1 Lezione del 05-11-25

Riprendiamo la trattazione della comunicazione fra processi, in particolare riferendoci al livello transport implementato alternativamente dai protocolli *TCP* e *UDP*.

1.1 Multiplexing

Abbiamo detto che l'obiettivo è implementare la comunicazione fra *processi*, a determinate *porte* (che sono astrazioni costruite al livello transport).

Abbiamo che se si vuole implementare questa trasmissione per *porte*, bisogna prevedere un passo di **multiplexing** al client dei messaggi di tutte le applicazioni (e quindi tutte le porte attive), e di **demultiplexing** al server dei messaggi provenienti verso tutte le applicazioni (e quindi tutte le porte attive).

1.1.1 Demultiplexing connectionless

Possiamo parlare di demultiplexing **connectionless** quando il protocollo non ha concetto di "connessione" (si pensi a UDP). In questo caso, l'host riceve datagrammi IP che contengono:

- Indirizzi IP sorgente e destinazione;
- Il segmento livello transport (nel payload);
- Numeri di porta sorgente e destinazione, contenuto nell'header del segmento di livello transport.

Quello che fa l'host è usare l'indirizzo IP e il numero di porta per dirigere il segmento verso il socket corrispondente (cioè il socket che è assaciato alla porta destinataria).

1.1.2 Demultiplexing connection-oriented

Nel caso di protocolli che invece hanno concetto di connessione (si pensi a TCP), si parla di demultiplexing **connection-oriented**.

Un socket TCP è identificato da più informazioni rispetto al semplice socket connectionless. In particolare, si adota una 4-upla:

1. Indirizzo IP sorgente;
2. Indirizzo IP destinatario;
3. Numero di porta sorgente;
4. Numero di porta destinatario.

Il vantaggio di questo approccio è che possiamo associare più socket alla stessa porta, in quanto la connessione è identificata anche dai dati provenienti dal client.

Questo è il classico esempio dei server TCP, che prevedono un socket di *ascolto* (magari il socket alla porta 80 nei web server), e poi tutta una serie di socket, individuati dalla 4-upla vista sopra, che gestiscono i singoli client. Chiaramente, i singoli client hanno prima parlato col socket di ascolto per ricevere un socket alla loro 4-upla.

1.2 UDP

Veniamo quindi alla trattazione vera e propria del protocollo **UDP** (*User Datagram Protocol*).

Ricordiamo che questo era un protocollo semplice, privo di servizi aggiuntivi, e basato sul modello *best service* di servizio. Questo significa che i pacchetti possono essere persi, o consegnati fuori ordine all'applicazione.

Inoltre, caratteristica fondamentale del protocollo UDP è che è *connectionless*: non c'è alcun concetto di connessione fra trasmettitore e ricevitore, nessun tipo di handshaking prima o dopo la comunicazione, e ogni segmento UDP viene considerato come a sé stante.

I vantaggi dell'UDP sono:

- L'estrema semplicità e velocità: il RTT di handshaking è inesistente, e il supporto software al ricevitore e al trasmettitore può essere estremamente semplice (non bisogna mantenere informazioni di stato);
- La dimensione ridotta degli header;
- Non esiste controllo di congestione e di flusso: oltre a risparmiare tempo sui controlli, si ha che in alcuni casi UDP può funzionare anche nonostante la congestione.

Le applicazioni di UDP comprendono il DNS, SNMP, e anche protocolli RDT come HTTP/3 (che implementa il suo livello di RDT, detto QUIC). Inoltre, UDP è largamente usato in applicazioni ad alte prestazioni come lo streaming e i videogiochi.

1.2.1 Segmento UDP

L'header di segmento UDP è estremamente semplice:

```
16 bit         16 bit
<source port> <destination port>  % header
<length>      <checksum>
<payload> (lunghezza variabile)   % payload
```

- Abbiamo l'indicazione del **numero di porta** e l'**indirizzo** destinatario, usati per il demultiplexing;
- La **lunghezza** del payload, cioè il messaggio di livello application, che può essere variabile;
- Un campo di **checksum** per l'intero segmento.

  Questo è calcolato prenendo i contenuti del segmento come sequenze di numeri su 16 bit e sommandoli bit a bit in complemento a 1. Il checksum è memorizzato in complemento a 1 nel campo `<checksum>`: quello che fa il ricevitore e ricalcolare il checksum e confrontarlo con quello nel pacchetto (fare la AND).

  Se risultano uguali, tutto è (apparentemente) a posto. Altrimenti il segmento è corrotto.
- Segue il **payload** vero e propri.

1.3 Protocollo TCP

Veniamo quindi alla discussione del protocollo **TCP** (*Transmission Control Protocol*).

Questo è un protocollo *punto-punto*, cioè astrae un canale di comunicazione (*pipe*) fra due host specifici, trasmettitore e ricevitore.

Implementa anche il concetto di *byte stream*, cioè un flusso di byte ordinato e affidabile (non si possono perdere byte).

Infine, ricordiamo che è *full duplex*, cioè permette la comunicazione bidirezionale sulla stessa connessione (trasmettitore e ricevitore possono scambiarsi di ruolo contemporaneamente sulla stessa connessione).

Come abbiamo anticipato, la dimensione massima di un segmento TCP su protocollo datalink di tipo Ethernet è 1460 byte, in quanto si perdono 20 byte di header datagramma IP e 20 byte di header segmento TCP (lo vedremo fra poco).

Notiamo inoltre che TCP supporta gli *ACK cumulativi* sui segmenti inviati, capacità di *pipelining*, e supporto al controllo di *flusso* e *congestione*.

1.3.1 Segmento TCP

Il segmento TCP è visibilmente più complesso del segmento UDP:

```
16 bit                          16 bit
<source port>                   <destination port>  % header
<sequence number>
<acknowledgement number>
<length> <inutilizzato> <flags> <receive window>
<checksum>                      <urgent data pointer>
<options> (lunghezza variabile)
<payload> (lunghezza variabile)                     % payload
```

- Immancabili sono il **numero di porta** e l'**indirizzo** destinatario, situati nella stessa posizione del segmento UDP;
- Segue il **numero di sequenza**, su 32 bit.

  Notiamo la particolarità che, essendo TCP orientato al byte, questo indicizza byte anziché segmenti.
- Il **numero di acknowledge** indica il numero di sequenza del prossimo byte aspettato: in congiunzione al flag A, segnala che il segmento è di ACK.
- Il campo `<length>` è riferito alla **lunghezza** del solo header TCP;
- Il campo dei **flag** prevede diversi bit, che sono:
  - Bit **C** ed **E**, pensati per notificare le congestioni (oggi di fatto inutilizzati);
  - Bit **U**, pensato per segnalare dati urgenti (oggi di fatto inutilizzato);
  - Bit **A**, già nominato, segnala che il segmento è di ACK;
  - Bit **P**, sempre legato ai dati urgenti (oggi di fatto inutilizzato);
  - Bit **R** (*RST*), **S** (*SYN*) e **F** (*FIN*), usati per la gestione della connessione.
- Il campo **receive window** è utile al flow control, in quanto sepcifica il numero di byte che il ricevitore è pronto ad accettare. In questo modo si riescono ad evitare buffer overflow quando si trasmettono più byte di quanti il ricevitore è capace di memorizzare;

- Prevediamo un campo **checksum** come in TCP;
- L'`<urgent data pointer>` è legato alla gestione di dati urgenti (come i bit U e P), oggi è perlopiù inutilizzato;
- Segue, come in IP, una sezione di **opzioni** TCP a lunghezza variabile;
- Infine c'è il **payload** vero e proprio che verrà trasmesso al socket TCP.

1.3.2 Numeri di sequenza TCP

Abbiamo detto che i **numeri di sequenza** in TCP sono orientati al byte (e non al segmento).

Quelli che gli host vogliono rendere disponibili alle applicazioni sono stream di byte. Il campo numero di sequenza di un segmento TCP contiene l'indice del primo byte contenuto nel campo payload del segmento stesso.

Avevamo visto in 11.2 come i byte (o comunque le unità di informazione minima che vogliamo spedire, al tempo le avevamo chiamate pacchetti) possono trovarsi, all'interno di una pipeline, in uno di 4 stati:

1. Inviati e seguiti da ACK del destinatario, quindi sostanzialmente "dimenticati" sia da ricevitore che da trasmettitore;
2. Inviati e non ancora seguiti da ACK, anche detti *in-flight*;
3. Utilizzabili ma non ancora inviati dal mittente;
4. Non ancora utilizzabili dal mittente.

Per inviare informazioni riguardo alla pipeline, usiamo i campi *numero di sequenza* e *numero di acknowledgement* del segmento TCP. In particolare, un segmento dal mittente che contiene un certo numero di sequenza segnala che i byte ivi contenuti stanno passando dallo stato 3 (non ancora spediti) a 2 (appena spediti). In risposta, un segmento di ACK (quindi con bit A alzato e numero di acknowledgement impostato) rappresenta una transizione da stato 3 (inviato e senza ACK) a stato 2 (inviato e con ACK).

In altre parole, possiamo dire che il *numero di sequenza* punta all'inizio della sezione **3** come vista dal mittente, mentre il *numero di acknowledgement* punta all'inizio della sezione **2** come vista dal destinatario.

1.3.3 Gestione connessione TCP

Veniamo quindi a come TCP implementa la **gestione delle connessione**. Iniziamo col considerare la fase di **apertura**.

Vediamo del tipico codice per creare una connessione fra client e server attraverso i classici socket di BSD:

- Lato client:

```
// creiamo un socket
int sc = socket(AF_INET, SOCK_STREAM, 0);
// richiediamo la connessione del socket all'indirizzo server
int sts = connect(sd, (struct sockaddr*) &serv_addr, sizeof(serv_addr
    ));
```

- Lato server:

```c
// creiamo un socket di ascolto
int sl = socket(AF_INET, SOCK_STREAM, 0);
// leghiamo il socket di ascolto all'indirizzo del server
bind(sl, (struct sockaddr*) &serv_addr, sizeof(serv_addr));
// mettiamo in ascolto il socket
listen(sl, 10);
// accettiamo una connessione TCP col client dal socket di ascolto
int sc = accept(sl, NULL, NULL);
```

Quello che due host che eseguono questo codice mettono in atto in fase di connessione è il cosiddetto **handshake** a 3-vie:

1. Inizia il client che si trova nel cosiddetto stato di LISTEN, che invia un messaggio TCP SYN al server. Il numero di sequenza viene scelto in maniera arbitraria. Con questa operazione il client transisce a stato SYNSENT;

2. Il server, anch'esso in stato di LISTEN risponde con un messaggio SYNACK, cioè l'ACK del messaggio SYN del client. Il numero di acknowledgement è impostato per seguire quello inviato dal client. Un altro numero di sequenza arbitrario è fornito dal server, e con questa operazione il server transisce a stato SYN RCVD;

3. Il client risponde con un ultimo ACK per il SYN ACK. Il segmento che contiene questo ACK potrebbe contenere dati per il server. Inoltre, il numero di acknowledgement è impostato per seguire quello inviato dal server. Con questa operazione, il client transisce a stato di ESTAB.

   Ad ogni modo, questo ultimo ACK segnala al server che il client è vivo e pronto a trasmettere, e quindi di poter transire allo stato ESTAB.

I numeri di sequenza vengono scelti in maniera casuale per disambiguare, sul socket di ascolto, fra più richieste di connessione TCP concorrenti.

Veniamo all'inevitabile (a meno di errori) fase di **chiusura** della connessione. Questa è più complessa dell'apertura, in quanto l'ultimo ricevitore potrebbe avere dei dati ancora da leggere in fase di chiusura.

Il procedimento è quindi simile al seguente, ponendo che sia il client a richiedere la chiusura della connessione (non si riporta codice in quanto consiste semplicemente in due `close()` sui descrittori di socket):

1. Il client richiede la chiusura attraverso un segmento FIN;

2. Il server riceve il segmento FIN e risponde con un ACK.

   Chiude quindi la connessione dal suo lato, inviando a sua volta un FIN;

3. Il client riceve il FIN del server, ed entra in una fase di *timed wait*, dove risponderà con ACK ad eventuali FIN;

4. Il server riceve l'ACK del cliente, e la connessione è effettivamente chiusa.

Interroghiamoci sui dettagli di questo processo. Abbiamo che in fase di creazione della connessione, sia server che client devono allocare risorse necessarie alla gestione della connessione (TCP è *stateful*, per cui richiede informazioni allocate sugli host). Queste informazioni dovrebbero ragionevolmente essere deallocate in fase di chiusura della connessione.

Questo però non è esattamente il caso: come abbiamo accennato, se l'host che chiude la connessione elimina subito i suoi descrittori, potrebbe verificarsi una situazione dove non è più in grado di fare ACK ad eventuali ultime richieste dell'altro host (mettiamo che il suo ACK si perde, ecc...). Per questo prevediamo una fase di timed wait (passo 3 nella sequenza appena vista), dove l'host che ha richiesto la chiusura aspetta, dopo la richiesta di chiusura dell'altro host, in modo da poter fare ACK ad eventuali richieste.