File e Socket

Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma
Luca Iezzi Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Reti di Calcolatori A.A. 2022/23

Prof.ssa Chiara Petrioli Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma
Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma

1. File I/O

Archiviazione di massa

Diversamente da quanto avviene con variabili, array, ecc all'interno di un programma, i file sono strumenti per l'archiviazione di massa:

- Persistenza dopo il termine dell'esecuzione di un programma
- Dimensioni dati potenzialmente maggiori rispetto a quanto possibile in RAM

Esattamente come avviene in RAM, con i file è possibile:

- Accedere sequenzialmente → tipicamente file di testo
- Accedere in modo casuale → tipicamente file binari

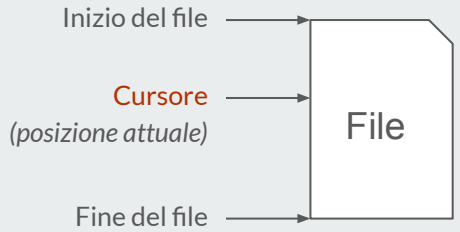
Operazioni necessarie per l'utilizzo dei file

In C, un apposito tipo di dato `FILE` è reso disponibile dalla libreria `stdio.h`. Usando la libreria C per l'I/O, tutte le interazioni con i file si basano su puntatori a variabili `FILE`. Questi oggetti sono chiamati *file stream*.

Affinché i file siano utilizzati con successo, è necessario seguire determinate operazioni nel corretto ordine:

1. Apertura del file
   Questa operazione restituisce `FILE*`, se l'apertura va a buon fine rappresenta l'interfaccia con il file e può essere usato nelle successive operazioni; in caso di errori, viene restituito `NULL` → controllare il valore di ritorno!
2. Operazioni varie sul file
   Tramite il puntatore a file è possibile effettuare letture, scritture, ecc
3. Chiusura del file
   Al termine delle operazioni è buona norma chiudere esplicitamente i file precedentemente aperti; dopo la chiusura di un file, il `FILE*` associato non dovrebbe essere più usato → undefined behavior!

Le interazioni con i file avvengono per mezzo di un cursore interno, gestito automaticamente e aggiornato a ogni operazione su uno stesso file.
L'apertura di un file determina la posizione iniziale di tale cursore.



Apertura e chiusura file

- `fopen(const char *filename, const char *mode)`

  È la funzione per l'apertura di un file; `filename` è una stringa C che rappresenta il percorso del file di interesse, mentre `mode` è una stringa C che rappresenta la modalità di apertura del file → determina le operazioni ammesse sul `FILE*` restituito

  Modalità di apertura supportate:
  - "`r`" → sola lettura; il file deve esistere, altrimenti ritorna `NULL`
  - "`w`" → sola scrittura; se non esiste, il file viene creato; se già esistente, tutti i contenuti sono azzerati
  - "`a`" → scrittura in *append*: simile a "`w`", ma file esistenti non sono azzerati, bensì nuove aggiunte al termine del file
  - "`r+`" → lettura e scrittura: simile a "`r`"; è possibile modificare in qualsiasi punto del file partendo dall'inizio
  - "`w+`" → lettura e scrittura: analogo a "`w`"
  - "`a+`" → lettura e scrittura: simile ad "`a`"; è possibile modificare in qualsiasi punto del file partendo dalla fine

Modalità	Nome	File non esiste	File esiste	Posizione iniziale
`r`	read	`NULL`	Lettura	Inizio
`w`	write	Crea	Scrittura, azzera contenuti	Inizio
`a`	append	Crea	Scrittura	Fine

Modalità	Nome	File non esiste	File esiste	Posizione iniziale
`r+`	read extended	`NULL`	Lettura e scrittura	Inizio
`w+`	write extended	Crea	Lettura e scrittura, azzera contenuti	Inizio
`a+`	append extended	Crea	Lettura e scrittura	Fine

Apertura e chiusura file

`fclose(FILE *fp)`

È la funzione per la chiusura di un file, tramite il suo puntatore associato `fp` (restituito da `fopen`)
Valore di ritorno `int`:

- `0` → chiusura del file riuscita
- `EOF` → errore in chiusura

`EOF` è una costante fornita dalla libreria `stdio.h` e rappresenta il termine del file; nel caso sia restituito da `fclose`, questo valore va però interpretato come errore in chiusura

Tentativo di apertura in read extended: lettura/scrittura

Percorso file relativo

Tentativo di apertura in write extended, se "`r+`" non ha aperto il file

Chiusura del file
Il risultato può essere inserito in uno `switch`, essendo di tipo `int`

```c
FILE *myFile = fopen("lab_c.txt", "r+");
if(myFile == NULL){
  printf("Errore nell'apertura del file in lettura\n");
  myFile = fopen("lab_c.txt", "w+");
}
if(myFile == NULL){
  printf("Il file non può essere aperto nemmeno in scrittura\n");
}

// File aperto correttamente
else {
  // Operazioni sul file

  switch(fclose(myFile)){
    case 0: {
      printf("File chiuso correttamente\n");
      break;
    }

    case EOF: {
      printf("Errore nella chiusura del file\n");
      break;
    }
  }
}
```

Operazioni comuni su file

Le principali operazioni su file:

- `int fgetc(FILE *fp)`
  Legge il prossimo byte dal file rappresentato tramite il puntatore `fp`, aggiornando il cursore del file.
  Il valore di ritorno `int`:
  - Byte letto dal file, interpretato come `unsigned char` e convertito ad `int` → lettura con successo
  - `EOF` (*End-of-File*) → lettura fallita
- `int fputc(int ch, FILE *fp)`
  Scrive il byte fornito `ch` nel file rappresentato da `fp`, dopo averlo convertito ad `unsigned char`, ed aggiorna il cursore del file.
  Il valore di ritorno `int`:
  - Byte scritto sul file → scrittura con successo
  - `EOF` → scrittura fallita

Lo stesso valore di ritorno `EOF` indica un'operazione di lettura o scrittura fallita in diversi casi:

- Lettura oltre il termine del file: non va considerato un errore → può essere determinato da `feof(fp) != 0`
- Qualsiasi altro caso: va considerato un errore → può essere determinato da `ferror(fp) != 0`

Operazioni comuni su file

- `size_t fread(void *buffer, size_t objSize, size_t count, FILE *fp)`

  Legge un numero di oggetti (`count`) dal file e li restituisce all'interno di un array `buffer`; `objSize` è la dimensione di un singolo oggetto in byte.
  Equivale ad effettuare `count` invocazioni consecutive di `fgetc`, salvando ordinatamente ogni byte restituito in `buffer`.
  Il valore di ritorno `size_t` è il numero di oggetti letti con successo → potrebbe non coincidere con `count`!

Verificare il valore di `fp`!

```c
double data[5];
FILE *fp = fopen("lab_C.txt", "r");
size_t objs = fread(data, sizeof(*data), 5, fp);
// Controllo objs == 5 --> lettura di 5 double con successo?
// ...
fclose(fp);
```

`*data = data[0]`

- `size_t fwrite(const void *buffer, size_t objSize, size_t count, FILE *fp)`

  Scrive un numero di oggetti (`count`) nel file, forniti tramite l'array `buffer`; `objSize` è la dimensione di un singolo oggetto in byte.
  Equivale ad effettuare `count` invocazioni consecutive di `fputc`, iterando `buffer` a partire dall'inizio.
  Il valore di ritorno `size_t` è il numero di oggetti scritti con successo → potrebbe non coincidere con `count`!

```c
double data[5] = {0.1, 0.2, 0.3, 0.4, 0.5};
FILE *fp = fopen("lab_C.txt", "w");
size_t objs = fwrite(data, sizeof(*data), 5, fp);
// Controllo objs == 5 --> scrittura di 5 double con successo?
// ...
fclose(fp);
```

Analogo ad `fread`

Funzioni per stringhe C utili anche su file

- `int fprintf(FILE *fp, const char *frmStr, ...)`

  Analoga a `printf`, la stringa formattata viene scritta sul file invece che `stdout`. Il valore di ritorno è:
  - Il numero di caratteri trasmessi al file stream → scrittura con successo
  - Un valore negativo → scrittura fallita
- `int fscanf(FILE *fp, const char *fmrStr, ...)`

  Analoga a `scanf`, la stringa viene letta da file invece che `stdin`, ed il parsing effettua i vari assegnamenti. Il valore di ritorno è:
  - Il numero di assegnamenti effettuati con successo → lettura e format match con successo
  - `0` → lettura con successo ma format match fallito prima del primo assegnamento
  - `EOF` → lettura fallita prima del primo assegnamento
- `char fgets(char *dst, int count, FILE *fp)`

  Già vista nella lettura da tastiera, in cui `fp` = `stdin`; legge un massimo di `count - 1` caratteri, che vengono salvati nell'array `dst`. La lettura viene interrotta al primo `'\n'` incontrato oppure se viene raggiunto `EOF`. Se la lettura va a buon fine, '\0' viene scritto nella posizione immediatamente successiva all'ultimo carattere. Il valore di ritorno è:
  - `dst` → lettura con successo
  - `NULL` → lettura fallita

Altre funzioni utili per file

- `int fflush(FILE *fp)`
  Gli oggetti file stream sono internamente bufferizzati per aumentare le prestazioni. Ad esempio: scritture su disco non sono effettuate al momento di invocazione delle funzioni di scrittura bensì in batch alla chiusura del file.
  Questa funzione permette di svuotare il buffer interno di un dato file stream ed applicare le scritture nel momento di invocazione di `fflush` → undefined behavior se il file stream è di tipo input oppure l'ultima operazione è stata una lettura!
  Restituisce `0` se il flush ha successo, altrimenti EOF (anche impostando il *flag di errore* sullo stream).
- `void rewind(FILE *fp)`
  Sposta il cursore all'inizio del file → utile per leggere di nuovo il file dall'inizio una volta raggiunta la fine
- `int remove(const char *filename)`
  Elimina il file identificato dal percorso `filename` → se il file è aperto in contemporanea, dipende dall'implementazione!
  Restituisce `0` se l'eliminazione ha successo, altrimenti un valore `!= 0`.
- `int rename(const char *filename, const char *newFilename)`
  Rinomina il file identificato dal percorso `filename` con un nuovo valore `newFilename` → se `newFilename` già esiste, dipende dall'implementazione!
  Restituisce `0` se la rinominazione ha successo, altrimenti un valore `!= 0`.

2. File I/O in POSIX

API POSIX per C

Portable Operating System Interface (POSIX) è una famiglia di standard IEEE ai fini di garantire compatibilità tra diversi sistemi operativi. Come base degli standard fu scelto Unix, ma data la numerosità di diverse versioni rilevanti, la necessità di un'interfaccia comune era comunque evidente.

Le API POSIX sono librerie per sviluppare programmi con garanzie di portabilità, e includono sia funzioni a livello di sistema (es. *syscall*) che user-level.

Lo sviluppo delle API è avvenuto in contemporanea con la definizione dello standard ANSI C: ci sono quindi molte similitudini nel funzionamento e una limitata compatibilità tra i due standard. Il risultato è l'esistenza di alcune funzioni disponibili nelle API POSIX che tuttavia non sono state incluse in ANSI C.

In un sistema *POSIX-compliant* è possibile compilare ed eseguire programmi C che sfruttano queste API: la maggior parte delle funzioni è dichiarata nella libreria `unistd.h` e, nel caso dei file, anche nella libreria `fcntl.h`.

Gestione file con API POSIX

Diversamente dalla libreria standard C, le API POSIX usano un'altra formalizzazione per interagire con i file.

Al posto dei file stream (`FILE*`), con POSIX i file sono gestiti tramite variabili di tipo `int` che rappresentano il file descriptor. Nei sistemi Unix, un file descriptor è un identificatore univoco all'interno dello stesso processo che può essere associato a file o *altre risorse di I/O*.

Equivalenti POSIX delle funzioni della libreria C (Librerie: `unistd.h` e `fcntl.h`)

- `fopen` → `int open(const char *filename, int flags)`
  Analogamente ad `fopen`, vengono forniti percorso al file e modalità di apertura del file. Il valore restituito è un file descriptor > `0` se l'apertura ha successo, altrimenti `-1` → la variabile `errno` permette di interpretare il tipo di errore.
  In POSIX, le `flags` di apertura possono essere combinate tramite operazioni bitwise.

stdlib C	"r"	"r+"	"w"	"w+"	"a"	"a+"
POSIX	O_RDONLY	O_RDWR	O_WRONLY | O_CREAT | O_TRUNC	O_RDWR | O_CREAT | O_TRUNC	O_WRONLY | O_CREAT | O_APPEND	O_RDWR | O_CREAT | O_APPEND

- `fclose` → `int close(int fd)`
  Chiude un file descriptor → restituisce `0` se ha successo, altrimenti `-1`
  Diversamente da `fclose` i contenuti possono rimanere nel buffer: invocare `fsync` (equivalente di `fflush`)!

Gestione file con API POSIX

Equivalenti POSIX delle funzioni della libreria C (Librerie: `unistd.h` e `fcntl.h`)

- `fread` → `ssize_t read(int fd, void *buf, size_t count)`
  Tenta di leggere `count` byte dal file descriptor fornito, salvandoli nell'array `buf` nello stesso ordine di lettura.
  Il valore di ritorno rappresenta il numero di byte letti con successo; `0` indica la lettura su un file che ha raggiunto `EOF`; `-1` invece indica un errore di lettura.
- `fwrite` → `ssize_t write(int fd, const void *buf, size_t count)`
  Tenta di scrivere `count` byte nel file descriptor fornito contenuti nell'array `buf`, secondo lo stesso ordine.
  Il valore di ritorno rappresenta il numero di byte scritti con successo; `-1` indica un errore di scrittura. Nei casi in cui il ritorno sia minore di `count`, è possibile che il disco sia pieno o che un *segnale di sistema* abbia interrotto la scrittura.
- `rewind` → `off_t lseek(int fd, off_t offset, int whence)`
  Il concetto equivalente al cursore del file in POSIX viene fornito dall'offset: è la distanza in numero di byte dall'inizio del file associato a un file descriptor ed hanno un tipo dedicato `off_t`. La funzione `lseek` permette spostamenti arbitrari e non solo di tornare all'inizio del file (*sarebbe l'equivalente di `fseek` in stdlib C*). Si può ottenere lo stesso risultato di `rewind` con:

```
1 off_t startOffset = lseek(fd, 0, SEEK_SET);
```

- SEEK_SET
- SEEK_CUR
- SEEK_END

Altre funzioni POSIX utili per gestione file

- `FILE* fdopen(int fd, const char *mode)`
  Restituisce un file stream conforme alla libreria standard C a partire da un file descriptor `fd`, precedentemente ottenuto tramite una chiamata a `open`. `mode` rappresenta la modalità di apertura del file come visto nell'uso di `fopen` → deve essere compatibile con le flag di apertura usate per `fd`!
  In caso di errore, restituisce `NULL` pointer e il messaggio di errore può essere ottenuto leggendo la variabile globale `errno`.
  Attenzione: `fd` non va sottoposto a `close`! Il file descriptor viene automaticamente chiuso contestualmente all'invocazione di `fclose` sul file stream ritornato da questa funzione, che quindi diventa necessario.

Gestione directory (Librerie: sys/types.h e dirent.h)

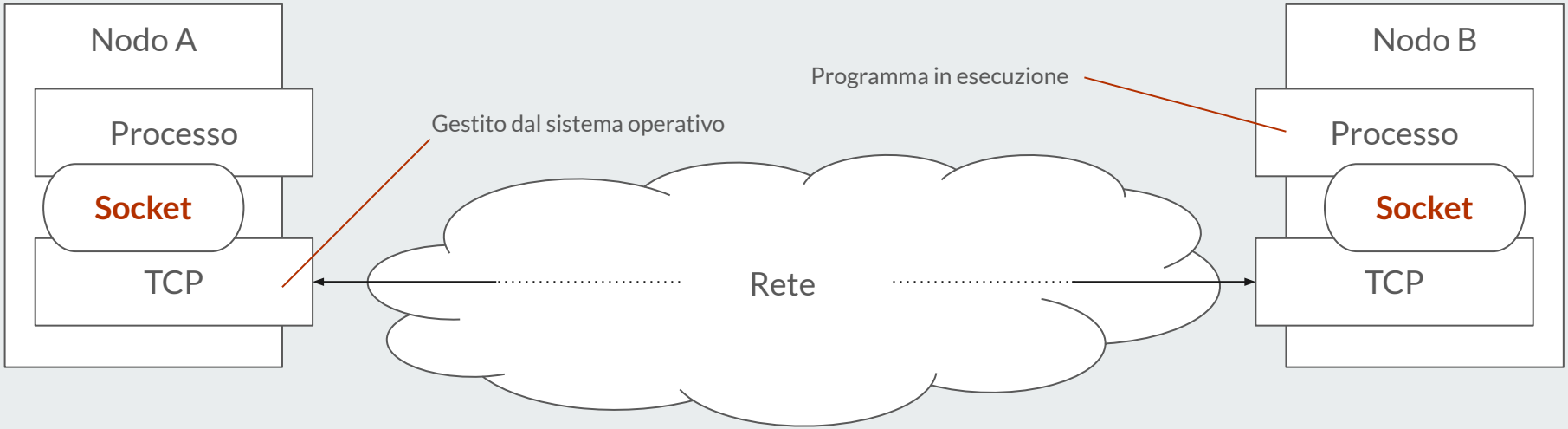
- `DIR* opendir(const char *name)`
  Apre la directory presente al percorso `name` e ne restituisce un directory stream. `NULL` pointer restituito in caso di errori → `errno`
- `struct dirent* readdir(DIR *dirp)`
  Restituisce la prossima directory entry (`struct dirent`, sotto forma di puntatore) presente nel directory stream fornito `dirp`.
  `NULL` pointer viene invece restituito al termine del directory stream o in caso di errore.
- `int closedir(DIR *dirp)`
  Chiude il directory stream `dirp`. Restituisce `0` se l'operazione ha successo, altrimenti `-1` → `errno`.

3. Socket I/O

API Socket

Le API Socket permettono la comunicazione tra diversi processi con un approccio basato su stream, non necessariamente eseguiti su uno stesso nodo.

- Sullo stesso nodo: Socket usati per comunicare tra diversi processi in modo efficiente
  Alternative: file I/O → prestazioni limitate dalle interazioni con il disco
- Su nodi diversi: Socket usati per comunicare all'interno di una rete → anche Internet!



API Socket

In generale i socket vengono utilizzati per gestire la comunicazione tra processi e protocolli:

- Asincroni
- Sincroni

} Bisogna implementare un meccanismo di buffering in uscita/entrata

Usare i socket in C

La libreria standard C non include funzioni relative all'uso dei socket: useremo perciò l'implementazione fornita tramite API POSIX.

Nelle API POSIX sono incluse diverse funzionalità riguardanti i socket:

- Strutture dati specifiche
- Funzioni per apertura, chiusura, e interazione
- System calls

Tipi di socket

Dominio di comunicazione di un socket

Il dominio influenza dove sia possibile la comunicazione (solo locale nello stesso nodo *vs.* locale e remoto) e come sono specificati nomi e indirizzi (filename *vs.* indirizzo IP e porta).

- Dominio UNIX – `AF_UNIX`
  - Solo comunicazione locale tra processi sullo stesso nodo
  - Identificazione basata su nomi file
  - Supporto sia per stream che datagram socket
- Dominio Internet – `AF_INET`
  - Sia per comunicazione locale che attraverso la rete
  - Identificazione basata su indirizzi IP e porte
  - Socket sono nativi per comunicazione di rete

Tipi di socket

- Datagram socket – `SOCK_DGRAM`
- Stream socket – `SOCK_STREAM`
- Raw socket – `SOCK_RAW`

Tipi di socket

Datagram socket

- Anche chiamati socket UDP
- Connection-less
  Non è necessario effettuare un setup di connessione prima della comunicazione
- Consegna best-effort
  Il livello di trasporto ammette la perdita di alcuni *pacchetti* e la consegna non ordinata
- Usi tipici: applicazioni fault-tolerant o con necessità di bassa latenza
  Le applicazioni dovranno implementare ritrasmissioni e/o consegna ordinata
  Esempio: Skype (VoIP)

Stream socket

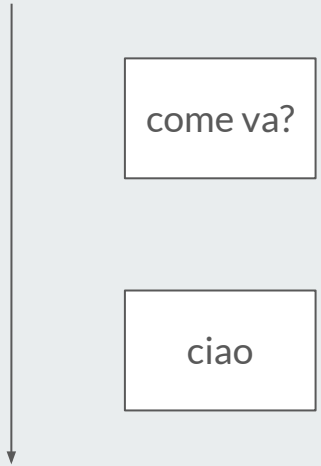
- Anche chiamati socket TCP
- Connection-oriented
  Un setup di connessione con handshake tra i due nodi è necessario
- Consegna affidabile
  Meccanismi di ritrasmissione, controllo del flusso, …
- Consegna ordinata
  I pacchetti mantengono l'ordine di invio
- Usi tipici: applicazioni basate su garanzie di affidabilità e consegna
  Esempi: HTTP, FTP, …

Tipi di socket

Invio di due messaggi: “ciao” “come va?”

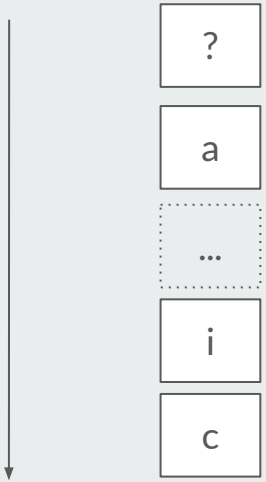
Datagram socket

- Singoli messaggi, indipendenti l’un l’altro



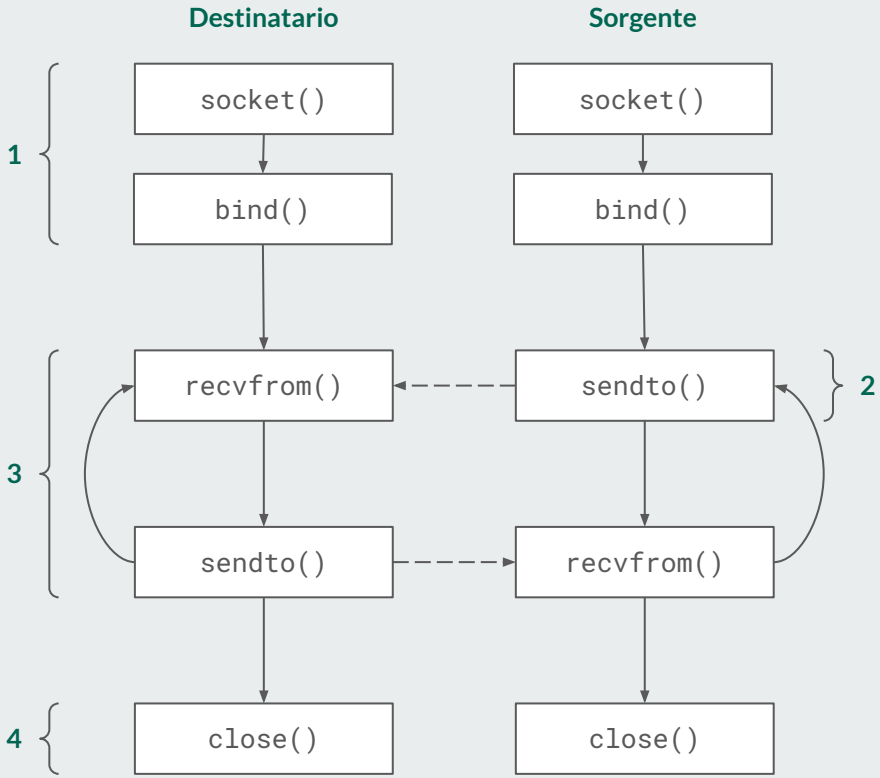
Stream socket

- Unico byte stream, i byte sono correlati tra loro



Ciclo di vita socket UDP

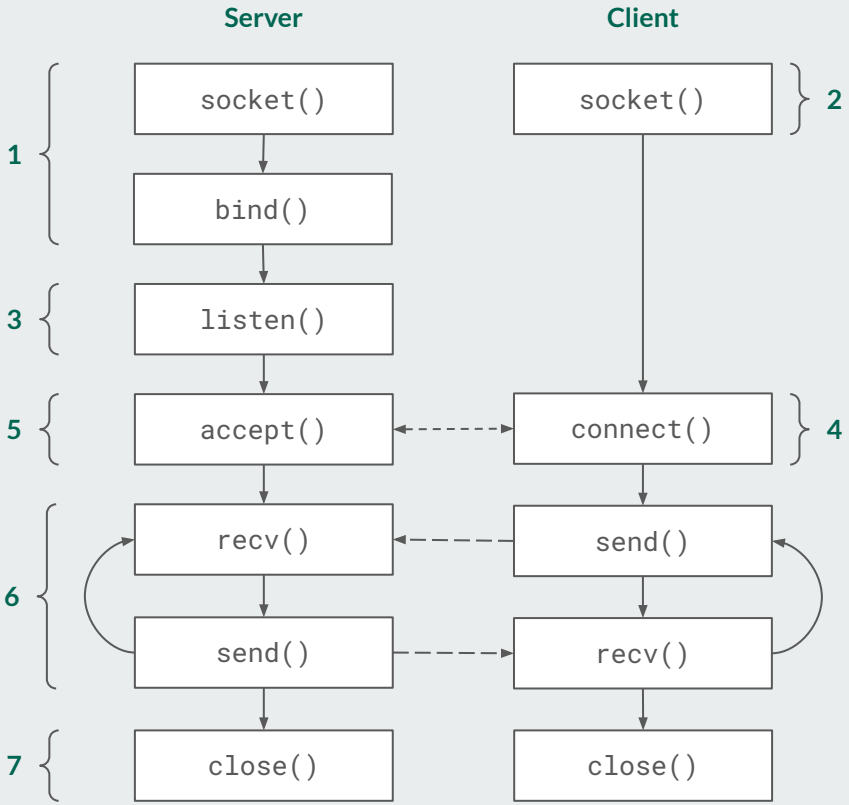
1. Entrambi i processi aprono un'istanza di socket UDP
2. Il nodo sorgente invia dati a una porta specifica sul nodo destinatario
3. Trasmissione dati
4. Chiusura delle istanze socket



Ciclo di vita socket TCP

In un socket TCP il nodo sorgente viene denominato client e il nodo destinatario server.

1. Entrambi i processi aprono un'istanza di socket TCP
2. Setup connessione con una specifica porta del server
3. Il server rimane in attesa di una connessione da un client
4. Il client si connette
5. Il server accetta la connessione
6. Trasmissione dati
7. Chiusura delle istanze socket



Tipi di socket

Raw socket

Permette di utilizzare protocolli definiti dall'utente, interfacciandosi direttamente con i pacchetti IP grezzi che vengono inviati o ricevuti dalla scheda di rete, prima che vengano elaborati dallo stack di rete del sistema operativo, senza l'intermediazione del livello di trasporto.

- Es. Wireshark, tcpdump, ...

Interazione POSIX con i socket

Precedentemente abbiamo visto l'introduzione del concetto di file descriptor tramite le API POSIX, descrivendolo come un identificatore univoco all'interno dello stesso processo che può essere associato a file o altre risorse di I/O.

I socket infatti sono implementati tramite file descriptor, quindi possono essere usate le funzioni POSIX `read` e `write`, ad esempio.

Utilizzo dei socket

Libreria: `sys/socket.h`

- `int socket(int domain, int type, int protocol)`
  Creazione di una nuova istanza di socket; `domain` rappresenta il dominio di comunicazione del socket (`AF_UNIX` o `AF_INET`) e `type` rappresenta il tipo di socket da creare (`SOCK_DGRAM`, `SOCK_STREAM`, o `SOCK_RAW`).
  `protocol` definisce il protocollo da utilizzare con questo socket: con `0` si allinea automaticamente al tipo di socket richiesto.

```c
int sock = socket(AF_UNIX,
                  SOCK_STREAM,
                  0);
```

```c
int sock = socket(AF_INET,
                  SOCK_DGRAM,
                  0);
```

L'indirizzo di un socket è gestito tramite una `struct sockaddr` generica ma i cui membri vengono valorizzati diversamente in base al dominio.

```c
struct sockaddr{u_short sa_family; char sa_data[14];};
```

Generica

AF_UNIX

```c
struct sockaddr_un{
  short sun_family;
  char sun_path[108];
};
```

AF_INET

```c
struct in_addr { unsigned long s_addr;};
struct sockaddr_in {
  short sin_family; u_short sin_port;
  struct in_addr sin_addr; char sin_zero[8];
};
```

Utilizzo dei socket: `server.c`

Librerie relative ai socket:

- `sys/types.h`
  Tipi di dati usati per socket programming
- `sys/socket.h`
  Tipi principali dei socket
- `netinet/in.h`
  Costanti e `struct` per indirizzi Internet

Librerie per `read`/`write` POSIX su socket:

- `unistd.h`
- `fcntl.h`

Gestione buffer in modo sicuro:

- `string.h`
  Funzioni `memset` e `memcpy`, rispettivamente per azzerare e copiare contenuti di buffer (`char*`)

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>

#define BUFSIZE 256
#define PORT 95000

void show_error(const char *msg) {
  perror(msg);
  exit(1);
}
```

Funzione di utilità per segnalazione di errori.
Necessita di `stdlib.h`

Utilizzo dei socket: `server.c`

1. Dichiarazioni per socket di ascolto ed eventuale socket di connessione (`sock_fd` e `new_sock_fd`)
2. `port_num` e `cli_len` rappresentano la porta per le connessioni in ingresso e dimensione dell'indirizzo del client
3. `n` è una variabile di appoggio per la lettura e scrittura tramite socket
4. `buffer` è un array di appoggio per la lettura e scrittura tramite socket
5. `srv_addr` e `cli_addr` rappresentano gli indirizzi del server e del client

```c
int sock_fd, new_sock_fd, port_num, cli_len;
char buffer[BUFSIZE];
struct sockaddr_in srv_addr, cli_addr;
int n;

sock_fd = socket(AF_INET, SOCK_STREAM, 0);
if(0 > sock_fd){
  show_error("Error while opening socket");
}

memset((char*) &srv_addr, 0, sizeof(srv_addr));
port_num = PORT;
srv_addr.sin_family = AF_INET;
srv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
srv_addr.sin_port = htons(port_num);

if(0 > bind(sock_fd, (struct sockaddr *) &srv_addr, sizeof(srv_addr))){
  show_error("Error while binding socket");
}
```

int main

Utilizzo dei socket: `server.c`

6. Apertura del socket di ascolto tramite la funzione `socket`
7. `memset` riempie il buffer con un valore {?} arbitrario (in questo caso `0`): viene inizialmente usato sull'indirizzo del server, assicurandosi che non siano utilizzati membri non inizializzati
8. `srv_addr` viene poi valorizzato con i parametri dell'applicazione
   a. `sin_addr.s_addr` rappresenta l'indirizzo del socket, che va impostato a quello dell'host; questo è ottenuto tramite `INADDR_ANY` {?}
   b. `sin_port` deve essere impostata alla porta desiderata dall'applicazione

```c
int sock_fd, new_sock_fd, port_num, cli_len;
char buffer[BUFSIZE];
struct sockaddr_in srv_addr, cli_addr;
int n;

sock_fd = socket(AF_INET, SOCK_STREAM, 0);
if(0 > sock_fd){
  show_error("Error while opening socket");
}

memset((char*) &srv_addr, 0, sizeof(srv_addr));
port_num = PORT;
srv_addr.sin_family = AF_INET;
srv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
srv_addr.sin_port = htons(port_num);

if(0 > bind(sock_fd, (struct sockaddr *) &srv_addr, sizeof(srv_addr))){
  show_error("Error while binding socket");
}
```

`int main`

9. Syscall `bind` permette di legare un socket ad un indirizzo. Un socket che è stato solamente aperto non è infatti associato ad alcun indirizzo.

`bind` può fallire per diversi motivi, ma il più comune è che ci sia già un socket sulla stessa porta

Utilizzo dei socket: `server.c`

10. Syscall `listen` fornisce un modo per ascoltare le connessioni in ingresso su un dato socket
    Il primo argomento è un file descriptor relativo al socket di interesse, mentre il secondo (`int`) è la dimensione del backlog: massimo numero di connessioni in attesa durante la gestione di una connessione
11. Syscall `accept` permette al server di accettare connessioni in entrata
    Restituisce un nuovo file descriptor di un socket, da utilizzare per le successive comunicazioni su questa connessione. Gli argomenti sono: file descriptor socket di ascolto, puntatore all'indirizzo client, dimensione dell'indirizzo client.
    Può essere usata nei socket di tipo stream
12. `memset` usato per azzerare il `buffer` prima della lettura dal socket di connessione tramite `read`
    `read` ha lo stesso comportamento visto nel caso di file I/O tramite API POSIX

```
 1 listen(sock_fd, 5);
 2
 3 cli_len = sizeof(cli_addr);
 4 new_sock_fd = accept(sock_fd, (struct sockaddr *) &cli_addr, &cli_len);
 5 if(0 > new_sock_fd){
 6   show_error("Error while accepting connection");
 7 }
 8
 9 memset(&buffer, 0, BUFSIZE);
10 n = read(new_sock_fd, buffer, BUFSIZE-1);
11 if(0 > n){
12   show_error("Error reading from socket");
13 }
```

int main

Utilizzo dei socket: `server.c`

13. Viene mostrato il messaggio ricevuto a video
14. Il server invia un messaggio di risposta usando `write` sul socket di connessione
    Analogamente a `read`, `write` mostra lo stesso comportamento visto con file I/O POSIX

```c
printf("Message from client: %s\n", buffer);

char ack_msg[] = "Message received!";
n = write(new_sock_fd, ack_msg, sizeof(ack_msg));
if(0 > n){
  show_error("Error writing to socket");
}

return 0;
```

int main

Utilizzo dei socket: `client.c`

Unica differenza tra le librerie rispetto al server:

- `netdb.h`
  Tipo usato per salvare informazioni riguardo un host remoto e funzioni per ottenere l'indirizzo in formato utile da usare in socket programming

Oltre a `BUFSIZE` e `PORT`, una terza macro rappresentante una costante è definita: `HOSTNAME`

Il valore "`127.0.0.1`" è lo string literal contente il formato IPv4 di `localhost`, ovvero il socket di dominio Internet che useremo implementerà una connessione locale.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>

#define BUFSIZE 256
#define PORT 95000
#define HOSTNAME "127.0.0.1"

void show_error(const char *msg) {
  perror(msg);
  exit(1);
}
```

Funzione di utilità per segnalazione di errori.
Necessita di `stdlib.h`

Utilizzo dei socket: `client.c`

Differenze rispetto al server

1. srv_addr conterrà l'indirizzo del server al quale connettersi
2. `srv` è una variabile di tipo `hostent` (libreria `netdb.h`) utilizzata per contenere informazioni riguardo l'host
3. `srv` è quindi valorizzata tramite la funzione `gethostbyname`, che restituisce le informazioni di un dato host, identificato per *nome*

```
1  int sock_fd, port_num, n;
2  struct sockaddr_in srv_addr;
3  struct hostent *srv;
4  char buffer[BUFSIZE];
5
6  port_num = PORT;
7  sock_fd = socket(AF_INET, SOCK_STREAM, 0);
8  if(0 > sock_fd){
9    show_error("Error while opening socket");
10 }
11
12 srv = gethostbyname(HOSTNAME);
13 if(NULL == srv){
14   fprintf(stderr, "Error unknown host\n");
15   exit(1);
16 }
```

int main

Utilizzo dei socket: `client.c`

Differenze rispetto al server

4. La `struct hostent` restituisce una lista di indirizzi, supportando la possibilità di un host di connettersi a più di una rete contemporaneamente
   Nel nostro caso viene selezionato il primo indirizzo di questa lista (`srv->h_addr_list[0]`) e copiato in `srv_addr.sin_addr.s_addr` usando `memcpy`

```
1 memset((char*) &srv_addr, 0, sizeof(srv_addr));
2 srv_addr.sin_family = AF_INET;
3
4 memcpy((char*) &srv_addr.sin_addr.s_addr,
5        (char*) srv->h_addr_list[0],
6        srv->h_length);
7 srv_addr.sin_port = htons(port_num);
8
9 if(0 > connect(sock_fd, (struct sockaddr *) &srv_addr, sizeof(srv_addr))){
10   show_error("Error while connecting to socket");
11 }
```

`int main`

5. `connect` serve al client per stabilire una connessione con il server
   Gli argomenti sono: il file descriptor del socket aperto, l'indirizzo dell'host remoto a cui connettersi, la dimensione dell'indirizzo host

Utilizzo dei socket: `client.c`

Esempio di utilizzo

- Viene chiesto all'utente di inserire un messaggio da inviare al server tramite `fgets`

```
1 printf("Please enter a message for the server:\n");
2 memset((char*) buffer, 0, BUFSIZE);
3 fgets(buffer, BUFSIZE - 1, stdin);
4
5 n = write(sock_fd, buffer, strlen(buffer));
6 if(0 > n){
7   show_error("Error while writing to socket");
8 }
```

int main

- Leggendo la risposta del server, questa viene mostrata sullo standard output con `printf`

```
1 memset((char*) buffer, 0, strlen(buffer));
2 n = read(sock_fd, buffer, BUFSIZE);
3 if(0 > n){
4   show_error("Error while reading from socket");
5 }
6
7 printf("Message from the server: %s\n", buffer);
8 return 0;
```

Riferimenti

- https://en.cppreference.com/w/c/io
- `fopen`, `fclose`
- `fgetc`, `fputc`, `feof`, `ferror`
- `fread`, `fwrite`
- `fprintf`, `fscanf`, `fgets`
- `fflush`, `rewind`, `remove`, `rename`
- POSIX
- API POSIX non presenti in libreria C
- Gestione file POSIX non standard C (1), Gestione file POSIX non standard C (2)
- `open`, `close`, `fsync`
- `read`, `write`, `lseek`
- `fdopen`, `opendir`, `readdir`, `closedir`
- `socket`, `bind`, `listen`, `accept`, `gethostbyname`, `connect`