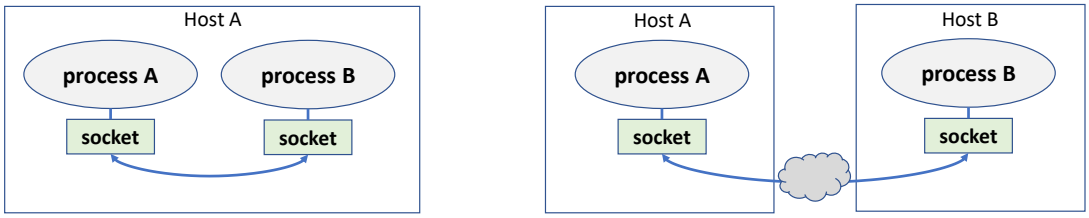
Comunicazione tramite socket

Sono un meccanismo di comunicazione inter-processo

Socket

- Un *socket* è un'astrazione per un endpoint della comunicazione tra due processi
  - Stessa macchina o macchine diverse (in rete)
  - *Network IPC*



- Socket descriptor come file descriptor
  - Una volta ottenuto un socket descriptor, potremmo usare le system call `read`/`write`

Un socket permette a processi di comunicare, indipendentemente dal fatto che si trovino sulla stessa macchina o su machine diverse (quindi in rete).
Il vantaggio dei socket è la flessibilità nel supportare comunicazioni tra processi. Il processo crea il socket e tutto quello che deve fare è scrivere dati su questa struttura dati. Il sistema operativo si occupa poi di trasferire i dati alla destinazione giusta.

Tipi di comunicazione

message-oriented

- Invio/ricezione di messaggi (datagrams)
- Messaggi consecutivi sono indipendenti
  - Self-contained messages

VS

stream-oriented

- Invio/ricezione di sequenze di byte (stream)
- Una sequenza di byte può comporre o meno un messaggio completo a livello applicativo

connectionless

- Non esiste una connessione logica tra i endpoint della comunicazione
- I dati inviati devono contenere informazioni di indirizzamento

VS

connection-oriented

- Gli endpoint della comunicazione devono prima stabilire una connessione logica
- I dati inviati non contengono informazioni di indirizzamento

La comunicazione tramite socket può essere di diversi tipi, e le possiamo classificare con due categorizzazioni, ortogonali tra loro.

Connectionless vs Connection-oriented = Posta vs telefono
- Connectionless: ogni invio deve contenere l'indirizzo destinatario, come una lettera o una e-mail
- Connection-oriented: ciascun invio non contiene informazioni di indirizzamento perché è già stata definito un canale di comunicazione uno a uno

Creazione di un socket

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

int socket(int domain, int type, int protocol);
```

File (socket) descriptor se ok, `-1` in caso di errore

Specifica le modalità di comunicazione

SOCK_STREAM	Stream-oriented; connection-oriented; con garanzie di ordinamento e consegna
SOCK_DGRAM	Message-oriented con messaggi di lunghezza massima fissata; connectionless; senza garanzia di ordinamento né di consegna,
SOCK_SEQPACKET	Message-oriented con lunghezza massima fissata, connection-oriented, con garanzia di ordinamento e consegna
SOCK_RAW	Accesso diretto all'interfaccia di rete; applicazione responsabile di gestire la comunicazione

AF_INET	Internet IPv4
AF_INET6	Internet IPv6
AF_UNIX	Comunicazione locale
AF_UNSPECIFIED	qualsiasi

Specifica la natura della comunicazione e le modalità di indirizzamento

`man socket`

La system call socket e' simile alla system call open per le operazioni su file. In entrambi i casi si ottiene un file descriptor su cui fare operazioni di I/O. Quando il socket non serve piu', basta invocare close().

Con socket di tipo datagram, tutto cio' che si deve fare è inviare un messaggio al destinatario, mentre con socket di tipo stream si richiede di stabilire una connessione logica tra gli endpoint prima di iniziare a comunicare.
Come la posta ordinaria, un socket di tipo datagram non da' garanzie che un messaggio arrivi a destinazione, ne' tanto meno che due messaggi arrivino nello stesso ordine in cui sono stati inviati.

Creazione di un socket

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

int socket(int domain, int type, int protocol);
```

Protocollo di comunicazione

IPPROTO_IP	Internet Protocol v4 (IPv4)
IPPROTO_IPV6	Internet Protocol v6 (IPv6)
IPPROTO_ICMP	Internet Control Message Protocol
IPPROTO_RAW	Raw IP packets
IPPROTO_TCP	Transmission Control Protocol
IPPROTO_UDP	User Datagram Protocol

- L'argomento `protocol` è solitamente impostato a 0
  - utilizza il protocollo di default in base a `domain` e `type`
  - `domain==AF_INET, type==SOCK_STREAM` → `protocol=IPPROTO_TCP`
  - `domain==AF_INET, type==SOCK_DGRAM` → `protocol=IPPROTO_UDP`
- Si usa la system call `close()` per chiudere il file descriptor quando non serve più

`man socket`

Tipicamente non si sceglie esplicitamente il protocollo, ma lo si lascia scegliere alla system call impostandolo al valore 0.
Se scegliamo il dominio AF_INET significa che vogliamo utilizzare i protocolli dello stack TCP/IP per la comunicazione in rete. Dalla teoria, sapete che TCP è un protocollo connection-oriented e stream-oriented, mentre UDP è un protocollo connectionless e message-oriented.
Se scegliamo SOCK_STREAM, allora verrà creato un socket di tipo TCP, se scegliamo un socket di tipo SOCK_DGRAM, allora verrà creato un socket di tipo UDP.

Indirizzamento

- Un endpoint della comunicazione (processo) viene identificato da un **indirizzo** e da un **numero di porta**

- Il formato dell'indirizzamento dipende dal dominio
  - `AF_INET` → Indirizzo IPv4
  - `AF_INET6` → Indirizzo IPv6
  - …

Per poter connettersi o inviare un messaggio a un processo su un'altra macchina, dobbiamo avere un modo per identificarlo, tramite un **indirizzo**. Un macchina in una rete TCP/IP è identificata da un'indirizzo IP.
Tale macchina può eseguire diversi processi, per cui non basta l'indirizzo della macchina, ma ci serve anche il numero di porta per identificare lo specifico processo a cui vogliamo connetterci o inviare un messaggio.

Indirizzamento

Le funzioni che implementano i socket usano una struttura dati generica per gli indirizzi, valida per tutti i domini:

```
struct sockaddr {
    sa_family_t sa_family;
    char        sa_data[];
};
```

dominio dell'indirizzo

indirizzo con lunghezza variabile

Nel dominio AF_INET, si usa `struct sockaddr_in`, che verrà convertita (cast) a `struct sockaddr`

```
struct in_addr {
    in_addr_t      s_addr;
};

struct sockaddr_in {
    sa_family_t    sin_family;
    in_port_t      sin_port;
    struct in_addr sin_addr;
};
```

Definito come `uint32_t`

Definito come `uint16_t`

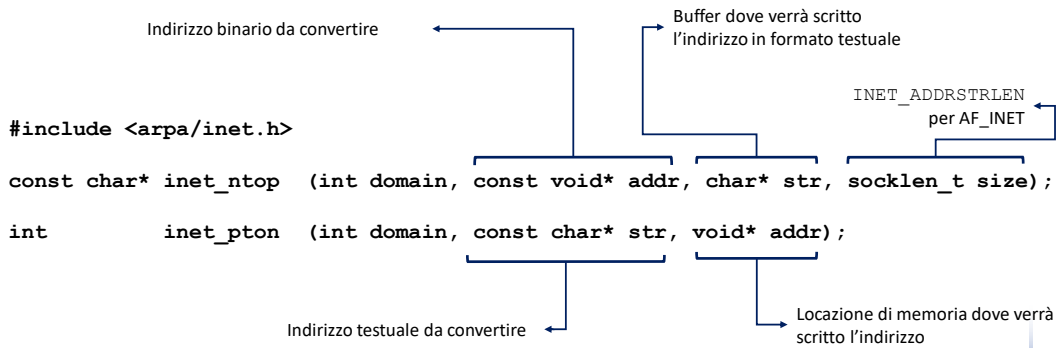
Gli indirizzi per i socket utilizzano una struttura dati generica sockaddr, valida per accomodare qualsiasi tipo di dominio.

Nel caso di AF_INET, si usa la struttura dati sockaddr_in che contiene un indirizzo IP (sin_addr) e un numero di porta (sin_port)

Indirizzo in formato testuale

- Gli indirizzi IPv4 vengono normalmente scritti in formato human-readble usando la notazione a punto (dotted notation)

```
11000000 10101000 00000001 00000001 → "192.168.1.1"
```



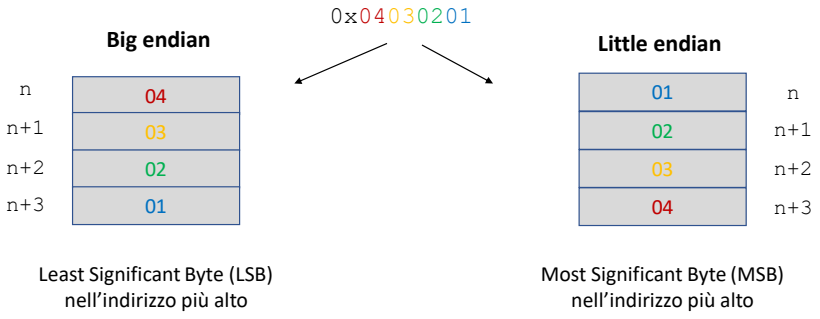
Le strutture dati della slide precedenti devono contenere un indirizzo in formato binario (un indirizzo IPv4 è lungo 32 bit), ma noi siamo abituati a pensare a un indirizzo in formato testuale, e.g. 192.168.1.1
Per questo la libreria socket ci fornisce delle funzioni per effettuare la conversione:

- da formato binario a formato testuale: inet_ntop
- da formato testuale a formato binario: inet_pton

In questo modo l'utente del programma può interagire speficando gli indirizzi in un formato più facile da memorizzare rispetto al formato binario.

Ordinamento dei byte

Due host diversi possono utilizzare byte ordering differenti!



Least Significant Byte (LSB)
nell'indirizzo più alto

Most Significant Byte (MSB)
nell'indirizzo più alto

Si consideri un numero intero che viene memorizzato su 4 byte nella memoria di un processo, e che questi 4 byte vengono memorizzati in locazioni contigue della memoria. Non c'è alcuno standard che dice in quale ordine questi byte devono essere memorizzati.
Stiamo considerando una comunicazione tra due processi che risiedono potenzialmente su due macchine diverse. È possibile che queste due macchine seguano differenti ordinamenti dei byte. Ciò può causare problemi di compatibilità: dovendo inviare un numero di 4 byte, il processo A potrebbe inviare i byte dal meno significativo al più significativo mentre il processo B potrebbe considerarli al contrario, ottenendo un numero totalmente diverso.

Network byte order

- Due host comunicanti devono "accordarsi" per usare lo stesso ordinamento
- Necessaria una conversione → TCP/IP usa **big endian**

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostint32);

uint16_t htons(uint16_t hostint16);

uint32_t ntohl(uint32_t netint32);

uint16_t ntons(uint16_t netint16);
```

h = host
n = network
l = long int
s = short int

- Funzioni che convertono un numero a 32 o 16 bit da ordinamento host a network e viceversa
- Utilizzato per
  - Indirizzi e numero di porta
  - Scambio di informazioni formattate (e.g., invio di un numero intero tramite socket)

Per risolvere il problema della slide precedente, lo standard TCP/IP utilizza l'ordinamento Big Endian. Per cui la libreria arpa/inet.h ci mette a disposizione delle funzioni per effettuare la conversione di un numero prima di inviarlo tramite un socket e/o dopo averlo ricevuto da un socket.
La pratica comune, ogni volta che si tratta di un numero da utilizzare con i socket (per invio/ricezione o per specificare un indirizzo/porta) è di convertirlo in formato network prima di usarlo. Per usare il numero localmente al processo, invece, è necessario convertirlo sempre in formato host.
Notare che formato network e formato host potrebbero coincidere (se la macchina in questione utilizza anch'essa Big Endian): in quel caso, queste funzioni non hanno effetto.

Associazione di un indirizzo a un socket

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

int bind(int sockfd, struct sockaddr* addr, socklen_t len);
```

- Informa il kernel che il processo vuole ricevere dati in arrivo alla coppia indirizzo/porta specificata da `addr`
- Il numero di porta deve essere maggiore di 1023
  - Numeri minori di 1024 sono dedicati a servizi well-known (e.g., 80 per HTTP, 21 per SMTP)
- Di solito si usa solo per il processo server
  - Al socket sul processo client viene assegnata la coppia indirizzo/porta dal kernel
- Specificando l'indirizzo `INADDR_ANY`, il socket accetta connessioni da qualsiasi interfaccia di rete dell'host
- Possibile usare indirizzo dell'interfaccia di loopback → 127.0.0.1
  - `$ ip a` → interfaccia `lo`

Creazione di una connessione

Lato server:

```
int listen(int sockfd, int backlog);
```

- Annuncio che il socket pronto ad accettare richieste di connessioni
  - `backlog` è il numero massimo di richieste di connessioni in sospeso

```
int accept(int sockfd, struct sockaddr* addr, socklen_t* len);
```

- Prelievo di una richiesta e creazione di una connessione socket
  - Restituisce un nuovo socket descriptor, da usare per ricevere i dati
  - Ottiene l'indirizzo del richiedente
  - Bloccante se non ci sono richieste in coda

Lato client:

```
int connect(int sockfd, const struct sockaddr* addr, socklen_t len);
```

- Inizio della procedura per stabilire una connessione con il socket dell'altro endpoint della comunicazione
  - Invio di una richiesta di connessione all'indirizzo specificato
  - Possibile usare indirizzo di loopback 127.0.0.1 per comunicare con un socket sullo stesso host

La listen() informa il kernel che il socket è pronto a ricevere richieste di connessione. Predispone una coda in cui vanno a finire tutte le richieste di connessione da altre macchine, in attesa di essere accettate e gestite (richieste pendenti). La coda ha una dimensione massima definito da backlog, dopodichè eventuali altre richieste vengono automaticamente scartate.

La accept() pesca dalla coda di richieste pendenti. Se la coda è vuota, si mette in attesa dell'arrivo di una connessione (e il processo si blocca). Quando si riesce ad accettare una nuova richiesta, viene creato un nuovo socket (il cui socket descriptor viene restituito dalla accept). Il processo userà questo nuovo socket per ricevere i dati. Il motivo della creazione di un nuovo socket è che quello originario può continuare a restare in attesa di nuove connessioni da altri processi.

Per far sì che la connect() vada a buon fine, è necessario che l'altro processo sia in esecuzione, che sia in ascolto (listen) allo stesso indirizzo specificato dalla connect e che abbia ancora spazio nella coda delle richieste di connessioni pendenti.

Scambio di dati su un socket connesso

- Può avvenire usando le system call `read()` e `write()`
  - socket descriptor = file descriptor
- Disponibili system call apposite, con maggiori funzionalità:

```
ssize_t send(int sockfd, const void* buf, size_t nbytes, int flags);
```

- Prelievo di `nbytes` dal buffer e invio sul socket

`MSG_NOSIGNAL`
`MSG_OOB`
…

```
ssize_t recv(int sockfd, const void* buf, size_t nbytes, int flags);
```

- Ricezione di `nbytes` dal socket e inserimento in un buffer locale al processo

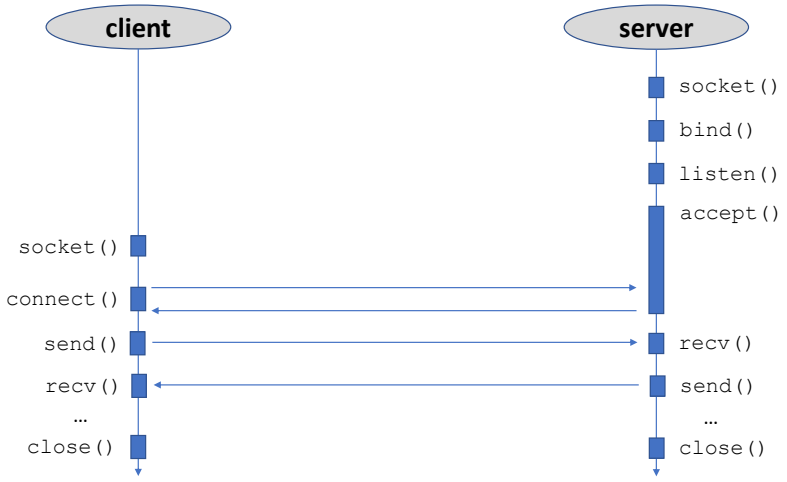
`MSG_DONTWAIT`
`MSG_WAITALL`
`MSG_OOB`
…

Send è praticamente uguale alla write, ma permette di aggiungere dei flag con opzioni aggiuntive su come devono essere trattati i dati inviati sul socket. Se la send restituisce senza errore, **non** significa che il processo ricevitore abbia effettivamente ricevuto i dati. La send mi dice solo che i dati sono stati inviati con successo dal processo. La consegna al ricevitore è compito del sistema operativo e dei protocolli di rete sottostanti (i dati potrebbero essere scartati sulla rete, da un router intermedio)

La recv() è del tutto simile alla read (con dei flag in più) e restituisce il numero di byte effettivamente ricevuti dal socket.

Sarebbe possibile anche usare read/write per leggere/scrivere dal socket descriptor, ma solo per socket connection-oriented.

Socket connection-oriented



Scambio di dati su socket connectionless

Lato client:

```
ssize_t sendto(int sockfd, const void* buf, size_t nbytes, int flags,
                      const struct sockaddr* addr, socklen_t addrlen);
```

- Prelievo di `nbytes` dal buffer e invio sul socket, all'indirizzo specificato da `addr`

Lato server:

```
ssize_t recvfrom(int sockfd, void* buf, size_t len, int flags,
                       const struct sockaddr* addr, socklen_t* addrlen);
```

- Ricezione di `nbytes` dal socket e inserimento in un buffer locale al processo
- Permette di ottenere l'indirizzo del sender dentro `addr`

Sendto è la duale di send, ma ovviamente dobbiamo specificare l'indirizzo a cui vogliamo inviare il messaggio
Stesso discorso per la recvfrom. La recvfrom restituisce anche l'indirizzo della macchina che ha inviato il messaggio, nell'argomento addr.

Per socket connectionless, non potremmo usare read/write perché non sapremmo come specificare l'indirizzo di destinazione dei dati.

Socket connectionless