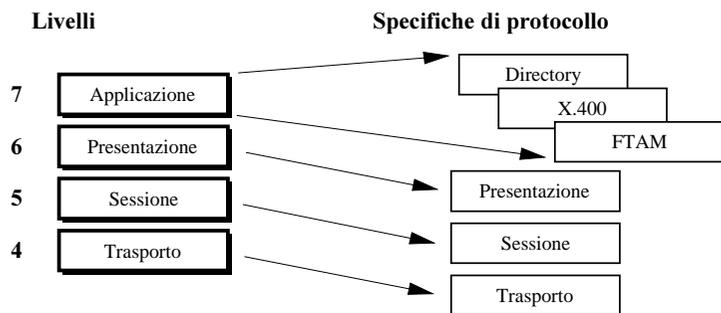
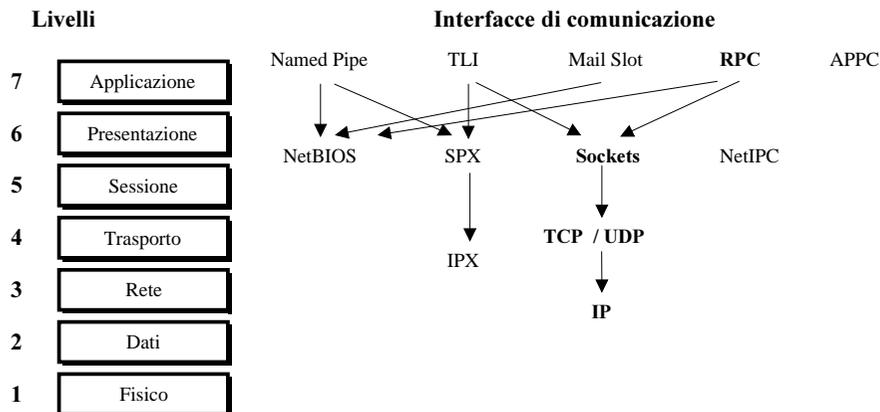


Strumenti di Comunicazione

Livelli di strumenti diversi



Necessità di definire e di diffondere Standard di comunicazione

socket come endpoint per comunicare in modo flessibile, differenziato ed efficiente

UNIX primitive comunicazione/sincronizzazione

In UNIX Berkeley

Socket meccanismo

strumenti di *comunicazioni locali o remote*

con politiche differenziate

Alternativa anche locale superando i problemi degli strumenti concentrati (**pipe** solo tra *processi in gerarchia*) **ben integrata con processi e trasparente**

SOCKET

- processi UNIX possono **scrivere/leggere** messaggi e stream, molte opzioni

Requisiti

- **eterogeneità:** la comunicazione fra processi su architetture diverse
- **trasparenza:** la comunicazione fra processi deve potere avvenire indipendentemente dalla localizzazione fisica
- **efficienza:** l'applicabilità di questi strumenti limitata dalla sola performance
- **compatibilità:** i *naive process* (leggono dallo standard input e scrivono sullo standard output e standard error) devono potere lavorare in ambienti distribuiti senza subire alcuna modifica
- **protocolli di comunicazione** diversi con convenzioni differenziate

concetto di **communication domain**

definizione di più semantiche di comunicazione
e di naming

organizzati in **DOMINI di COMUNICAZIONE**

Semantica di comunicazione include

- affidabilità di una trasmissione
- possibilità di lavorare in *multicast*

Naming è il meccanismo usato per indicare i punti terminali di una comunicazione

Si sceglie il **dominio** più appropriato mediante un'interfaccia standard

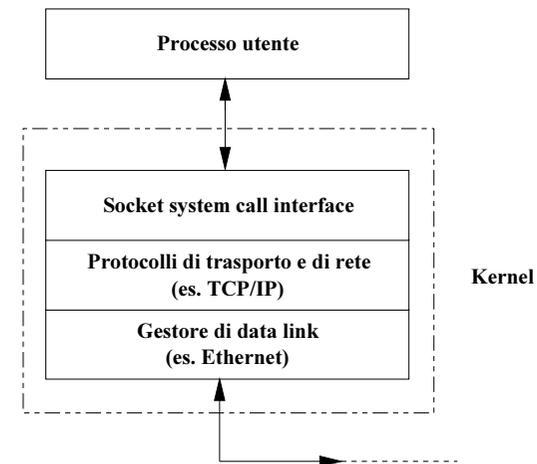
Astrazione unificante **SOCKET**

- comunicazione *in modo omogeneo*
- semantica *uniforme*
scelta connection-oriented e connectionless
consistente nei domini

Proprietà differenziate:

- *trasferimento di dati in modo sequenziale*
- *trasferimento dei dati senza duplicazioni*
- *trasferimento dei dati in modo affidabile*
- *manca di limiti alla lunghezza del messaggio*
- *invio di messaggi out-of-band*
- *comunicazione connection-oriented o connection-less*

TRASPARENZA nella COMUNICAZIONE



Le socket come **interfaccia omogenea** per i servizi

Socket terminale della comunicazione

Socket descriptor integrato con i file descriptor

USO di protocolli di trasporto diversi

anche altri protocolli, diversi da quello che si considera
uso di suite TCP/IP (sia UDP sia TCP)

Chiamata	Significato
<i>open()</i>	Prepara un dispositivo o un file a operazioni di input/output
<i>close()</i>	Termina l'uso di un dispositivo o un file precedentemente aperto
<i>read()</i>	Ottiene i dati da un dispositivo di input o da un file, e li mette nella memoria del programma applicativo
<i>write()</i>	Trasmette i dati dalla memoria applicativa a un dispositivo di output o un file
<i>lseek()</i>	Muove I/O pointer ad una specifica posizione in file /dispositivo
<i>fcntl()</i>	Controlla le proprietà di un file descriptor e le funzioni di accesso
<i>ioctl()</i>	Controlla i dispositivi o il software usato per accedervi

SOCKET

SERVIZI vari tipi di socket

datagram scambio di messaggi senza garanzie supporto **best effort**

stream (stream virtuale)
scambio messaggi *in ordine* nei *due sensi*,
senza errori, non duplicati, out-of-band
flusso nessun confine di messaggio

seqpacket (XNS)

raw (Internet debug)

PROTOCOLLI

anche più **domini di comunicazione**

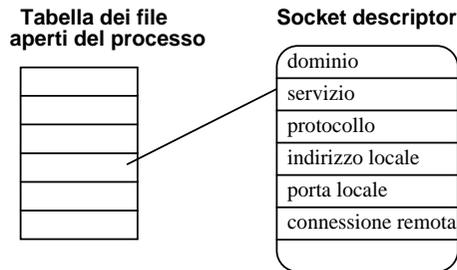
UNIX (AF_UNIX)

Internet (AF_INET)

XEROX (AF_NS)

CCITT (AF_CCITT) X.25

In ogni dominio ci sono **protocolli** diversi



Combinazioni possibili fra **dominio** e **tipo** con indicazione del protocollo

Tipo socket	AF_UNIX	AF_INET	AF_NS
Stream socket	Possibile	TCP	SPP
Datagram socket	Possibile	UDP	IDP
Raw socket	No	IP	Possibile
Seq-pack socket	No	No	SPP

Protocolli più probabili nello standard Berkeley

AF_INET	Stream	IPPROTO_TCP	TCP
AF_INET	Datagram	IPPROTO_UDP	UDP
AF_INET	Raw	IPPROTO_ICMP	ICMP
AF_INET	Raw	IPPROTO_RAW	(raw)
AF_NS	Stream	NSRPROTO_SPP	SPP
AF_NS	Seq-pack	NSRPROTO_SPP	SPP
AF_NS	Raw	NSRPROTO_ERROR	Error Protocol
AF_NS	Raw	NSRPROTO_RAW	(raw)
AF_UNIX	Datagram	IPPROTO_UDP	UDP
AF_UNIX	Stream	IPPROTO_TCP	TCP

prefisso AF ==> Address Family

simboli PF_UNIX, PF_INET, PF_NS, PF_CCITT

la sigla PF significa *Protocol Family* e

coincide completamente con la address family

simboli PF_SNA, PF_DECnet e PF_APPLETALK

NAMING PER LE SOCKET

Nomi logici delle socket (VALIDI LOCALMENTE)

un indirizzo di socket dipende dal dominio

Nomi fisici da associare (end-point VALIDI GLOBALMENTE)

una socket deve essere collegata al sistema fisico ==>

è necessario il **binding**, cioè

legare la socket ad una **porta del nodo**

si realizza una entità detta **half-association**

half-association

- dominio Internet:

{famiglia indirizzo, indirizzo Internet, numero di porta}

socket collegata ad un *numero di porta*

- dominio UNIX:

{famiglia indirizzo, path nel filesystem, file associato}

- dominio CCITT:

indirizzamento legato al protocollo di rete X.25

Internet

nodi **{identificatore_rete, identificatore_host}**

numeri di porta all'interno del nodo (port number)

alcuni riservati (*well-known addresses*) per *servizi noti*

FTP è associato il port number 21

In ambito Internet tutti i port number da 1 a 1023 sono riservati al super-user del sistema.

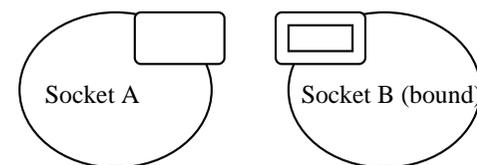
Primitive sulle socket

creazione **socket** in ogni processo

Nome logico LOCALE

s = **socket** (dominio, tipo, protocollo)

```
int s,          /* file descriptor associato alla socket */
    dominio,    /* UNIX, Internet, etc. */
    tipo,       /* datagram, stream, etc. */
    protocollo; /* quale protocollo */
```



- il client ha creato la socket

- il server ha creato la socket

- il server ha collegato la socket ad un indirizzo

Client e Server in stato di pre-connessione

aggancio al sistema 'fisico': **nodi e porte GLOBALI**

rok = **bind** (s, nome, lungnome)

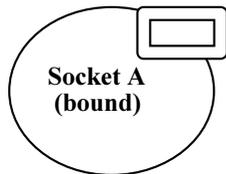
```
int rok, s;
struct sockaddr *nome; /* porta locale per la socket */
int lungnome;
```

In **Internet** a porte **fisiche** ed **host**
(quindi anche in altre macchine)

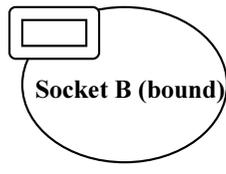
==>

Un processo possiede

socket associate a **porte fisiche**
di un nodo



- il client ha creato la socket
- il client ha collegato la socket ad un indirizzo



- il server ha creato la socket
- il server ha collegato la socket ad un indirizzo

Le **socket** con le **porte** formano **half-association**
(interne ad un solo processo)

sufficiente per scambio messaggi
scambio datagrammi

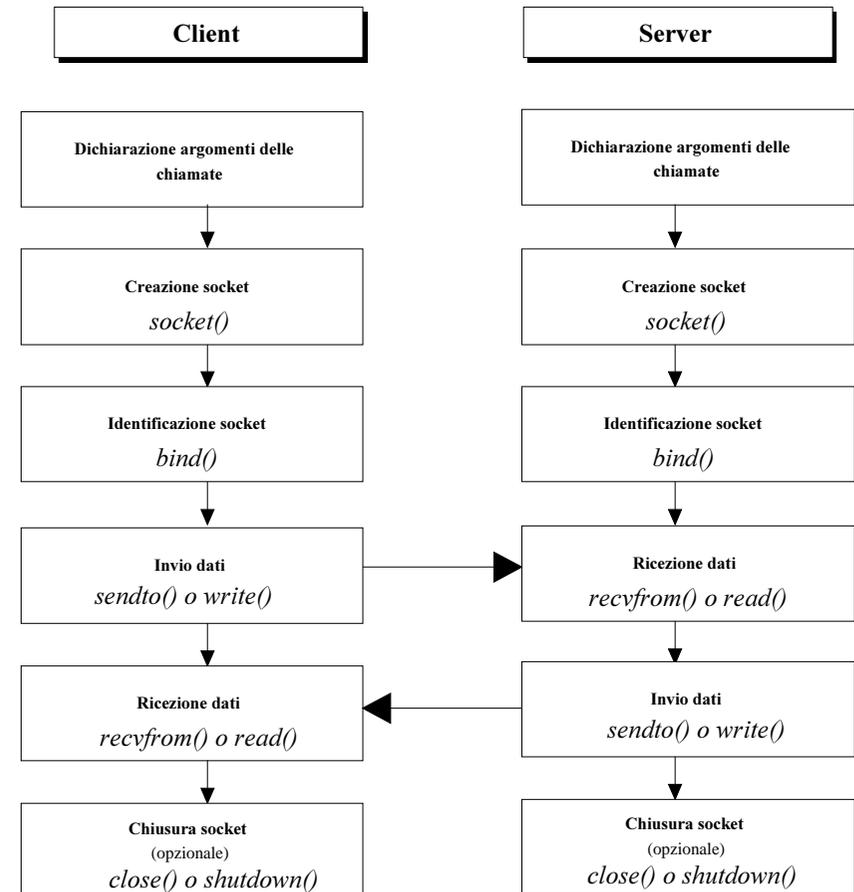
processo Client

- dichiarazione delle variabili di riferimento a socket
- conoscenza dell'*indirizzo Internet* del nodo remoto
- conoscenza della *porta* del servizio da usare

processo Server

- dichiarazione delle variabili di riferimento a socket
- conoscenza della *porta* per il servizio da offrire
- assegnazione di *wildcard address* per l'indirizzo Internet locale (*server multiporta*)

datagrammi



Creazione di una socket datagramma ed uso

```
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
s = socket (AF_INET,SOCK_DGRAM,0);
int s;
```

```
nbytes = sendto (s, msg, len, flags, to, tolen)
int s, nbytes; char *msg;
int len, flags;
struct sockaddr_in *to; int tolen;
```

```
nbytes = recvfrom (s, buf, len, flags, from, fromlen)
int s, nbytes; char *buf;
int len, flags;
struct sockaddr_in *from; int *fromlen;
```

s == *socket descriptor*
buf == *puntatore al messaggio da spedire*
len == *sua lunghezza*
flags == *flags attualmente non sono implementati*
(a parte MSG_PEEK)
to/from == *puntatore all'indirizzo della socket partner*
tolen/fromlen == *sua lunghezza*
nbytes == *lunghezza del messaggio inviato/ricevuto*

Il risultato di ritorno il numero dei byte trasmessi/ricevuti

il flag MSG_PEEK non estrae dalla socket il messaggio

Datagrammi

Lunghezza massima del messaggio
(ad esempio 9K byte o 16K byte)

Uso del protocollo UDP e IP
non affidabile intrinsecamente

A livello utente si può ottenere maggiore affidabilità prevedendo una ritrasmissione dei messaggi

recvfrom ritorna solo un messaggio per volta
per prevenire situazioni di perdita di byte di un messaggio,
len più alto possibile

CLIENTE (MITTENTE)

```
struct sockaddr_in *servaddr;
char msg[2000];
int count;
...
count= sendto (s, msg, strlen(msg), 0, servaddr,
sizeof(struct sockaddr_in));
```

SERVITORE (RICEVENTE)

```
struct sockaddr_in *clientaddr;
char buffer[BUFFERSIZE];
int count;
...
addrlen=sizeof(sockaddr_in);
count= recvfrom (s, buffer, BUFFERSIZE, 0, clientaddr,
&addrlen);
```

Socket con connessione **STREAM socket**
per stabilire una **CONNESSIONE**

PROTOCOLLO DI INIZIALIZZAZIONE

Sincronizzazione CLIENTE/SERVITORE

ruolo **attivo/passivo** in **relazione mutua**

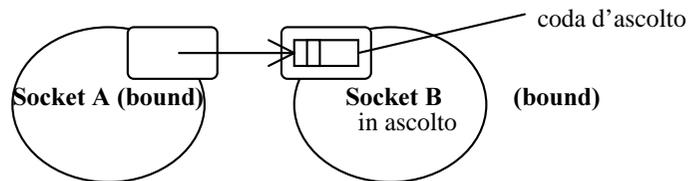
una entità (attiva) richiede la relazione di coppia

una entità (passiva) accetta la relazione

modello **cliente/servitore**

una entità (cliente) richiede il servizio

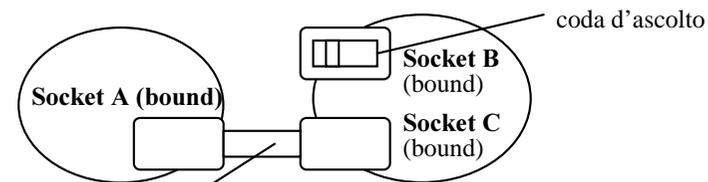
una entità (server) accetta il servizio e risponde



- il client ha fatto una richiesta di connessione

- il server ha ricevuto una richiesta nella coda d'ascolto

Client e Server al momento della richiesta di connessione

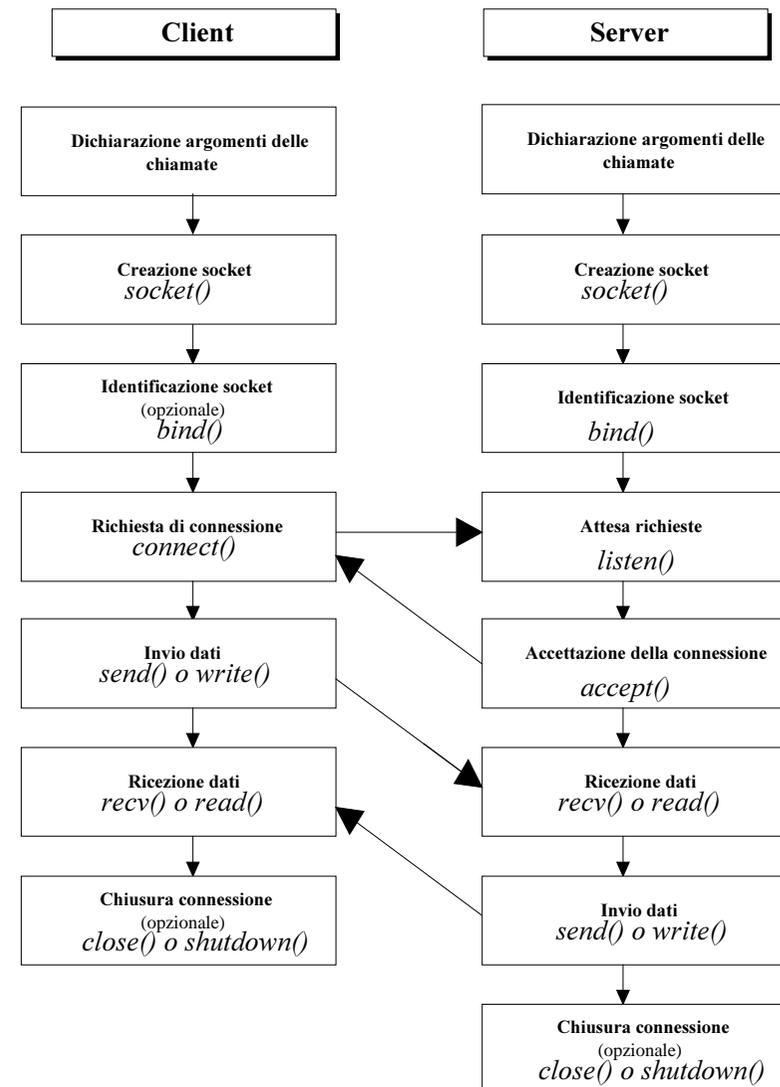


- il server ha accettato la richiesta di connessione
- il server ha stabilito una connessione con il client mediante una nuova socket con le stesse caratteristiche della originale
- la socket originale continua ad ascoltare le eventuali richieste di connessione

Client e Server nello stato di connessione avvenuta

Passi di realizzazione del protocollo

per la connessione



Connessione tra la attiva e la nuova socket

SOCKET CONNESSE

CONNESSIONE permane fino alla chiusura di una delle due half-association

CANALE VIRTUALE

I processi collegati si possono mandare messaggi (**send**, **recv**) ma possono fare **read** e **write**

recnum = **read** (s, buff, length);

sentnum = **write** (s, buff, length);

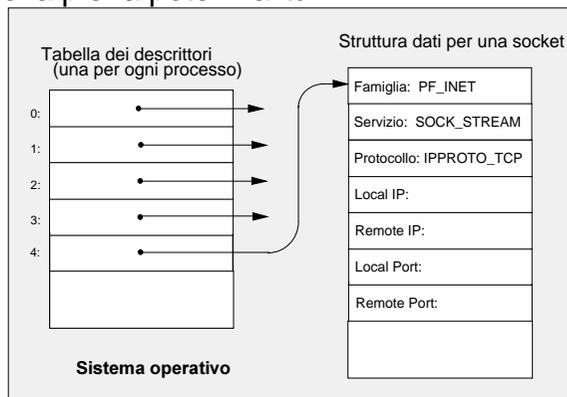
SOCKET usate in modo differenziato

processi naive

Quindi un processo può avere **input/output** ridiretto su una socket e lavorare in remoto

processi più intelligenti

sfruttano la piena potenzialità



Molte possibilità non di default (avanzate)

come ritrovare i dati **out-of-band**

come realizzare **attese multiple** o **operazioni asincrone**

UNIX - Variabili per socket address

Nomi collegati alle socket

socket address in due tipi di strutture

sockaddr_in (famiglia AF_INET)

sockaddr

```
struct sockaddr {
```

```
    u_short    sa_family;
```

```
    char       sa_data[14];
```

```
}
```

```
struct sockaddr_in {
```

```
    u_short    sin_family;
```

```
    u_short    sin_port;
```

```
    struct in_addr sin_addr;
```

```
/* char       sin_zero [8]; non usata */
```

```
}
```

```
struct in_addr {u_long    s_addr};
```

```
struct sockaddr_in myaddr;
```

sin_family == famiglia di indirizzamento AF_INET

sin_port == port address

da assegnare prima del binding o

prima della richiesta di connessione

il server deve conoscere l'indirizzo della socket

il client può ottenerlo dalla primitiva

sin_addr == una struttura Internet del nodo remoto

Si vedano i file di inclusione corrispondenti

```
#include <sys/types.h>
```

```
#include <netinet/in.h>
```

```
#include <sys/socket.h>
```

Indirizzo Internet di un Host remoto

NOMI GLOBALI

corrispondenza tra **nome logico** di dominio del nodo e **nome fisico** Internet di nodo ==>

primitiva *gethostbyname()*

restituisce l'indirizzo Internet e lunghezza

```
#include <netdb.h>
```

```
struct hostent * gethostbyname (name)
```

```
char *name;
```

name dichiarato in /etc/hosts

la tabella riporta per ogni nome logico di nodo

indirizzo Internet

(primi 2 bytes per la rete ed 1 per la sottorete, 1 per host)

aliases

```
137.204.56.11  didahp1  hp1
```

```
137.204.56.12  didahp2  hp2
```

```
137.204.56.13  didahp3  hp3
```

gethostbyname restituisce

un puntatore alla struttura *hostent*

oppure NULL se fallisce

La ricerca avviene solo localmente

```
struct hostent {
```

```
char * h_name; /* nome ufficiale dell'host */
```

```
char ** h_aliases; /* lista degli aliases */
```

```
int h_addrtype; /* tipo dell'indirizzo dell'host */
```

```
int h_length; /* lunghezza dell'indirizzo */
```

```
char ** h_addr_list;
```

```
/* lista degli indirizzi presa dal server dei nomi host */
```

```
#define h_addrh_addr_list[0]
```

```
/* indirizzo dell'host in compatibilità con altre versioni
```

```
La struttura si trova in /usr/include/netdb.h per
```

```
HP-UX v8.0. */
```

```
}
```

Esempio di utilizzo

```
#include <netdb.h>
```

```
struct hostent * hp;
```

```
struct sockaddr_in peeraddr;
```

```
peeraddr.sin_family= AF_INET;
```

```
peeraddr.sin_port= 22375;
```

```
if (hp=gethostbyname(argv[1]) )
```

```
peeraddr.sin_addr.s_addr=
```

```
((struct in_addr*)(hp->h_addr))-> s_addr;
```

```
else /* errore */
```

Utile esercizio

Si può pensare di estendere la ricerca nel nodo di servizio:
i sistemi son andati in questa direzione (**NCS**)

Port address per un servizio

getservbyname() restituisce il numero di porta relativo ad un servizio noto

tabella di corrispondenze fra **servizi** e **porte**
(e **protocollo**)

file /etc/services

<i>echo</i>	<i>7/tcp</i>	<i># Echo</i>
<i>systat</i>	<i>11/tcp users</i>	<i># Active Users</i>
<i>daytime</i>	<i>13/tcp</i>	<i># Daytime</i>
<i>daytime</i>	<i>13/udp</i>	<i>#</i>
<i>qotd</i>	<i>17/tcp quote</i>	<i># Quote of the Day</i>
<i>chargen</i>	<i>19/tcp ttytst source</i>	<i># Character</i>
<i>chargen</i>	<i>19/udp ttytst source</i>	<i># Generator</i>
<i>ftp-data</i>	<i>20/tcp</i>	<i># File Transfer Protocol (Data)</i>
<i>ftp</i>	<i>21/tcp</i>	<i># File Transfer Protocol (Control)</i>

utilizzata *prima del bind*
se il client non lo compie,
prima della richiesta di connessione

```
#include <netdb.h>
struct servent *getservbyname (name,proto)
    char *name,*proto;
(se TCP è l'unico registrato con quel servizio allora 0)
```

Esempio

```
#include <netdb.h>
struct servent *sp; struct sockaddr_in peeraddr;
sp=getservbyname("example","tcp");
peeraddr.sin_port=sp->s_port;
```

Wildcard address per trovare indirizzo Internet locale

Indirizzo in socket address locale INADDR_ANY interpretato come qualunque indirizzo valido

In caso di workstation con porte multiple
(e più indirizzi Internet)
Il server può accettare le connessioni da ogni indirizzo

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
struct sockaddr_in sin;
sin.sin_addr.s_addr= INADDR_ANY;
<<identificazione della socket>>
```

In questo modo, si possono attendere richieste da qualunque potenziale cliente anche per server con indirizzi di rete multipli

Creazione di una socket

Chiamata *socket()* crea un punto terminale di comunicazione fra processi

{famiglia d'indirizzamento, tipo, protocollo di trasporto}

```
#include <sys/types.h>
#include <sys/socket.h>
int socket (af, type, protocol)
    int af,type,protocol;
```

La primitiva **socket()** restituisce *socket descriptor* o
-1 se la creazione fallisce

protocollo di trasporto anche 0 oppure uno specifico

Uso di *getprotobyname()*

uso del file /usr/include/netinet/in.h

```
int s;
s=socket (AF_INET,SOCK_STREAM,0);
```

aggancio della socket in half-association

La chiamata *bind()* collega alla **socket creata la porta**

Il server **deve** legare un port address
prima di accettare connessione

Il client può fare il bind o meno

il sistema assegna un numero di porta fra le libere

il cliente tenta la connessione con la porta del server

Il server risponde ad una richiesta di porta corretta

Half-association per una socket: BIND

```
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
int bind ( s, addr, addrlen)
    int s;
    struct sockaddr_in *addr;
    int addrlen;
```

s == socket descriptor che identifica la socket

addr == struttura con indirizzo di porta e
indirizzo internet del nodo stesso

addrlen == la lunghezza di addr

Socket address con sin_port a 0 in alcuni sistemi
porta alla assegnazione automatica
di un numero di porta libero
(come fare nel caso questo non sia possibile?)

```
struct sockaddr_in myaddr;
<<assegnazione del socket address>>
bind (s,&myaddr,sizeof(struct sockaddr_in));
```

CLIENT - Richiesta di connessione

Il processo **client** richiede una connessione con

```
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <errno.h>
```

```
int connect(s, addr, addrlen)
```

```
int s, addrlen;
```

```
struct sockaddr_in *addr;
```

s == socket descriptor

addr == puntatore al socket address remoto

addrlen == lunghezza di questa struttura

Casi di errore in errno:

- ETIMEDOUT: tentativo di connessione in time-out: la coda d'ascolto del server è piena o non creata
- EISCONN: una socket non può essere connessa più volte per non duplicare association
- ECONNREFUSED: impossibilità di connessione

connect() sincrona

se non modalità asincrona *nonblocking I/O*

Connect() può anche fare il **bind()** automaticamente

Il **client** al successo della connect() usa la connessione

sperando che il tutto sia andato a buon fine (?)

```
struct sockaddr_in *peeraddr;
```

```
<< assegnazione del socket address remoto >>
```

```
s=socket(AF_INET,SOCK_STREAM,0);
```

```
connect(s,&peeraddr,sizeof(struct sockaddr_in));
```

SERVER - Attesa di richieste di connessione

Dopo il bind, il server deve preparare una **coda** di accettazione per le **richieste** di connessione dei client

primitiva *listen()*

si crea la coda per il socket descriptor

la connessione viene stabilita solo dopo l'accodamento di una richiesta di un client

la socket **deve** essere collegata alla porta dal sistema

```
<<nessun file incluso>>
```

```
int listen(s,backlog)
```

```
int s,backlog;
```

s == socket descriptor

backlog == intero fra 1 e 20

massimo numero di richieste in coda (consigliato <=5)

Evento *richiesta non accodata*

segnalato **solo** al **processo client** (ETIMEDOUT)

non segnalato al **server**

Variabile globale segnala il tipo di errore dalla chiamata

file /usr/include/errno.h

sistema *errno* ==> valore EINVAL

```
<< preparazione argomenti >>
```

```
s = socket(AF_INET,SOCK_STREAM,0);
```

```
<< identificazione della socket >>
```

```
listen(s,5);
```

SERVER - Accettazione di una richiesta di connessione

Il **server** accetta le richieste di connessione

primitiva `accept()` ==>

si crea una nuova socket per la connessione e si fornisce il socket descriptor relativo

La **nuova socket**:

- ha una semantica di comunicazione come la vecchia
- ha lo stesso port address della vecchia socket
- è connessa alla socket del client

La primitiva **`accept()` sincrona**

(a meno di modalità di tipo nonblocking I/O)

```
#include <sys/types.h>
```

```
#include <netinet/in.h>
```

```
#include <sys/socket.h>
```

```
int accept (ls, addr, addrlen)
```

```
int ls, *addrlen; struct sockaddr_in *addr;
```

ls == socket descriptor

altri parametri passati per indirizzo

addr == indirizzo del socket address connesso

addrlen == la lunghezza espressa in byte

Area in spazio utente per la socket

La `accept()` non offre la possibilità di filtrare le richieste devono essere accettate tutte oppure nessuna

Sono possibili **errori** (vedi `errno`)

Esempio

```
struct sockaddr_in peeraddr;
```

```
<< allocazione del socket address ed inizializzazione dell'area di memoria associata: socket, bind, listen >>
```

```
addrlen=sizeof(sockaddr_in);
```

```
s=accept(ls, &peeraddr, &addrlen);
```

Invio e Ricezione di dati sulla connessione

Uso dei socket descriptor come file descriptor UNIX

`read()` / `write()` e `send()` / `recv()`

```
#include <sys/types.h>
```

```
#include <sys/socket.h>
```

```
int send (s, msg, len, flags)
```

```
int s; char *msg; int len,flags;
```

```
int recv (s, buf, len, flags)
```

```
int s; char *buf; int len,flags;
```

s == socket descriptor

buf /msg == puntatore al messaggio

len == lunghezza del messaggio

flags == opzioni di comunicazione

• **flags** `send()`:

• 0 per un funzionamento normale,

• `MSG_OOB` per un messaggio out-of-band.

• **flags** `recv()`:

• 0 per un funzionamento normale,

• `MSG_OOB` per un messaggio out-of-band,

• `MSG_PEEK` per una lettura non distruttiva dallo stream

Il risultato della primitive rappresenta il numero di byte rispettivamente inviati o ricevuti (-1 se fallimento)

chiamate sincrone

(a meno delle modalità nonblocking I/O)

```
char buf[30]="Messaggio di prova";
int count;
count=send(s,buf,30,0);
.....
count=recv(s,buf,5,0);
```

I messaggi sono realmente inviati a primitiva terminata?

==> i dati sono bufferizzati dal protocollo di trasporto TCP: non è detto che siano immediatamente inviati. Solo alla prima comunicazione 'vera' otteniamo indicazioni

Soluzione ==> messaggi di *lunghezza pari al buffer* o si comanda il flush del buffer

Come preservare i messaggi in ricezione?

==> ogni receive restituisce i dati pronti: *stream mode* TCP non implementa marcatori di fine messaggio

Soluzioni ==> messaggi a *lunghezza fissa*
messaggi a *lunghezza variabile*
si alternano un messaggio a lunghezza fissa e uno vero e proprio variabile in lunghezza
il primo contiene la lunghezza del secondo

Chiusura di una socket di Connessione

Primitiva *close()*

int **close** (s)

int s;

La chiamata *close()* decrementa il contatore dei processi referenti al socket descriptor

il chiamante non lavora più con quel descrittore

dopo un intervallo di tempo controllato da un'opzione avanzata (SO_LINGER)

vedi **demon** che ha creato la connessione e passa al server il socket descriptor
il demon deve chiudere la socket

int sd;

sd=socket(AF_INET,SOCK_STREAM,0);

.....

close(sd);

Alla chiusura, ogni messaggio nel buffer è spedito, mentre ogni dato non ricevuto viene perduto

shutdown() più elegante per abbattimento della connessione per una migliore gestione delle risorse

Le operazioni di chiusura sono garanzie di collegamento con il pari sul canale

Shutdown

Per chiudere una connessione in uno solo dei due versi
la trasmissione passa da full-duplex ad half-duplex

tutti i processi che usano lo stesso socket descriptor

<<primitiva locale: nessun inclusione>>

int **shutdown** (s,how)

int s, how;

s == socket descriptor della socket

how == direzione

0 non si **ricevono** più dati, ma si può trasmettere al pari
send() (e ogni scrittura successiva) ritorna con -1 ed il
processo pari riceve SIGPIPE
(importante in caso **non bloccante**)

1 si possono ricevere i dati dalla socket
senza **ulteriori trasmissioni** di dati
abbattimento graceful della connessione
Il processo collegato alla connessione alla lettura
riceve end-of-file

2 entrambi gli effetti

Esempio di chiusura di un canale di trasmissione di una
socket ed invia *end-of-file* al processo remoto

```
int s; /* socket descriptor aperto su stream*/  
shutdown (s,1);
```

Considerando il protocollo sottostante

In TCP

CLOSE

la **close** (fd) segnala la immediata intenzione di
non fare più operazioni sullo stream =>

tutti i dati dell'altra direzione in attesa non vengono ricevuti e
il ricevente ottiene una fine del file =>

deve fare una close

I segmenti non ancora trasmessi dalla parte di chi chiude
possono essere buttati via

I segmenti non ancora trasmessi dalla parte del pari
possono essere buttati via e non trasmessi

SHUTDOWN

la **shutdown** (fd, 1) segnala la intenzione di
non fare **più invii** sullo stream =>

il ricevente è avvertito da una fine del file =>

può fare anche molte ricezioni

può fare anche molti invii

decide quando fare la shutdown dal suo verso

si attua una chiusura dolce e

i segmenti non ancora trasmessi dal pari possono essere
inviati ed effettivamente ricevuti

Trasparenza di comunicazioni fra processi

Uso di write() e read() al posto delle send() e recv()

```
#include <unistd.h>
ssize_t write (fileds,buf,nbytes)
ssize_t read (fileds,buf,nbytes)
    int fileds;
    char *buf;
    size_t nbytes;
```

trasparenza della comunicazione

Prima **close()** dei file descriptors 0,1,2

Uso di **fork()** di un comando locale
dopo aver creato tre socket
con socket descriptor *stdin*, *stdout* e *stderr*

Poi **exec()**

Il processo locale naif è così
connesso ad un processo remoto

Funzioni ausiliarie

obiettivo portabilità

Manipolazione interi

Quattro funzioni di libreria per convertire da formato di rete in formato interno per interi (lunghi o corti)

```
/* trasforma un intero da formato esterno in interno */
shortlocale = ntohs(shortrete);
longlocale = ntohl(longrete);
```

```
/* trasforma un intero da formato interno in esterno */
shortrete = htons(shortlocale);
longrete = htonl(longlocale);
```

Manipolazione indirizzi IP

Conversione tra indirizzi IP a 32 bit e corrispondente notazione col punto decimale (ascii: "123.34.56.78")

inet_addr() converte l'indirizzo dalla forma con punto decimale
indirizzo = **inet_addr**(stringa);
stringa contiene l'indirizzo in formato punto decimale
risultato *indirizzo* IP a 32 bit

inet_ntoa() esegue la funzione inversa
stringa = **inet_ntoa**(indirizzo);
indirizzo indirizzo IP a 32 bit (cioè un *long integer*)
risultato *stringa* di caratteri con indirizzo in forma con punto

FNZIONI che non richiedono fine stringa (solo blocchi di byte)

bcmp (addr1, addr2, length)	memset (addr1, char, length)
bcopy (addr1, addr2, length)	memcpy (addr1, addr2, length)
bzero (addr1, length)	memcmp (addr1, addr2, length)

Principali chiamate dell'interfaccia socket

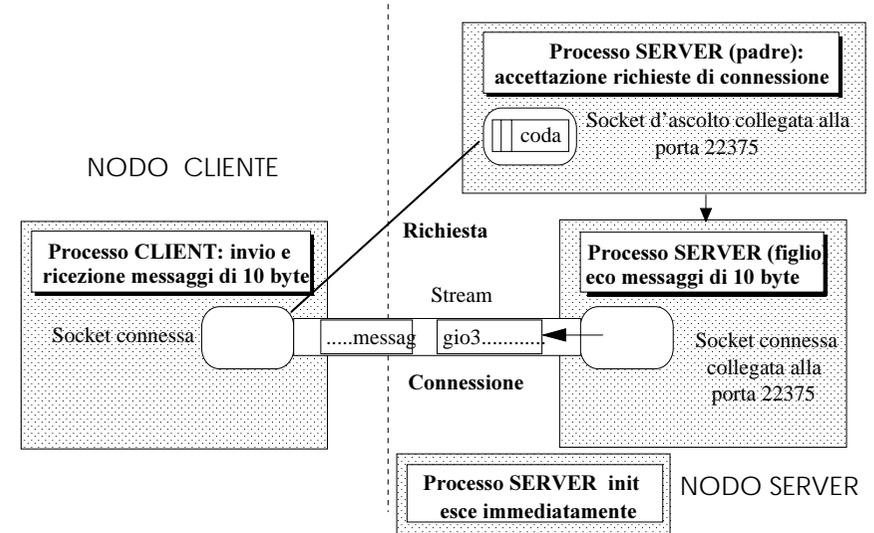
di kernel dal BSD 4.2

Oltre alle funzioni viste sopra,
(si sperimenti la *gethostbyname* per il nodo locale)
si vedano le azioni

Chiamata	Significato
<i>socket()</i>	Crea un descrittore da usare nelle comunicazioni di rete
<i>connect()</i>	Connette la socket a una remota
<i>write()</i>	Spedisce i dati attraverso la connessione
<i>read()</i>	Riceve i dati dalla connessione
<i>close()</i>	Termina la comunicazione e dealloca la socket
<i>bind()</i>	Lega la socket con l'endpoint locale
<i>listen()</i>	Socket in modo passivo e predispose la lunghezza della coda per le connessioni
<i>accept()</i>	Accetta le connessioni in arrivo
<i>recv()</i>	Riceve i dati in arrivo dalla connessione
<i>recvmsg()</i>	Riceve i messaggi in arrivo dalla connessione
<i>recvfrom()</i>	Riceve i datagrammi in arrivo da una destinazione specificata
<i>send()</i>	Spedisce i dati attraverso la connessione
<i>sendmsg()</i>	Spedisce messaggi attraverso la connessione
<i>sendto()</i>	Spedisce i datagrammi verso una destinazione specificata
<i>shutdown()</i>	Termina una connessione TCP in una o in entrambe le direzioni
<i>getsockname()</i>	Permette di ottenere la socket locale legata dal kernel (vedi parametri <i>socket, sockaddr, length</i>)
<i>getpeername()</i>	Permette di ottenere l'indirizzo del pari remoto una volta stabilita la connessione (vedi parametri <i>socket, sockaddr, length</i>)
<i>getsockopt()</i>	Ottiene le opzioni settate per la socket
<i>setsockopt()</i>	Cambia le opzioni per una socket
<i> perror()</i>	Invia un messaggio di errore in base a <i>errno</i> (stringa su <i>stderr</i>)
<i> syslog()</i>	Invia un messaggio di errore sul file di log (vedi parametri <i>priority, message, params</i>)

Si sperimentino le primitive non note (e note)

Esempio con CONNESSIONE e PARALLELISMO



/* CLIENT */

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
```

```
char *ctime(); /* routine di formattazione dell'orario */
int ricevi () /* routine di ricezione di un messaggio*/
```

```
int s; /* socket descriptor */
struct hostent *hp; /* puntatore alle informazioni host remoto */
long timevar; /* contiene il ritorno dalla time() */
struct sockaddr_in myaddr_in; /* socket address locale */
struct sockaddr_in peeraddr_in; /* socket address peer */
```

```

main(argc, argv)
int argc; char *argv[];
{   int addrlen, i;
    char buf[10];/* L'esempio scambia messaggi da 10 bytes */

if (argc != 3)   {
    fprintf(stderr, "Uso: %s <host remoto> <nric>\n", argv[0]);
    exit(1);    }

/* azzera le strutture degli indirizzi */
memset ((char *)&myaddr_in, 0, sizeof(struct sockaddr_in));
memset ((char *)&peeraddr_in, 0, sizeof(struct sockaddr_in));

/* assegnazioni per il peer address da connettere */
peeraddr_in.sin_family = AF_INET;
/*richiede informazioni a proposito del nome dell'host
passato alla chiamata */
hp = gethostbyname (argv[1]);
if (hp == NULL) {
    fprintf(stderr, "%s: %s non trovato in /etc/hosts\n",
            argv[0], argv[1]);
    exit(1); }
peeraddr_in.sin_addr.s_addr =
    htonl(((struct in_addr *) (hp->h_addr))->s_addr);

/*definisce il numero di porta senza la chiamata
getservbyname(): non registrato in /etc/services

```

Se fosse registrato nel nodo cliente avremmo:

```

struct servent *sp;
/* puntatore alle informazioni del servizio */

sp = getservbyname ("example", "tcp");
if (sp == NULL) {
    fprintf(stderr, "%s: non trovato in /etc/services\n", argv[0]);
    exit(1); }
peeraddr_in.sin_port = htons (sp->s_port);
*/

peeraddr_in.sin_port = htons(22375);

/* creazione della socket */
s = socket (AF_INET, SOCK_STREAM, 0);
if (s == -1) {
    perror(argv[0]);
    fprintf(stderr, "%s: non posso creare la socket\n", argv[0]);
    exit(1); }

/* No bind(): il numero di porta del client assegnato dal sistema.
Il server lo conosce alla richiesta di connessione; il processo
client lo ricava con la primitiva getsocketname()
Modificare nel caso non si abbia questa feature */
/*tentativo di connessione al server remoto */
if ( connect (s, &peeraddr_in, sizeof(struct sockaddr_in)) == -1)
{   perror(argv[0]);
    fprintf(stderr, "%s: impossibile connettersi con il server \n",
            argv[0]);
    exit(1);
}
/* lo stream è stato ottenuto in modo completo (!?) */

```

```

addrrlen = sizeof(struct sockaddr_in);
if ( getsockname(s, &myaddr_in, &addrrlen) == -1)
    { perror(argv[0]);
      fprintf(stderr, "%s: impossibile leggere il socket address\n",
              argv[0]);
      exit(1); }

```

/ scrive un messaggio iniziale per l'utente */*

```

time(&timevar);
printf("Connessione a %s sulla porta %u alle %s",
       argv[1], ntohs(myaddr_in.sin_port), ctime(&timevar));

```

/ Il numero di porta espresso in byte in formato interno: conversione nel formato host mediante **ntohs()**. Per alcuni host i due formati coincidono (sistemi HP-UX), maggiore portabilità*

**/*

```

sleep(5); /* attesa che simula un'elaborazione */

```

/ NON C'È PRESENTAZIONE DEI DATI*

*Invio di **messaggi** al processo server remoto*

Si mandano un insieme di interi successivi

****buf=i*** pone i primi 4 byte del vettore buf uguali alla codifica dell'intero in memoria

Il server rispedisce gli stessi messaggi al client (senza usarli)

Aumentando il numero e la dimensione dei messaggi, potremmo anche occupare troppa memoria dei gestori di trasporto =>

*sia il server che il client stabiliscono un limite alla memoria associata alla coda delle socket */*

/* invio di tutti i messaggi */

```

for (i=1; i<=atoi(argv[2]); i++) {
    *buf = htonl( i );
    if ( send(s, buf, 10, 0) != 10)
        { fprintf(stderr, "%s: Connessione terminata per errore",
                  argv[0]);
          fprintf(stderr, "sul messaggio n. %d\n", i);
          exit(1);
        }
}

```

/*Shutdown() della connessione per successivi invii (modo 1): Il server riceve un end-of-file dopo le richieste e riconosce che non vi saranno altri invii di messaggi */

```

if ( shutdown(s, 1) == -1) {
    perror(argv[0]);
    fprintf(stderr, "%s: Impossibile eseguire lo shutdown\
della socket\n", argv[0]);
    exit(1); }

```

/*Ricezione delle risposte dal server

Il loop termina quando la recv() fornisce zero, cioè la terminazione end-of-file. Il server la provoca quando chiude il suo lato della connessione/*

```

while (ricevi(s, buf, 10) )
    printf("Ricevuta la risposta n. %d\n", ntohl( *buf) );
/* Per ogni messaggio ricevuto, diamo un'indicazione locale */

```

/ Messaggio per indicare il completamento del programma. */*

```

time(&timevar);
printf("Terminato alle %s", ctime(&timevar));
}

```

```

int ricevi (s, buf, n)
    int s; char * buf; int n;
{ /* ricezione di un messaggio di specificata lunghezza
   da una socket */
int i, j;
/*Il ciclo interno verifica che la recv() non ritorni un
messaggio più corto di quello atteso (n=10 byte)
recv ritorna appena vi sono dati e non attende tutti i
dati richiesti
Il loop interno garantisce la ricezione fino al decimo byte
per permettere alla recv successiva di partire sempre
dall'inizio di una risposta

while (i = recv (s, buf, n, 0))
{   if (i == -1) {
        perror(argv[0]);
        fprintf(stderr, "%s: errore nella lettura delle risposte\n",
                argv[0]);
        exit(1); }
    while (i < 10) {
        j = recv (s, &buf[i], n-i, 0);
        if (j == -1) {
            perror(argv[0]);
            fprintf(stderr, "%s: errore nella lettura delle risposte\n",
                    argv[0]);
            exit(1); }
        i += j; }
}
*/ Per messaggi di piccola dimensione la
frammentazione è improbabile, ma con dimensioni
superiori (già a qualche Kbyte) il pacchetto può venire
suddiviso dai livelli sottostanti, e una ricezione parziale
diventa più probabile. In ogni caso, in ricezione
dobbiamo sempre aspettare l'intero messaggio, se questo
è significativo */

```

/* SERVER con parallelismo*/

```

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <signal.h>
#include <stdio.h>
#include <netdb.h>

long timevar; /* contiene il valore fornito da time() */
int s; /* socket descriptor */
int ls; /* socket per la listen()*/
struct hostent *hp; /* puntatore all' host remoto */
struct sockaddr_in myaddr_in; /* per il socket address locale */
struct sockaddr_in peeraddr_in; /* per il socket address peer */

main(argc, argv)
int argc; char *argv[];
{   int addrlen;

/* Azzera le strutture degli indirizzi */
memset ((char *)&myaddr_in, 0, sizeof(struct sockaddr_in));
memset ((char *)&peeraddr_in, 0, sizeof(struct sockaddr_in));

/* Assegna la struttura d'indirizzo per la listen socket. */
myaddr_in.sin_family = AF_INET;
/*Il server ascolta su un qualunque suo indirizzo
(wildcard address), invece che sul suo indirizzo di rete.
Convenzione per server connessi a più reti: consente di
attendere richieste su ogni rete
==> maggiore portabilità del codice */

myaddr_in.sin_addr.s_addr = INADDR_ANY;
/* assegna il numero di porta */
myaddr_in.sin_port = 22375;

```

```

/* Crea la socket d'ascolto. */
ls = socket (AF_INET, SOCK_STREAM, 0);
if (ls == -1) {
    perror(argv[0]);
    fprintf(stderr, "%s: impossibile creare la socket.\n", argv[0]);
    exit(1); }

```

```

/* Collega la socket all'indirizzo fissato */
if ( bind (ls, &myaddr_in, sizeof(struct sockaddr_in)) == -1) {
    perror(argv[0]);
    fprintf(stderr, "%s: impossibile eseguire il collegamento.\n",
        argv[0]);
    exit(1); }

```

```

/* Inizializza la coda d'ascolto richieste (massimo 5 pendenti)*/
if ( listen (ls, 5) == -1) {
    perror(argv[0]);
    fprintf(stderr, "%s: impossibile l'ascolto sulla socket\n",
        argv[0]);
    exit(1); }

```

/*Inizializzazione completata.

Il programma crea un processo daemon. La chiamata setsid() sgancia il processo dal terminale di controllo e lo stacca dal gruppo del processo padre (il processo diventa leader di una nuova sessione non collegata a nessun terminale) */

```

setsid();          /* Unix System V */

```

```

switch (fork()) {
case -1:/* Impossibilità di creazione di un processo figlio. */
    perror(argv[0]);
    fprintf(stderr, "%s: impossibile creare un daemon.\n", argv[0]);
    exit(1);

```

```

case 0: /* FIGLIO e schema di processo DEMONE */

```

/*Il demone entra in un loop di attesa e, ad ogni richiesta, crea un processo figlio per servire la chiamata. Il daemon chiude lo stdin e lo stderr, mentre lo stdout è assunto come ridiretto ad un file di log per registrare gli eventi di esecuzione*/

```

    close(stdin);
    close(stderr);

```

/*Il segnale SIGCLD è ignorato (SIG_IGN) per evitare di mantenere processi zombi per ogni servizio eseguito. Il daemon non deve occuparsi degli zombies */

```

signal(SIGCLD, SIG_IGN);

```

```

for (;;) {
    addrlen = sizeof(struct sockaddr_in);

```

/*accept() bloccante in attesa di richieste di connessione. Completa fornendo l'indirizzo del chiamante e la sua lunghezza, oltre che un socket descriptor per la connessione instaurata */

```

    s = accept (ls, &peeraddr_in, &addrlen);
    if ( s == -1) exit(1);

```

```

switch (fork()) {
case -1:
/* Non è possibile generare un figlio ed allora esce */
    exit(1);
case 0:
/* Esecuzione del processo figlio che gestisce il servizio */
    server(); /* ulteriore figlio per il servizio */
    exit(0);

default:
/* Il processo daemon chiude il socket descriptor e
termina (potrebbe tornare ad accettare ulteriori richieste). Questa operazione consentirebbe al daemon
di non superare il massimo dei file descriptor ed al
processo figlio fare una close() effettiva sui file */
    close(s);    }
}

default: /* processo INIZIALE */
/*Il processo iniziale esce per lasciare libera la console
*/
    exit(0); }
}

```

```

char *inet_ntoa(); /* routine formattazione indirizzo Internet */
char *ctime(); /* routine di formattazione dell'orario ottenuto da
una time () */

int ricevi ();

/* procedura SERVER: routine eseguita dal processo
figlio che il daemon crea per gestire il servizio
Si ricevono alcuni pacchetti dal processo client, si
elaborano e si ritornano i risultati al mittente; inoltre
alcuni dati sono scritti sullo stdout */
server()
{ int reqcnt = 0; /* conta il numero di messaggi */
char buf[10]; /* l'esempio usa messaggi di 10 bytes */
char *hostname; /* nome dell'host richiedente */
int len, len1;

/* Chiude la socket d'ascolto ereditata dal processo daemon */
close (ls);

/*Cerca le informazioni relative all'host connesso
mediante il suo indirizzo Internet
Primitiva gethostbyaddr() */
hp = gethostbyaddr ((char *) & ( ntohl( peeraddr_in.sin_addr) ,
sizeof (struct in_addr), peeraddr_in.sin_family);

if (hp == NULL) hostname = inet_ntoa (peeraddr_in.sin_addr);
/* Non trova host ed allora assegna l'indirizzo formato Internet */
else
{ hostname = hp->h_name; /* punta al nome dell'host */ }
/*stampa un messaggio d'avvio*/
time (&timevar);
printf("Inizio dal nodo %s porta %u alle %s",
hostname, ntohs(peeraddr_in.sin_port), ctime(&timevar));

```

*/*Loop di ricezione messaggi del cliente*

Uscita alla ricezione dell'evento di shutdown del processo client, cioè all'evento fine del file ricevuto dal server

*Si possono anche assegnare opzioni avanzate alla socket, come, ad esempio, la chiusura ritardata per offrire più sicurezza ai dati (SO_LINGER) */*

```
while (ricevi (s, buf, 10) )
{ reqcnt++; /* Incrementa il contatore di messaggi */
  sleep(1); /* Attesa per simulare l'elaborazione dei dati */
  /*Invio della risposta al cliente per ogni messaggio*/
  if ( send (s, buf, 10, 0) != 10)
    {printf("Connessione a %s abortita sulla send\n", hostname);
      exit(1); }
  /* sui dati mandati e ricevuti non facciamo nessuna
  trasformazione */
}
/*Il loop termina se non vi sono più richieste da servire
La close() può avere un effetto ritardato per attendere la
spedizione di tutte le risposte al processo client.
Questa modalità avanzata può permettere al processo
server di rilevare l'istante di cattura dell'ultima risposta
e stabilire così la durata della connessione */
close (s);
/* Stampa un messaggio di fine. */
time (&timevar);
printf("Terminato %s porta %u, con %d messaggi, alle %s\n",
      hostname, ntohs(peeraddr_in.sin_port),
      reqcnt, ctime(&timevar));
}
```

Si modifichi il processo server per aggiungere indicazioni di tempo nei messaggi *dal servitore al cliente* e *viceversa* per temporizzare la durata delle operazioni

Primitive sospensive

Una primitiva sospensiva interrotta da un segnale deve essere riattivata dall'inizio

vedi schema seguente (errno == **EINTR** verifica tale evento)

```
for (;;)
{ int g, len = sizeof (from);
  g=accept (f, (struct sockaddr *)&from, &len);
  if (g < 0) { if (errno != EINTR) syslog(LOG_ERR, ..."p);
              continue; } /* necessità di ripetizione primitiva */
  ... /* altro codice in caso di successo */
}
```

STREAM socket e CONNESSIONI

Giocano più **modelli di interazione**

modello **attivo/passivo** per connessione

- una entità (attiva) richiede la relazione di coppia
- una entità (passiva) accetta la relazione

requisiti?

- Conoscenza reciproca
- Richiesta attiva deve precedere quella passiva

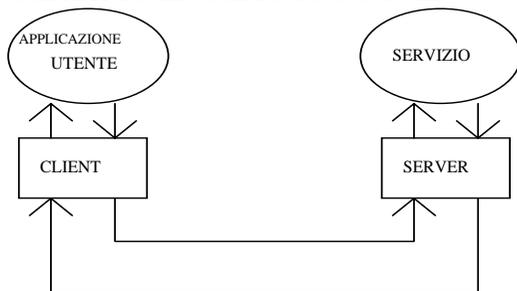
modello **cliente/servitore**

- una entità (cliente) richiede il servizio
- una entità (server) accetta il servizio e propone la risposta

È possibile?

- che un client richieda il servizio e non il server
- che il servizio si possa fornire a più clienti insieme
- che si instauri un canale tra cliente e servitore per una molteplicità di richieste di servizio

Modelli CLIENTE SERVITORE



FAIRNESS Problema generale di come rispondere alle richieste in modo giusto

Modelli di SERVIZIO

Per una classificazione dei servizi

- servizio **sequenziale** vs **concorrente**
- servizio **con** vs **senza connessione**
- servizio **con** vs **senza stato**

servizio sequenziale

- il servitore considera le richieste una alla volta
- ritardi nel servizio stesso ai clienti*

servizio concorrente

- il servitore serve più richieste insieme
- maggior complessità progettuale del server*
- NON è necessario il parallelismo nel server**

servizio senza connessione

- (usando UDP) costo basso, ma *si devono realizzare ordinamenti e reliability*

servizio con connessione

- (usando TCP) *costo superiore della realizzazione*

servizio senza stato (stateless server)

- il servitore considera le richieste e le dimentica appena fornite

- problemi nella gestione di richieste replicate*
- senza rieseguire il servizio*

servizio con stato (stateful server)

- il servitore tiene traccia dello stato di interazione dei servizi con i clienti

- maggior complessità progettuale del server*
- NON facile decidere lo stato in concorrenza**

Server concorrente/sequenziale

Distinguiamo dal punto di vista cliente

- **tempo di elaborazione (di servizio) di una richiesta T_S**
tempo impiegato dal server per una richiesta isolata
- **tempo di risposta osservato dal cliente T_R**

$$T_R = T_S + 2 T_C + T_Q ;$$

T_C tempo di comunicazione medio

T_Q tempo di accodamento medio

ritardo totale tra la spedizione della richiesta e l'arrivo della risposta dal server

Con **lunghe code di richieste**, il tempo di risposta può diventare anche molto maggiore del tempo di elaborazione della richiesta

Server sequenziale

una richiesta alla volta e accoda le altre richieste

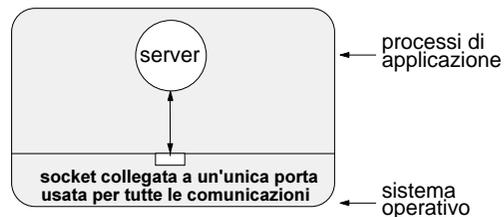
Tralasciando il tempo di comunicazione, se

N è la lunghezza della coda ==>

$$T_R \text{ (medio)} = (N/2+1) * T_S$$

Soluzione:

- limitare la lunghezza della coda
- le richieste a coda piena vengono rifiutate



Un server sequenziale può servire un traffico di **K** clienti con **R** richieste/secondo, se la richiesta mediamente ha un tempo di elaborazione minore di **$1/RK$**

Server concorrenti

Concorrenza può migliorare il **tempo di risposta**:

- se la risposta richiede un **tempo di attesa** significativo di I/O o di **sospensione** del servizio con possibilità di interleaving;
- se le richieste richiedono **tempi di elaborazione** molto variabili;
- se il server è eseguito in un **multiprocessore**, cioè con servizi in reale parallelismo

$$T_R = T_S + 2 T_C + T_Q + T_I + T_G ;$$

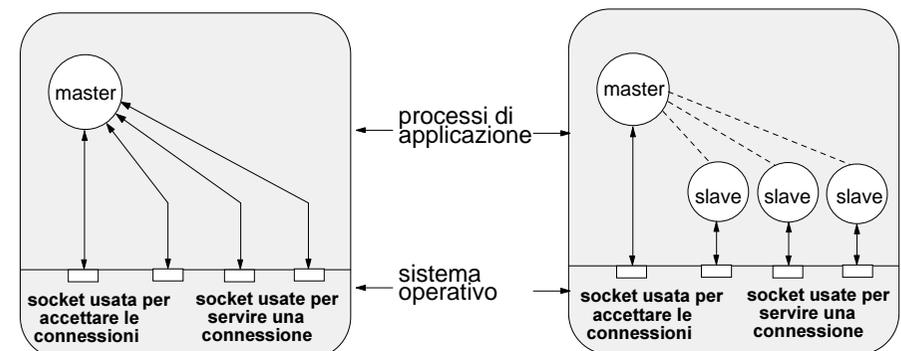
T_C tempo di comunicazione medio

T_Q tempo di accodamento medio (trascurabile)

T_I tempo di interleaving con altri servizi concorrenti

T_G tempo di generazione di un eventuale servitore

Sono possibili **scemi diversi di realizzazione** di un server di questo tipo

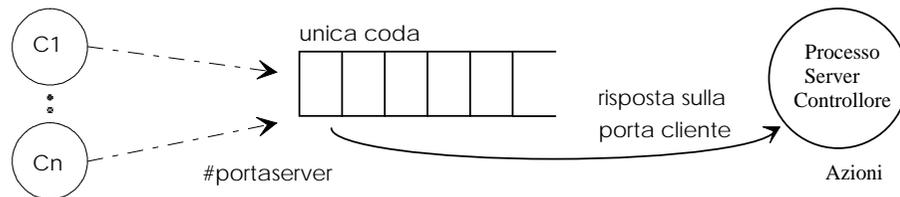


Caso 1: Server sequenziale

una richiesta alla volta (con connessione o meno) | tempi di risposta al servizio aumentano ==> **servizi brevi**

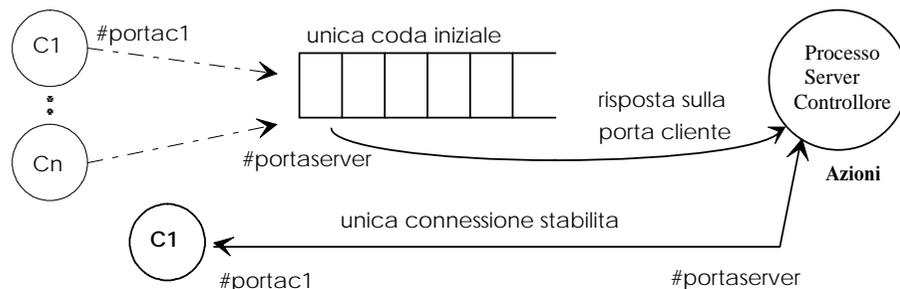
1a) server sequenziale senza connessione

servizi senza stato (che introduce ritardi) e non soggetti a guasti



1b) server sequenziale con connessione

*servizi con requisiti di reliability limitando lo stato (uso di TCP)
difficile realizzazione pratica
overhead delle azioni di controllo della connessione*



Caso 2: Server concorrente

più richieste alla volta

2a) SERVER PARALLELO:

uso di processi multipli, un master server
*generazione di **processi interni** per ogni servizio
massimo parallelismo per servizi non interferenti*

Si deve garantire che il costo di generazione del processo non ecceda il guadagno ottenuto

Soluzione con **processi creati in precedenza**

che consentano di essere velocemente smistati al servizio necessario

numero prefissato di processi iniziali e altri creati su necessità e mantenuti per un certo intervallo

2b) Un unico server, che gestisce la concorrenza

*servizi devono condividere lo stato
uso di asincronismi interni*

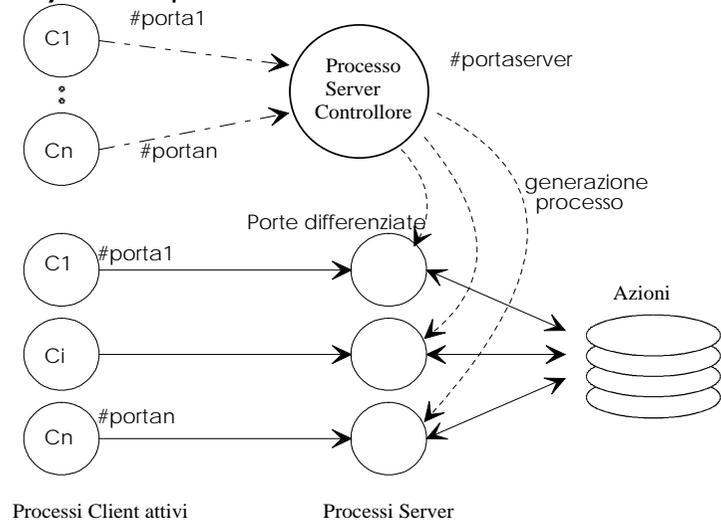
con connessione

più processi, generati dal server per i servizi contemporanei

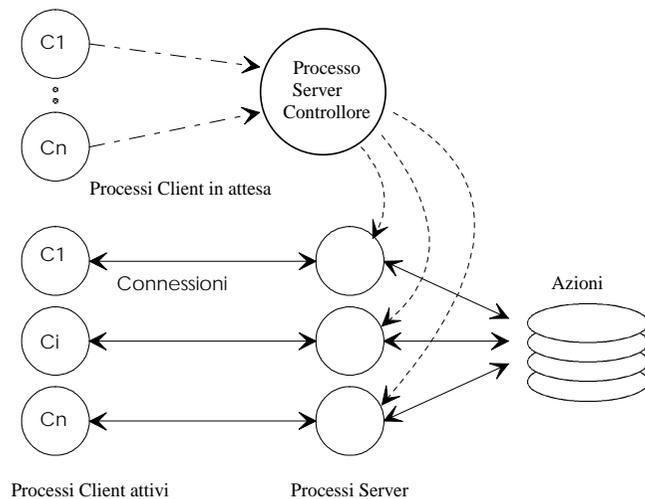
senza connessione

più processi ciascuno con una porta privata del cliente

2a) Server parallelo senza connessione



2a) Server parallelo con connessione



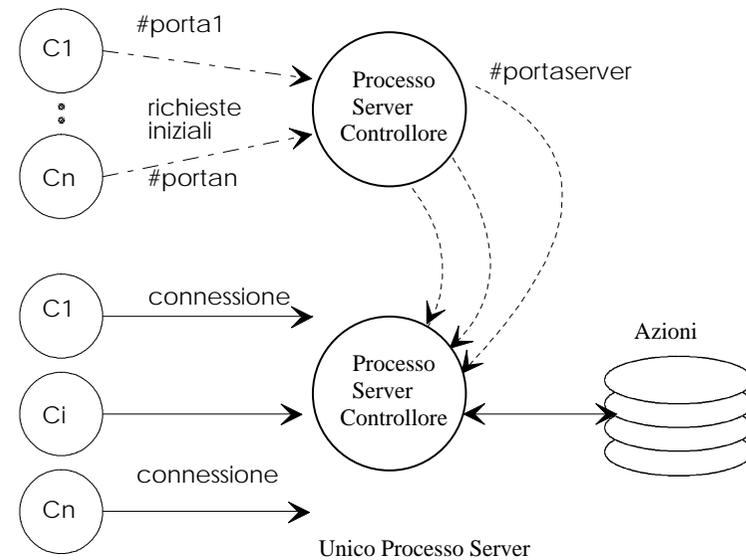
Un canale unico e un processo per ogni servizio

2b) **SERVER concorrente non parallelo**

in caso di connessione e stato (limitati)
realizzato con un unico processo che produce effetti di concorrenza

l'unico **processo master** deve considerare tutte le richieste ed servirle al meglio (mantenendo le connessioni)

Facciamo il caso di connessione (ma anche senza)



Processi Client attivi

Necessità di rispondere in modo veloce alle richieste successive e sollecitazioni

Ricezione eventi multipli

attesa contemporanea di eventi di I/O su più socket descriptor (o file)

legata alla gestione di più eventi all'interno di uno stesso processo

per ovviare ai problemi causati da **single operazioni bloccanti (lettura) o con attese di completamento**

PRIMITIVA select *select()*

blocca il processo in attesa di **almeno un** evento fra più eventi attesi possibili (range da **0** a **soglia** intera) anche per un predeterminato intervallo

attesa multipla globale sincrona o con durata massima (time-out) bloccante o meno

condizioni del tipo

- **evento di lettura: operazione possibile**
 - . in una socket sono presenti dati da leggere *recv()*
 - . in una socket si può accettare una chiamata *accept()*
 - . in un socket si è verificato un errore
- **evento di scrittura: operazione completata**
 - . in una socket la connessione è completata *connect()*
 - . in una socket si possono spedire altri dati con *send()*
 - . in una socket si è verificato un errore
- **condizioni eccezionali: errore o urgenza**
 - . arrivo di dati *out-of-band*,
 - . inutilizzabilità della socket, *close()* o *shutdown()*

```
#include <time.h>
```

```
int select (nfd, readfds, writefds, exceptfds, timeout)  
size_t nfd; /* numero di eventi attesi: limite superiore */  
int *readfds, *writefds, *exceptfds;  
const struct timeval *timeout;
```

La chiamata esamina gli eventi per i file descriptors specificati nelle maschere *readfds*, *writefds*, *exceptfds*. Queste maschere sono codificate con bit a 1

9	8	7	6	5	4	3	2	1	0	file descriptor
0	0	1	0	1	1	0	0	0	0	maschera

I bit della maschera corrispondono ai file descriptor

si esaminano solo i file descriptor il cui bit è ad 1

$fds[(f / BITS_PER_INT)] \& (1 \ll (f \% BITS_PER_INT))$

Al ritorno della chiamata le tre maschere sono modificate in relazione agli eventi per i corrispondenti file descriptor 1 se evento verificato, 0 altrimenti

azione bloccante fino al timeout

Se timeout NULL (0) == >

attesa indefinita di un evento (**sincrona** con il primo)

se punta ad una struttura con un tempo ==>

intervallo di **time-out**

alla scadenza, si ritorna 0 e si cancellano le maschere

Massimo intervallo di attesa 31 giorni

azione non bloccante

polling dei canali se timeout ad una struttura con tempo 0

Esempio di uso della **select()** e macro di manipolazione delle **relative maschere**

```
#include <stdio.h>
do_select(s)
    int s; /* socket descriptor */
{struct fd_set read_mask, write_mask;
/* dichiarazione delle maschere */
int nfd; int nfd;
for (;) {
/* azzera la maschera */
    FD_ZERO(&read_mask);
    FD_ZERO(&write_mask);
    FD_SET(s,&read_mask);
/* inserisce il file descriptor nella maschera */
    FD_SET(s,&write_mask);
    nfd=s+1;
    nfd=select (nfd,&read_mask,&write_mask,NULL,
                (struct timeval*)0);

    if (nfd==-1)
/* per select generali, potrebbe essere timeout*/
        {perror("select: condizione inattesa."); exit(1);}
    if (FD_ISSET(s,&read_mask)) do_read(s);
/*cerca il file descriptor nella maschera*/
    if (FD_ISSET(s,&write_mask)) do_write(s);
}
}
```

Esempio di Server Concorrente con connessioni

```
#define LEN 100
typedef struct { long len; char name[LEN]; } Request ;
typedef struct { long value; long errno; /* 0 successo */
                } Response ;
typedef int HANDLE;

int recv_request (HANDLE h, Request *req)
/* funzione di ricezione di una intera richiesta */
{ int r_bytes, n;
  int len = sizeof *req;

  for (r_bytes = 0; r_bytes < len; r_bytes += n)
    { n = recv (h, ((char *) req) + r_bytes, len - r_bytes, 0);
      if (n <= 0) return n; }
/* Decodifica len */
  req->len = ntohl (req->len);
  return r_bytes;
}

int send_response (HANDLE h, long value)
/* funzione di invio al cliente di una risposta */
{ Response res; size_t w_bytes;
  size_t len = sizeof res;
  res.errno = value == -1 ? htonl (errno) : 0;
  res.value = htonl (value);
  for (w_bytes = 0; w_bytes < len; w_bytes += n)
    { n = send (h, ((char *) &res) + w_bytes, len - w_bytes, 0);
      if (n <= 0) return n; }
  return w_bytes;
}
```

```

HANDLE create_server_endpoint (u_short port)
/* funzione di attesa di richieste di canale */
{ struct sockaddr_in addr; HANDLE h;
  h = socket (PF_INET, SOCK_STREAM, 0);
  memset ((void *) &addr, 0, sizeof addr);
  addr.sin_family = AF_INET;
  addr.sin_port = htons (port);
  addr.sin_addr.s_addr = INADDR_ANY;
  bind (h, (struct sockaddr *) &addr, sizeof addr);
  listen (h, 5);
  return h;
}

long action (Request *req);
/* azione qualunque di servizio */
{ ... }

long handle (HANDLE h) {
/* per ogni possibile evento da parte di un cliente
connesso, si esegue la funzione handle
in questa funzione si riceve la richiesta (lettura pronta)
si attua l'azione e
si invia la risposta
*/
struct Request req; long value;
if (recv_request (h, &req) <= 0) return 0;
  /* azione */
value = action (&req);
return send_response (h, value);
}

```

```

int main (int argc, char *argv[])
/* servitore concorrente con connessione */
{ u_short port = argc > 1 ? atoi(argv[1]) : 10000;
  /* Porta di listen per connessioni */
/* attesa di richieste di connessione */
  HANDLE listener = create_server_endpoint(port);
  HANDLE maxhp1 = listener + 1;
  /* numero corrente di possibili socket da verificare */
  fd_set read_hs, temp_hs;
  FD_ZERO(&read_hs); FD_ZERO(&temp_hs);
  FD_SET(listener, &read_hs); FD_SET(listener, &temp_hs);
  temp_hs = read_hs;

  for (;;)
  { HANDLE h;
    select (maxhp1, &temp_hs, 0, 0, 0);
    /* verifica delle richieste presenti */
    for (h = listener + 1; h < maxhp1; h++)
    /* richieste sulle connessioni */
    { if (FD_ISSET(h, &temp_hs))
      if (handle (h) == 0)
        /* Chiusura della connessione da parte del cliente */
        { FD_CLR(h, &read_hs); close(h); }
    }
    /* richieste di nuove connessioni */
    if (FD_ISSET(listener, &temp_hs)) {
      h = accept (listener, 0, 0);
      FD_SET(h, &read_hs);
      if (maxhp1 <= h) maxhp1 = h + 1; }
    temp_hs = read_hs;
  }
}

```

Servitore Multiplo (multiservizio)

Possibilità di un unico servitore per più servizi

Vantaggi: un unico collettore attivo che si incarica di smistare le richieste

Il **servitore multiplo** può

- **portare a termine** completamente i servizi per richiesta
- **incaricare** altri processi del servizio (specie in caso di connessione e stato) e **tornare al servizio** di altre richieste

Unico processo master per molti servizi

riconosce i servizi e attiva il servizio stesso

Un master può ricevere più richieste

- servire direttamente la richiesta
- generare un processo a parte

Vedi BSD UNIX inetd (/etc/services)

inetd

alcuni servizi sono svolti in modo diretto (interno)

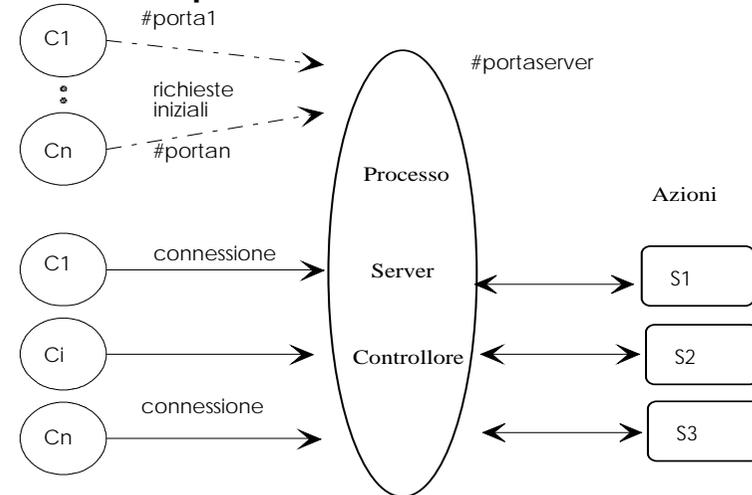
altri sono invece delegati a processi creati su richiesta

Problemi:

Il server può diventare il **collo di bottiglia** del sistema

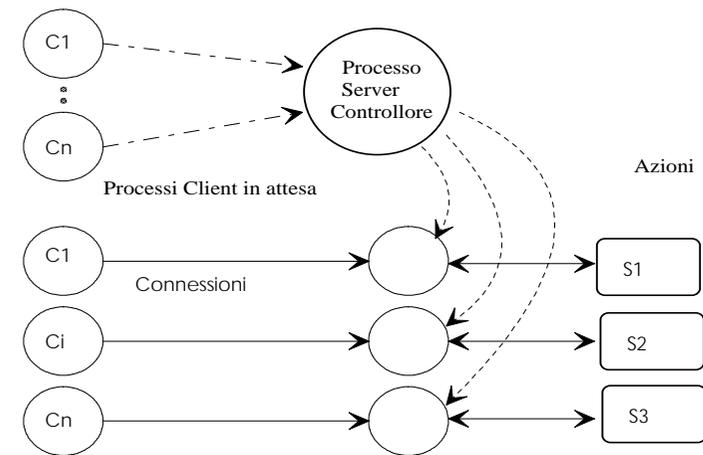
Necessità di decisioni rapide e leggere

Server multiplo con connessione



Processi Client attivi

Server multiplo con processi interni



Processi Client attivi

Processi Server

Esempio di file di configurazione del demone INETD

```
# @(#)inetd.conf 1.24 92/04/14 SMI
# Configuration file for inetd(8). See inetd.conf(5).
#
# To re-configure the running inetd process, edit this file, then
# send the inetd process a SIGHUP.

# Internet services syntax:
# <service_name> <socket_type> <proto> <flags> <user>
#           <server_pathname> <args>

#
# Ftp and telnet are standard Internet services.
ftp      stream tcp nowait root    /usr/etc/in.ftpd  in.ftpd
telnet  stream tcp nowait root    /usr/etc/in.telnetd in.telnetd
#
# Tnamed serves the obsolete IEN-116 name server protocol.
#
# Shell, login, exec, comsat and talk are BSD protocols.
shell   stream tcp nowait root    /usr/etc/in.rshd  in.rshd
login  stream tcp nowait root    /usr/etc/in.rlogind in.rlogind
exec   stream tcp nowait root    /usr/etc/in.rexecd in.rexecd
talk   dgram  udp wait  root    /usr/etc/in.talkd in.talkd
#
# Finger, systat and netstat give out user information which may be
# valuable to potential "system crackers." Many sites choose to disable
# some or all of these services to improve security.
finger stream tcp nowait nobody /usr/etc/in.fingerd in.fingerd
#
#systat stream tcp nowait root /usr/bin/ps      ps -auwwx
#netstat stream tcp nowait root /usr/ucb/netstat netstat -f inet
#
# Time service is used for clock synchronization.
time    stream tcp nowait root internal
timedgram udp waitrootinternal
#
```

```
# Echo, discard, daytime, and chargen are used primarily for testing.
echo    stream tcp nowait root internal
echo    dgram  udp wait  root internal
discard stream tcp nowait root internal
discard dgram  udp wait  root internal
daytime stream tcp nowait root internal
daytime dgram  udp wait  root internal
#
# RPC services syntax:
# <rpc_prog>/<vers> <socket_type> rpc/<proto> <flags> <user>
#           <pathname> <args>

#
# Rerexecution usually obtained in other ways.
#rexrd/1  stream rpc/tcp wait root    /usr/etc/rpc.rexd  rpc.rexd
#
# Rquotad serves UFS disk quotas to NFS clients.
rquotad/1 dgram  rpc/udp wait root    /usr/etc/rpc.rquotad rpc.rquotad
#
# Rstatd is used by programs such as perfmeter.
rstatd/2-4 dgram  rpc/udp wait root /usr/etc/rpc.rstatd  rpc.rstatd
#
# The rusers service gives out user information.
rusersd/1-2 dgram  rpc/udp wait root /usr/etc/rpc.rusersd  rpc.rusersd
#
# The spray server is used primarily for testing.
sprayd/1    dgram  rpc/udp wait root /usr/etc/rpc.sprayd  rpc.sprayd
#
# The rwall server lets anyone on the network bother everyone on your machine.
walld/1     dgram  rpc/udp wait root /usr/etc/rpc.rwalld  rpc.rwalld
#
# rpc.cmsd is a data base daemon which manages calendar data backed
# by files in /usr/spool/calendar
100068/2-3 dgram  rpc/udp wait root /usr/etc/rpc.cmsd  rpc.cmsd

# Sun ToolTalk Database Server
100083/1   stream  rpc/tcp  wait root /usr/etc/rpc.ttdbserverd
                                                    rpc.ttdbserverd
```

Dalla parte del cliente

anche concorrenza dalla parte del cliente
per ottenere i vantaggi di servizi multipli

a) soluzione parallela

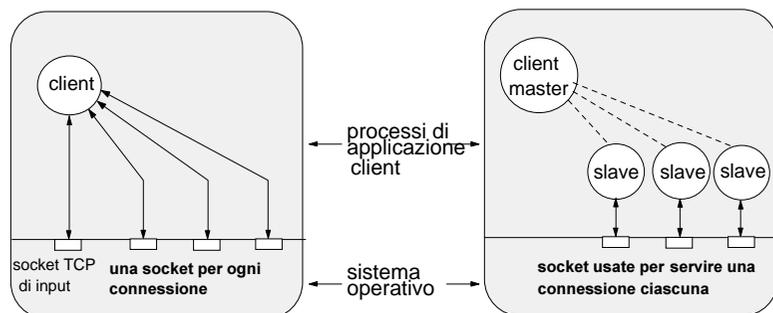
possibilità di generare più processi (slave) che gestiscono ciascuno una diversa interazione con un server
possibilità di interagire con più server contemporaneamente (multicast)

*come interagire con il server in modo appropriato?
condivisione di socket?*

b) soluzione concorrente

possibilità che il cliente unico gestisca più interazioni
necessità di asincronismo

uso di select e politiche di servizio opportune



Gestione opzioni per Stream Socket

funzioni primitive per configurare socket

getsockopt() / setsockopt()

leggere e fissare le modalità di utilizzo delle socket
(tipicamente il valore del campo vale o 1 o 0)

```
#include <sys/types.h>
```

```
#include <sys/socket.h>
```

```
char optval = 1;
```

```
int getsockopt (s, level, optname, optval, optlen)
```

```
int s, level, optname, *optlen;
```

```
int setsockopt (s, level, optname, &optval, optlen)
```

```
int optlen;
```

s == socket descriptor legato alla socket

level == livello di protocollo per socket SOL_SOCKET

optname == nome dell'opzione

optval == puntatore ad un'area di memoria per valore

optlen == lunghezza (o puntatore) quarto argomento

Opzioni	Descrizione
SO_DEBUG	abilita il debugging (valore diverso da zero)
SO_REUSEADDR	riuso dell'indirizzo locale
SO_DONTROUTE	abilita il routing dei messaggi uscenti
SO_LINGER	ritarda la chiusura per messaggi pendenti
SO_BROADCAST	abilita la trasmissione broadcast
SO_OOBINLINE	messaggi prioritari pari a quelli ordinari
SO_SNDBUF	setta dimensioni dell'output buffer
SO_RCVBUF	setta dimensioni dell'input buffer
SO_SNDLOWAT	setta limite inferiore di controllo di flusso out
SO_PCVALLOWAT	limite inferiore di controllo di flusso in ingresso
SO_SNDTIMEO	setta il timeout dell'output
SO_RCVTIMEO	setta il timeout dell'input
SO_USELOOPBACK	abilita network bypass
SO_PROTOCOL	setta tipo di protocollo

Riutilizzo del socket address (STREAM)

Opzione **SO_REUSEADDR** modifica bind()

Il sistema tende a non ammettere più di un utilizzo di un indirizzo locale

con l'opzione, si convalida l'indirizzo di una socket **senza controllare la unicità di associazione**

Socket con wildcard address per una porta in uso

Controllo al momento della connessione

Protocollo	Coda di rx	Coda di tx	Indirizzo locale	Indirizzo remoto	Stato della connessione
TCP	0	0	*.2000	*.*	ASCOLTO

netstat segnala demone sulla porta 2000 all'indirizzo Internet * (wildcard address)

Un processo client richiede una connessione

Proto-collo	Coda di rx	Coda di tx	Indirizzo locale	Indirizzo remoto	Stato cns
TCP	0	0	137.204.57.33.2000	137.204.57.32.4000	STABILITA
TCP	0	0	*.2000	*.*	ASCOLTO

Il processo demone originale continua ad attendere sulla socket legata alla porta 2000 ed al wildcard address

Se termina, il riavvio necessita **SO_REUSEADDR**

Senza, il processo tenta il collegamento alla porta 2000 con wildcard address ==> bind() errore ==>

già presente una connessione per lo stesso socket address

```
int optval=1;
```

```
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval)); bind(s,&sin,sizeof(sin));
```

Controllo periodico della connessione

Il protocollo di trasporto può inviare messaggi di controllo periodici per analizzarne lo stato di una connessione (**SO_KEEPALIVE**)

Se problemi ==>

connessione è considerata abbattuta

i processi avvertiti da un **SIGPIPE** *chi invia dati*
da **end-of-file** *chi li riceve*

Verifica ogni 45 secondi e dopo 6 minuti di tentativi
opzione in dominio Internet tipo boolean

Utilizzo di servizi di routing non standard

opzione **SO_DONTROUTE** in dominio Internet

i messaggi scavalcano i servizi di routing e sono diretti all'interfaccia di rete per trovare il cammino

Dimensioni buffer di trasmissione/ricezione

Opzioni **SO_SNDBUF** e **SO_RCVBUF**

Aumento della dimensione buffer di trasmissione ==>

invio messaggi più grandi senza attesa

massima dimensione possibile 65535 byte

```
int result; int buffersize=10000;
```

```
result=setsockopt(s, SOL_SOCKET, SO_RCVBUF, &buffersize, sizeof(buffersize));
```

Azioni eseguite alla chiusura di socket

Con opzione **SO_LINGER** si influenzano i messaggi in caso di **close**

struttura di dati *linger* in `/usr/include/sys/socket.h`:
`struct linger { int l_onoff; int l_linger; /* attesa in sec */ }`

l_onoff	l_linger	Graceful/Hard Close	Chiusura Con/ Senza attesa
0	don't care	G	Senza
1	0	H	Senza
1	valore > 0	G	Con

chiusura graceful della connessione

l_onoff ad 1 e **l_linger** **valore positivo**

la chiamata `close()` avviene dopo la trasmissione di tutti i dati nel buffer

chiusura hard della connessione

l_linger a 0

ogni dato non inviato è perso

valore di default **l_onoff** 0

disconnessione graceful

L'opzione non influenza lo shutdown()

IPC fra processi locali

dominio Internet processi locali

opzione **SO_LOOPBACK**

IP usa loopback locale per i dati

Ricezione messaggi out-of-band in linea

Opzione in `recv()` per ricevere messaggi out-of-band

Opzione **SO_OOBLINE**

dato out-of-band nello stream di normale ricezione

non sono necessarie letture con flag

Analisi del tipo della socket

Opzione **SO_TYPE** (`getsockopt`)

si può risalire al tipo di socket

Analisi degli errori

Opzione **SO_ERROR** (`getsockopt`)

variabile globale **so_error**

leggere e cancellare errore su una socket

Broadcasting

Opzione **SO_BROADCAST**

messaggi broadcast

spedizione in Internet

datagrammi broadcast attraverso una socket

Primitive messaggi modo Scatter/Gather

Primitive `writv()` e `readv()` per *scatter/gather I/O*

scrittura/lettura da buffer non contigui

Incapsulamento delle informazioni a livello applicativo

senza delegarlo ai protocolli di trasporto

le primitive richiedono i dati già incapsulati

operazioni di compattazione (scrittura)

frammentazione nei buffer (lettura)

Struttura dati utilizzata per contenere i buffer non contigui presente nel file <sys/uio.h>:

```
struct iovec {
    caddr_t iov_base;    /* indirizzo base di un buffer */
    int iov_len;        /* dimensione del buffer */
}
```

Le due primitive:

```
#include <sys/types.h>
#include <sys/uio.h>
```

```
int readv (s, iov, iovcount)
int writev (s, iov, iovcount)
    int s;
    struct iovec iov[];
    int iovcount;
```

s == socket descriptor

iov == puntatore al primo dei buffer di tipo iovec

iovcount == numero di buffer

ritorno il numero di byte effettivamente letti o scritti

L'effetto di queste chiamate è quello di leggere o scrivere questi buffer **uno per volta** fino al termine

Esecuzione sequenziale e atomica

Esempio:

trasmissione di byte preceduti da una testata di informazioni

classiche chiamate read() e write()

```
struct info header;
char buffer[200];
< imposta le informazioni in header >
< trasmissione unica di un messaggio alla volta >
if (write(s,header,sizeof(header))!=sizeof(header))
    return(-1)
if (write(s,buffer,200)!=200) return(-1)
.....
```

uso di funzioni di scatter/gather I/O

```
struct iovec iov[2];
struct info header;
char buffer[200];
< imposta le informazioni in header >
< senza caricare i dati in un buffer ad-hoc >
iov[0].iov_base = (char *)&header;
iov[0].iov_len = sizeof(header);
iov[1].iov_base = buffer;
iov[1].iov_len = 200;
if (writev (s,&iov[0],2) != sizeof(header)+200) return(-1)
.....
```

Scambio di dati con modalità asincrona

A default => modello sincrono delle socket

Socket asincrone

segnale **SIGIO** segnala ogni cambiamento di stato della socket (ad esempio per l'arrivo di dati)

gestore di **SIGIO** robusto per situazioni inaspettate

SIGIO ignorato dai processi che non hanno definito un gestore

select() per determinare quale condizione si sia verificata

SIGIO

socket asincrona

Attributo FIOASYNC con primitiva *ioctl()*

```
#include <sys/ioctl.h>
```

```
int ioctl (int filedesc, int request, ... /* args */)
    filedesc == file descriptor
```

```
    request == tipo di attributo da assegnare
```

```
    poi == valori da assegnare all'attributo
```

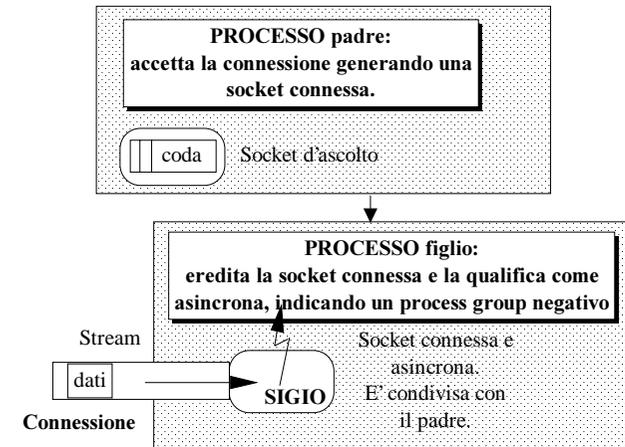
parametro *process group* del processo alla socket asincrona. Primitiva *ioctl()* attributo SIOCSPGRP

Consegna di SIGIO:

```
ioctl (filedesc, SIOCSPGRP, &flag) flag
```

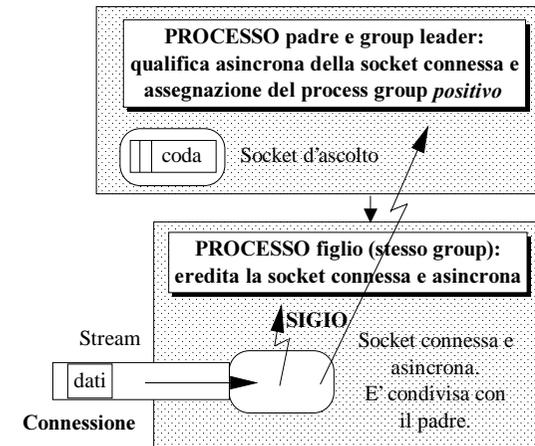
valore **negativo** ==> segnale per il processo con pid uguale al process group

valore **positivo** ==> segnale arriva a tutti i processi del process group



Valore **negativo**

Consegna SIGIO al solo processo del gruppo



Valore **positivo**

Consegna di SIGIO a tutti i processi del gruppo

Due esempi di socket asincrone con consegna del segnale al singolo processo ed al gruppo di processi:

```
int ls;      /* socket d'ascolto */
int flag=1;
/* valore per FIOASYNC per socket asincrona */
int handler; /* gestore delle I/O sulle socket */

signal(SIGIO,handler);
if (ioctl (ls,FIOASYNC,&flag)== -1) {
    perror("non posso rendere asincrona la socket");
    exit(1);
}
flag= -getpid(); /* identificatore di processo negativo */
if (ioctl (ls,SIOCSGRP,&flag) == -1)
{ perror("non si assegna il process group alla socket");
  exit(1); }
```

Le opzioni sono poco standard.

Per Linux:

```
int flag, ret;

#ifdef __linux__
flag = fcntl (s, F_GETFL);
if (fcntl (s, F_SETFL, flag | FASYNC ) == -1)
    { perror("fcntl failed");
      exit(-1);}
flag = -getpid();
if (fcntl (s, F_SETOWN, flag) == -1)
    { perror("fcntl F_SETOWN"); exit (-2);}
#endif
```

Nonblocking I/O

Primitive bloccanti

anche modificata con primitiva ioctl()

parametro FIONBIO

valore 0 / 1
modalità **bloccante** / **non bloccante**

chiamate modificate

- **accept()**
restituisce errore di tipo EWOULDBLOCK
- **connect()**
condizione d'errore di tipo EINPROGRESS
- **recv() e read()**
condizione d'errore di tipo EWOULDBLOCK
- **send() e write()**
condizione d'errore EWOULDBLOCK

Esempio di assegnazione dell'attributo non bloccante

```
#include <sys/ioctl.h>
int s;      /* socket descriptor */
int arg=1; /* valore per la qualifica non blocking */
ioctl(s,FIONBIO,&arg);
ioctl(s,FIOASYNC,&arg);
```

Funzionamento **non bloccante** delle primitive anche mediante chiamata *fcntl()*

FCNTL () dedicata al controllo dei file aperti

FCNTL

```
#include <fcntl.h>
int fcntl (fileds, cmd, .../* argomenti */)
    int fileds; /* descrittore del file */
    int cmd;
    /* argomenti */

if (fcntl (descr, F_SETFL, FNDELAY) < 0)
{ perror("non si riesce a rendere asincrona la socket");
  exit(1); }
```

anche O_NDELAY

I comandi hanno significati diversi in

System V

- O_NDELAY

le chiamate read(), recv(), send(), write() senza successo immediato ritornano un valore 0.

Possibile problema: equivale alla condizione di end-of-file sulle socket

- O_NONBLOCK (POSIX.1 standard, System V vers.4)

Le chiamate senza successo immediato ritornano -1 e la condizione d'errore EAGAIN

BSD

- FNDELAY, O_NONBLOCK

le chiamate read(), recv(), send(), write() senza successo immediato ritornano con valore -1 e la condizione d'errore EWOULDBLOCK

Nonblocking I/O

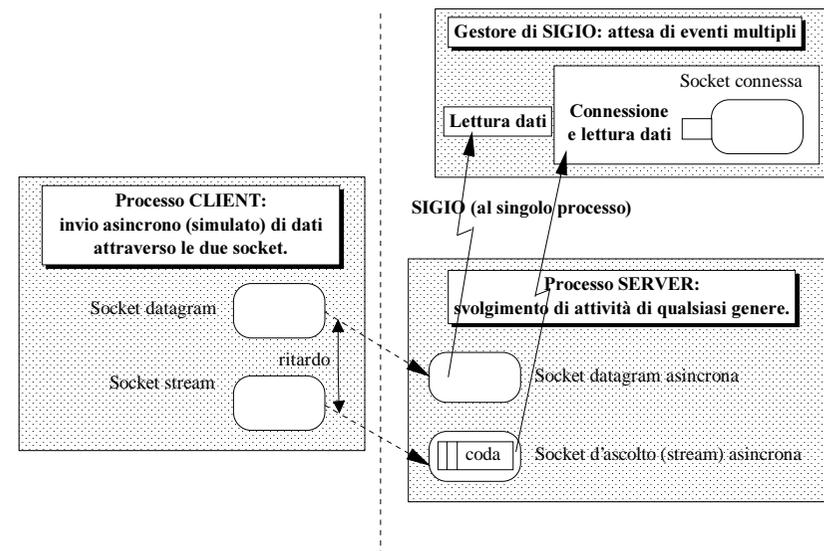
Primitive bloccanti a default

Per asincronismo

uso primitiva ioctl() con FIONBIO a valore 1
recvfrom(), *recv()*, *read()*, *sendto()*, *send()* e *write()*
condizione d'errore (-1) di tipo EWOULDBLOCK

Anche con uso di fcntl()

flags O_NDELAY oppure O_NONBLOCK



Processo server con socket asincrone

(manca la gestione del segnale SIGIO)

- Assegnazione di un **gestore per SIGIO**
- Assegnazione della **address family**
- Assegnazione di un **indirizzo Internet locale wildcard**
- **Preparazione della datagram socket**
 - Assegnazione del numero di porta
 - Creazione della datagram socket
 - Collegamento al socket address
 - Qualifica di **asincronismo**
- **Preparazione della stream socket**
 - Assegnazione del numero di porta
 - Creazione della stream socket
 - Collegamento al socket address
 - Qualifica di **asincronismo**
 - Creazione della coda d'ascolto

Gestore di SIGIO

- Notifica all'utente dell'arrivo del segnale
- Prepara la **maschera** con i socket descriptor delle datagram e stream socket
- Esegue una **select** con time-out pari a zero
- A secondo della maschera, sceglie **datagram e/o stream** per leggere e visualizzare i dati

/ SERVER */*

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <signal.h>
#include <stdio.h>
#include <netdb.h>
int ds; /* datagram socket descriptor */
int ss; /* streams socket descriptor */
struct hostent *hp; /* puntatore all' host remoto */
struct sockaddr_in myaddr; /* socket address locale */
struct sockaddr_in peeraddr_in; /* socket address remoto */

int io_handler(); /* gestore del segnale SIGIO */

main(argc, argv)
int argc; char *argv[];
{ struct sigvec vec;
  /* Assegna la struttura per la notifica dell'evento */
  vec.sv_handler = (int *) io_handler;
  vec.sv_mask = 0;
  vec.sv_flags = 0;
  if ( sigvector(SIGIO, &vec, (struct sigvec *) 0) == -1)
      perror(" sigvector(SIGIO)");
  myaddr.sin_family = AF_INET;
  myaddr.sin_addr.s_addr = INADDR_ANY;
  myaddr.sin_port = 22373;
  ds = socket(myaddr.sin_family, SOCK_DGRAM, 0);
  if (ds == -1) { perror(argv[0]);
                printf("%s: Non datagram socket\n", argv[0]);
                exit(1); }
  /* Attribuisce la qualifica di asincrona alla socket */
  set_up_async(ds);
  myaddr.sin_port = 22374;
  ss = socket(myaddr.sin_family, SOCK_STREAM, 0);
  if (ss == -1) { perror(argv[0]);
```

```

        printf("%s: Non stream socket\n", argv[0]);
        exit(1); }
set_up_async (ss);
if (listen(ss, 5) == -1 ) {perror(argv[0]);
        printf("%s: impossibile ascolto \n", argv[0]);
        exit(1); }
printf(" Il Server entra in un loop infinito.\n\n");
for ( ; ; ) { } /* questo loop simula qualunque esecuzione*/
}
/*
* SET_UP_ASYNC(S) la socket s diventa asincrona
*/
set_up_async(s)
int s;
{ int flag = 1;
if (bind (s, &myaddr, sizeof(myaddr)) == -1) {
        perror(" impossibile collegare indirizzo a socket\n");
        exit(1);}

/* Assegna lo stato di asincrona alla socket */
if (ioctl (s, FIOASYNC, &flag) == -1) {
        perror(" non posso qualificare asincrona la socket ");
        exit(1); }

/* Il process group viene posto negativo per la consegna del
segnale al solo processo */
flag = -getpid();
if (ioctl (s, SIOCSPGRP, &flag) == -1)
        { perror("can't get the process group."); }
}

```

```

/*
* IO_HANDLER() gestore del segnale SIGIO
*/
#include <sys/param.h>
#define BPI 32 /* bit per intero */
#define DONT_CARE (char *) 0
#define BUFLLEN 100
io_handler ()
{ struct fd_mask { u_long fds_bits[NOFILE/BPI+1];};
/* NOFILE è il massimo numero di descrittori per processo */
struct fd_mask readmask;
int numfds, count, s;
char buf[BUFLLEN];
struct timeval {
        unsigned long tv_sec; /* secondi */
        long tv_usec; /* microsecondi */
} timeout;

memset (buf, 0, BUFLLEN); printf(" SIGIO ricevuto!\n\n");
FD_ZERO(&readmask);
FD_SET(ds, &readmask);
FD_SET(ss, &readmask);
timeout.tv_sec = 0; timeout.tv_usec = 0;

if ((numfds = select (ss + 1, &readmask, DONT_CARE,
        DONT_CARE, &timeout) < 0)
        { perror("select fallita "); exit(1); }
if (numfds == 0) { printf("condizione inattesa.\n");
        exit(1); }

/* si trattano i possibili casi in sequenza */
if (FD_ISSET(ds, &readmask)) {
        count = recvfrom (ds, buf, BUFLLEN, 0, &peeraddr_in,
                sizeof(struct sockaddr_in ));
        buf[count] = '\0'; printf(" ricevuto datagram: %s\n\n", buf);}
}

```

```

if (FD_ISSET(ss, &readmask)) {
    s=accept (ss, &peeraddr_in, sizeof(struct sockaddr_in ));
    if (s == -1 ) { perror(" accettazione fallita "); exit(1); }
    printf(" accettazione di richiesta di connessione\n");
    count = recv(s, buf, BUFLen, 0);
    /* la ricezione è bloccante, anche se per correttezza
    dovrebbe non esserlo in un server come questo */
    if (count == -1 ) {perror(" errore di ricezione" );
        exit(1);}
    buf[count] = '\0';
    /* i dati si devono stampare come stringa */
    printf(" ricevuti dalla stream socket i dati: %s\n\n", buf);
    printf(" Server terminato!\n");
    exit(0); }
}

```

Processo client

- *Lettura degli argomenti di chiamata*
- *Assegnazione della address family*
- *Acquisizione dell'indirizzo Internet host remoto*
- *Preparazione della **datagram socket***
 - Creazione della datagram socket
 - Assegnazione del numero di porta per il servizio remoto
- ***Invio di messaggio** attraverso la datagram socket*
- ***Attesa di 5 secondi***
- *Preparazione della **stream socket***
 - Assegnazione numero di porta del servizio remoto
 - Creazione della stream socket
 - Richiesta di connessione alla stream socket remota
- ***Invio di dati** attraverso la stream socket*

```

/* CLIENT */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
int ds; int ss; struct hostent *hp; struct servent *sp;
struct sockaddr_in myaddr_in; /* socket address locale */
struct sockaddr_in peeraddr_in; /* socket address remoto */

main(argc, argv)
int argc; char *argv[];
{ int addrlen;
if (argc != 4) { fprintf(stderr,"Usage: %s <host remoto>\
<dati datagram> <dati stream> \n\n", argv[0]);
    exit(1); }
peeraddr_in.sin_family = AF_INET;

```

```

hp = gethostbyname (argv[1]);
if (hp == NULL) { fprintf(stderr, "%s: %s non in /etc/hosts\n",
                        argv[0], argv[1]); exit(1); }
peeraddr_in.sin_addr.s_addr =
    ((struct in_addr *) (hp->h_addr))->s_addr;
ds = socket (AF_INET, SOCK_DGRAM, 0);
if (ds == -1) { perror(argv[0]);
              fprintf(stderr, "%s: no datagram socket\n", argv[0]);
              exit(1); }
peeraddr_in.sin_port = 22373;
if (sendto (ds, argv[2], strlen(argv[2]), 0, &peeraddr_in,
           sizeof(struct sockaddr_in )) == -1) {
    fprintf(stderr, "%s: invio Datagram fallito. ", argv[0]);
    perror(argv[0]); exit(1); }
printf(" %s : Messaggio Datagram inviato. \n", argv[0]);

sleep(5); /* attesa per simulare asincronismo del client*/

ss = socket (AF_INET, SOCK_STREAM, 0);
if (ss == -1) { perror(argv[0]);
              fprintf(stderr, "%s: No stream socket\n", argv[0]);
              exit(1); }
peeraddr_in.sin_port = 22374;
if (connect (ss, &peeraddr_in, sizeof(struct sockaddr_in )) == -1)
{ perror(argv[0]);
  fprintf(stderr, "%s: impossibile connettersi\n", argv[0]);
  exit(1); }
/* consegna del segnale SIGIO al server */
if (send (ss, argv[3], strlen(argv[3]), 0) == -1) {
    fprintf(stderr, "%s: invio Stream fallito. ", argv[0]);
    perror(argv[0]); exit(1); }
printf("%s : Dati stream inviati. Fine Cliente\n", argv[0]);

```

Invio/ricezione dati out-of-band (STREAM)

In caso di connessione, problema di informazioni urgenti
segnalazione out-of-band

Uso di un canale *logicamente indipendente* dal
 canale normale utilizzato dalla connessione
 TCP un *solo canale* di segnalazione out-of-band
 recapitati messaggi di un *solo byte*

***I dati out-of-band usano un'area indipendente da quella
 utilizzata per i dati normali (1 solo byte)***

Il messaggio out-of-band è in **sequenza** nello **stream**
consegnato al destinatario **indipendentemente** dai dati

DEFAULT

**Uso di send() e recv() con opzione MSG_OOB
 flussi distinti a livello utente**

send() con MSG_OOB anche bloccante
recv() con MSG_OOB non bloccante ricezione **a polling**
 errore **EINVAL** se il dato non disponibile
 anche uso di select per ottenere il valore

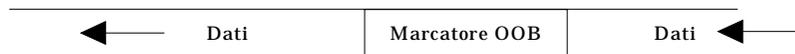
Senza dovere fare polling, si può anche segnalare un gestore
 dei dati urgenti:

Due metodi di segnalazione

- 1 se abilitato **SIGURG** (ioctl con SIOCSPGRP) si invoca il **gestore del segnale** all'arrivo dei dati
- 2 se il processo è bloccato su una select(), la **select esce** con la maschera degli eventi eccezionali

Trattamento Dati Urgenti

Esiste un *marcatore logico* nello stream per indicare il punto legato alla *condizione eccezionale*



Esempio

```
#include <signal.h>
int onurg(); /* gestore dei dati out-of-band */
int pid,s;
signal (SIGURG,onurg);
/* associa il gestore dell'evento al segnale SIGURG */

pid=-getpid();
if (ioctl (s,SIOCSPGRP,&pid)<0) {
    perror("ioctl(SIOCSPGRP)");
    exit(1); }
/* notifica il segnale solo al processo in questione */
.....
```

Forzatura e Recupero dei dati in linea

Opzione **SO_OOBINLINE** (setsockopt) forza un solo flusso anche a livello utente

il dato out-of-band è letto con normale recv()

Non si deve oltrepassare il marcatore

recv restituisce la distanza tra l'attuale puntatore dello stream di dati al puntatore del byte out-of-band

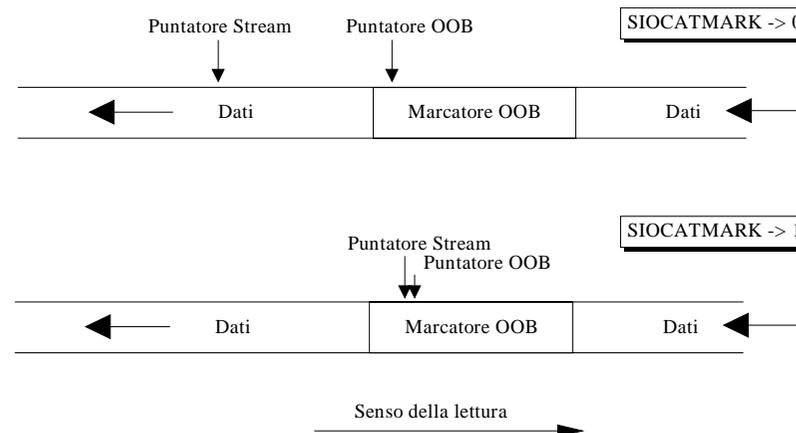
recv() successive non possono più leggerlo

Uso di ioctl() con opzione SIOCATMARK

Il puntatore del byte out-of-band viene raggiunto dal puntatore di lettura nel buffer di ricezione

Se restituisce 0 le prossime letture portano a dati precedenti la segnalazione out-of-band

Se 1 il puntatore ad out-of-band è stato raggiunto



Segnalazione out-of-band

per ignorare tutti i dati presenti nello stream di byte fino al puntatore del byte out-of-band

Il processo può svuotare il buffer di ricezione fino al raggiungimento di tale puntatore

Esempio di svuotamento del flusso

```
onurg()
{ int atmark; char mark; char flush[100];

while(1) {
/* ciclo di attesa del raggiungimento del puntatore del
byte out-of-band */

if (ioctl (s,SIOCATMARK,&atmark) < 0) {
/* s socket descriptor */
perror("ioctl(SIOCATMARK)");
return;
}
if (atmark) break;
/* se è 1 esce: ha raggiunto il byte out-of-band */

if (read (s, flush, sizeof(flush))<=0) return;
/*condizione d'errore sulla lettura: si ignora la lettura */

}
recv (s,&mark,1,MSG_OOB);
/* riceve il byte out-of-band per decodificarlo. */
printf("ricevuto messaggio OOB: %c\n",mark);
return;
}
```

Gestione *commit*

Scambio di una serie di dati

Alla ricezione si elabora solo al mark out-of-band

Gestione avanzata Datagram Socket

Broadcasting

setsockopt()

opzione SO_BROADCAST

la specifica di broadcast permette di indicare nel socket address un indirizzo Internet che consenta un indirizzo di multicast

Problemi di sovraccarico della rete

Socket address di default

Non si deve specificare l'indirizzo della socket remota *sendto()* e *sendmsg()* producono errore se chiamate con un indirizzo destinatario diverso da zero
Uso di *recv()* o *read()*

Specifica in *connect()* con due indirizzi speciali

INADDR_ANY

socket connessa all'interfaccia per comunicazioni di tipo loopback (locale)

INADDR_BROADCAST

solo con i nodi presenti sulla rete locale

L'effetto della *connect()* consiste in una semplice registrazione dell'indirizzo di socket
si può ripetere più volte per variare il socket address di default

Esempio di socket datagram con broadcast

```
#define MYPORT 24000
int on =1;
struct sockaddr sin;
/* socket datagramma */
s=socket(AF_INET, SOCK_DGRAM, 0);
/* The socket is marked broadcast */
setsockopt(s, SOL_SOCKET, SO_BROADCAST,
           &on, sizeof (on));

/* numero di porta per la socket */
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = IN_ADDR_ANY;
sin.sin_port = MYPORT;
bind(s, (struct sockaddr*)&sin, sizeof (sin));
```

In caso di invio la destinazione deve essere opportuna:

INADDR_BROADCAST (definito in <netinet/in.h >)

```
#define INADDR_BROADCAST (u_long)0xffffffff
```

In caso di rete con submask bisogna tenere conto dell'eventuale ridefinizione del broadcast

Il valore si verifica usando la **ifconfig** del device di rete

se ridefinito come "137.204.57.255"

```
broadaddr_in.sin_family = AF_INET;
broadaddr_in.sin_port = htons( PEER_PORT);
broadaddr_in.sin_addr.s_addr= inet_addr("137.204.57.255");
if ((sendto(s,mess,10,0,&broadaddr_in,sizeof(broadaddr_in)))<0)
    perror("ERRORE GRAVE. No broadcast");
printf("Send broadcast avvenuta\n");
```

Primitive messaggi: modo Scatter/Gather

Scambiare dati provenienti da buffer non contigui

Oltre a `readv()` e `writew()`
anche `recvmsg()` e `sendmsg()`

```
#include sys/socket.h
struct msghdr {
    caddr_t msg_name;
    /* indirizzo socket remota (opzionale con socket connessa)
    */
    int msg_namelen; /* dimensione indirizzo */
    struct iovec * msg_iov;
    /* vettore di buffer non contigui */
    int msg_iovlen;
    /* numero di elementi nel vettore msg_iov */
    caddr_t msg_accrights; /* diritti d'accesso */
    int msg_accrightlen; /*lunghezza campo */
}
```

Due nuove primitive:

```
#include <sys/types.h>
#include <sys/socket.h>
int recvmsg(s, msg, flags)
int sendmsg(s, msg, flags)
    int s, flags;
    struct msghdr msg[];
```

s == socket descriptor

msg == puntatore alla struttura msghdr

flag == per `send()` e `recv()`

Atomicità

uso di molte chiamate send() con datagrammi
possono arrivare al destinatario fuori ordine

Unica chiamata sendmsg()

buffer non contigui vengono incapsulati
in un unico messaggio

Gestione sincrona di eventi multipli

Uso della chiamata select()

differenza sugli eventi che provocano il ritorno della funzione

- in lettura:
 - . *recv asincrona*: ioctl con FIONREAD restituisce il numero di bytes disponibili,
- eventi eccezionali:
 - . all'arrivo dati out-of-band,
 - . socket non utilizzabile (close o shutdown)

Scambio di dati con modalità asincrona

Esempio di processi comunicanti in modo asincrono

Specifiche di IPC nel Dominio UNIX

Solo comunicazione locale

le socket in dominio Internet hanno prestazioni inferiori a le socket locali
overhead di protocollo IP

Identificazione della socket

```
#include <sys/un.h>
```

```
struct sockaddr_un {  
    short sun_family;  
    u_char sun_path[92];  
}
```

sun_family == **AF_UNIX**

sun_path == *path name*

Si noti la **identità del servitore** (attraverso il nome di file)

Ad esempio

```
struct sockaddr_un myaddr;  
myaddr.sun_family = AF_UNIX;  
strcpy(myaddr.sun_path, "/tmp/c");
```

bind() con un *inode* corrispondente al path name

Collegamento ad un file

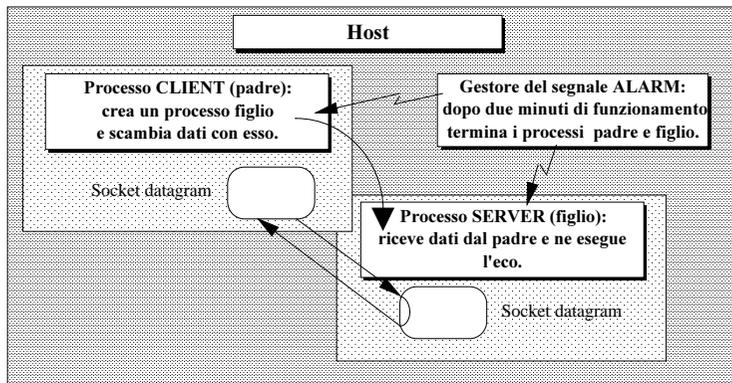
Uso di chiamata *unlink()* oppure cancellare il file
mediante il comando *rm*

Attenzione: *a volte, mescolare domini diversi provoca problemi*

```
#define SOCKET_PATH /tmp/myserver
struct sockaddr_un servaddr;

servaddr.sun_family=AF_UNIX;
strcpy(servaddr.sun_path,SOCKET_PATH);
s=socket(AF_UNIX,SOCK_DGRAM,0);
bind(s,&servaddr,sizeof(struct sockaddr_un));
.....
close(s);
unlink(SOCKET_PATH);
```

Esempio di server e client



Processi locali con datagram socket

Il server attende un numero di messaggi da 2000 bytes ciascuno e, una volta arrivati, ne esegue l'eco
 Il server ha un time-out per terminare dopo 2 minuti

* Processo SERVER */

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/un.h>
#include <stdio.h>
#include <signal.h>
#include <netdb.h>
#define bzero(ptr, len) memset((ptr), NULL, (len))
/* azzerare strutture */
int timeout();

main( argc, argv)
int argc; char *argv[];
{ int sock, slen, rlen, expect;
  unsigned char sdata[5000];
  struct sockaddr_un servaddr; /* indirizzo del server */
  struct sockaddr_un from;
  int fromlen;
  signal (SIGALRM, timeout);
  alarm ((unsigned long) 120); /* assegnazione time-out */
  if ((sock = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0)
    { perror("server: socket"); exit(1); }
  bzero(&servaddr, sizeof(servaddr));
  servaddr.sun_family = AF_UNIX;
  strcpy(servaddr.sun_path,argv[1]);
  if (bind (sock, &servaddr, sizeof(servaddr)) < 0)
    { close(sock); perror("server: bind"); exit(2);}
```

```

expect = atoi (argv[2]) * 2000;
while (expect > 0)
{ fromlen = sizeof(from);
  rlen = recvfrom (sock, sdata, 2000, 0, &from, &fromlen);
  if (rlen == -1) { perror("server : recv\n"); exit(3); }
  else {
    expect -= rlen;
    printf("server : ricevuti %d bytes\n", rlen);
    slen = sendto (sock, sdata, rlen, 0, &from, fromlen);
    /* risponde al mittente */
    if (slen < 0) { perror ("server : sendto\n"); exit (4); }
  }
}
unlink(argv[1]);
/* unlink() serve a rendere disponibile l'inode per la prossima
esecuzione */
close(sock);
printf("Server terminato.\n");
exit(0);}

timeout() /* gestore del timeout per evitare blocchi indefiniti */
{ printf( "server: tempo di attesa scaduto.\n" );
  fprintf(stderr, "fermo il processo server\n");
  exit(5);}

```

Il processo server viene impegnato dalla attesa

Il processo client esegue una fork()
per creare un processo slave che esegue il servizio
 L'utente non deve lanciarlo in background

/* Processo CLIENT */

```

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/un.h>
#include <stdio.h>
#include <signal.h>
#include <netdb.h>
#define bzero(ptr, len) memset((ptr), NULL, (len))
int timeout();

main( argc, argv)
  int argc;  char *argv[];
{  int sock;
   int j, slen, rlen, fromlen;
   unsigned char  sdata[2000]; /* invio dati */
   unsigned char  rdata[2000]; /* dati ricevuti */
   struct sockaddr_un servaddr; /* indirizzo del server */
   struct sockaddr_un clntaddr; /* indirizzo del client */
   struct sockaddr_un from;

  if (argc!=4) { fprintf(stderr,"Uso: %s <clientpath> <serverpath> \
<N messaggi>\n",argv[0]);exit(1); }
  signal (SIGALRM, timeout);
  alarm ((unsigned long) 120);
  /* crea un processo servitore per ricevere i dati */
  printf("Client : fork del server\n");
  if (fork() == 0 ) {
    execl("./suudp", "suudp", argv[2], argv[3], 0 );
    printf("impossibile eseguire ./suudp.\n");
    exit(1); }

```

```

/* Inizializza il messaggio da spedire */
for (j = 0; j < sizeof(sdata); j++)  sdata[j] = (char) j;
if ((sock = socket (AF_UNIX, SOCK_DGRAM, 0)) < 0) {
    perror("client: socket");  exit(2); }

/*Il client esegue il bind ad una socket in modo che il server
  possa acquisire il suo indirizzo e rispondere ai messaggi */
bzero(&clntaddr, sizeof(clntaddr));
clntaddr.sun_family = AF_UNIX;
strcpy(clntaddr.sun_path, argv[1]);
if (bind (sock, &clntaddr, sizeof(clntaddr)) < 0)
    { close(sock); perror("client: bind"); exit(3); }
bzero(&servaddr, sizeof(servaddr));
servaddr.sun_family = AF_UNIX;
strcpy(servaddr.sun_path, argv[2]);
for (j = 0; j < atoi(argv[3]); j++)
    { sleep(1);
      slen = sendto (sock, sdata, 2000, 0,
                    (struct sockaddr *) &servaddr, sizeof(servaddr));
      if (slen < 0) {  perror("client: sendto");  exit(4); }
      else {  printf("client : inviati %d bytes\n", slen);
              fromlen = sizeof(from);
              rlen = recvfrom (sock, rdata, 2000, 0, &from, mlen);
              if (rlen == -1) { perror("client: recvfrom\n");
                              exit(5); }
              else printf("client : ricevuti %d bytes\n", rlen);}
      sleep(1); unlink(argv[1]);  close(sock);
      printf("Client terminato.\n");  exit(0);}

timeout()
{printf( "client: tempo di attesa scaduto.\n" );
  fprintf(stderr, "fermo il processo client\n");  exit(6);}

```

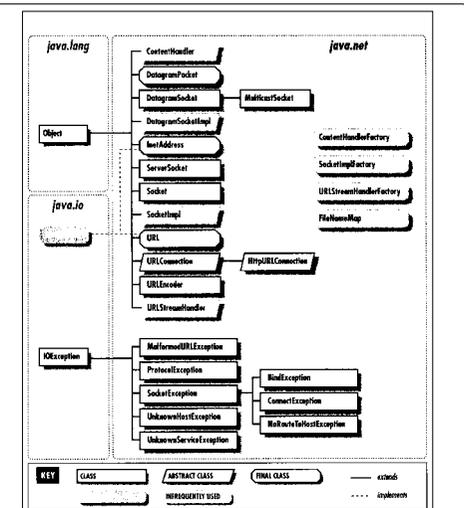
La Programmazione di rete in Java

Java fornisce per la gestione della comunicazione le classi del package di networking `java.net`

Possibilità di utilizzare SOCKET e comunicazione:

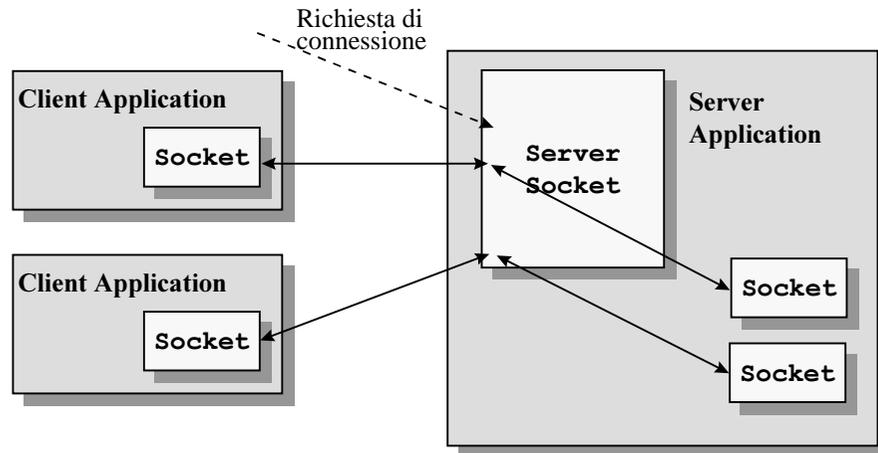
- con **connessione**
 - classe `Socket`, per socket lato **Client**, con connessione (TCP)
 - classe `ServerSocket`, per socket lato **Server**, con connessione (TCP)
- senza **connessione**
 - classe `DatagramSocket`, per socket (C/S) senza connessione (UDP)

Gerarchia delle classi del package `java.net`



SOCKET per STREAM

Protocollo di Comunicazione **con Connessione** in Java



Classi distinte per ruoli **Cliente** e **Servitore**

La classe `java.net.Socket`

La classe `java.net.ServerSocket`

Lato CLIENTE

La classe `Socket` consente di creare una socket con **connessione**, stream (TCP) per il collegamento di un **Client** a un Server

I costruttori della classe creano la socket, la legano a una porta locale e la connettono a una porta di una macchina remota (*Unix: socket, bind, connect*)

Operazioni CLIENTE

```
public Socket(InetAddress remoteHost,  
              int remotePort) throws ...
```

Crea una **socket stream cliente** e la collega alla porta specificata della macchina all'indirizzo IP dato (equivalente in Unix a: `socket, bind, connect`)

```
public Socket(String remoteHost,  
              int remotePort) throws ...
```

Crea una **socket stream cliente** e la collega alla porta specificata della macchina di nome logico `String`

```
public Socket(InetAddress remoteHost,  
              int remotePort,  
              InetAddress localInterface,  
              int localPort) throws ...
```

Crea una **socket stream cliente** e la collega

sia a una porta della macchina locale

(se `localPort` vale zero, il numero di porta è scelto automaticamente dal sistema)

sia a una porta della macchina remota

(macchine multihomed)

La creazione della socket produce in modo atomico anche la connessione al server corrispondente

Operazioni CLIENTE - supporto

Per ottenere informazioni sulle socket si possono utilizzare i metodi:

```
public InetAddress getInetAddress()  
    restituisce l'indirizzo della macchina remota a cui la socket  
    è connessa
```

```
public InetAddress getLocalAddress()  
    restituisce l'indirizzo della macchina locale
```

```
public int getPort()  
    restituisce il numero di porta sulla macchina remota a cui  
    la socket è connessa
```

```
public int getLocalPort()  
    restituisce il numero di porta sulla macchina locale a cui la  
    socket è legata
```

Esempio: `int porta = oggettoSocket.getPort();`

Operazioni di comunicazione su stream

Lettura o scrittura da/su una socket con i metodi:

```
public InputStream getInputStream()  
    ritorna l' input stream per leggere dei byte dalla socket  
public OutputStream getOutputStream()  
    ritorna l' output stream per scrivere dei byte sulla socket
```

I due metodi restituiscono un oggetto stream (`InputStream` e `OutputStream`) che incapsula il canale di comunicazione. Attraverso gli stream si possono spedire/ricevere solo byte, senza nessuna formattazione. Altri oggetti possono incapsulare questi stream, per fornire funzionalità di più alto livello (es., `DataInputStream`)

Limiti alle aperture contemporanee di sessioni

Il numero di connessioni che un processo (Client o Server) può aprire è limitato, => si chiudono le connessioni non utilizzate

Il metodo **close()** chiude l'oggetto socket e disconnette il Client dal Server

```
public synchronized void close()
```

Client di echo (il Server Unix e' sulla porta 7)

```
try {  
    oggSocket = new Socket(hostname, 7);  
    out = new PrintWriter  
        (oggSocket.getOutputStream(), true);  
    in = new BufferedReader(new InputStreamReader  
        (oggSocket.getInputStream()));  
    userInput = new newBufferedReader  
        (new InputStreamReader(System.in));  
    while (true)  
    {  
        oggLine = userInput.readLine();  
        if (oggLine.equals(".")) break;  
        out.println(oggLine);  
        System.out.println(in.readLine());  
    }  
    oggSocket.close();  
    // fine try  
    catch (UnknownHostException e)  
        { System.err.println(e); }+  
    catch (IOException e)  
        { System.err.println(e); }  
    . . .
```

La classe `java.net.ServerSocket`

Lato server

La classe `ServerSocket` definisce una socket capace di accettare richieste di connessione provenienti dai Client

Una volta **stabilita la connessione** (tramite il metodo `accept`) la trasmissione dei dati avviene attraverso un normale oggetto `Socket` (restituito dalla `accept`).

Al momento della creazione effettua le operazioni di socket, bind e listen.

Costruttori:

```
public ServerSocket(int localPort) throws...
```

crea una socket in ascolto sulla porta specificata

```
public ServerSocket(int localPort, int count)
```

crea una socket in ascolto sulla porta specificata con una coda di lunghezza count

```
public Socket accept() throws...
```

la chiamata di `accept` mette il servitore in attesa di nuove richieste di connessione. Se non ci sono richieste, il servitore si blocca in attesa.

La `accept` restituisce un oggetto della classe `Socket` su cui avviene la fase di comunicazione tra il Client e il Server

Informazioni sulle socket connesse :

```
public InetAddress getInetAddress()
```

restituisce l'indirizzo della macchina **locale** a cui la socket server è connessa

```
public int getLocalPort()
```

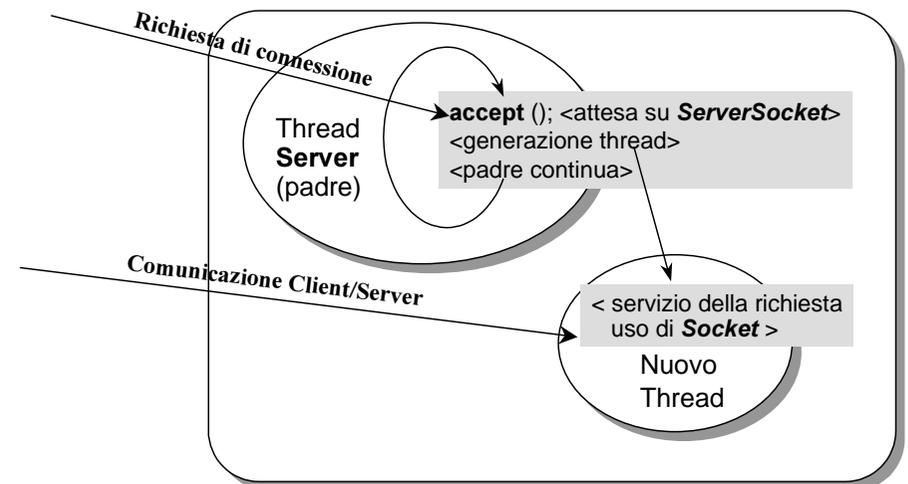
restituisce il numero di porta sulla macchina **locale**

Server daytime (il Server Unix su porta 13)

```
...
oggServer = new ServerSocket(portaDaytime);
try {
    while (true) {
        oggConnessione = oggServer.accept();
        out = new PrintWriter
            (oggConnessione.getOutputStream(), true);
        out.println(new Date());
        oggConnessione.close();
    }
}
catch (IOException e) {
    oggServer.close(); System.err.println(e);
}
...

```

Server Multi-Threaded connection-oriented



La classe `java.net.DatagramSocket`

```
public final class
    DatagramSocket extends Object
```

Costruttore

```
DatagramSocket(InetAddress localAddress,
                int localPort) throws...
```

`DatagramSocket` crea socket UDP legata localmente

Uso di `send` e `receive`

```
Sock.sendMessage("stringa");
Sock.send(DatagramPacket);
Sock.receive(DatagramPacket);
```

`DatagramPacket`

Esistono anche le classi per usare e preparare datagrammi che consentono di specificare come comunicare

Un pacchetto specifica un array di byte da/su cui scrivere e con indicazioni di comunicazione con diversi costruttori

```
DatagramPacket(
    byte [] buf, int offset, int length,
    InetAddress address,
    int port);
```

`InetAddress`

classe per gli indirizzi IP

Opzioni delle Socket

Si possono vedere alcune opzioni delle socket in Java con funzioni definite

SetTcpNoDelay (boolean on) throws ...
il pacchetto è inviato immediatamente, senza bufferizzare

SetSoLinger (boolean on, int linger)
dopo la close, il sistema tenta di consegnare i pacchetti ancora in attesa di spedizione. Questa opzione permette di scartare i pacchetti in attesa (linger in sec)

SetSoTimeout (int timeout) throws...
la lettura da socket (es., `read()`) è bloccante.
Questa opzione definisce un timeout (in msec), trascorso il quale si sblocca la read (ma viene lanciata una eccezione da gestire)

SetSendBufferSize (int size) throws...
il buffer di invio dello stream può essere variato
SetReceiveBufferSize (int size)
il buffer di invio dello stream può essere variato

SetKeepAlive (boolean on) throws...
abilita, disabilita la opzione di keepalive

Sono previste le get corrispondenti
alcune opzioni sono nella classe `SocketOptions`

Socket Multicast

Sono anche possibili ulteriori classi per inviare **messaggi multicast** e **gruppi di multicast**

Preparazione del gruppo: IP classe D e porta libera

```
InetAddress gruppo = InetAddress.  
    getByName("229.5.6.7");  
MulticastSocket s = new  
    MulticastSocket(6666);
```

Operazioni di ingresso/uscita dal gruppo

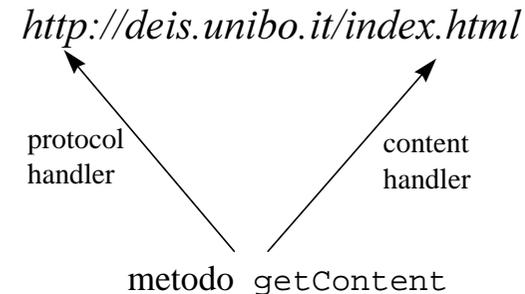
```
// unisciti al gruppo ...  
s.joinGroup(gruppo);  
  
byte[] msg = {'H', 'e', 'l', 'l', 'o'};  
DatagramPacket packet=new DatagramPacket  
    (msg, msg.length, gruppo, 6666);  
s.send(packet);  
  
// get their responses!  
byte[] buf = new byte[1000];  
DatagramPacket recv = new  
    DatagramPacket (buf, buf.length);  
s.receive(recv);  
...  
// esci dal gruppo ...  
s.leaveGroup(group);
```

URL

Inoltre visto l'ampio uso di **Indirizzi Web**

La classe `java.net.URL`

La classe URL descrive le risorse Internet



Scaricamento di una pagina

```
...  
URL deis = new URL  
    ("http://www.deis.unibo.it/");  
BufferedReader in = new BufferedReader(  
    new InputStreamReader(  
        deis.openStream()));  
  
String inLine;  
while ((inLine = in.readLine()) != null)  
    System.out.println(inLine);  
in.close();  
...
```

non solo la possibilità di agganciarsi dinamicamente ad una URL ma di gestire il protocollo HTTP