Università di Messina - CdL in Informatica - Reti e Sistemi Distribuiti Mod.B

Hands-On #2

Il protocollo TCP ed il problema della frammentazione

Last Update: March 10, 2026

Scenario

Hai scritto nell' HO precedente un server TCP che riceve messaggi da un client e li stampa a schermo. Il codice funziona perfettamente sul tuo laptop. Se però lo testi su due macchine collegate in rete ... inizia a dare risultati strani. A volte i messaggi arrivano incompleti. A volte due messaggi si fondono insieme. A volte il server si blocca senza motivo.

Nessun bug nel tuo codice? In realtà il bug c'è — ed è concettuale.

Questa esercitazione ti aiuterà a capire perché, e a scrivere codice di rete robusto.

Concetti chiave da tenere a mente

TCP non è un protocollo che trasmette messaggi

TCP garantisce che i byte arrivino *nell'ordine giusto* e *senza corruzione*. Non garantisce nulla sulla dimensione dei *blocchi* in cui arrivano. Una singola chiamata `send()` può corrispondere a zero, una o più chiamate `recv()` dall'altra parte.

```
Mittente:  send(1000 byte)
Ricevitore (caso A):  recv → 1000 byte                    // fortunato
Ricevitore (caso B):  recv → 512 byte,  recv → 488 byte
Ricevitore (caso C):  recv → 200 byte,  recv → 600 byte,  recv → 200
byte
Ricevitore (caso D):  recv → 1 byte × 1000                 // valido!
```

Quando scrivi un programma (a livello applicazione) e chiami `send()` infatti, nella tua testa stai "mandando un messaggio". Ma `TCP` non sa nulla di messaggi — lui vede solo una sequenza continua di byte, esattamente come un file su disco. Non c'è nessun separatore, nessun confine, nessuna unità atomica.
Confrontalo con `UDP`: lì ogni `sendto()` produce un datagramma distinto, e dall'altra parte ogni `recvfrom()` restituisce esattamente quel datagramma — intero o niente. `UDP` è un protocollo di messaggi nel senso che preserva i confini. `TCP` non funziona così. Quando chiami `send(sockfd, buf, 100)`, stai dicendo al kernel "aggiungi questi 100 byte al flusso". Il kernel li mette in un buffer (TCP transmit buffer 87KB), poi decide lui quando e come spedirli — magari aspetta altri dati per riempire un segmento, magari li spezza per rispettare la MTU, magari li unisce con la `send()` successiva. Tutto questo è trasparente all'applicazione. Dall'altra parte, `recv()` non sa nulla di tutto ciò. Svuota semplicemente il suo buffer (TCP recv buffer 16KB) di ricezione e ti dà i byte che ci sono in quel momento — che siano 1 o 1000, indipendentemente da come sono stati spediti.

Il valore di ritorno di `recv()` è importante!

```
1 ssize_t r = recv(sockfd, buf, 1000, 0);
2 // r puo' valere: 1000, 512, 1, ... qualsiasi numero da 1 a 1000
```

```
// r == 0    => connessione chiusa dall'altro lato
// r == -1   => errore (controlla errno)
```

Il codice che ignora il valore di ritorno di `recv()` è **sempre sbagliato**.

Il network byte order

Quando trasmetti un intero su rete, macchine diverse possono interpretare i byte in ordine diverso (*little-endian* vs *big-endian*). Le funzioni `htons`/`htonl` e `ntohs`/`ntohl` fanno la conversione verso e dal *network byte order* (big-endian per convenzione).

Funzione	Direzione	Dimensione
`htons`	host → network	16 bit
`htonl`	host → network	32 bit
`ntohs`	network → host	16 bit
`ntohl`	network → host	32 bit

Codice di partenza

Il seguente server presenta il problema di cui abbiamo parlato. Analizza attentamente prima di procedere.

Listing 1: server_broken.c — riesci a trovare i problemi?

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT   8080
#define BUFLEN 2048

int main(void) {
    int server_fd, client_fd;
    struct sockaddr_in addr;
    char buf[BUFLEN];

    server_fd = socket(AF_INET, SOCK_STREAM, 0);

    addr.sin_family      = AF_INET;
    addr.sin_port        = htons(PORT);
    addr.sin_addr.s_addr = INADDR_ANY;

    bind(server_fd, (struct sockaddr *)&addr, sizeof(addr));
    listen(server_fd, 5);

    printf("Server in ascolto su porta %d\n", PORT);
    client_fd = accept(server_fd, NULL, NULL);

    while (1) {
        /* BUG: recv potrebbe restituire meno byte del previsto, il
           buffer potrebbe contenere meno byte di quello richiesti
           dalla recv() */
        int n = recv(client_fd, buf, BUFLEN, 0);
        if (n <= 0) break;

```

```
32          buf[n] = '\0';
33          printf("Ricevuto: %s\n", buf);
34      }
35
36      close(client_fd);
37      close(server_fd);
38      return 0;
39  }
```

Listing 2: client_broken.c

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4   #include <unistd.h>
5   #include <arpa/inet.h>
6
7   #define PORT 8080
8
9   int main(void) {
10      int sockfd;
11      struct sockaddr_in addr;
12
13      sockfd = socket(AF_INET, SOCK_STREAM, 0);
14
15      addr.sin_family = AF_INET;
16      addr.sin_port   = htons(PORT);
17      inet_pton(AF_INET, "127.0.0.1", &addr.sin_addr);
18
19      connect(sockfd, (struct sockaddr *)&addr, sizeof(addr));
20
21      char *messages[] = {
22          "Primo messaggio di testo",
23          "Secondo messaggio, piu lungo del primo",
24          "Terzo"
25      };
26
27      for (int i = 0; i < 3; i++) {
28          /* BUG: send non garantisce di inviare tutti i byte, il
                buffer di invio del kernel potrebbe essere pieno */
29          send(sockfd, messages[i], strlen(messages[i]), 0);
30          usleep(1000);
31      }
32
33      close(sockfd);
34      return 0;
35  }
```

Esercizi

Esercizio 1 — Prova ad immaginare il funzionamento su rete reale ed osserva il problema

1. Dove si trova esattamente il problema della frammentazione nel client e nel server?
2. Perchè avviene in trasmissione?
3. Perchè avviene in ricezione?

Esercizio 2 — Scrivi `recv_all()`

Il tuo obiettivo è scrivere una funzione che **garantisce** di ricevere esattamente `n` byte, indipendentemente da quante chiamate `recv()` siano necessarie.

Listing 3: Prototipo della funzione da implementare

```c
/*
 * Riceve esattamente 'n' byte dal socket 'sockfd' nel buffer 'buf'.
 * Ritorna:
 *   n      se la lettura e' avvenuta con successo
 *   0      se la connessione e' stata chiusa prima di ricevere n byte
 *  -1      in caso di errore (controlla errno)
 */
ssize_t recv_all(int sockfd, void *buf, size_t n);
```

Implementa la funzione, poi scrivine anche una analoga per `send`:

```c
ssize_t send_all(int sockfd, const void *buf, size_t n);
```

▷ Suggerimento

Usa un puntatore "mobile" che avanza nel buffer ad ogni iterazione. Tieni un contatore di quanti byte hai già ricevuto. Il loop termina quando il contatore raggiunge `n`.

[!] Attenzione

Ricorda: `recv()` può restituire un numero di byte inferiore a quello richiesto anche senza errori. Non è un errore — è il comportamento normale.

Esercizio 3 — Il problema della lunghezza

`recv_all()` risolve la frammentazione, ma solo se sai *già* quanti byte aspettarti. In una comunicazione reale i messaggi hanno lunghezze variabili: come fa il ricevitore a sapere quando un messaggio è finito?

Progetta (su carta o in pseudocodice) un **protocollo minimo** client-server in cui ogni messaggio è preceduto da un *header* di lunghezza fissa che indica la dimensione del payload. Lo schema è:

4 byte	N byte
`lunghezza (uint32_t)`	`payload`

Rispondi alle seguenti domande *prima* di scrivere il codice:

a) Il mittente deve mandare prima la lunghezza o prima il payload?

b) Se la lunghezza è `uint32_t` (4 byte), qual è la dimensione massima di un singolo messaggio?

c) Perché è necessario usare `htonl()` sul campo lunghezza prima di inviarlo? Cosa succede se non lo si fa?

▷ **Suggerimento**

Pensa al ricevitore come a qualcuno che non sa nulla di ciò che sta arrivando. L'unica cosa che sa con certezza è la struttura dell'header — ovvero quanti byte leggere *per primo*, e cosa significano.

Esercizio 4 — Implementa il protocollo

Implementa due funzioni:

```c
/* Invia un messaggio con header di lunghezza.
   Ritorna 0 in caso di successo, -1 in caso di errore. */
int send_message(int sockfd, const char *payload, uint32_t len);

/* Riceve un messaggio con header di lunghezza.
   Alloca dinamicamente il buffer e lo restituisce in *out_buf.
   Scrive la lunghezza in *out_len.
   Ritorna 0 in caso di successo, -1 in caso di errore o
      connessione chiusa.
   Il chiamante e' responsabile di fare free(*out_buf). */
int recv_message(int sockfd, char **out_buf, uint32_t *out_len);
```

Modifica il client per inviare i tre messaggi usando `send_message()`, e il server per riceverli con `recv_message()`. Verifica che:

- Ogni messaggio arrivi completo e separato dagli altri.
- Il programma funzioni correttamente anche con messaggi da 64 KB.
- La memoria allocata da `recv_message()` venga liberata.

▷ **Ordine consigliato delle operazioni in `recv_message`**

1. Leggi 4 byte con `recv_all()` nell'header.
2. Converti la lunghezza da network byte order.
3. Alloca `len + 1` byte (il +1 serve per poter aggiungere `\0` se il payload è una stringa).
4. Leggi esattamente `len` byte con `recv_all()`.

Esercizio 5 — Aggiungi il checksum

Estendi il protocollo aggiungendo un byte di checksum **dopo** il payload. Il checksum è lo XOR di tutti i byte del payload (XOR: vedi Kernighan & Ritchie sul sito del docente).

4 byte	N byte	1 byte
lunghezza	payload	checksum (XOR)

Il ricevitore deve:

1. Calcolare lo XOR del payload ricevuto.
2. Confrontarlo con il byte di checksum ricevuto.
3. Stampare un avviso se non coincidono.

Questo esercizio anticipa direttamente la struttura degli header di protocollo reali (IP,

TCP, UDP) che studierete con i socket raw.

Domande di riflessione

Rispondi a queste domande *dopo* aver completato gli esercizi.

Q1. Perché il codice di partenza funzionava sempre in locale ma potrebbe fallire su rete reale?

Q2. Qual è la differenza tra il valore di ritorno di `recv()` uguale a `0` e uguale a `-1`?

Q3. Se il campo lunghezza dell'header fosse `uint16_t` (2 byte) invece di `uint32_t`, quale sarebbe la dimensione massima del payload?

Q4. Il protocollo che hai implementato è adatto a trasferire file binari (immagini, eseguibili)? Perché sì o perché no?

Consegna: inviare zip con report e sorgenti al docente (roberto.marino@unime.it)
Prossimo passo: Sei pronto per costruire strutture analoghe direttamente sui pacchetti IP — ed è esattamente quello che farai con i socket raw.