodi più importanti

Gli stessi di `InputStream`. Di fatto l'implementazione di default non fa altro che chiamare i corrispondenti metodi dell'`InputStream` attaccato.

3.2.3.2) Classe FilterOutputStream

Estende `OutputStream` ed è a sua volta la classe da cui derivano tutti gli stream di filtro per l'output. Si attacca a un `OutputStream`.

Costruttore

```
protected FilterOutputStream(OutputStream out);
```

Crea uno stream di filtro attaccato a `out`.

Variabili

```
protected OutputStream out;
```

Reference all'`OutputStream` a cui è attaccato.

Metodi più importanti

Gli stessi di `OutputStream`. Di fatto l'implementazione di default non fa altro che chiamare i corrispondenti metodi dell'`OutputStream` attaccato.

Vari stream di filtro predefiniti sono disponibili in Java. I più utili sono descritti nel seguito.

3.2.3.3) BufferedInputStream e BufferedOutputStream

Forniscono, al loro interno, meccanismi di buffering per rendere più efficienti le operazioni di I/O.

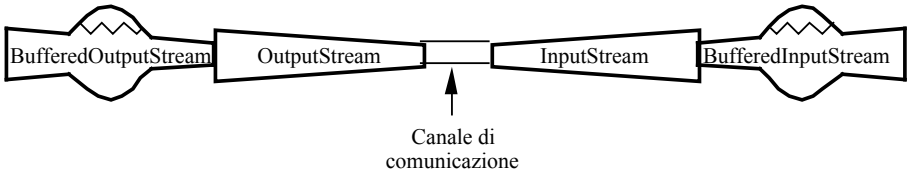


Figura 3-6: Stream di filtro per il buffering

Essi diminuiscono il numero di chiamate al sistema operativo (ad esempio nel caso di accessi a file) o di TPDU spediti (nel caso delle connessioni di rete).

Costruttori

```
public BufferedInputStream(InputStream in);
```

Crea un `BufferedInputStream` attaccato a `in` (con un buffer interno di 512 byte).

```
public BufferedInputStream(InputStream in, int size);
```

Secondo costruttore in cui si specifica la dimensione del buffer interno.

```
public BufferedOutputStream (OutputStream out);
```


Crea un `BufferedOutputStream` attaccato a `out` (con un buffer interno di 512 byte).

```
public BufferedOutputStream (OutputStream out, int size);
```

Secondo costruttore in cui si specifica la dimensione del buffer interno.

Metodi più importanti

Gli stessi degli stream da cui derivano. In più, `BufferedOutputStream` implementa il metodo `flush()`.

3.2.3.4) DataInputStream e DataOutputStream

Forniscono metodi per leggere e scrivere dati a un livello di astrazione più elevato (ad esempio interi, reali, stringhe, booleani, ecc.) su un canale orientato al byte.

I valori numerici vengono scritti in ***network byte order*** (il byte più significativo viene scritto per primo), uno standard universalmente accettato. In C, ciò si ottiene mediante l'uso di apposite funzioni:

- `htons()`: host to network short;
- `htonl()`: host to network long;
- `ntohs()`: network to host short;
- `ntohl()`: network to host long.

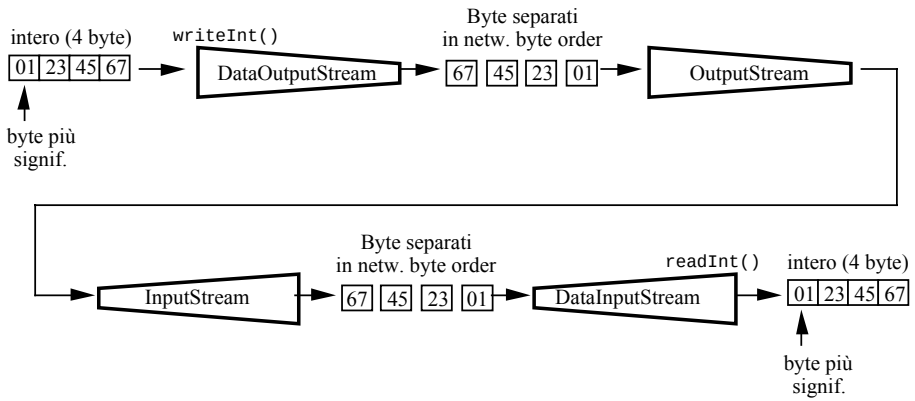


Figura 3-7: Uso di `DataInputStream` e `DataOutputStream`

Costruttori

```
public DataInputStream(InputStream in);
```

Crea un `DataInputStream` attaccato a `in`.

```
public DataOutputStream(OutputStream out);
```

Crea un `DataOutputStream` attaccato a `out`.

Metodi più importanti

```
public writeBoolean(boolean v) throws IOException;
public boolean readBoolean() throws IOException;
public writeByte(int v) throws IOException;
public byte readByte()throws IOException;
public writeShort(int v) throws IOException;
public short readShort()throws IOException;
public writeInt(int v) throws IOException;
public int readInt() throws IOException;
public writeLong(long v) throws IOException;
public long readLong()throws IOException;
public writeFloat(float v) throws IOException;
public float readFloat() throws IOException;
public writeDouble(double v) throws IOException;
public double readDouble() throws IOException;
public writeChar(int v) throws IOException;
public char readChar() throws IOException;
```

Letture e scritture dei tipi di dati primitivi.

```
public writeChars(String s) throws IOException;
```

Scrittura di una stringa.

```
public String readLine() throws IOException;
```

Lettura di una linea di testo, terminata con ***CR***, ***LF*** oppure ***CRLF***.

3.2.3.5) PrintStream

Fornisce metodi per scrivere, come sequenza di caratteri ASCII, una rappresentazione di tutti i tipi primitivi e degli oggetti che implementano il metodo `toString()`, che viene in tal caso usato.

Costruttori

```
public PrintStream(OutputStream out);
```

Crea un `PrintStream` attaccato a `out`.


```
public PrintStream(OutputStream out, boolean autoflush);
```

Se `autoflush` è impostato a `true`, lo stream effettua un `flush()` ogni volta che incontra la fine di una linea di testo (CR, LF oppure CR+LF).

Metodi più importanti

```
public void print(...) throws IOException;
```

Il parametro può essere ogni tipo elementare, un array di caratteri, una stringa, un Object.

```
public void println(...) throws IOException;
```

Il parametro può essere ogni tipo elementare, un array di caratteri, una stringa, un Object.

`println()` dopo la scrittura va a linea nuova (appende CR, LF oppure CR+LF).

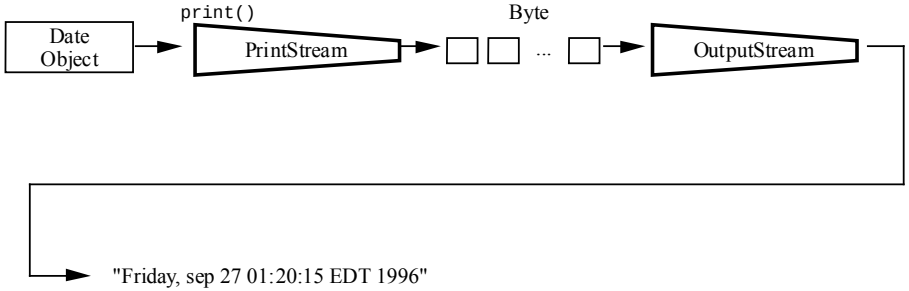


Figura 3-8: Uso di `PrintStream`

Esempi di uso di FilterStream

3.2.3.6) Esempi di uso degli stream di filtro

Vediamo ora alcuni usi tipici degli stream di filtro. In generale ci sono due modi di crearli, a seconda che si vogliano mantenere o no dei riferimenti agli stream a cui vengono attaccati.

Efficienza

Si usano gli stream `BufferedInputStream` e `BufferedOutputStream`, che offrono la funzione di buffering. Supponiamo di lavorare con un `FileOutputStream` (che vedremo nel seguito).

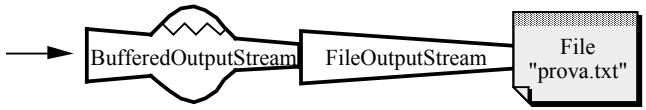


Figura 3-9: Buffering nell'accesso a un file

La prima possibilità è di mantenere in una apposita variabile anche il riferimento allo stream cui si attacca il filtro (cioè al `FileOutputStream`):

```
FileOutputStream fileStream;
BufferedOutputStream bufferedStream;
fileStream = new FileOutputStream("prova.txt");
bufferedStream = new BufferedOutputStream(fileStream);
```

La seconda possibilità è creare il `FileOutputStream` direttamente dentro il costruttore dello stream di filtro :

```
BufferedOutputStream bufferedStream;
bufferedStream = new BufferedOutputStream(new FileOutputStream("prova.txt"));
```

Si noti che in questo caso non vi è modo, nel resto del codice, di riferirsi esplicitamente al `FileOutputStream`.

Lettura e scrittura ASCII
Si usano `DataInputStream` (chiamando ad esempio `readLine()`) e `PrintStream` (chiamando ad esempio `println(...)`).

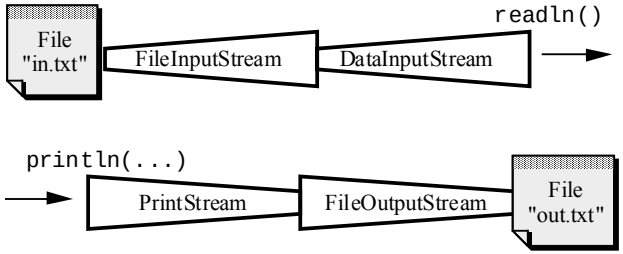


Figura 3-10: Lettura e scrittura ASCII su file


```
DataInputStream asciiIn;
PrintStream asciiOut;
asciiIn = new DataInputStream(new FileInputStream("in.txt"));
asciiOut = new PrintStream(new FileOutputStream("out.txt"));
```

Efficienza e scrittura caratteri ASCII
Usiamo un `PrintStream` attaccato a un `BufferedOutputStream` attaccato a sua volta a un `FileOutputStream`.

Figura 3-11: Scrittura ASCII, bufferizzata, su file

```
PrintStream asciiOut;
asciiOut = new PrintStream(new BufferedOutputStream(
                           new FileOutputStream("out.txt")));
```

Analogo discorso per la lettura bufferizzata:

```
DataInputStream asciiIn;
asciiIn = new DataInputStream(new BufferedInputStream(
                                new FileInputStream("in.txt")));
```

3.2.4) Tipi base di Stream

Le classi `InputStream` e `OutputStream` sono astratte, e da esse si derivano quelle che rappresentano le connessioni a reali canali di comunicazione di vario tipo. Ne esistono molte; noi parleremo di:

- stream per l'accesso a file;
- stream per l'accesso ad array di byte esistenti in memoria centrale;
- stream per l'accesso a connessioni di rete.

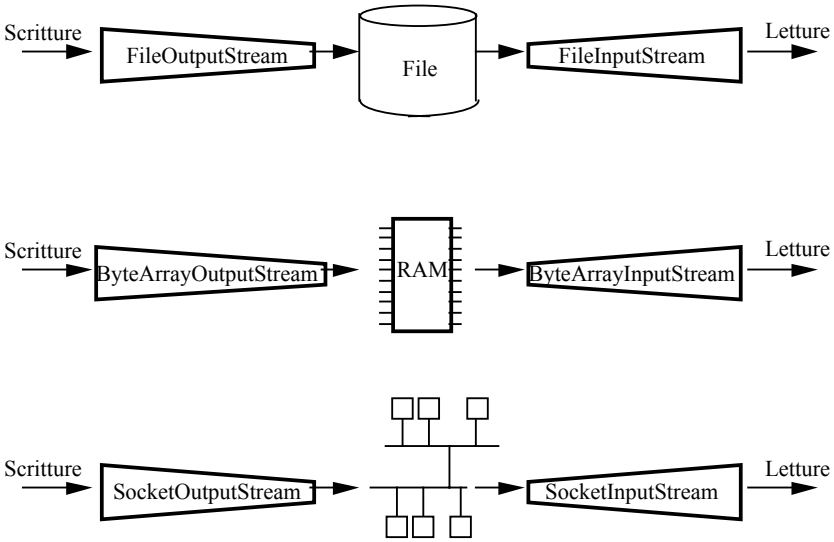


Figura 3-12: Tipi base di stream

Si noti che `SocketInputStream` e `SocketOutputStream` sono classi private, per cui una connessione di rete viene gestita per mezzo di `InputStream` e `OutputStream`.

3.2.5) Stream per accesso a file

L'accesso a un file si ottiene con due tipi di Stream:

- `FileInputStream` per l'accesso in lettura;
- `FileOutputStream` per l'accesso in scrittura.

La classe `FileInputStream` permette di leggere sequenzialmente da un file (che deve già esistere).

Figura 3-13: `FileInputStream`

Costruttori

```
public FileInputStream(String fileName) throws IOException;
public FileInputStream(File fileName) throws IOException;
```

Sono i più importanti. La classe `File` rappresenta un nome di file in modo indipendente dalla piattaforma, ed è di solito utilizzata in congiunzione con un `FileDialog` che permette all'utente la scelta di un file. Vedremo un esempio più avanti. Se si usa una stringa, essa può essere:

- Nome del file (il direttorio di lavoro è implicito);
- Path completo più nome del file.

Metodi più importanti

Tutti quelli di `InputStream`, più uno (`getFD()`) che non vedremo.

La classe `FileOutputStream` permette di scrivere sequenzialmente su un file, che viene creato quando si istanzia lo stream. Se esiste già un file con lo stesso nome, esso viene distrutto.

Figura 3-14: `FileOutputStream`

Costruttori

```
public FileOutputStream(String fileName) throws IOException;
public FileOutputStream (File fileName) throws IOException;
```

Sono i più importanti. Per la classe `File` vale quanto detto sopra.

Metodi più importanti

Tutti quelli di `OutputStream`, più uno (`getFD()`) che non vedremo.

Si noti che questi stream, oltre che eccezioni di I/O, possono generare eccezioni di tipo `SecurityException` (ad esempio se un applet cerca di aprire uno stream verso il file system locale).

Esempio 3

Applicazione che mostra in una `TextArea` il contenuto di un file di testo, e che salva il contenuto di tale `TextArea` in un file.

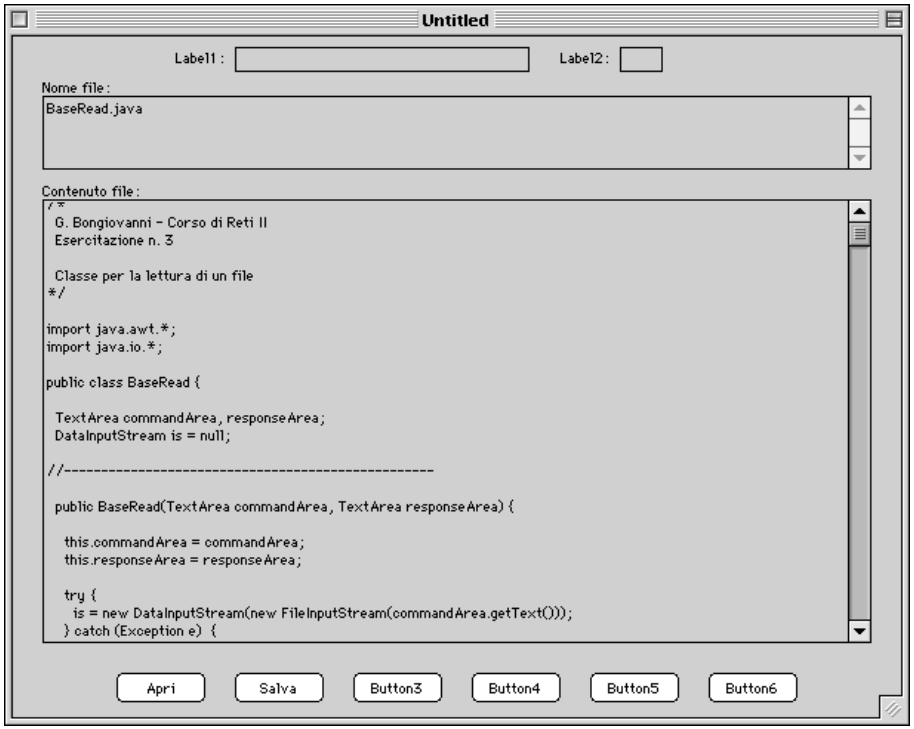


Figura 3-15: Interfaccia utente dell'esempio 3

Il codice è costituito da tre classi. La prima, `BaseAppE3`, è basata su quella dell'esempio 1 ed istanzia a comando una delle altre due quando si preme un bottone. Se ne mostrano qui solo i frammenti di codice rilevanti.

```
import java.awt.*;

public class BaseAppE3 extends Frame    {
...ecc.
    //-------------------------------------------------
    void button1_Clicked(Event event) {
        baseRead = new BaseRead(textArea1, textArea2);
        baseRead.readFile();
    }
//-------------------------------------------------
```



```
    void button2_Clicked(Event event) {
        baseWrite = new BaseWrite(textArea1, textArea2);
        baseWrite.writeFile();
    }
//-------------------------------------------------
```

Le altre due si occupano della lettura e scrittura sul file.

```
import java.awt.*;
import java.io.*;

public class BaseRead {
    TextArea commandArea, responseArea;
    DataInputStream is = null;
//-------------------------------------------------
    public BaseRead(TextArea commandArea, TextArea responseArea) {
        this.commandArea = commandArea;
        this.responseArea = responseArea;
        try {
            is=new DataInputStream(new FileInputStream(commandArea.getText()));
        } catch (Exception e)   {
            responseArea.appendText("Exception" + "\n");
        }
    }
//-------------------------------------------------
    public void readFile() {
        String inputLine;
        try {
            while ((inputLine = is.readLine()) != null) {
                responseArea.appendText(inputLine + "\n");
            }
        } catch (IOException e) {
            responseArea.appendText("IO Exception" + "\n");
        } finally {
            try {
                is.close();
            } catch (IOException e) {
                responseArea.appendText("IO Exception" + "\n");
            }
        }
    }
}
```

```
import java.awt.*;
import java.io.*;

public class BaseWrite {
    TextArea commandArea, responseArea;
    PrintStream os = null;
//-------------------------------------------------
    public BaseWrite(TextArea commandArea, TextArea responseArea) {

        this.commandArea = commandArea;
        this.responseArea = responseArea;

        try {
            os = new PrintStream(new FileOutputStream(commandArea.getText()));
        } catch (Exception e)   {
            responseArea.appendText("Exception" + "\n");
        }
    }
//-------------------------------------------------
```

```
    public void writeFile() {
        os.print(responseArea.getText());
        os.close();
    }
}
```

Se si vuole selezionare il file da leggere o da creare in modo interattivo, si usa la classe `FileDialog`, che serve a entrambi gli scopi.
Il codice per aprire un file in lettura è tipicamente:

```
...
FileDialog openDialog = new FileDialog(null, "Titolo", FileDialog.LOAD);
openDialog.show();
if (openDialog.getFile() != null) {
    is = new DataInputStream(
         new FileInputStream(
         newFile(openDialog.getDirectory(), openDialog.getFile())));
}
...
```

E quello per creare un file in scrittura:

```
...
FileDialog saveDialog = new FileDialog(null, "Titolo", FileDialog.SAVE);
saveDialog.show();
if (saveDialog.getFile() != null) {
    os = new PrintStream(
         new FileOutputStream(
         new File(saveDialog.getDirectory(), saveDialog.getFile()));
}
...
```

3.2.6) Stream per accesso alla memoria

Esistono due tipi di stream che consentono di leggere e scrivere da/su buffer di memoria:

- `ByteArrayInputStream`
- `ByteArrayOutputStream`

Il `ByteArrayInputStream` permette di leggere sequenzialmente da un buffer (ossia da un array di byte) di memoria.

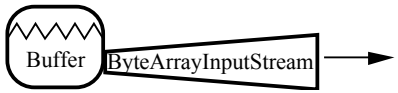


Figura 3-16: `ByteArrayInputStream`

Costruttori

A noi ne serve solo uno:

```
public ByteArrayInputStream (byte[] buf);
```

Crea un `ByteArrayInputStream`, attaccato all'array `buf`, dal quale vengono poi letti i dati.

Metodi più importanti

Quelli di `InputStream`. Da notare che:
- un `EOF` si incontra quando si raggiunge la fine dell'array;
- il metodo `reset()` ripristina lo stream, per cui le successive letture ripartono dall'inizio dell'array.

Il `ByteArrayOutputStream` permette di scrivere in un buffer (array di byte) in memoria. Il buffer viene allocato automaticamente e si espande secondo le necessità, sempre automaticamente.

Figura 3-17: `ByteArrayOutputStream`

Costruttori

I più utilizzati sono:

```
public ByteArrayOutputStream();
```

Crea un `ByteArrayOutputStream` con una dimensione iniziale del buffer di 32 byte. Il buffer comunque cresce automaticamente secondo necessità.

```
public ByteArrayOutputStream (int size);
```

Crea un `ByteArrayOutputStream` con una dimensione iniziale di `size` byte. E' utile quando si voglia massimizzare l'efficienza e si conosca in anticipo la taglia dei dati da scrivere.

Metodi più importanti

Oltre a quelli di `OutputStream` ve ne sono alcuni per accedere al buffer interno.

```
public void reset();
```

Ricomincia la scrittura dall'inizio, di fatto azzerando il buffer.

```
public int size();
```

Restituisce il numero di byte contenuti (cioè scritti dopo l'ultima `reset()`) nel buffer.

```
public byte[] toByteArray();
```

Restituisce una copia dei byte contenuti nel buffer, che non viene resettato.

```
public String toString();
```

Restituisce una stringa che è una copia dei byte contenuti. L'array non viene resettato.

```
public writeTo(OutputStream out) throws IOException;
```

Scrive il contenuto del buffer sullo stream `out`. Il buffer non viene resettato. E' più efficiente che chiamare `toByteArray()` e scrivere quest'ultimo su `out`.

3.2.7) Stream per accesso a connessioni di rete

Il dialogo di un'applicazione Java con una peer entity attraverso la rete può avvenire sia appoggiandosi a TCP che a UDP.

Poiché la modalità connessa è molto più versatile ed interessante per i nostri scopi, ci occuperemo solo di essa.

Come noto, la vita di una connessione TCP si articola in tre fasi:
1. apertura della connessione;
2. dialogo per mezzo della connessione;
3. chiusura della connessione;

Anche un'applicazione Java segue lo stesso percorso:
1. apertura della connessione. Esistono per questo due classi:
   - `Socket`: per aprire una connessione con un server in ascolto;
   - `ServerSocket`: per mettersi in ascolto di richieste di connessione;
2. dialogo per mezzo della connessione. Con opportune istruzioni, dal `Socket` precedentemente creato si ottengono:
   - `InputStream` per ricevere i dati dalla peer entity;
   - `OutputStream` per inviare i dati alla peer entity;
3. chiusura della connessione. Con opportune istruzioni, si chiudono:
   - l'`InputStream`;
   - l'`OutputStream`;
   - il `Socket` (per ultimo, preferibilmente).

3.2.7.1) Classe Socket

Questa classe rappresenta in Java l'estremità locale di una connessione TCP. La creazione di un oggetto `Socket`, con opportuni parametri, stabilisce automaticamente una connessione TCP con l'host remoto voluto.

Se la cosa non riesce, viene generata una eccezione (alcuni dei motivi possibili sono: host irraggiungibile, host inesistente, nessun processo in ascolto sul server).

Una volta che l'oggetto `Socket` è creato, da esso si possono ottenere i due stream (di input e output) necessari per comunicare.

Costruttore

```
public Socket(String host, int port) throws IOException;
```

A noi basta questo (ce ne sono due). `host` è un nome DNS o un indirizzo IP in ***dotted decimal notation***, e `port` è il numero di port sul quale deve esserci un processo server in ascolto.

Ad esempio:

```
...
mySocket = new Socket("cesare.dsi.uniroma1.it", 80);
mySocket = new Socket("151.100.17.25", 80);
```

. . .

Metodi più importanti

```
public InputStream getInputStream() throws IOException;
```

Restituisce un `InputStream` per leggere i dati inviati dalla peer entity.

```
public OutputStream getOutputStream() throws IOException;
```

Restituisce un `OutputStream` per inviare i dati inviati dalla peer entity.

```
public void close() throws IOException;
```

Chiude il `Socket` (e quindi questa estremità della connessione TCP).

```
public int getPort() throws IOException;
```

Restituisce il numero di port all'altra estremità della connessione.

```
public int getLocalPort() throws IOException;
```

Restituisce il numero di port di questa estremità della connessione.

Esempio 4

Applicazione che apre una connessione via Socket con un host e su un port scelti per mezzo dei due rispettivi campi. Va usata con ***protocolli ASCII***, cioè protocolli che prevedono lo scambio di sequenze di caratteri ASCII.

L'applicazione permette di:

- stabilire la connessione con un server mediante il bottone "Connect";
- inviare un comando col bottone "Send";
- leggere ***una singola linea di testo*** della risposta col bottone "Receive";
- chiudere la connessione col bottone "Close".

Si noti che, a seconda del numero di port specificato, si può dialogare con server differenti. Ad esempio, usando il port 21 ci si connette con un server FTP, mentre con il port 80 (come nell'esempio di figura 4-16) ci si collega ad un server HTTP.

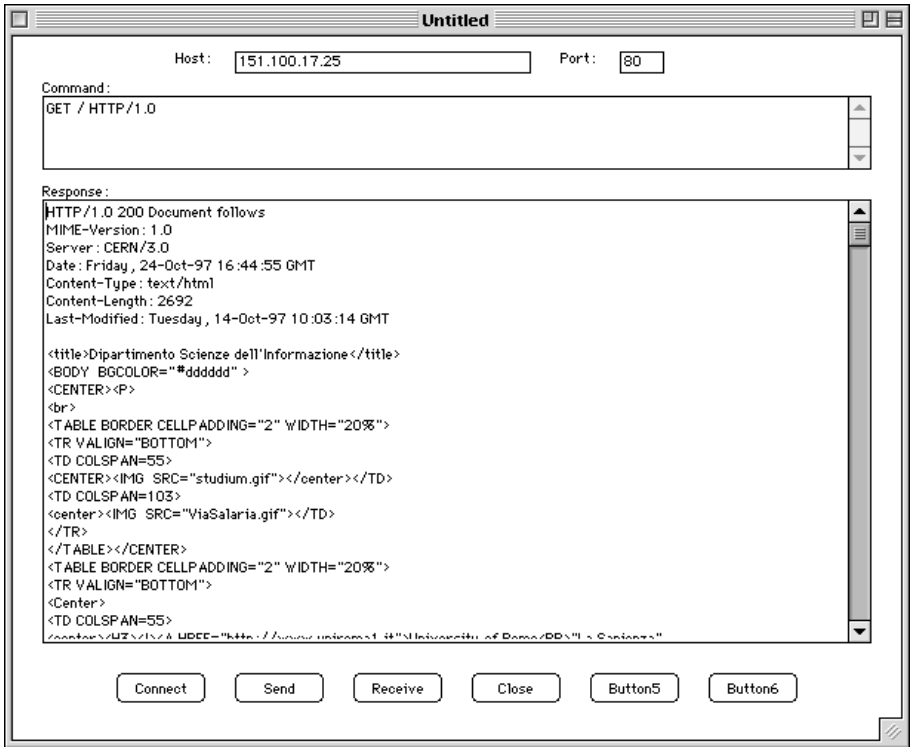


Figura 3-18: Interfaccia utente dell'esempio 4

Il codice è costituito da due classi. La prima, `BaseAppE4`, è basata su quella dell'esempio 1 e provvede ad istanziare la seconda che ha il compito di attivare la connessione e di gestire la comunicazione. Se ne mostrano qui solo i frammenti di codice rilevanti.

```
import java.awt.*;

public class BaseAppE4 extends Frame    {
...ecc.
//-------------------------------------------------
    void button1_Clicked(Event event) {
        baseConn = new BaseConn(textField1.getText(),
                                textField2.getText(),
                                textArea1, textArea2);
}
//-------------------------------------------------
    void button2_Clicked(Event event) {
        baseConn.send();
}
//-------------------------------------------------
void button3_Clicked(Event event) {
        baseConn.receive();
```



```
}
//-------------------------------------------------
    void button4_Clicked(Event event) {
        baseConn.close();
}
//-------------------------------------------------
```

La seconda si occupa della comunicazione.

```
import java.awt.*;
import java.lang.*;
import java.io.*;
import java.net.*;

public class BaseConn {
    TextArea commandArea, responseArea;
    Socket socket = null;
    PrintStream os = null;
    DataInputStream is = null;
//-------------------------------------------------
    public BaseConn(String host, String port,
                    TextArea commandArea, TextArea responseArea) {
        this.commandArea = commandArea;
        this.responseArea = responseArea;
        try {
            socket = new Socket(host, Integer.parseInt(port));
            os = new PrintStream(socket.getOutputStream());
            is = new DataInputStream(socket.getInputStream());
            responseArea.appendText("***Connection established" + "\n");
        } catch (Exception e)   {
            responseArea.appendText("Exception" + "\n");
        }

    }
//-------------------------------------------------
    public void send() {
        os.println(commandArea.getText());
    }
//-------------------------------------------------
    public void receive() {
        String inputLine;
        try {
            inputLine = is.readLine();
            responseArea.appendText(inputLine + "\n");
        } catch (IOException e) {
            responseArea.appendText("IO Exception" + "\n");
        }
    }
//-------------------------------------------------
    public void close() {
        try {
            is.close();
            os.close();
            socket.close();
            responseArea.appendText("***Connection closed" + "\n");
        } catch (IOException e) {
            responseArea.appendText("IO Exception" + "\n");
        }
    }
}
```

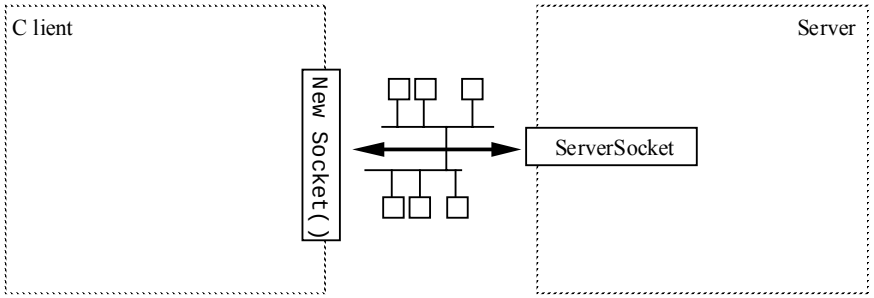
3.2.7.2) Classe ServerSocket

Questa classe costituisce il meccanismo con cui un programma Java agisce da server, e cioè accetta richieste di connessioni provenienti dalla rete.

La creazione di un `ServerSocket`, seguita da una opportuna istruzione di "ascolto", fa si che l'applicazione si metta in attesa di una richiesta di connessione.

Quando essa arriva, il `ServerSocket` crea automaticamente un `Socket` che rappresenta l'estremità locale della connessione appena stabilita. Da tale `Socket` si derivano gli stream che permettono la comunicazione.

Fase di stabilimento della connessione



Connessione stabilita

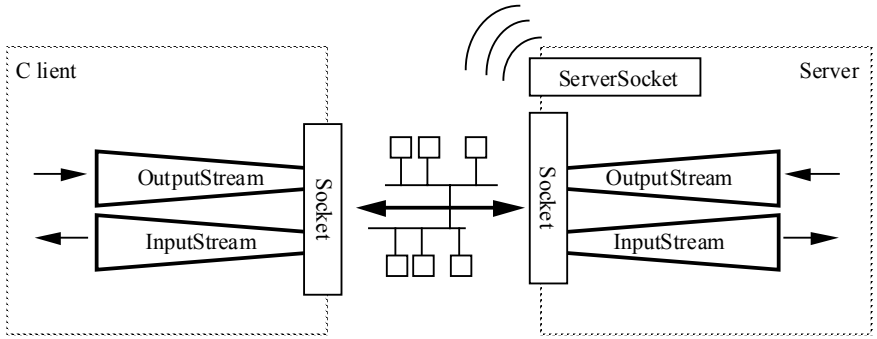


Figura 3-19: Connessione a un `ServerSocket`

Costruttori

```
public ServerSocket(int port) throws IOException;
```

A noi basta questo (ce ne sono due). `port` è il numero del port sul quale il `ServerSocket` si mette in ascolto.

Metodi più importanti

```
public Socket accept() throws IOException;
```

Questo metodo blocca il chiamante finché qualcuno non cerca di connettersi. Quando questo succede, il metodo ritorna restituendo un `Socket` che rappresenta l'estremità locale della connessione.

```
public void close() throws IOException;
```

Chiude il `ServerSocket` (e quindi non ci sarà più nessuno in ascolto su quel port).

```
public int getLocalPort() throws IOException;
```

Restituisce il numero di port su cui il `ServerSocket` è in ascolto.

Esempio 5

Semplice Server che:

- accetta una sola connessione sul port 5000;
- restituisce al Client tutto ciò che riceve.

Il codice è il seguente.

```
import java.io.*;
import java.net.*;

public class SimpleServer   {
//-------------------------------------------------
    public static void main(String args[]) {
        ServerSocket server;
        Socket client;
        String inputLine;
        DataInputStream is = null;
        PrintStream os = null;
        try {
            server = new ServerSocket(5000);
            System.out.println("Accepting one connection...");
            client = server.accept();
            server.close();
            System.out.println("Connection from one client accepted.");
            is = new DataInputStream(client.getInputStream());
            os = new PrintStream(client.getOutputStream());
            os.println("From SimpleServer: Welcome!");
```

```
            while ((inputLine = is.readLine()) != null) {
                System.out.println("Received: " + inputLine);
                os.println("From SimpleServer: " + inputLine);
            }
            is.close();
            os.close();
            client.close();
            System.out.println("Connection closed.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Sia il client che il server visti precedentemente hanno pesanti limitazioni, derivanti dal fatto che essi consistono di un unico flusso di elaborazione:

- client: può leggere una sola linea di testo ad ogni pressione del bottone "Receive". Infatti, se si inserisse un ciclo infinito di lettura, esso bloccherebbe l'esecuzione (dopo l'ultima linea arrivata dal server) in attesa di ulteriore input che non arriverà mai, dato che è impossibile inviare un altro comando;
- server: può gestire una sola connessione. Infatti, un singolo flusso di elaborazione non può, in maniera semplice, gestire una connessione e rimanere nel contempo anche in ascolto di nuove richieste.

Per superare queste limitazioni la soluzione più potente e versatile è ricorrere al ***multithreading***, cioè alla gestione di multipli flussi di elaborazione all'interno dell'applicazione.

3.3) Multithreading

Java supporta il multithreading in modo nativo, a livello di linguaggio. Ciò rende la programmazione di multipli thread molto più semplice che dovendo usare librerie apposite, come è il caso di altri linguaggi.

L'ambiente Java fornisce la classe `Thread` per gestire i thread di esecuzione. Ogni oggetto istanziato da tale classe (o da una sua derivata) costituisce un flusso separato di esecuzione (ossia un thread di esecuzione).

Si noti in proposito che il metodo `main()` di un qualunque oggetto attiva un thread, che termina con la terminazione del `main()` stesso.

3.3.1) Classe Thread

E' la rappresentazione runtime di un thread di esecuzione.

Costruttori

Ce ne sono vari, i più usati sono i tre elencati sotto. Il terzo serve nel caso si faccia uso dell'***interfaccia*** `Runnable` (che vedremo più avanti).

```
public Thread();
public Thread(String name);
public Thread(Runnable target);
```

Metodi più importanti

Ci sono vari metodi statici (che vengono chiamati sul thread corrente), i più utili dei quali sono:

```
public static void sleep(long millis) throws InterruptedException;
```

Questo metodo mette a dormire il thread corrente per `millis` millisecondi. In questo periodo, altri thread possono avanzare nell'esecuzione. Se in questo tempo qualcun altro chiama il suo metodo `interrupt()`, il thread viene risvegliato da una `InterruptedException`.

```
public static void yield() throws InterruptedException;
```

Questo metodo fa si che il thread corrente ceda la Cpu ad altri thread in attesa. Poiché l'ambiente Java non può garantire la ***preemption*** (essa dipende dal sistema operativo) è consigliabile usarlo, quando un thread deve effettuare lunghe computazioni, a intervalli regolari.

I metodi di istanza più utili (che possono essere chiamati su qualunque thread) sono:

```
public synchronized void start() throws IllegalThreadStateException;
```

E' il metodo che si deve chiamare per far partire un thread, una volta creato; è un errore chiamarlo su un thread già avviato.

```
public void run();
```

E' il metodo che l'ambiente runtime chiama quando un thread viene avviato con il metodo `start()`. costituisce il corpo eseguibile del thread, e determina il suo comportamento. Quando esso finisce, il thread termina. E' dunque il metodo che ogni classe derivata da `Thread` deve ridefinire.

```
public void stop();
```

Termina il thread.

```
public void suspend();
```

Sospende il thread; altri possono eseguire.

```
public void resume();
```

Rende il thread nuovamente eseguibile, cioè ***ready*** (Nota: questo non significa che diventi anche ***running***, in quanto ciò dipende anche da altri fattori).

In più, ci sono altri metodi per gestire la priorità, avere notizie sullo stato del thread, ecc.

Esempio 6

Programma che consiste di due thread, i quali scrivono un messaggio ciascuno sullo standard output con cadenze differenti.

```
import java.io.*;
import java.lang.*;

public class PingPong extends Thread {
    String word;
    int delay;
//--------------------------------------------------
    public PingPong (String whatToSay, int delayTime) {
        word = whatToSay;
        delay = delayTime;
    }
//--------------------------------------------------
    public void run () {
        try {
            while (true) {
                System.out.println(word);
                sleep(delay);
            }
        } catch (InterruptedException e) {
            return;
        }
    }
//--------------------------------------------------
    public static void main(String args[]) {
        new PingPong("ping", 250).start();  //1/4 sec.
        new PingPong("PONG", 100).start();  //1/10 sec.
    }
}
```

3.3.2) Interfaccia Runnable

Vi sono alcune situazioni nelle quali il ricorso a una estensione della classe `Thread` non è adatto agli scopi, poiché:

- l'oggetto per il quale si vuole avviare un thread è già estensione di una qualche altra classe (e quindi non può estendere anche `Thread`);
- si vogliono avviare molteplici thread dentro la stessa istanza di un oggetto, cosa che è impossibile estendendo la classe `Thread`.

La risposta a questo problema è data dalla interfaccia `Runnable`, che include un solo metodo:

```
public abstract void run();
```

dall'ovvio significato: in esso si specifica il comportamento del thread da creare, come visto prima.

Basta quindi definire una classe che implementa tale interfaccia:

```
public MyClass implements Runnable {
...
}
```

ed includervi un metodo `run()`:

```
public abstract void run(){
...
}
```

per avere la possibilità di lanciare thread multipli all'interno di un oggetto di tale classe.

Tali thread vengono creati col terzo dei costruttori visti per la classe `Thread` e successivamente vengono attivati invocandone il metodo `start()`, che a sua volta causa l'avvio del metodo `run()` dell'oggetto che è passato come parametro nel costruttore:

```
...
new Thread(theRunnableObject).start();
...
```

Esempio 7

Funzionalità simile all'esempio precedente ma ottenuta con l'interfaccia `Runnable`.

```
import java.io.*;
import java.lang.*;

public class RPingPong /*extends AnotherClass*/ implements Runnable {
    String word;
    int delay;
//--------------------------------------------------
    public RPingPong (String whatToSay, int delayTime) {
        word = whatToSay;
        delay = delayTime;
    }
//--------------------------------------------------
    public void run () {
        try {
            while (true) {
                System.out.println(word);
                Thread.sleep(delay);
            }
        } catch (InterruptedException e) {
```

```
            return;
        }
    }
//---------------------------------------------------
    public static void main(String args[]) {
        Runnable ping = new RPingPong("ping", 100); //1/10 sec.
        Runnable PONG = new RPingPong("PONG", 100); //1/10 sec.
        new Thread(ping).start();
        new Thread(ping).start();
        new Thread(PONG).start();
        new Thread(PONG).start();
        new Thread(PONG).start();
        new Thread(PONG).start();
        new Thread(PONG).start();
    }
}
```

Esempio 8

Applicazione che apre una connessione di rete, come nell'esempio 4. La differenza è che un thread separato rimane in ascolto delle risposte del server: questo permette di ricevere risposte costituite da più linee di testo senza alcun problema.

Il codice è costituito da due classi. La prima, `BaseAppE8`, è basata su quella dell'esempio 4 e provvede ad istanziare la seconda che ha il compito di attivare la connessione e di gestire la comunicazione. Se ne mostrano qui solo i frammenti di codice rilevanti, sottolineando il fatto che non c'è più bisogno del bottone "Receive", visto che un thread separato rimane in costante ascolto delle risposte.

```
import java.awt.*;

public class BaseAppE8 extends Frame    {
...ecc.
//---------------------------------------------------
    void button1_Clicked(Event event) {
        baseTConn = new BaseTConn(textField1.getText(),
                                  textField2.getText(),
                                  textArea1, textArea2);
    }
//---------------------------------------------------
    void button2_Clicked(Event event) {
        baseTConn.send();
    }
//---------------------------------------------------
    void button3_Clicked(Event event) {
        baseTConn.close();
    }
//---------------------------------------------------
```

La seconda classe si occupa della comunicazione, ed inolte attiva tramite il metodo `run()` un thread separato costituito da un ciclo infinito in cui si ricevono le risposte del server e si provvede a mostrarle sullo schermo.

```
import java.awt.*;
```



```
import java.lang.*;
import java.io.*;
import java.net.*;

public class BaseTConn implements Runnable {
    TextArea commandArea, responseArea;
    Socket socket = null;
    PrintStream os = null;
    DataInputStream is = null;
//---------------------------------------------------
    public BaseTConn(String host, String port,
                     TextArea commandArea, TextArea responseArea) {
        this.commandArea = commandArea;
        this.responseArea = responseArea;
        try {
            socket = new Socket(host, Integer.parseInt(port));
            os = new PrintStream(socket.getOutputStream());
            is = new DataInputStream(socket.getInputStream());
            responseArea.appendText("***Connection established" + "\n");
            new Thread(this).start();
        } catch (Exception e)   {
            responseArea.appendText("Exception" + "\n");
        }
    }
//---------------------------------------------------
    public void run() {
        String inputLine;
        try {
            while ((inputLine = is.readLine()) != null) {
                responseArea.appendText(inputLine + "\n");
            }
        } catch (IOException e) {
            responseArea.appendText("IO Exception" + "\n");
        }
    }
//---------------------------------------------------
    public void send() {
            os.println(commandArea.getText());
    }
//---------------------------------------------------
    public void close() {
        try {
            is.close();
            os.close();
            socket.close();
            responseArea.appendText("***Connection closed" + "\n");
        } catch (IOException e) {
            responseArea.appendText("IO Exception" + "\n");
        }
    }
}
```

3.4 Sincronizzazione

Come in tutti i regimi di concorrenza, anche in Java possono sorgere problemi di consistenza dei dati condivisi se i thread sono ***cooperanti***.

I dati condivisi fra thread differenti possono essere:
- variabili statiche di una classe che estende `Thread`, che sono quindi condivise da tutte le sue istanze;
- variabili di istanza di un oggetto `Runnable`, che sono condivise da tutti i thread attivati dentro tale oggetto.

Per risolvere i problemi legati alla concorrenza, in Java è stata incorporata nella classe `Object` (e quindi in tutte le altre, che derivano da essa) e nelle sue istanze la capacità potenziale di funzionare come un ***monitor***.

3.4.1) Metodi sincronizzati

In particolare, tale funzionalità si attiva ricorrendo ai ***metodi sincronizzati***, definiti come:

```
public void synchronized aMethod(...) {
...
}
```

Quando un thread chiama un metodo sincronizzato di un oggetto, acquisisce un ***lucchetto*** su quell'oggetto. Nessun altro thread può chiamare un qualunque metodo sincronizzato dello stesso oggetto finché il lucchetto non viene rilasciato. Se lo fa, verrà messo in attesa che ciò avvenga.

Si noti che eventuali metodi non sincronizzati di quell'oggetto possono comunque essere eseguiti da qualunque thread, in concorrenza col thread che possiede il lucchetto.

Dunque:
- relativamente all'insieme dei suoi metodi sincronizzati, un oggetto si comporta come un monitor, garantendo per essi un accesso in mutua esclusione;
- di conseguenza, è sufficiente gestire i dati condivisi per mezzo di metodi sincronizzati per garantire un corretto funzionamento dei thread cooperanti.

Ad esempio, consideriamo una classe che rappresenta un acconto bancario, sul quale possono potenzialmente essere fatte molte operazioni contemporaneamente:

```
public class Account    {
    private double balance;
//-------------------------------------------------
    public Account(double initialDeposit) {
        balance = initialDeposit;
    }
//-------------------------------------------------
    public synchronized double getBalance() {
        return balance;
    }
//-------------------------------------------------
    public synchronized void deposit(double amount) {
        balance += amount;
    }
}
```

Non può succedere che un thread legga il valore del conto mentre un altro thread lo aggiorna, o che due thread lo aggiornino in concorrenza.

Si noti che il costruttore non ha bisogno di essere sincronizzato, perché è eseguito solo per creare un oggetto, il che avviene una sola volta per ogni nuovo oggetto.

Anche i metodi statici possono essere sincronizzati. In questo caso il lucchetto è relativo alla classe e non alle sue istanze. In altre parole, sia una classe che le sue istanze possono funzionare come monitor. Va notato come tali monitor siano indipendenti, cioè il lucchetto sulla classe non ha alcun effetto sui metodi sincronizzati delle sue istanze e viceversa.

3.4.2) Istruzioni sincronizzate

E' possibile eseguire del codice sincronizzato, che quindi attiva il lucchetto di un oggetto, anche senza invocare un metodo sincronizzato di tale oggetto. Ciò si ottiene con le ***istruzioni sincronizzate***, che hanno questa forma:

```
...
synchronized(anObject) {
... // istruzioni sincronizzate
}
...
```

Questo modo di ottenere la sincrinizzazione fra thread cooperanti richiede maggior attenzione da parte del programmatore, che deve inserire blocchi di istruzioni sincronizzate in tutte le parti di codice interessate.

Esempio 9

Applicazione che realizza un server multithreaded, il quale:

- accetta connessioni da molteplici client;
- invia a tutti quelli connessi, in ***broadcast***, ciò che riceve da uno qualunque di loro.

Costituisce, dunque, un server per una semplice ***chatline***. Può essere usato con una semplice variazione del client visto nell'esempio 8: il client, appena si connette, deve inviare una linea di testo che il server usa come identificativo dell'utente connesso.

In tal modo si realizza di fatto una minimale architettura client-server.

Il codice è costituito da due classi. La prima, `ChatServer`, accetta richieste di connessione sul port 5000 e, ogni volta che ne arriva una, istanzia un oggetto della classe `ChatHandler` che si occupa di gestirla.

```
import java.net.*;
import java.io.*;
import java.util.*;

    public class ChatServer {
//-------------------------------------------------
    public ChatServer() throws IOException {
        ServerSocket server = new ServerSocket(5000);
        System.out.println ("Accepting connections...");
        while(true) {
            Socket client = server.accept();
            System.out.println ("Accepted from " + client.getInetAddress());
            new ChatHandler(client).start();
        }
    }
//-------------------------------------------------
    public static void main(String args[]) throws IOException {
        new ChatServer();
    }
}
```

La seconda si occupa della gestione di una singola connessione e dell'invio a tutte le altre, in broadcast, dei dati provenienti da tale connessione.

```
import java.net.*;
import java.io.*;
import java.util.*;

public class ChatHandler extends Thread {
  protected static Vector handlers = new Vector();
  protected String userName;
  protected Socket socket;
  protected DataInputStream is;
  protected PrintStream os;
//-------------------------------------------------
  public ChatHandler(Socket socket) throws IOException {
    this.socket = socket;
```



```
    is = new DataInputStream(new BufferedInputStream(socket.getInputStream()));
    os = new PrintStream(new BufferedOutputStream(socket.getOutputStream()));
  }
//-------------------------------------------------
  public void run() {
    try {
      userName = is.readLine();
      os.println("Salve " + userName + ", benvenuto nella chatline di Reti II");
      os.flush();
      broadcast(this, "Salve a tutti! Ci sono anch'io.");
      handlers.addElement(this); //e' un metodo sincronizzato
      while (true) {
        String msg = is.readLine();
        if (msg != null) broadcast(this, msg);
        else break;
      }
    } catch(IOException ex) {
      ex.printStackTrace();
    } finally {
      handlers.removeElement(this); //e' un metodo sincronizzato
      broadcast(this, "Arrivederci a presto.");
      try {
        socket.close();
      } catch(Exception ex) {
        ex.printStackTrace();
      }
    }
  }
//-------------------------------------------------
  protected static void broadcast(ChatHandler sender, String message) {
    synchronized (handlers) { //ora nessuno puo' aggiungersi o abbandonare
      Enumeration e = handlers.elements();
      while (e.hasMoreElements()) {
        ChatHandler c = (ChatHandler) e.nextElement();
        try {
          c.os.print("Da " + sender.userName);
          c.os.print("@" + sender.socket.getInetAddress().getHostName() + ": ");
          c.os.println(message);
          c.os.flush();
        } catch(Exception ex) {
          c.stop();
        }
      }
    }
  }
}
```

3.4.3) Wait() e Notify()

In Java esiste anche un meccanismo per sospendere e risvegliare i thread che si trovano all'interno di un monitor, analogo a quello offerto dalle variabili di condizione.

Esistono tre metodi della classe Object che servono a questo:

```
public final void wait() throws InterruptedException;
```

Un thread che chiama questo metodo di un oggetto viene sospeso finché un altro thread non chiama `notify()` o `notifyAll()` su quello stesso oggetto. Il lucchetto su quell'oggetto viene temporaneamente e atomicamente rilasciato, così altri thread possono entrare.

```
public final void notify() throws IllegalMonitorStateException;
```

Questo metodo, chiamato su un oggetto, risveglia un thread (quello in attesa da più tempo) fra quelli sospesi dentro quell'oggetto. In particolare, tale thread riprenderà l'esecuzione solo quando il thread che ha chiamato `notify()` rilascia il lucchetto sull'oggetto.

```
public final void notifyAll() throws IllegalMonitorStateException;
```

Questo metodo, chiamato su un oggetto, risveglia tutti i thread sospesi su quell'oggetto. Naturalmente, quando il chiamante rilascerà il lucchetto, solo uno dei risvegliati riuscirà a conquistarlo e gli altri verranno nuovamente sospesi.

E' importante ricordare che questi metodi possono essere usati solo all'interno di codice sincronizzato sullo stesso oggetto per il quale si invocano, e cioè:

- dentro metodi sincronizzati di quell'oggetto;
- dentro istruzioni sincronizzate su quell'oggetto.

Esempio 10

Esempio di produttore-consumatore che fa uso di `wait()` e `notify()`. La classe `Consumer` rappresenta un consumatore, che cerca di consumare un oggetto (se c'è) da un `Vector`.

Usa un thread separato per questo, e sfrutta `wait()` per rimanere in attesa di un oggetto da consumare.

Il produttore è costituito dall'utente, che attraverso il thread principale (quello del `main()`) aggiunge oggetti e invia `notify()` per risvegliare il consumatore.

```
import java.util.*;
import java.io.*;

public class Consumer extends Thread {
    protected Vector objects;
//-------------------------------------------------
    public Consumer () {
        objects = new Vector();
    }
//-------------------------------------------------
    public void run () {
        while (true) {
            Object object = extract ();
```



```
            System.out.println (object);
        }
    }
//-------------------------------------------------
    protected Object extract () {
        synchronized (objects) {
            while (objects.isEmpty ()) {
                try {
                    objects.wait ();
                } catch (InterruptedException ex) {
                    ex.printStackTrace ();
                }
            }
            Object o = objects.firstElement ();
            objects.removeElement (o);
            return o;
        }
    }
//-------------------------------------------------
    public void insert (Object o) {
        synchronized (objects) {
            objects.addElement (o);
            objects.notify ();
        }
    }
//-------------------------------------------------
    public static void main (String args[]) throws IOException,
                                                   InterruptedException {
        Consumer c = new Consumer ();
        c.start ();
        DataInputStream i = new DataInputStream (System.in);
        String s;
        while ((s = i.readLine ()) != null) {
            c.insert (s);
            Thread.sleep (1000);
        }
    }
}
```

4) Utilizzo dei messaggi

Vedremo ora come costruire, su uno stream quale quello offerto da una connessione TCP, una comunicazione basata su un flusso di messaggi anziché, come fatto finora, su un flusso di byte.

Un ***messaggio*** è una quantità arbitraria di informazioni (in genere tali da avere una propria autosufficienza logica e funzionale) che vengono corredate da ***informazioni di controllo*** (***CI***) e quindi inviate come un'unità a se stante che prende il nome di ***pacchetto***.

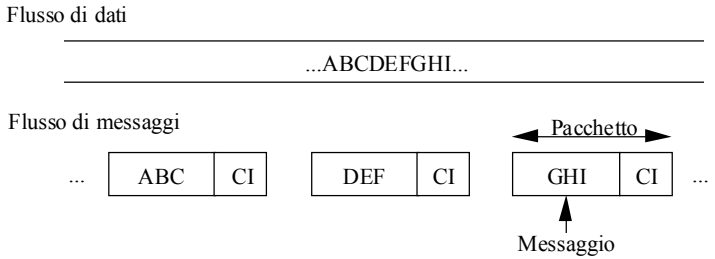


Figura 4-1: Dati, messaggi e pacchetti

Si noti come, per una comunicazione di questo tipo, non sia strettamente necessaria una connessione basata su TCP.

Anche UDP sarebbe adeguato, se non fosse per il fatto che non è un protocollo affidabile. Di conseguenza, noi ricaveremo il supporto alla gestione di messaggi a partire da stream affidabili quali quelli offerti da TCP.

I principali benefici derivanti dall'uso di messaggi sono i seguenti:

1. facilità di multiplexing e demultiplexing. Se su una singola connessione incidono più trasmissioni indipendenti, è più facile gestire tale situazione incapsulando i dati da trasmettere. Il server può estrarli indipendentemente gli uni dagli altri, senza entrare nel merito dei contenuti;
2. gli errori sono confinati all'interno di un messaggio. Per la stessa ragione di prima, eventuali errori presenti in un messaggio non influenzano i messaggi successivi. Senza l'incapsulamento l'errore potrebbe rendere impossibile determinare il confine fra due messaggi;
3. facilità di trasmettere dati di natura diversa sulla stessa connessione. Ciò è strettamente legato al punto 1. Infatti, dato che ogni messaggio è corredato di informazioni di controllo, esse consentono di comunicare metainformazioni sulla natura dei dati contenuti nel messaggio.

Tipicamente i messaggi possono essere usati per:

- corredare le informazioni trasmesse di una minimale informazione di controllo, ad esempio la lunghezza del messaggio (sarà la prima cosa che vedremo);
- consentire a parti diverse di una stessa applicazione di condividere lo stesso stream per la comunicazione (la seconda cosa che vedremo);
- consentire a un'applicazione di specificare, per mezzo delle informazioni di controllo, una lista di destinatari. Ciò consente trasmissioni di tipo ***multicast***. Questa tecnica non la vedremo in dettaglio, ma sarà chiaro come possa essere implementata.

4.1) Classi di base per la gestione di messaggi

4.1.1) Classe MessageOutput

E' la superclasse astratta da cui derivano tutti gli stream per la gestione di messaggi (che indicheremo col termine generico di ***message stream***) di output.

E' un `FilterOutputStream` (e quindi si deve attaccare a un `OutputStream`) e, per convenienza, estende `DataOutputStream`.

Ai metodi di quest'ultima classe aggiunge tre metodi `send()`.

Su un message stream di output si opera in questo modo:

- usando i normali metodi di `DataOutputStream` si scrive un messaggio, finché esso non è completato (tipicamente usando un buffer interno);
- usando uno dei metodi `send()` lo stream effettua le seguenti operazioni:
  - incapsula il messaggio in un pacchetto, corredandolo delle informazioni di controllo;
  - invia il pacchetto sul canale di comunicazione;
  - azzera il buffer interno.

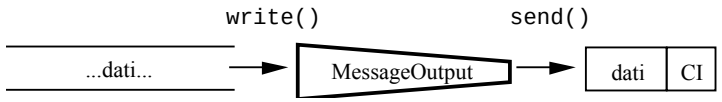


Figura 4-2: Funzionamento di un message stream di output

La definizione della classe è la seguente (dal libro "Java Network Programming" di Merlin Hughes et al., Mannig Publications Co, Greenwich CT, 1997).

```
/* Copyright (c) 1996, 1997 Prominence Dot Com, Inc.        *
 * See the file legal.txt in the txt directory for details. */

package prominence.msg;

import java.io.*;

/**
 * The superclass for all message output streams.
 * <p>Extends <tt>DataOutputStream</tt> and adds <tt>send()</tt>
 * methods that send the current message to the attached channel.
 *
 * @version 1.0 1 Nov 1996
 * @author Merlin Hughes
 * @see prominence.msg.MessageInput
 */
public abstract class MessageOutput extends DataOutputStream {
  /**
   * Creates a new <tt>MessageOutput</tt>.
   * @param out Stream to which to write message data.
   */
  public MessageOutput (OutputStream out) {
    super (out);
  }

  /**
   * Sends the current message to the attached channel.
   * <p>Subclasses will extend this class and implement this
   * method as appropriate for a particular communications
   * channel.
   * @exception IOException Occurs if there is a problem sending
   * a message.
   */
  public abstract void send () throws IOException;

  /**
   * Sends the current message to the attached channel with
   * a routing header that indicates a list of recipients.
   * <p>Subclasses that support this method will override it
   * with an appropriate implementation. The default implementation
   * is to throw an exception.
   * @param dst The list of intended recipients
   * @exception IOException Occurs if there is a problem sending
   * a message or this method is not supported.
   */
  public void send (String[] dst) throws IOException {
    throw new IOException ("send[] not supported");
  }

  /**
   * Sends the current message to the attached channel with
   * a routing header that indicates a single recipient.
   * <p>The default implementation of this method calls the
   * previous method with a single-element array.
   * @param dst The intended recipient
   * @exception IOException Occurs if there is a problem sending
   * a message or targeted sending is not supported.
   */
  public void send (String dst) throws IOException {
    String[] dsts = { dst };
    send (dsts);
  }
}
```

Note

- Si estende `DataOutputStream`, ereditando quindi i suoi metodi.
- Il costruttore accetta un `OutputStream` a cui attaccarsi e lo passa al costruttore della superclasse. I vari metodi di scrittura della superclasse scriveranno direttamente su tale stream. Esso, nelle classi derivate, tipicamente sarà un `ByteArrayOutputStream`, ossia uno stream che scrive in un buffer (costituito da un array di byte) in memoria.
- `send()` trasmette materialmente il messaggio (dopo averlo incapsulato) sul canale di comunicazione, cioé scrive l'intero pacchetto sull'`OutputStream` a cui è attaccato.
- Le altre due varianti del metodo `send(…)`, che noi non useremo, predispongono alla trasmissione multicast.

4.1.2) Classe MessageInput

E' la superclasse astratta da cui derivano tutti i message stream di input.

E' un `FilterInputStream` (e quindi si deve attaccare a un `InputStream`) e, per convenienza, estende `DataInputStream`.

Ai metodi di quest'ultima classe aggiunge un metodo `receive()`.

Con un message stream di input opera in questo modo:

- chiamando `receive()` ci si blocca in attesa che un pacchetto sia ricevuto dal canale di comunicazione. Quando ciò accade, il messaggio viene estratto dal pacchetto e diviene disponibile;
- chiamando i vari metodi di lettura di `DataInputStream` si leggono i dati del messaggio;
- chiamando nuovamente `receive()` si attende un nuovo pacchetto, il cui contenuto sovrascrive il messaggio precedente (anche se quest'ultimo non è stato completamente letto).

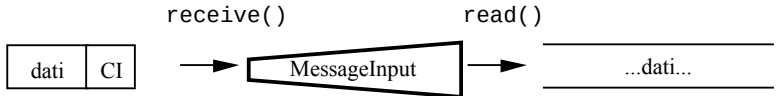


Figura 4-3: Funzionamento di un message stream di input

La definizione della classe è la seguente (dal libro "Java Network Programming" di Merlin Hughes et al., Mannig Publications Co, Greenwich CT, 1997).

```
/* Copyright (c) 1996, 1997 Prominence Dot Com, Inc.        *
 * See the file legal.txt in the txt directory for details. */

package prominence.msg;

import java.io.*;

/**
 * The superclass for all message input streams.
 * <p>Extends <tt>DataInputStream</tt> and adds a <tt>receive()</tt>
 * method that receives a new message from the attached channel.
 *
 * @version 1.0 1 Nov 1996
 * @author Merlin Hughes
 * @see prominence.msg.MessageOutput
 */
public abstract class MessageInput extends DataInputStream {
  /**
   * Creates a new <tt>MessageInput</tt>.
   * @param in Stream from which to read message data
   */
  public MessageInput (InputStream in) {
    super (in);
  }

  /**
   * Receives a new message from the attached channel and makes
   * it available to read through the standard <tt>DataInputStream</tt>
   * methods.
   * <p>Subclasses will extend this class and implement this method
   * as appropriate for a particular communications channel.
   * @exception IOException Occurs if there is a problem receiving
   * a new message.
   */
  public abstract void receive () throws IOException;
}
```

Note

- Si estende `DataInputStream`, ereditando i suoi metodi.
- Il costruttore accetta un `InputStream` a cui attaccarsi e lo passa al costruttore della superclasse. I vari metodi di lettura della superclasse leggeranno direttamente da tale stream. Esso, nelle classi derivate, sarà tipicamente uno stream che legge da un buffer in memoria.
- `receive()` si blocca finché non arriva un pacchetto, estrae il messaggio e lo rende disponibile nel buffer di cui sopra.

4.1.3) Classe MessageOutputStream

Questa è la prima implementazione di `MessageOutput` che vedremo, e si attacca a un `OutputStream`.

Questa classe, in particolare, incapsula i messaggi in pacchetti la cui Control Information è costituita dalla lunghezza in byte del messaggio. Ciò consente al destinatario di sapere in anticipo quanto è grande il messaggio, senza doverne analizzare alcuna parte.

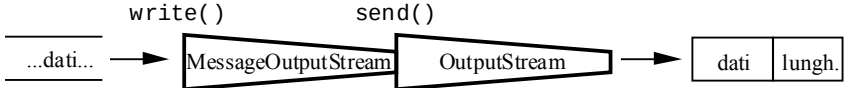


Figura 4-4: Funzionamento di `MessageOutputStream`

Internamente, il meccanismo di buffering si ottiene per mezzo di un `ByteArrayOutputStream`.

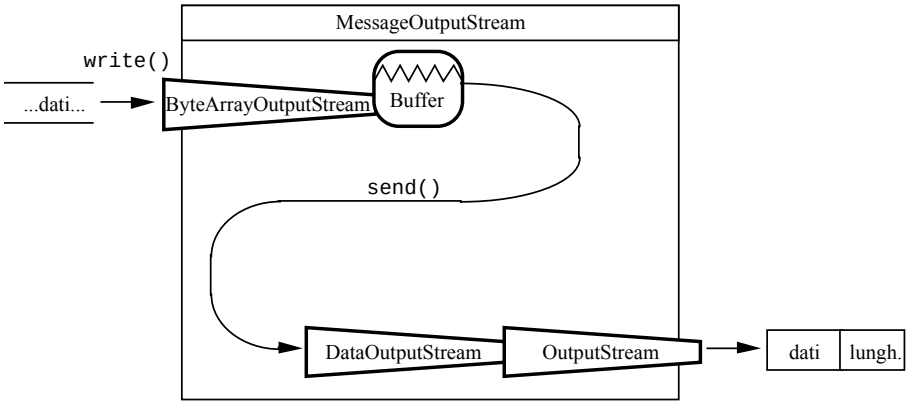


Figura 4-5: Implementazione di `MessageOutputStream`

La definizione della classe è la seguente (dal libro "Java Network Programming" di Merlin Hughes et al., Mannig Publications Co, Greenwich CT, 1997).

```
/* Copyright (c) 1996, 1997 Prominence Dot Com, Inc.        *
 * See the file legal.txt in the txt directory for details. */

package prominence.msg;

import java.io.*;
```



```
/**
 * A <tt>MessageOutput</tt> that writes messages to an attached
 * <tt>OutputStream</tt>.
 *
 * @version 1.0 1 Nov 1996
 * @author Merlin Hughes
 * @see prominence.msg.MessageInputStream
 */
public class MessageOutputStream extends MessageOutput {
  /**
   * The attached <tt>OutputStream</tt>.
   */
  protected OutputStream o;
  /**
   * A <tt>DataOutputStream</tt> attached to <tt>o</tt>.
   */
  protected DataOutputStream dataO;
  /**
   * A <tt>ByteArrayOutputStream</tt> used to buffer the current message.
   */
  protected ByteArrayOutputStream byteO;

  /**
   * Creates a new <tt>MessageOutputStream</tt>. Message data is
   * buffered internally until <tt>send()</tt> is called.
   * @param o The <tt>OutputStream</tt> to which to write messages
   */
  public MessageOutputStream (OutputStream o) {
    super (new ByteArrayOutputStream ());
    byteO = (ByteArrayOutputStream) out;
    this.o = o;
    dataO = new DataOutputStream (o);
  }

  /**
   * Sends a message to the attached stream. The message body length
   * is written as an <tt>int</tt>, followed by the message body itself;
   * the internal message buffer is then reset.
   * <p>This method synchronizes on the attached stream.
   * @exception IOException Occurs if there is a problem writing to
   * the attached stream.
   */
  public void send () throws IOException {
    synchronized (o) {
      dataO.writeInt (byteO.size ());
      byteO.writeTo (o);
    }
    byteO.reset ();
    o.flush ();
  }
}
```

Note

- la classe estende `MessageOutput` e quindi `DataOutputStream`;
- i metodi di `DataOutputStream` scrivono nel buffer interno;
- `send()` invia un pacchetto, formato dalla lunghezza dati più il contenuto del buffer, sullo stream di output a cui il `MessageOutputStream` è attaccato.

Costruttore

- Chiama quello di `MessageOutput`, il quale a sua volta chiama quello di `DataOutputStream`. Però, al costruttore della superclasse non viene passato il vero `OutputStream` a cui ci si deve attaccare, ma un nuovo `ByteArrayOutputStream`, creato

per l'occasione, che costituirà il buffer interno. Di conseguenza, la superclasse sarà attaccata a tale buffer, e i suoi metodi di scrittura opereranno su di esso.

- Ricordiamo che `FilterOutputStream` tiene in una variabile `out` una reference all'`OutputStream` a cui è attaccato (cioè quello che gli viene passato nel costruttore). Usiamo tale variabile per ricavare una reference al buffer interno, `byteO`, che serve per usare i metodi di `ByteArrayOutputStream`.
- Dopodiché attacchiamo al vero `OutputStream`, cioè ad `o`, un `DataOutputStream` (per poterne usare i metodi) e utilizziamo `dataO` come reference ad esso.
- Chiudere con `close()` il MessageOutputStream, che non è attaccato al vero `OutputStream`, non chiude quest'ultimo ma il buffer interno `byteO`, il che non ha alcun effetto.

Metodi

- Tutti i normali metodi di `DataOutputStream` scriveranno dunque dentro `byteO`, il buffer interno.
- Il metodo `send()`, invece, compie le seguenti azioni (bloccandosi se necessario):
  - si sincronizza sul vero `OutputStream`, garantendo così una corretta gestione del canale di comunicazione nel caso vi fossero attacati molteplici `MessageOutputStream`;
  - scrive sull'`OutputStream` un intero pari alla dimensione corrente del buffer;
  - scarica il contenuto del buffer sull'`OutputStream`;
  - resetta il buffer interno;
  - chiama `flush()` per inviare subito il pacchetto (lunghezza più dati).

4.1.4) Classe MessageInputStream

Questa classe, che estende `MessageInput`, si attacca a un `InputStream` e va usata per leggere i dati che provengono da un `MessageOutputStream`.

Essa, col metodo `receive()`, estrae il messaggio dal pacchetto (eliminando la CI, costituita dalla dimensione del messaggio) e quindi rende disponibile il contenuto del messaggio ai normali metodi di lettura (quelli di `DataInputStream`), memorizzandolo in un buffer interno.

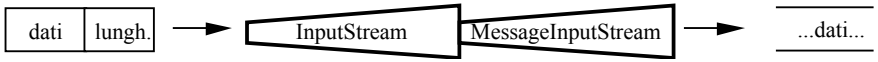


Figura 4-6: Funzionamento di `MessageInputStream`

Internamente, si usa un ByteArrayInputStream per implementare il meccanismo di buffering.

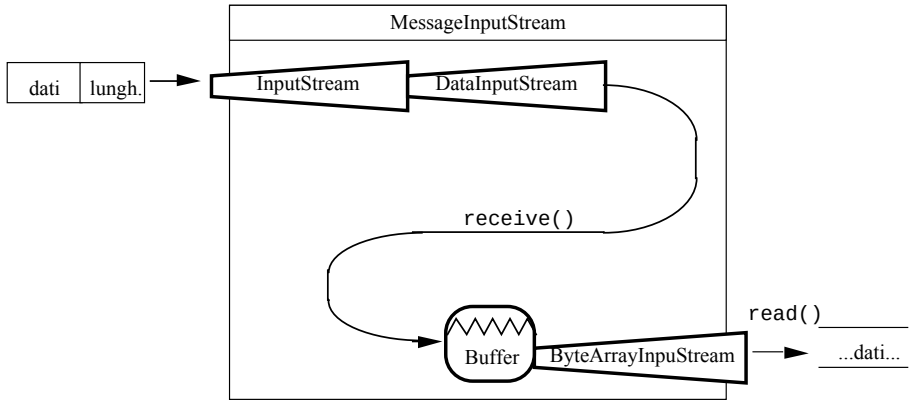


Figura 4-7: Implementazione di `MessageInputStream`

La definizione della classe è la seguente (dal libro "Java Network Programming" di Merlin Hughes et al., Mannig Publications Co, Greenwich CT, 1997).

```
/* Copyright (c) 1996, 1997 Prominence Dot Com, Inc.        *
 * See the file legal.txt in the txt directory for details. */

package prominence.msg;

import java.io.*;

/**
 * A <tt>MessageInput</tt> that reads messages from an
 * attached <tt>InputStream</tt>.
 *
 * @version 1.0 1 Nov 1996
 * @author Merlin Hughes
 * @see prominence.msg.MessageOutputStream
 */
public class MessageInputStream extends MessageInput {
  /**
   * The attached <tt>InputStream</tt>.
   */
  protected InputStream i;
  /**
   * A <tt>DataInputStream</tt> attached to <tt>i</tt>.
   */
  protected DataInputStream dataI;

  /**
   * Creates a new <tt>MessageInputStream</tt>.
   * @param i The <tt>InputStream</tt> from which to read messages.
   */
  public MessageInputStream (InputStream i) {
    super (null);
    this.i = i;
    dataI = new DataInputStream (i);
  }

  /**
   * A buffer containing the most recently received message.
```

```
   * <p>This variable is exposed to permit potential optimizations.
   */
  byte[] buffer;

  /**
   * Receives a message from the attached stream. An <tt>int</tt>
   * header is read, which indicates the length of the message body.
   * The message body is then read and made available through the
   * usual superclass <tt>read()</tt> methods.
   * <p>This method synchronizes on the attached stream.
   * @exception IOException Occurs if there is a problem reading
   * from the attached stream.
   */
  public void receive () throws IOException {
    synchronized (i) {
      int n = dataI.readInt ();
      buffer = new byte[n];
      dataI.readFully (buffer);
    }
    in = new ByteArrayInputStream (buffer);
  }
}
```

Note

- la classe estende `MessageInput` e quindi `DataInputStream`;
- i metodi di `DataInputStream` leggeranno dal buffer interno (se è stata fatta almeno una `receive()`);
- `receive()` crea un buffer, lo riempie col messaggio e lo attacca al `MessageInputStream`.

Costruttore

- Chiama quello di `MessageInput`, e quindi quello di `DataInputStream`, che però (come prima) non vengono attaccati al vero `InputStream`. In questo caso anzi vengono attaccati a un `null`, perché non ha senso fare normali letture prima che sia arrivato un pacchetto: in tal caso si genera un'eccezione.
- crea in `dataI` un nuovo `DatainputStream` attaccato al canale di comunicazione, per poterne usare i metodi.

Metodi

- Tutti i normali metodi di `DataInputStream` leggeranno da `in` e quindi dal buffer interno.
- Il metodo `receive()` compie le seguenti operazioni (bloccandosi finché non arriva un messaggio):
  - si sincronizza sul canale di comunicazione;
  - legge un intero, che indica la dimensione del messaggio;
  - crea un array di byte di pari dimensioni;
  - legge dal canale di comunicazione un corrispondente numero di byte;
  - crea un nuovo `ByteArrayInputStream`, attaccato all'array appena riempito;
  - attacca il `MessageInputStream` al `ByteArrayInputStream` appena creato, assegnando quest'ultimo a `in` (la variabile di `FilterInputStream` nella quale si memorizza l'`InputStream` a cui il `FilterInputStream` è attaccato).

4.2) Un'applicazione client-server per la gestione di transazioni

Si deve notare che:

- molti `MessageOutputStream` possono essere attaccati a un singolo `OutputStream`
- molti `MessageInputStream` possono essere attaccati a un singolo `InputStream`.

I message stream opereranno comunque correttamente, anche se pilotati da corrispondenti thread in concorrenza, grazie all'incapsulamento e alla sincronizzazione.

Ciò è utile, ad esempio, in un'applicazione per la gestione di transazioni, che si svolgono ciascuna secondo il seguente schema:

1. una richiesta del client;
2. una elaborazione conseguente effettuata dal server (che può richiedere anche molto tempo);
3. una risposta del server.

Pensiamo a un server multithreaded, in cui ogni thread gestisce sequenzialmente una transazione dopo l'altra.

I vari thread operano in concorrenza, per cui mentre uno elabora una transazione, un altro ne accetta una nuova, e così via.

Naturalmente, molte richieste viaggiano assieme in una direzione, e molte risposte viaggiano assieme nell'altra direzione.

Grazie alla struttura di messaggi, queste trasmissioni non si disturbano a vicenda.

Il server:

- contiene una `HashTable` che mappa attributi in valori;
- attiva un numero di thread deciso dall'utente per gestire un corrispondente numero massimo di transazioni in concorrenza.

Il client:

- può leggere (con `get()`) o modificare (con `put()`) il valore di un attributo (queste sono le uniche due possibili transazioni).

4.2.1) Classe TransactionClient

E' il cliente che si connette al corrispondente server.

Usa `MessageOutputStream` e `MessageInputStream` per comunicare. Presenta all'utente due campi testo e due bottoni:

- col bottone "get" si chiede il valore di un attributo;
- col bottone "put" si cambia il valore di un attributo.

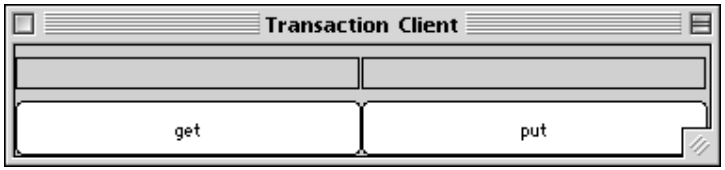


Figura 4-8: Interfaccia utente del client

La definizione della classe è la seguente; si noti che rispetto alla versione del libro ("Java Network Programming" di Merlin Hughes et al., Mannig Publications Co, Greenwich CT, 1997) c'è una variazione sostanziale, in quanto la ricezione dei dati del server avviene in un thread separato, altrimenti sia il client che il server si bloccano su `receive()`.

```
import java.awt.*;
import java.io.*;
import java.net.*;

import prominence.msg.MessageInputStream;
import prominence.msg.MessageOutputStream;

public class TransactionClient extends Frame implements Runnable {
  protected MessageInputStream mI;
  protected MessageOutputStream mO;
  protected Button get, put;
  protected TextField attr, value;

  public TransactionClient (InputStream i, OutputStream o) {
    super ("Transaction Client");
    mI = new MessageInputStream (i);
    mO = new MessageOutputStream (o);
    attr = new TextField (24);
    value = new TextField (24);
    get = new Button ("get");
    put = new Button ("put");
    setLayout (new GridLayout (2, 2));
    add (attr);
    add (value);
    add (get);
    add (put);
    pack ();
    show ();
  }

  public void run () {
    while (true) {
      try {
        mI.receive ();
        System.out.print ("attr: " + mI.readUTF ());
        System.out.println (" value: " + mI.readUTF ());
      } catch (IOException e) {
            e.printStackTrace();
      }
    }
  }
```



```
  public boolean handleEvent (Event e) {
    if ((e.id == e.ACTION_EVENT) && (e.target instanceof Button)) {
      try {
        if (e.target == get) {
          mO.writeUTF ("get");
          mO.writeUTF (attr.getText ());
        } else if (e.target == put) {
          mO.writeUTF ("put");
          mO.writeUTF (attr.getText ());
          mO.writeUTF (value.getText ());
        }
        mO.send ();
      } catch (IOException ex) {
        ex.printStackTrace ();
      }
    }
    return super.handleEvent (e);
  }

  static public void main (String args[]) throws IOException {
    if (args.length != 2)
      throw new RuntimeException ("Syntax: TransactionClient <server> <port>");
    Socket s = new Socket (args[0], Integer.parseInt (args[1]));
    InputStream i = s.getInputStream ();
    OutputStream o = s.getOutputStream ();
    TransactionClient c = new TransactionClient (i, o);
    //c.listen ();
    new Thread(c).start();
  }
}
```

Note

- Il costruttore riceve i due stream per la comunicazione, vi attacca due message stream e costruisce l'interfaccia utente.
- Il `main()` crea il socket per la connessione al server, estrae i due stream, e crea il `TransactionClient`. Quindi avvia un thread separato per la ricezione dei dati.
- Il metodo `run()`, e dunque il thread separato, è un ciclo infinito che:
  - riceve un pacchetto spedito dal server (contenente una coppia di stringhe attributo-valore) con `receive()`;
  - stampa sullo standard output la coppia attributo e valore.
- Il metodo `handleEvent()` gestisce la interazione coll'utente, dando via via inizio alle transazioni:
  - bottone get: si invia un pacchetto il cui messaggio è costituito da due stringhe UTF:
    - la prima è "get"
    - la seconda è il nome dell'attributo di cui si vuole conoscere il valore;
  - bottone put: si invia un pacchetto il cui messaggio è costituito da tre stringhe UTF:
    - la prima è "put"
    - la seconda è il nome dell'attributo che si vuole modificare;
    - la terza è il valore che si vuole assegnare all'attributo.

4.2.2) Classe TransactionServer

Questo è il server che gestisce le transazioni. Anche lui usa gli stessi tipi di message stream del client, e grazie ad essi può attivare molteplici thread che lavorano in concorrenza per gestire molte transazioni contemporaneamente. Un ritardo simula artificialmente il tempo necessario a gestire una transazione.

Implementa l'interfaccia `Runnable` per poter lanciare molti thread sullo stesso oggetto.

Il costruttore riceve i due stream per la comunicazione e crea una `HashTable` (essenzialmente una classe che implementa una memoria associativa, contenente coppie chiave-valore e dotata di metodi per l'inserimento e la ricerca di elementi) per memorizzare e ricercare le coppie attributo-valore.

La definizione della classe è la seguente (dal libro "Java Network Programming" di Merlin Hughes et al., Mannig Publications Co, Greenwich CT, 1997).

```
/* Copyright (c) 1996, 1997 Prominence Dot Com, Inc.        *
 * See the file legal.txt in the txt directory for details. */

import java.io.*;
import java.util.*;
import java.net.*;

import prominence.msg.MessageOutput;
import prominence.msg.MessageInputStream;
import prominence.msg.MessageOutputStream;

public class TransactionServer implements Runnable {
  protected Hashtable h;
  protected InputStream i;
  protected OutputStream o;

  public TransactionServer (InputStream i, OutputStream o) {
    this.i = i;
    this.o = o;
    h = new Hashtable ();
  }
  public void run () {
    MessageInputStream mI = new MessageInputStream (i);
    MessageOutputStream mO = new MessageOutputStream (o);
    try {
      while (true) {
        mI.receive ();
        try {
          Thread.sleep (1000);
        } catch (InterruptedException ex) {
        }
        String cmd = mI.readUTF ();
      System.out.println (Thread.currentThread () + ": command " + cmd);
        if (cmd.equals ("get")) {
          get (mI, mO);
        } else if (cmd.equals ("put")) {
          put (mI);
        }
      }
    } catch (IOException ex) {
      ex.printStackTrace ();
    }
  }

  void get (DataInputStream dI, MessageOutput mO) throws IOException {
    String attr = dI.readUTF ();
    mO.writeUTF (attr);
    if (h.containsKey (attr))
      mO.writeUTF ((String) h.get (attr));
    else
      mO.writeUTF ("null");
    mO.send ();
  }

  void put (DataInputStream dI) throws IOException {
    String attr = dI.readUTF ();
    String value = dI.readUTF ();
    h.put (attr, value);
  }

  static public void main (String args[]) throws IOException {
    if (args.length != 2)
      throw new RuntimeException ("Syntax: TransactionServer <port> <threads>");
    ServerSocket server = new ServerSocket (Integer.parseInt (args[0]));
    Socket s = server.accept ();
    server.close ();
    InputStream i = s.getInputStream ();
    OutputStream o = s.getOutputStream ();
    TransactionServer t = new TransactionServer (i, o);
    int n = Integer.parseInt (args[1]);
    for (int j = 0; j < n; ++ j)
      new Thread (t).start ();
  }
}
```

Note

- Il `main()` opera come segue:
  - crea un `ServerSocket`;
  - accetta una connessione e chiude subito il `ServerSocket`;
  - deriva i due stream per la comunicazione;
  - crea il `TransactionServer`;
  - lancia su di esso un certo numero (passato come parametro) di thread.
- Il metodo `run()` di ogni thread è un ciclo infinito, in cui:
  - si creano i due message stream, locali al thread, per la comunicazione;
  - si aspetta un messaggio dal client, con `receive()`;
  - appena arriva il messaggio, che è relativo a una nuova transizione, la si gestisce:
    - si simula un ritardo di 1 secondo;
    - si legge il comando ("get" o "put");
    - si chiama il corrispondente metodo di gestione (`get()` o `put()`):
      - se il comando è "get", il thread spedisce un messaggio di risposta costituito da una coppia di stringhe (attributo e valore);
      - se il comando è "put", il thread cambia il valore dell'attributo (se c'è nella tavola) o inserisce una nuova coppia (se non c'è l'attributo). Questo è ottenuto automaticamente col metodo `put()` di `HashTable`.

Lo scopo fondamentale di questa applicazione client-server è mostrare come l'uso dei message stream ci consenta facilmente di condividere un unico canale di comunicazione fra molti thread concorrenti. L'incapsulamento ci protegge dal mescolamento dei dati.

4.3) Accodamento di messaggi

Avendo la possibilità di gestire stream di messaggi anziché di byte, è possibile crearsi ulteriori strumenti di utilità:

- una coda di messaggi;
- degli stream di messaggi (di input e output) che estraggono (e inseriscono) messaggi dalla / nella coda anziché da una connessione di rete.

Ciò significa avere la possibilità di disaccoppiare fra loro:

- il canale di comunicazione fisico sul quale viaggiano i dati (che partono e arrivano da / nella coda);
- la componente applicativa che utilizza o produce i dati (che legge / scrive da / nella coda).

Le principali conseguenze positive di questo disaccoppiamento sono:

- isolamento dell'applicazione dagli errori di comunicazione;
- isolamento dell'applicazione da eventuali ritardi di trasmissione: le scritture dell'applicazione avvengono nella coda e quindi sono rapide. Eventuali ritardi nell'invio sulla rete penalizzano solo il thread (di norma separato) che li gestisce.

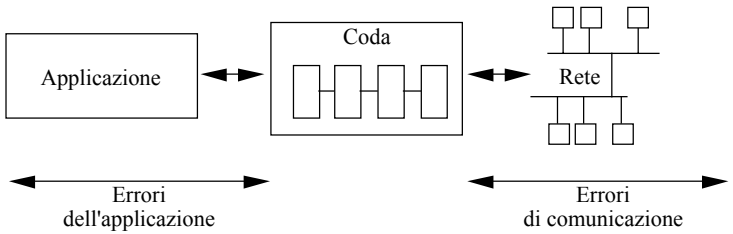


Figura 4-9: Coda di messaggi

4.3.1) Classe Queue

E' una implementazione della ***coda*** (intesa come struttura dati), i cui elementi sono messaggi.

Si noti che nella coda ci sono messaggi e non pacchetti, perché la coda provvede automaticamente a tenere separati tali elementi e quindi non c'è bisogno dell'informazione di controllo.

La definizione della classe è la seguente (dal libro "Java Network Programming" di Merlin Hughes et al., Mannig Publications Co, Greenwich CT, 1997).

```
/* Copyright (c) 1996, 1997 Prominence Dot Com, Inc.        *
 * See the file legal.txt in the txt directory for details. */

package prominence.util;

import java.util.Vector;

/**
 * A FIFO (first in, first out) data-structure; the opposite of a
<tt>Stack</tt>.
 * Objects are added to the front of the <tt>Queue</tt> and removed from the
back.
 * <p>This implementation blocks the caller who attempts to remove an object
from
 * an empty queue until the queue is non-empty again.
 *
 * @version 1.0 1 Nov 1996
 * @author Merlin Hughes
 */
public class Queue {
  /**
   * A <tt>Vector</tt> of the queue elements.
   */
  protected Vector queue;

  /**
   * Creates a new, empty <tt>Queue</tt>.
   */
  public Queue () {
    queue = new Vector ();
  }

  /**
   * Attempts to remove an object from the queue; blocks if there are no objects
   * in the queue. This call will therefore always return an object.
   * @returns The least-recently-added object from the queue
   */
  public Object remove () {
    synchronized (queue) {
      while (queue.isEmpty ()) {
        try {
          queue.wait ();
        } catch (InterruptedException ex) {}
      }
      Object item = queue.firstElement ();
      queue.removeElement (item);
      return item;
    }
  }

  /**
   * Adds an item to the front of the queue, wakes a caller who is waiting for
   * the queue to become non-empty.
   * @param item The object to be added
   */
  public void add (Object item) {
    synchronized (queue) {
```

```
      queue.addElement (item);
      queue.notify ();
    }
  }

  /**
   * Returns whether the queue is empty.
   * @returns Whether the queue is empty
   */
  public boolean isEmpty () {
    return queue.isEmpty ();
  }
}
```

Note

- La coda è implementata per mezzo di un `Vector`, classe che offre due metodi, `add()` e `remove()`, per inserire e recuperare un elemento. Va notato che un thread che cerca di estrarre un elemento da una coda vuota viene bloccato, e si risveglierà quando ci sarà qualcosa nella coda.
- Il costruttore crea un `Vector` vuoto, nel quale verrà mantenuta la coda.
- Il metodo `remove()` elimina un elemento dalla coda e lo restituisce al chiamante:
  - contiene un blocco di codice sincronizzato sul `Vector`, al fine di poter gestire la coda per mezzo di thread multipli;
  - si blocca con `wait()`, rilasciando quindi il lucchetto, se la coda è vuota.
- Il metodo `add()` aggiunge un elemento alla coda:
  - contiene un blocco di codice sincronizzato sul `Vector`;
  - provvede a risvegliare con `notify()` uno degli eventuali thread in attesa di un elemento.

4.3.2) Classe QueueOutputStream

Questa classe è un `MessageOutput` che, invece di scrivere messaggi su un `OutputStream`, li inserisce in una coda.

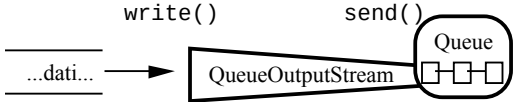


Figura 4-10: `QueueOutputStream`

`send()` aggiunge alla coda un messaggio, costituito dal corrente contenuto del buffer.

Come si noterà, in questo caso non si incapsula il messaggio in un pacchetto, in quanto la coda è una struttura costituita di elementi separati, ciascuno dei quali è un array che internamente ha l'informazione sulle proprie dimensioni. Il contenuto di ogni array coincide con quello del corrispondente messaggio.

Internamente, il buffer si implementa con un ByteArrayOutputStream.

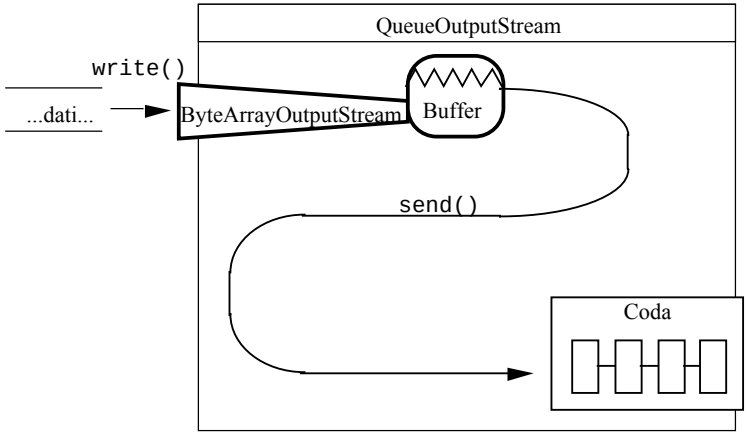


Figura 4-11: Implementazione di `QueueOutputStream`

La definizione della classe è la seguente (dal libro "Java Network Programming" di Merlin Hughes et al., Mannig Publications Co, Greenwich CT, 1997).

```
/* Copyright (c) 1996, 1997 Prominence Dot Com, Inc.        *
 * See the file legal.txt in the txt directory for details. */

package prominence.msg;

import java.io.*;
import prominence.util.Queue;

/**
 * A <tt>MessageOutput</tt> that inserts messages into a <tt>Queue</tt>
 * of byte arrays.
 *
 * @version 1.0 1 Nov 1996
 * @author Merlin Hughes
 * @see prominence.msg.QueueInputStream
 */
public class QueueOutputStream extends MessageOutput {
  /**
   * A <tt>ByteArrayOutputStream</tt> used to buffer the current message
   * contents.
   */
  protected ByteArrayOutputStream byteO;
  /**
   * The <tt>Queue</tt> of messages.
   */
  protected Queue q;

  /**
   * Creates a new <tt>QueueOutputStream</tt>.
   * @param q A <tt>Queue</tt> into which messages will be written
   */
  public QueueOutputStream (Queue q) {
    super (new ByteArrayOutputStream ());
    byteO = (ByteArrayOutputStream) out;
```

```
    this.q = q;
  }

  /**
   * Inserts the current message buffer into the <tt>Queue</tt>.
   */
  public void send () {
    byte[] buffer = byteO.toByteArray ();
    byteO.reset ();
    q.add (buffer);
  }
}
```

Note

- Il costruttore riceve come parametro la coda a cui si deve attaccare, e crea un buffer interno su cui avverranno le scritture (analogamente a quanto visto per `MessageOutputStream`). Inoltre, mantiene una reference alla coda cui è attaccato.
- Il metodo `send()` estrae il contenuto attuale del buffer (ossia il messaggio costruito con le scritture) e lo inserisce come nuovo elemento nella coda. Quindi resetta il buffer in modo che le prossime scritture possano costruire il prossimo messaggio.
- Si noti che molti `QueueOutputStream` possono essere attaccati alla stessa coda. Grazie al fatto che l'aggiunta di elementi avviene in un blocco sincronizzato, non si verificano interferenze fra eventuali thread concorrenti, che tipicamente gestiscono i vari `QueueOutputStream`.

4.3.3) Classe QueueInputStream

Questa classe è un `MessageInput` che, invece di leggere messaggi da un `InputStream`, li estrae da una coda.

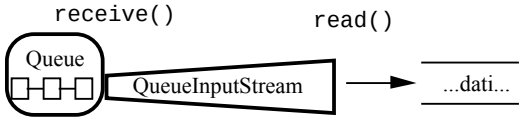


Figura 4-12: `QueueInputStream`

`Receive()` estrae un messaggio dalla coda e rende disponibile il suo contenuto per le letture. Se non ci sono messaggi, blocca il chiamante finché non ce n'è uno disponibile. Internamente il buffer si implementa con un `ByteArrayInputStream`.

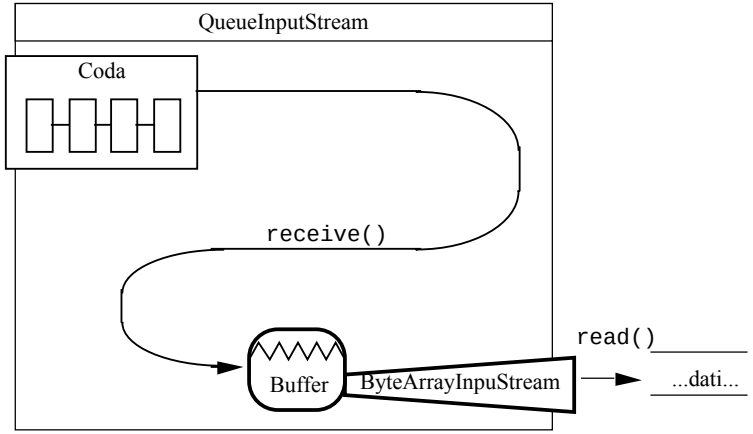


Figura 4-13: Implementazione di `QueueInputStream`

La definizione della classe è la seguente (dal libro "Java Network Programming" di Merlin Hughes et al., Mannig Publications Co, Greenwich CT, 1997).

```
/* Copyright (c) 1996, 1997 Prominence Dot Com, Inc.        *
 * See the file legal.txt in the txt directory for details. */

package prominence.msg;

import java.io.*;
import prominence.util.Queue;

/**
 * A <tt>MessageInput</tt> that reads messages from a <tt>Queue</tt> of
 * byte arrays.
 *
 * @version 1.0 1 Nov 1996
 * @author Merlin Hughes
 * @see prominence.msg.QueueOutputStream
 */
public class QueueInputStream extends MessageInput {
  /**
   * The <tt>Queue</tt> of messages.
   */
  protected Queue q;

  /**
   * Creates a new <tt>QueueInputStream</tt>.
   * @param q A <tt>Queue</tt> out of which messages will be read
   */
  public QueueInputStream (Queue q) {
    super (null);
    this.q = q;
  }

  /**
   * A buffer containing the most recently received message.
   * <p>This variable is exposed to permit potential optimizations.
   */
  byte[] buffer;

  /**
   * Extracts a message from the attached <tt>Queue</tt> and makes
   * it available to read through the usual superclass <tt>read()</tt>
   * methods.
   */
  public void receive () {
    buffer = (byte[]) q.remove ();
    in = new ByteArrayInputStream (buffer);
  }
}
```

Note

- Il costruttore riceve come parametro la coda a cui attaccarsi. Chiama il costruttore della superclasse passandogli `null` (non c'è alcun messaggio da leggere) e mantiene una reference alla coda.
- Il metodo `receive()` estrae un messaggio dalla coda (bloccandosi se non ce ne sono) e lo copia in un array di byte interno. Quindi, analogamente a `MessageInputStream`, crea un `ByteArrayInputStream` attaccato al buffer e si aggancia a tale `ByteArrayInputStream` per le successive letture.

- Anche in questo caso, molti `QueueInputStream` possono essere attaccati alla stessa coda senza problemi di interferenze.

4.3.4) Utilizzo tipico degli stream per l'accodamento di messaggi

Come abbiamo detto, l'accodamento dei messaggi permette di:

- isolare l'applicazione dagli errori di comunicazione;
- isolare l'applicazione dagli eventuali ritardi di trasmissione.

Entrambi questi obiettivi possono essere raggiunti con sistemi analoghi a quelli sotto esposti.

Uso di code in input
Si dedica un thread separato, che chiameremo ***thread copiatore***, alla lettura dei pacchetti dalla connessione di rete. Tale thread non fa altro che ricevere pacchetti dalla rete (attraverso un `MessageInput`) e ricopiarli in una coda (attraverso un `QueueOutputStream`).

Una applicazione che legga i dati da tale coda (attaccandovi un `QueueInputStream`) è quindi isolata dalla rete.

Il codice di tale thread copiatore sarà semplicemente un ciclo infinito di copiatura:

```
...
try{
    while (true) {
        mi.receive();
        byte[] buffer=new byte[mi.available()];
        mi.readFully(buffer);
        mo.write(buffer);
        mo.send();
    }
} catch (IOException e) {
    e.printStackTrace();
}
...
```

dove:

- `mi` è il `MessageInput`;
- `mo` è il `QueueOutputStream`.

Uso di code in output
Si dedica un thread copiatore alla scrittura dei pacchetti sulla connessione di rete. Tale thread preleva i pacchetti da una coda e li ricopia su una connessione di rete.

Una applicazione che scriva i dati nella coda (attaccandovi un `QueueOutputStream`) è quindi isolata dalla rete, sia per quanto riguarda errori che possibili ritardi.

Il codice del thread copiatore è uguale a prima, solo che ora si avrà che:

- `mi` è il `QueueInputStream`;
- `mo` è il `MessageOutput`.

Vedremo più avanti una classe (`Demultiplexer`) che fra le sue funzioni ha anche quella di thread copiatore per isolare una applicazione dalla rete.

4.4) Multiplexing di messaggi

Finora abbiamo sviluppato stream di messaggi piuttosto semplici, che non aggiungono particolari funzionalità ai normali stream, pur presentando già alcuni vantaggi quali:
- facilità di condivisione di un unico canale di comunicazione da parte di più stream di messaggi;
- disaccoppiamento fra rete e applicazione.

Comunque, gli stream visti finora formano la base necessaria sulla quale svilupparne di più sofisticati.

In particolare, ora vedremo come sviluppare degli stream di messaggi che siano in grado di effettuare il ***multiplexing***, su un unico canale di comunicazione, di vari flussi di dati eterogenei fra loro, tipicamente generati da componenti diverse di una applicazione in maniera del tutto trasparente a tali componenti, che in genere ignorano l'esistenza le une delle altre.

Lo scopo si raggiunge incapsulando i messaggi dentro pacchetti nei quali la Control Information è costituita da un' ***etichetta*** che caratterizza l'origine dei dati.

Quando tali pacchetti giungono a destinazione, i messaggi vengono instradati alla corretta componente grazie a un meccanismo corrispondente di ***demultiplexing***, che opera sulla base di tale Control Information.

In particolare, dopo aver introdotto le necessarie classi di utilità, svilupperemo un'applicazione client-server che implementa una chatline più raffinata di quella vista precedentemente.

Il client infatti è costituito da due parti indipendenti fra loro, che consentono l'una di inviare testo e l'altra disegni condividendo lo stesso canale fisico di comunicazione.

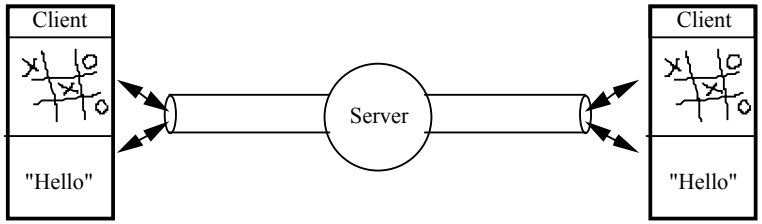


Figura 4-14: Chatline grafica e testuale

Il server ha il compito di effettuare il broadcast dei pacchetti che riceve da un client a tutti gli altri.

Come vedremo, esso non entra nel merito di ciò che riceve, e quindi rimarrà inalterato anche in caso si aggiungano ulteriori moduli funzionali ai client.

4.4.1) Classe MultiplexOutputStream

Questa classe estende `MessageOutput`, però si attacca a un altro `MessageOutput` e non a un semplice `OutputStream`.

Ciò consente di effettuare il multiplexing su un qualunque stream di messaggi (ad esempio su un `QueueOutputStream` oppure un `MessageOutputStream`).

Questo meccanismo ci consente di gestire più livelli di incapsulamento, uno dentro l'altro.

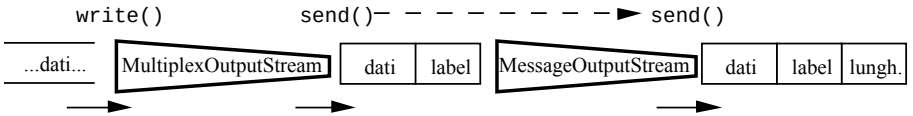


Figura 4-15: Incapsulamento multiplo

In sostanza, aggiungiamo un meccanismo di etichettatura all'`OutputStream` attaccato.

Dall'altra parte del canale, tale etichezza sarà usata per determinare l'origine dei dati.

Nel client che vedremo fra breve useremo questa classe per fare il multiplexing di due flussi di dati (che provengono dalle due componenti separate dell'applicazione) su un unico canale di comunicazione.

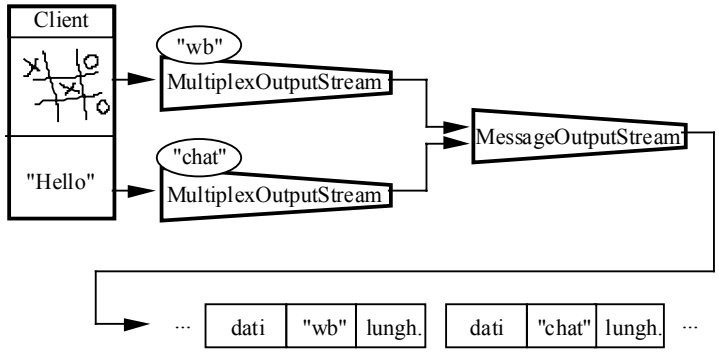


Figura 4-16: Multiplexing dei messaggi della chatline

La definizione della classe è la seguente (dal libro "Java Network Programming" di Merlin Hughes et al., Mannig Publications Co, Greenwich CT, 1997).

```
/* Copyright (c) 1996, 1997 Prominence Dot Com, Inc.        *
 * See the file legal.txt in the txt directory for details. */

package prominence.msg;

import java.io.*;

/**
 * A <tt>MessageOutput</tt> that attaches to an existing <tt>MessageOutput</tt>
 * and attaches a multiplexing label to the header of each message that
 * is transmitted.
 * <p>The label is specified in the constructor and so one stream always
 * attaches the same label.
 *
 * @version 1.0 1 Nov 1996
 * @author Merlin Hughes
 * @see prominence.msg.MultiplexInputStream
 */
public class MultiplexOutputStream extends MessageOutput {
  /**
   * The <tt>MessageOutput</tt> to which this is attached.
   */
  protected MessageOutput o;
  /**
   * A <tt>ByteArrayOutputStream</tt> used to buffer the current message
   * contents.
   */
  protected ByteArrayOutputStream byteO;
  /**
   * The multiplexing label.
   */
  protected String label;

  /**
   * Creates a new <tt>MultiplexOutputStream</tt>.
   * @param o The <tt>MessageOutput</tt> to which to send messages
   * @param label The multiplexing label to be used by this stream
   */
  public MultiplexOutputStream (MessageOutput o, String label) {
    super (new ByteArrayOutputStream ());
    byteO = (ByteArrayOutputStream) out;
    this.o = o;
    this.label = label;
  }

  /**
   * Sends the current message with a multiplexing label header to the
   * attached <tt>MessageOutput</tt>.
   * @exception IOException Occurs if there is a problem sending the
   * message.
   */
  public void send () throws IOException {
    synchronized (o) {
      o.writeUTF (label);
      byteO.writeTo (o);
      o.send ();
    }
    byteO.reset ();
  }

  /**
   * Sends the current message with a multiplexing label header to the
   * attached <tt>MessageOutput</tt>.
   * <p>If the attached <tt>MessageOutput</tt> supports targeted sending
   * then this method will succeed; otherwise an appropriate
   * <tt>IOException</tt> will be thrown.
   * @param dst The list of intended recipients
   * @exception IOException Occurs if there is a problem sending the
   * message or the targeted <tt>send()</tt> method is not supported by
   * the attached <tt>MessageOutput</tt>.
   */
  public void send (String[] dst) throws IOException {
    synchronized (o) {
      o.writeUTF (label);
      byteO.writeTo (o);
      o.send (dst);
    }
    byteO.reset ();
  }
}
```


Note

- Questo stream ci offre il vantaggio (rispetto a inserire "manualmente" un'etichetta) di offrire tale funzione in modo trasparente al chiamante, che usa i normali metodi di un `MessageOutput`. Solo al momento di chiamare il costruttore si è consapevoli di aver a che fare con un `MultiplexOutputStream`.
- Il metodo `send(String[] dst)` è implementato per predisporsi al caso in cui il `MessageOutput` a cui si attacca supporti tale metodo (spedizione ***multicast***).

Costruttore

- Riceve come parametri il `MessageOutput` (naturalmente si passerà una sua classe derivata) a cui attaccarsi e l'etichetta da usare come CI.
- Per il resto è analogo agli altri stream di messaggi di output: crea un `ByteArrayOutputStream` su cui convogliare le scritture in attesa della spedizione.

Metodi

- `send()` scrive sul `MessageOutput` l'etichetta (cioè la CI) e quindi il contenuto del buffer interno (cioè il messaggio). Quindi chiama il metodo `send()` del `MessageOutput`, il che fa sì che quest'ultimo invii sul canale di comunicazione:
  - la sua propria CI (se c'è, come nel caso di `MessageOutputStream`);
  - ciò che ha nel buffer, ossia nell'ordine:
    - l'etichetta;
    - il messaggio vero e proprio;
- E' necessario assicurarsi, dento la `send()`, che le scritture sul `MessageOutput` e la chiamata del metodo `send()` di quest'ultimo non possano essere interrotti, per evitare, se qualcun altro condivide il `MessageOutput`, che la costruzione e l'invio del pacchetto siano interrotti prima di essere completati. Ciò si ottiene sincronizzando tale codice sul `MessageOutput`.

4.4.2) Classe MultiplexInputStream

Questo è lo stream di input corrispondente a `MultiplexOutputStream`.

Questa classe estende `MessageInput`, si attacca a un altro `MessageInput` (non a un semplice `InputStream`) ed estrae l'etichetta da ogni messaggio che viene ricevuto.

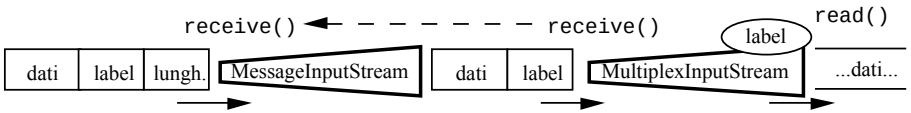


Figura 4-17: `MultiplexInputStream`

L'etichetta è resa accessibile all'esterno, rendendo così possibile determinare come deve essere elaborata l'informazione contenuta nel messaggio.

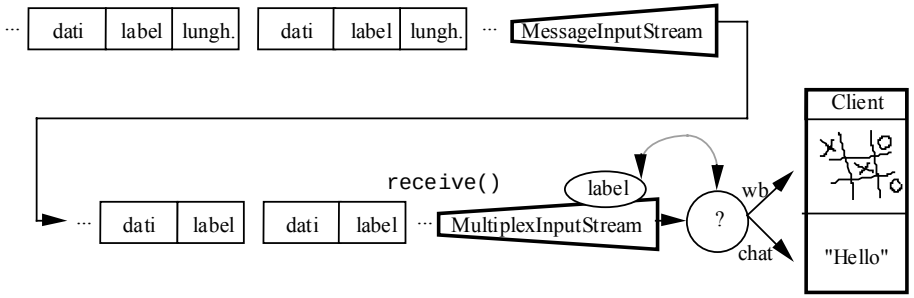


Figura 4-18: Demultiplexing dei messaggi della chatline

La definizione della classe è la seguente (dal libro "Java Network Programming" di Merlin Hughes et al., Mannig Publications Co, Greenwich CT, 1997).

```
/* Copyright (c) 1996, 1997 Prominence Dot Com, Inc.        *
 * See the file legal.txt in the txt directory for details. */

package prominence.msg;

import java.io.*;

/**
 * A <tt>MessageInput</tt> that attaches to an existing <tt>MessageInput</tt>
 * and strips the multiplexing label from each message that is received.
 * <p>The label is made publicly accessible in the <tt>label</tt> variable.
 *
 * @version 1.0 1 Nov 1996
 * @author Merlin Hughes
 * @see prominence.msg.MultiplexOutputStream
 */
public class MultiplexInputStream extends MessageInput {
  /**
   * The multiplexing label of the most recently received message.
   */
  public String label;
  /**
   * The <tt>MessageInput</tt> to which this is attached.
   */
  protected MessageInput i;

  /**
   * Creates a new <tt>MultiplexInputStream</tt>.
   * @param i The <tt>MessageInput</tt> from which messages should
   * be received
   */
  public MultiplexInputStream (MessageInput i) {
    super (i);
    this.i = i;
  }

  /**
   * Receives a new message from <tt>i</tt> and strips the multiplexing
   * label. The label is accessible in the <tt>label</tt> variable; the
   * contents of the message can be read through the usual superclass
   * <tt>read()</tt> methods.
   * @exception IOException Occurs if there is a problem receiving a
   * message or extracting the multiplexing label.
   */
  public void receive () throws IOException {
    i.receive ();
    label = i.readUTF ();
  }
}
```

Note

- Questa classe non effettua il demultiplexing. Si limita ad estrarre l'etichetta e a renderla accessibile dall'esterno. Il demultiplexing vero e proprio sarà effettuato da qualcun altro sulla base di tale informazione.

Costruttore

- Riceve come parametro il `MessageInput` (si passerà di fatto una sua sottoclasse) a cui attaccarsi.
- Chiama il costruttore della superclasse (cioè `MessageInput` e quindi `DataInputStream`) passandogli il `MessageInput` a cui si è attaccato, per cui i normali metodi di lettura della superclasse saranno direttamente passati a `i`.

Metodi

- `receive()` chiama innanzitutto il metodo `receive()` dell'`InputStream` a cui è attaccato (che provvederà a rimuovere la propria CI). Quindi provvede a leggere l'etichetta (ossia la CI di multiplexing) che viene memorizzata nella variabile pubblica `label`.
- A questo punto, i normali metodi di lettura (`read()`, ecc.) leggeranno dal corpo del messaggio.

4.4.3) Classe Demultiplexer

Questa è la classe che si incarica di effettuare il demultiplexing vero e proprio.

Riceve i messaggi da un `MultiplexInputStream` e, sulla base della loro etichetta, li consegna ad uno dei `MessageOutput` a cui è connesso.

In sostanza, è un thread copiatore che ha in più la capacità di effettuare il demultiplexing dei messaggi in arrivo. Include dei metodi per registrare i (e cancellare la registrazione dei) `MessageOutput` che devono essere associati, in qualità di destinatari, alle etichette.

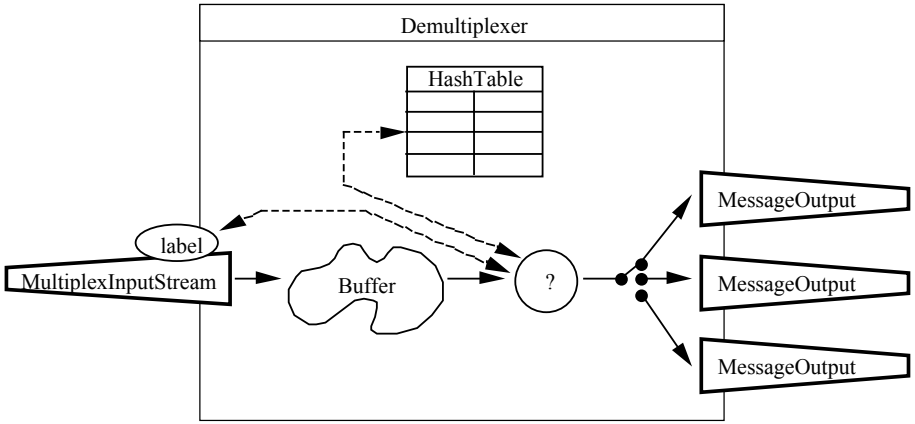


Figura 4-19: `Demultiplexer`

Tipicamente i `MessageOutput` sono dei `QueueOutputStream`, per le ragioni che abbiamo esposto in precedenza, oppure dei `DeliveryOutputStream` (che vedremo fra poco).

La definizione della classe è la seguente (dal libro "Java Network Programming" di Merlin Hughes et al., Mannig Publications Co, Greenwich CT, 1997).

```
/* Copyright (c) 1996, 1997 Prominence Dot Com, Inc.        *
 * See the file legal.txt in the txt directory for details. */

package prominence.msg;

import java.io.*;
import java.util.*;
/**
 * A class that reads messages from a <tt>MultiplexInputStream</tt>
 * and forwards them on to the <tt>MessageOutput</tt> identified by the
 * message label.
 *
 * @version 1.0 1 Nov 1996
 * @author Merlin Hughes
 * @see prominence.msg.MessageCopier
 */
public class Demultiplexer extends Thread {
  /**
   * The <tt>MultiplexInputStream</tt> from which messages are read.
   */
  protected MultiplexInputStream i;
  /**
   * The message routing table. Maps from message labels to
   * <tt>MessageOutput</tt>s.
   */
  protected Hashtable routes;

  /**
   * Current <tt>Demultiplexer</tt> ID.
   */
  static private int plexerNumber;
  /**
   * Assigns unique <tt>Demultiplexer</tt> ID's.
   * @return An unique <tt>Demultiplexer</tt> ID.
   */
  static private synchronized int nextPlexerNum () { return plexerNumber ++; }

  /**
   * Creates a new <tt>Demultiplexer</tt> reading from a specified
   * stream.
   * @param i The <tt>MultiplexInputStream</tt> from which mess. should be read
   */
  public Demultiplexer (MultiplexInputStream i) {
    super ("Demultiplexer-" + nextPlexerNum ());
    this.i = i;
    routes = new Hashtable ();
  }

  /**
   * Registers a <tt>MessageOutput</tt> as the destination for messages
   * with a particular label.
   * @param label The message label that is to be routed
   * @param o The destination for such messages
   */
  public void register (String label, MessageOutput o) {
    routes.put (label, o);
  }

  /**
   * Deregisters a particular message label.
   * @param label The label that is to be deregistered
   */
  public void deregister (String label) {
```

```
    routes.remove (label);
  }

  /**
   * Routes messages from the <tt>MultiplexInputStream</tt> to the
   * <tt>MessageOutput</tt> identified by their labels. <pre>This method
   * is called by a
   * new thread when the superclass <tt>start()</tt> method is called.
   * @see java.lang.Thread#start
   */
  public void run () {
    try {
      while (true) {
        i.receive ();
        MessageOutput o = (MessageOutput) routes.get (i.label);
        if (o != null) {
          byte[] message = new byte[i.available ()];
          i.readFully (message);
          synchronized (o) {
            o.write (message);
            o.send ();
          }
        }
      }
    } catch (IOException ex) {
      ex.printStackTrace ();
    }
  }
}
```

Costruttore

- Chiama il costruttore della superclasse, passandogli come nome una stringa concatenata con un numero progressivo (ottenuto con il metodo statico `nextPlexerNum()`).
- Crea una `HashTable` che servirà per correlare le etichette ai `MessageOutput`.
- Mantiene una reference al `MultiplexInputStream` a cui è attaccato.

Metodi

- Il metodo statico `nextPlexerNum()` restituisce il valore della variabile statica `plexerNumber` e successivamente lo incrementa di 1, fornendo così il numero progressivo per i nomi dei vari `Demultiplexer` che si volessero creare.
- Il metodo `register(...)` riceve come parametri un'etichetta e un `MessageOutput`, e inserisce un corrispondente nuovo elemento nella `HashTable`.
- Il metodo `deregister(...)` riceve come parametro un'etichetta e rimuove dalla `HashTable` il corrispondente elemento.
- Il metodo `run()` è un ciclo infinito che realizza le funzioni di:
  - thread copiatore;
  - demultiplexing dei messaggi.
- Il thread effettua nel ciclo i seguenti passi:
  - riceve un messaggio da `MultiplexInputStream`;
  - cerca nella `HashTable` l'`OutputStream` corrispondente all'etichetta del messaggio;
    - se esso esiste:
      - crea un buffer (array di byte) e lo riempie col messaggio;

      - scrive il buffer sul `MessageOutput` trovato prima e chiama `send()` di tale `MessageOutput`;
      - queste ultime due operazioni sono sincronizzate sul `MessageOutput` per evitare che il messaggio si possa mescolare con quelli di altri thread che condividono il `MessageOutput` prima di essere spedito.
    - se invece l'etichetta non è registrata nella `HashTable`, il messaggio viene scartato.

4.4.4) Classe DeliveryOutputStream e Interfaccia Recipient

Consegnare i messaggi in una coda e aspettarsi che l'applicazione cerchi attivamente di prelevarli da lì, il che richiede in genere un thread separato, non è talvolta necessario, sopratutto per una piccola applicazione.

Una alternativa è definire un diverso tipo di `MessageOutput`, che consegna i messaggi attivamente al destinatario invece di scriverli su uno stream (che nel caso sopracitato è la coda).

Ciò si può implementare facendo sì che questo nuovo tipo di `MessageOutput`, quando deve spedire un messaggio con `send()`, chiami direttamente un apposito metodo `receive()` (che deve quindi essere presente) del destinatario.

In tal modo, quando un messaggio è spedito viene immediatamente consegnato al destinatario per l'elaborazione.

La contropartita è che il thread che invia il messaggio con `send()` (tipicamente, un thread copiatore quale il `Demultiplexer`) deve aspettare il ritorno della `receive()` del destinatario per poter proseguire la propria elaborazione e ricevere quindi il prossimo messaggio.

La classe `DeliveryOutputStream` implementa un `MessageOutput` del tipo descritto.

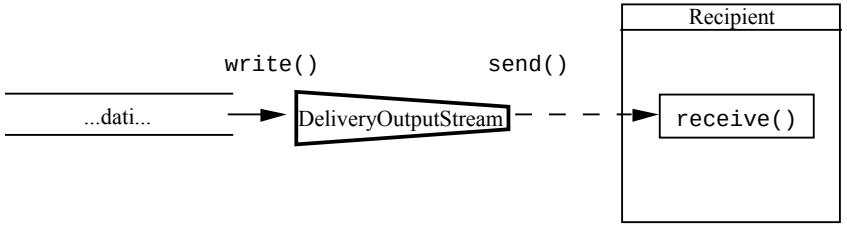


Figura 4-20: `DeliveryOutputStream`

La interfaccia `Recipient` dichiara il metodo `receive()`, e deve essere implementata da ogni destinatario che si voglia attaccare ad un `DeliveryOutputStream`.

4.4.4.1) Classe DeliveryOutputStream

E' un `MessageOutput` che si attacca a un oggetto che implementa l'interfaccia `Recipient`.

I messaggi sono costruiti dentro un `ByteArrayOutputStream` e sono consegnati al destinatario immediatamente, dopo averli spostati in un `ByteArrayInputStream` dal quale il destinatario provvederà a leggerli.

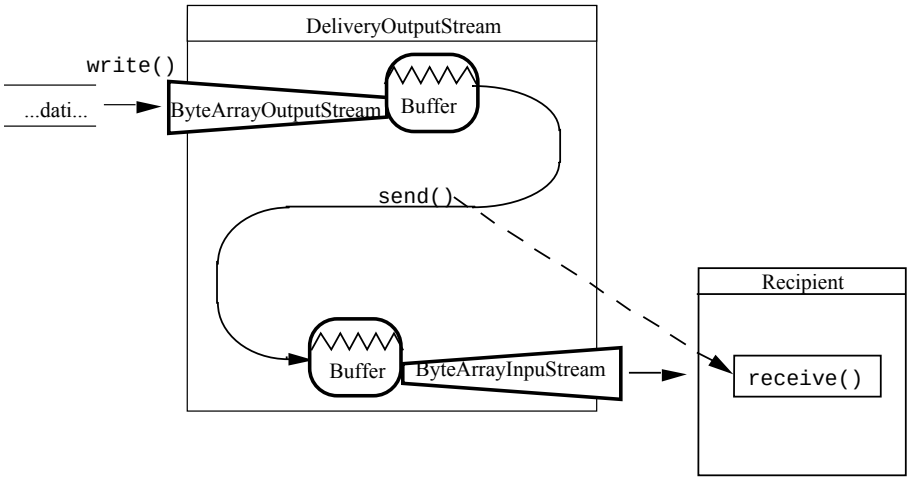


Figura 4-21: Implementazione di `DeliveryOutputStream`

La definizione della classe è la seguente (dal libro "Java Network Programming" di Merlin Hughes et al., Mannig Publications Co, Greenwich CT, 1997).

```
/* Copyright (c) 1996, 1997 Prominence Dot Com, Inc.        *
 * See the file legal.txt in the txt directory for details. */

package prominence.msg;

import java.io.*;

/**
 * A <tt>MessageOutput</tt> that immediately delivers its
 * contents to a recipient specified in the constructor when
 * send() is called.
 * <p>The message is delivered through the <tt>Recipient</tt> interface.
 *
 * @version 1.0 1 Nov 1996
 * @author Merlin Hughes
 * @see prominence.msg.Recipient
```



```
 */
public class DeliveryOutputStream extends MessageOutput {
  /**
   * A <tt>ByteArrayOutputStream</tt> used to buffer the message contents.
   */
  protected ByteArrayOutputStream byteO;
  /**
   * The recipient of messages sent to this stream.
   */
  protected Recipient r;

  /**
   * Creates a new <tt>DeliveryOutputStream</tt> with a specified recipient.
   * @param r The recipient for messages sent to this stream.
   */
  public DeliveryOutputStream (Recipient r) {
    super (new ByteArrayOutputStream ());
    byteO = (ByteArrayOutputStream) out;
    this.r = r;
  }

  /**
   * Delivers the current message contents to the designated recipient.
   */
  public void send () {
    byte buffer[] = byteO.toByteArray ();
    ByteArrayInputStream bI = new ByteArrayInputStream (buffer);
    r.receive (new DataInputStream (bI));
    byteO.reset ();
  }
}
```

Costruttore

- Chiama il costruttore della superclasse, attaccandolo a un `ByteArrayOutputStream` interno.
- Mantiene una reference in `byteO` a tale `ByteArrayOutputStream` ed un'altra, in `r`, all'oggetto che implementa l'interfaccia `Recipient`, che viene passato come parametro.

Metodi

- Il metodo `send()` effettua le seguenti operazioni:
  - crea, in un array di byte, una copia del contenuto attuale del `ByteArrayOutputStream` (cioè una copia del messaggio);
  - costruisce un `ByteArrayInputStream` attaccato alla copia del messaggio;
  - chiama il metodo `receive()` del destinatario, passandogli per convenienza un `DataInputStream` attaccato al `ByteArrayInputStream` e quindi al messaggio;
  - resetta il `ByteArrayOutputStream` per consentire la costruzione del prossimo messaggio.

4.4.4.2) Interfaccia Recipient

E' una semplice interfaccia che dichiara un solo metodo, `receive()`.

Esso ha come parametro un `DataInputStream` (che di fatto costituisce il corpo del messaggio) dal quale, con opportune letture, potrà essere acquisito il messaggio.

La definizione dell'interfaccia è la seguente (dal libro "Java Network Programming" di Merlin Hughes et al., Mannig Publications Co, Greenwich CT, 1997).

```
/* Copyright (c) 1996, 1997 Prominence Dot Com, Inc.        *
 * See the file legal.txt in the txt directory for details. */

package prominence.msg;

import java.io.*;

/**
 * The interface through which the <tt>DeliveryOutputStream</tt> delivers
 * messages.
 *
 * @version 1.0 1 Nov 1996
 * @author Merlin Hughes
 * @see prominence.msg.DeliveryOutputStream
 */
public interface Recipient {
  /**
   * Delivers a new message to the recipient.
   * @param in A <tt>DataInputStream</tt> from which the message contents can
   * be read
   */
  public void receive (DataInputStream in);
}
```

4.4.5) Client per la chatline grafica e testuale

Vediamo ora come è fatto il client per la chatline grafica e testuale; in seguito vedremo il server, che peraltro è molto semplice.

Come abbiamo già detto, il client è composto di due componenti distinte ed indipendenti, che comunicano con le loro controparti (grazie al server) per mezzo di multiplexing di messaggi.

Il client consiste di tre classi:

- un frame che contiene i componenti (`CollabTool`);
- la componente grafica (`WhiteBoard`);
- la componente testuale (`ChatBoard`).

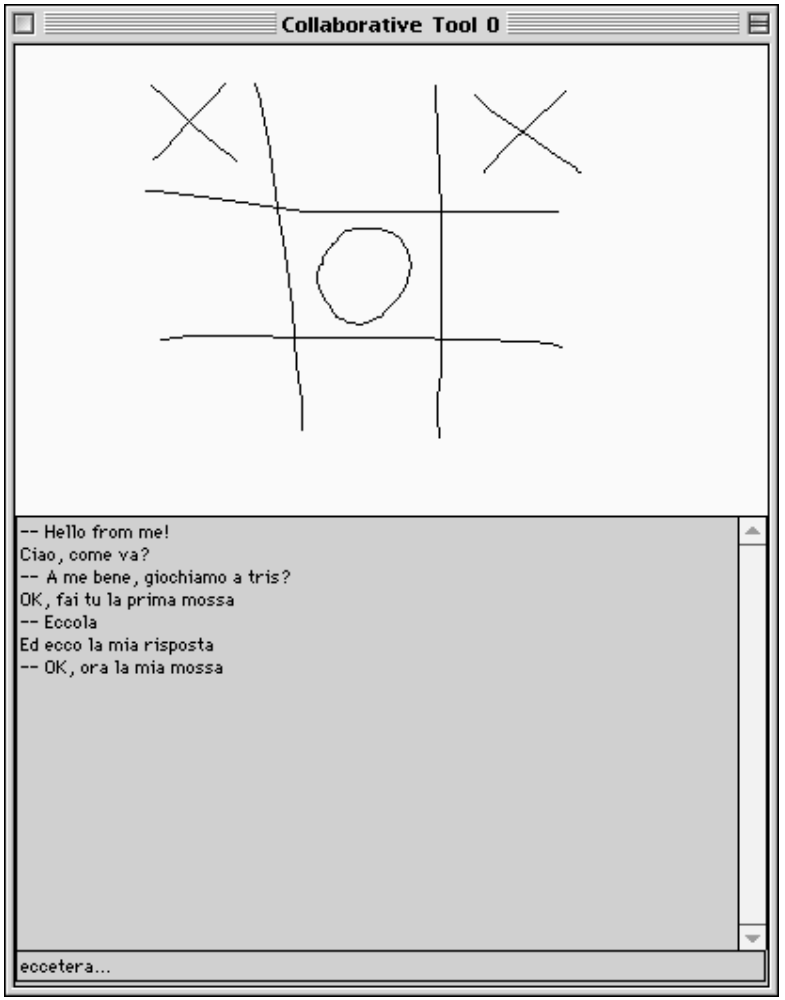


Figura 4-22: Interfaccia utente del client

4.4.5.1) Classe CollabTool

E' la classe principale (contiene il `main()`) ed ha diverse incombenze:

- aprire la connessione con il server;
- creare le due componenti (`ChatBoard` e `WhiteBoard`), inserirle in un frame e mostrare il tutto;
- predisporre tutti i necessari stream di messaggi (di input e output) per la comunicazione.

La definizione della classe è la seguente.

```
import java.io.*;
import java.awt.*;
import java.net.*;
import prominence.msg.*;

public class CollabTool extends Frame {
  protected static int id = 0;

  public CollabTool (InputStream i, OutputStream o) {
    super ("Collaborative Tool " + (id ++));
    setResizable(false);
    Whiteboard wb = new Whiteboard ();
    Chatboard cb = new Chatboard ();
    setLayout (new GridLayout (2, 1));
    add (wb);
    add (cb);
    resize (400, 500);

    MessageOutputStream mO = new MessageOutputStream (o);
    MessageInputStream mI = new MessageInputStream (i);

    cb.setMessageOutput (new MultiplexOutputStream (mO, "chat"));
    wb.setMessageOutput (new MultiplexOutputStream (mO, "wb"));

    Demultiplexer d = new Demultiplexer (new MultiplexInputStream (mI));
    d.register ("chat", cb.getMessageOutput ());
    d.register ("wb", wb.getMessageOutput ());
    d.start ();
  }

  static public void main (String args[]) throws IOException {

  Socket socket = new Socket(args[0], 5000);

    new CollabTool (socket.getInputStream(),socket.getOutputStream()).show ();
  }
}
```

Note

- Il metodo `main()` apre la connessione di rete con il server (il cui indirizzo IP o nome DNS deve essere passato come parametro) sul port 5000.
- Quando la connessione di rete è stabilita, da essa si derivano i due stream per la comunicazione che vengono passati al costruttore di `CollabTool`.

Costruttore

- Chiama il costruttore di `Frame` passandogli un titolo per la finestra.
- Crea `cb` (un `ChatBoard`) e `wb` (un `WhiteBoard`) e li aggiunge al `Frame`;
- Crea due message stream attaccati ai due stream esistenti da/verso il server:
  - `mO` è un `MessageOutputStream`;
  - `mI` è un `MessageInptStream`.
- Chiamando il metodo `setMessageOutput (...)` di `wb` e `cb`, aggancia il loro output ad un corrispondente `MultiplexOutputStream`.
- A questo punto l'output delle due componenti è sistemato, ora si deve occupare dell'input. Innanzitutto fa queste operazioni:
  - crea un `MultiplexInputStream` agganciato a `mI`;
  - crea un `Demultiplexer` agganciato a tale `MultiplexInputStream`.
  - Registra nel `Demultiplexer` gli stream di output ai quali dovranno essere inviati i messaggi di competenza di ognuna delle due componenti. Questo si fa chiedendo a loro quale dev'essere lo stream di output, mediante il loro metodo `getMessageOutput()` che, appunto, restituisce il `MessageOutput` che ognuna delle due componenti avrà provveduto a creare secondo le proprie inclinazioni.
- Infine, avvia il thread "copiatore" del `Demultiplexer`.

4.4.5.2) Classe ChatBoard

Questa classe implementa la componente testuale della chatline.

E' costituita da una `TextArea` in cui appaiono i messaggi precedenti e da un `TextField` in cui si immette di volta in volta un nuovo messaggio. Esso viene spedito non appena si preme il tasto Return dentro il `TextField`.

Nel contesto del client, questa componente dialoga direttamente con le corrispondenti componenti degli altri client collegati, ed ignora completamente l'esistenza della componente grafica.

Implementa l'interfaccia `Runnable` perché un thread gestisce l'interazione con l'utente (quello principale) mentre un thread separato (avviato con `run()`) si occupa di ricevere i messaggi che arrivano dalla connessione di rete.

La definizione della classe è la seguente (dal libro "Java Network Programming" di Merlin Hughes et al., Mannig Publications Co, Greenwich CT, 1997).

```
/* Copyright (c) 1996, 1997 Prominence Dot Com, Inc.        *
 * See the file legal.txt in the txt directory for details. */

import java.io.*;
import java.awt.*;
import prominence.msg.*;
import prominence.util.Queue;

public class Chatboard extends Panel implements Runnable {
  protected TextArea output;
  protected TextField input;
  protected Queue q;
  protected Thread exec;

  public Chatboard () {
    setLayout (new BorderLayout ());
    add ("Center", output = new TextArea ());
    output.setEditable (false);
    add ("South", input = new TextField ());
    q = new Queue ();
    exec = new Thread (this);
    exec.start ();
  }

  protected MessageOutput o;

  public void setMessageOutput (MessageOutput o) {
    this.o = o;
  }

  public MessageOutput getMessageOutput () {
    return new QueueOutputStream (q);
  }

  public boolean action (Event e, Object arg) {
    if (e.target == input) {
      try {
        o.writeUTF (input.getText ());
        o.send ();
      } catch (IOException ex) {
        ex.printStackTrace ();
      }
      output.appendText (input.getText () + "\n");
      input.setText ("");
      return true;
    }
    return super.action (e, arg);
  }

  public void sendMessage (String s) {
      try {
        o.writeUTF (s);
        o.send ();
      } catch (IOException ex) {
        ex.printStackTrace ();
      }
  }

  public void run () {
    QueueInputStream qI = new QueueInputStream (q);
    while (true) {
      try {
        qI.receive ();
```



```
        String msg = qI.readUTF ();
        output.appendText ("-- " + msg + "\n");
      } catch (IOException ex) {
        ex.printStackTrace ();
      }
    }
  }
}
```

Costruttore

- Crea i componenti dell'interfaccia utente e li dispone opportunamente.
- Crea una coda `q` a cui verranno attaccati i 2 stream per la comunicazione col `Demultiplexer`:
  - lo stream di output attaccato alla coda serve al `Demultiplexer` per depositarvi i messaggi arrivati;
  - lo stream di intput attaccato alla coda serve al `Chatboard` per prelevare (e quindi leggere) i messaggi arrivati.
- Avvia il thread separato per lettura dei messaggi in arrivo.

Metodi

- I metodi `setMessageOutput(MessageOutput o)` e `getMessageOutput()`, che vengono chiamati da `ColabTool` in fase di creazione della `ChatBoard`, predispongono gli stream di input e output. In particolare:
  - `setMessageOutput(MessageOutput o)` si aggancia al `MultiplexOutputStream`, creato da `CollabTool`, che serve in output;
  - `getMessageOutput()` invece crea un `QueueOutputStream` attaccato a `q` e lo passa a `CollabTool`, il quale così sa dove inviare i dati prodotti dal `Demultiplexer` per il `ChatBoard`.
- Il metodo `run()`, che fa partire il thread separato di lettura dei dati in arrivo, prima di tutto crea un `QueueInputStream` attaccato a `q`, e poi entra in un ciclo infinito di estrazione dei messaggi da tale stream.
- Il metodo `action(...)` gestisce l'evento di tipo `action` nel solo `TextField`, inviando un nuovo messaggio; rimanda alla superclasse la gestione di altri tipi di eventi.

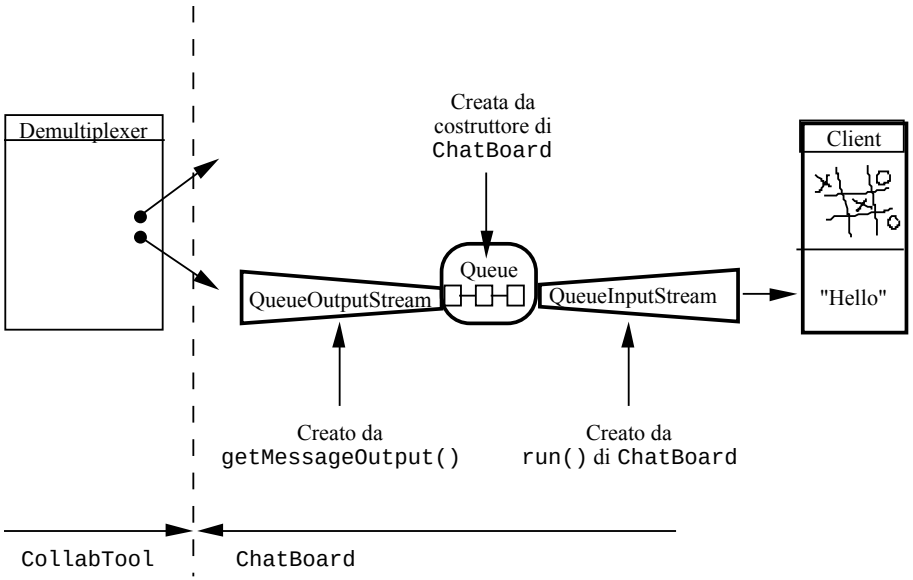


Figura 4-23: Collegamento degli stream per l'input di `ChatBoard`

4.4.5.3) Classe WhiteBoard

Questa classe implementa la componente grafica della chatline.

Offre una superficie sulla quale l'utente può disegnare a mano libera. Quando l'utente:

- preme il pulsante del mouse: inizia una nuova porzione del disegno;
- sposta il mouse (a bottone premuto, ***drag***): la porzione del disegno viene man mano costruita;
- rilascia il mouse: la porzione di disegno appena completata viene spedita sotto forma di messaggio.

Nel contesto del client, anche questa componente dialoga con le corrispondenti componenti degli altri client ed ignora l'esistenza della componente `ChatBoard`.

Implementa l'interfaccia `Recipient`, e quindi implementa un metodo `receive()` per la ricezione immediata dei messaggi.

La definizione della classe è la seguente (dal libro "Java Network Programming" di Merlin Hughes et al., Mannig Publications Co, Greenwich CT, 1997).

```
/* Copyright (c) 1996, 1997 Prominence Dot Com, Inc.        *
 * See the file legal.txt in the txt directory for details. */

import java.io.*;
import java.awt.*;
import prominence.msg.*;

public class Whiteboard extends Canvas implements Recipient {
  public Whiteboard () {
    setBackground (new Color (255, 255, 204));
  }

  protected MessageOutput o;

  public void setMessageOutput (MessageOutput o) {
    this.o = o;
  }

  public boolean mouseDown (Event e, int x, int y) {
    transmit (x, y);
    return super.mouseDown (e, x, y);
  }

  public boolean mouseDrag (Event e, int x, int y) {
    scribble (x, y);
    transmit (x, y);
    return super.mouseDrag (e, x, y);
  }

  public boolean mouseUp (Event e, int x, int y) {
    scribble (x, y);
    transmit (x, y);
    try {
      o.send ();
    } catch (IOException ex) {
      ex.printStackTrace ();
    }
    return super.mouseUp (e, x, y);
  }

  protected int oX, oY;

  protected void transmit (int x, int y) {
    try {
      o.writeInt (x);
      o.writeInt (y);
    } catch (IOException ex) {
      ex.printStackTrace ();
    }
    oX = x;
    oY = y;
  }

  protected void scribble (int x, int y) {
    Graphics g = getGraphics ();
    g.drawLine (oX, oY, x, y);
    g.dispose ();
  }

  public MessageOutput getMessageOutput () {
    return new DeliveryOutputStream (this);
  }
```

```
  public void receive (DataInputStream dI) {
    Graphics g = getGraphics ();
    try {
      int x0 = dI.readInt (), y0 = dI.readInt ();
      while (dI.available () > 0) {
        int x1 = dI.readInt (), y1 = dI.readInt ();
        g.drawLine (x0, y0, x1, y1);
        x0 = x1;
        y0 = y1;
      }
    } catch (IOException ex) {
      ex.printStackTrace ();
    }
    g.dispose ();
  }
}
```

Costruttore

- `WhiteBoard` è costituito da un unico `Canvas` (area di disegno), istanziato automaticamente dato che `WhiteBoard` estende proprio `Canvas`. Il costruttore si limita a cambiare il colore di fondo.

Metodi

- Come in `ChatBoard`, i metodi `setMessageOutput(MessageOutput o)` e `getMessageOutput()`, che vengono chiamati da `CollabTool` in fase di creazione della `WhiteBoard`, predispongono gli stream di I/O. In particolare:
  - `setMessageOutput(MessageOutput o)` si aggancia al `MultiplexOutputStream`, creato da `CollabTool`, che serve in output;
  - `getMessageOutput()` invece crea un `DeliveryOutputStream`, avente come target il `WhiteBoard` stesso, e lo passa a `CollabTool`, il quale così sa dove inviare i dati prodotti dal `Demultiplexer` e destinati al `WhiteBoard`.
- Il metodo `scribble()` disegna un trattino dall'ultima posizione del mouse (`ox, oy`) a quella corrente (`x, y`), e per far questo crea un contesto grafico che poi elimina.
- Il metodo `transmit()` scrive nel corpo del messaggio corrente le coordinate (`x, y`) attuali, che così si aggiungono a quelle precedenti; aggiorna quindi le coordinate (`ox, oy`) con i valori correnti.
- La gestione degli eventi si articola su tre metodi:
  - `mouseDown(...)` inizia a costruire il prossimo messaggio, chiamando `transmit(...)`;
  - `mouseDrag(...)` mostra l'ultimo tratto disegnato (con `scribble(...)`) e poi inserisce sul messaggio le nuove coordinate finali, chiamando `transmit(...)`;
  - `mouseUp(...)` disegna l'ultimo tratto (con `scribble(...)`), completa il messaggio (con `transmit(...)`) e poi invia effettivamente il messaggio con il metodo `send()` del `MultiplexOutputStream` a cui è attaccato.
- Il metodo `receive()` legge il contenuto di un messaggio (cioè una successione di coppie (`x, y`)) e provvede ad effettuare il corrispondente disegno.

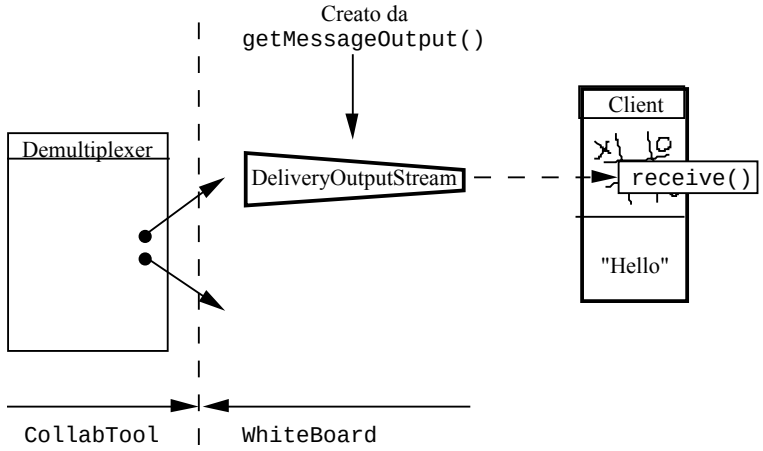


Figura 4-24: Collegamento degli stream per l'input di `WhiteBoard`

4.4.6) Server per la chatline grafica e testuale

E' il server a cui tutti i client si connettono. Invia in broadcast a tutti (tranne che al mittente) tutto ciò che riceve da uno qualunque di essi.

Concettualmente (e anche strutturalmente) è identico al server dell'esempio 9, solo che comunica attraverso l'uso di messaggi.

Si noti che i messaggi sono a loro volta pacchetti composti da:

- etichetta;
- dati.

Questo però il server lo ignora, e non ne preoccupa. Quindi, tale server continuerà a funzionare correttamente anche anche nel caso in cui ai client si dovessero aggiungere ultreriori componenti funzionali, purché anch'esse comunichino fra loro facendo uso di stream per il multiplexing dei messaggi.

Il codice è costituito da due classi. La prima, `CollabServer`, accetta richieste di connessione sul port 5000 e, ogni volta che ne arriva una, istanzia un oggetto della classe `CollabHandler` che si occupa di gestirla.

```
import java.net.*;
import java.io.*;
import java.util.*;

public class CollabServer {

//-------------------------------------------------

  public CollabServer() throws IOException {

    ServerSocket server = new ServerSocket(5000);
    System.out.println ("Accepting connections...");
    while(true) {
      Socket client = server.accept();
      System.out.println ("Accepted from " + client.getInetAddress());
      new CollabHandler(client).start();
    }
  }

//-------------------------------------------------

  public static void main(String args[]) throws IOException {

    new CollabServer();
  }

}
```

La seconda si occupa della gestione di una singola connessione e dell'invio a tutte le altre, in broadcast, dei dati provenienti da tale connessione.

```
import java.net.*;
import java.io.*;
import java.util.*;
import prominence.msg.*;

public class CollabHandler extends Thread {

  protected static Vector handlers = new Vector();
  protected Socket socket;
  protected MessageInputStream is;
  protected MessageOutputStream os;

//-------------------------------------------------

  public CollabHandler(Socket socket) throws IOException {

    this.socket = socket;
    is = new MessageInputStream(socket.getInputStream());
    os = new MessageOutputStream(socket.getOutputStream());
  }

//-------------------------------------------------

  public void run() {
```



```
    try {
      handlers.addElement(this); //e' un metodo sincronizzato
      while (true) {
        is.receive();
        byte[] buffer = new byte[is.available()];
        is.readFully(buffer);
        broadcast(this, buffer);
      }
    } catch(IOException ex) {
      ex.printStackTrace();
    } finally {
      handlers.removeElement(this); //e' un metodo sincronizzato
      try {
        socket.close();
      } catch(Exception ex) {
        ex.printStackTrace();
      }
    }
  }

//-------------------------------------------------

  protected static void broadcast(CollabHandler sender, byte[] buffer) {

    synchronized (handlers) { //ora nessuno puo' aggiungersi o abbandonare
      Enumeration e = handlers.elements();
      while (e.hasMoreElements()) {
        CollabHandler c = (CollabHandler) e.nextElement();
        if (c != sender) {
          try {
            c.os.write(buffer);
            c.os.send();
          } catch(Exception ex) {
            c.stop();
          }
        }
      }
    }
  }
}
```

Note

- `CollabTool` avvia mediante il `main()` il thread principale che:
  - rimane in ascolto sul port 5000;
  - attiva un thread separato (`CollabHandler`) per la gestione di ogni nuova connessione.
- Ogni nuovo `CollabHandler` attacca due message stream ai due canali di comunicazione, e quando si avvia:
  - si registra nel `Vector` statico delle connessioni attive;
  - riceve messaggi in input e li ritrasmette col metodo `broadcast()`;
  - quando termina, si rimuove dal `Vector` delle connessioni.
- Il metodo `broadcast()` di `CollabHandler` si sincronizza sul `Vector` delle connessioni e spedisce a tutti, tranne che a se stesso, il messaggio passato come parametro.

4.5) Ulteriori estensioni di funzionalità tramite messaggi

Abbiamo visto come l'uso e la concatenazione di message stream costituisca un potente e versatile meccanismo per lo sviluppo di applicazioni di rete.

Con tali conoscenze in mente, è relativamente semplice procedere ad ulteriori estensioni di funzionalità, quali ad esempio una trasmissione di tipo ***multicast***, ossia una trasmissione nella quale si specifica il sottoinsieme dei destinatari che deve ricevere il messaggio.

Tale funzionalità può essere ottenuta con una classe `RoutingOutputStream`, che avrà il compito di incapsulare ogni messaggio dotandolo di una CI costituita da un vettore di stringhe che specificano ciascuna un destinatario.

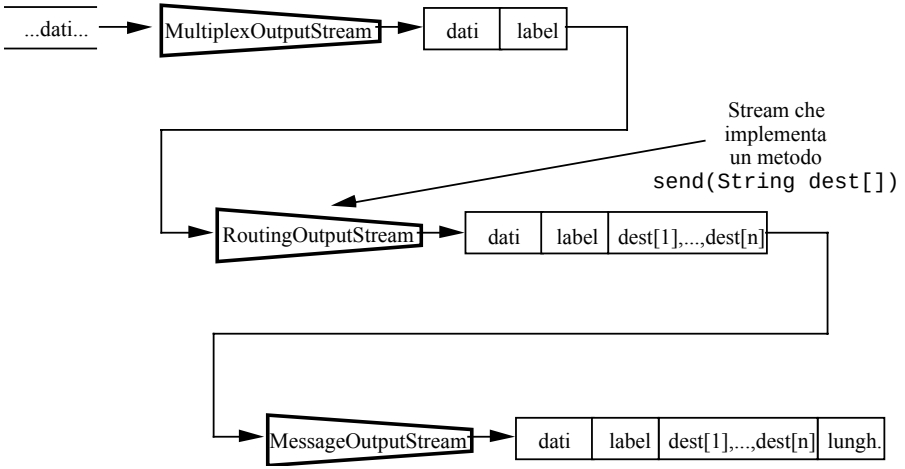


Figura 4-25: Uso di `RoutingOutputStream` per trasmissioni multicast,

All'altra estremità, tipicamente sul server, a valle di un `MessageInputStream` si potrà predisporre una corrispondente classe `RoutingInputStream` che estrae la lista di destinatari e la rende disponibile all'esterno, e una classe `Router` (che ha la funzione di thread copiatore) incaricata di inviare una copia di ogni messaggio solamente ai corrispondenti destinatari.

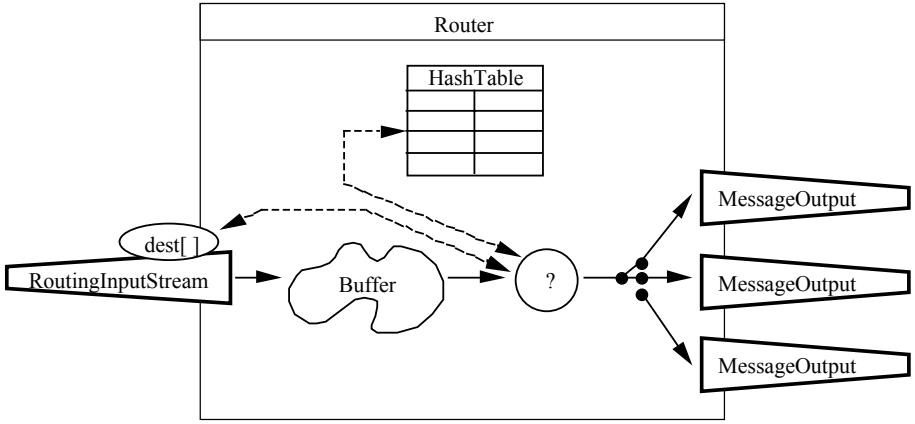


Figura 4-26: `Router`

Con riferimento alla specifica concatenazione illustrata nella figura 4-25, il messaggio che viene instradato dal `Router` è in realtà un pacchetto contenente la CI del `MultiplexOutputStream`, e cioè l'etichetta della componente applicativa che deve ricevere i dati.

Di conseguenza, a valle di ogni `MessageOutput` in uscita dal `Router` ci potrà essere un `MessageInput`, quindi un `MultiplexInputStream` ed infine un `Demultiplexer` che esamina tale etichetta e smista il messaggio alla corretta componente applicativa.

In tal modo è possibile costruire applicazioni di tipo client-server costituite da componenti indipendenti capaci di effettuare trasmissioni multicast, il che consente di soddisfare praticamente qualunque esigenza applicativa.