Reti

Matteo Genovese

September 2024

Contents

1 Reti di calcolatori e Internet

1.1 Cos'è internet?

Internet è una rete di calcolatori connesse tra di loro

1.1.1 Definizioni varie

host *o sistemi periferici* sistema terminale della rete, coloui che svolge operazioni.

rete di collegamenti
o *communication link* è un tipo di connessione per collegare gli host.

commutatori di pacchetti
o *packet switch* è un altro tipo di connessione per collegare gli host.

velocità di trasmissione
o *transimission rate* velocità con cui i vari tipi di collegamenti si scambiano dati, misurata in **bit/secondo (bps)**.

pacchetto
o packet divisione del dato in sottoparti (**segmento**) che gli host si devono scambiare con l'aggiunta di intestazione.

asd

1.2 Ai confini della rete

1.2.1 Le reti di accesso

1.2.2 Mezzi trasmissivi

1.3 Il nucleo della rete - importante

1.3.1 Commutazione di pacchetto

1.3.2 Commutazione di circuito

1.3.3 Una rete di reti

1.4 Ritardi, perdite e throughput nelle reti a commutazione di pacchetto - importante

1.4.1 Panoramica del ritardo nelle reti a commutazione di pacchetto

1.4.2 Ritardo di accodamento e perdita di pacchetti

1.4.3 Ritardo end-to-end

1.4.4 Throughput nelle reti di calcolatori

1.5 Livelli dei protocolli e loro modelli di servizio - importante

1.5.1 Architettura a livelli

1.5.2 Incapsulamento

2 Livello di applicazione

2.1 Princìpi delle applicazioni di rete

2.1.1 Architetture delle applicazioni di rete

Ci sono due tipi di architettura: **client-server** o **P2P (peer-to-peer)**.
L'**architettura client-server** si basa su un *server* che sarà sempre attivo, con un indirizzo IP statico e sarà connesso con altri server, mentre il *client* è colui che comunica con il server, non succederà mai che il server proverà a contattare il client, avranno degli indirizzi IP dinamici. Costi alti per via dell'installazione e manutenzione. Nel caso in cui il server cada e non sarà più raggiungibile tutta la rete cadrà, quindi abbiamo un **singolo punto di fallimento** che è il server.
L'**architettura P2P pura** si basa sulla comunicazione tra i vari *client*, non esiste un *host* sempre attivo. È un'architettura scalabile ma difficile da gestire per la sicurezza (tutti gli host devono essere protetti adeguamente, se uno solo non è protetto tutta la rete è insicura) visto che manca un punto centrale di controllo.
Esiste un'architettura **Ibrida** quindi un mix di architettura client-server e P2P. Il server serve come mezzo di ricerca, i peer mandano una richiesta al server che la inoltra agli altri peer, mettendo poi in comunicazione i peer nella modalità P2P.

2.1.2 Processi comunicanti

- **Processo**: programma in esecuzione su un host, più processi comunicano tramite **schemi interprocesso** e processi su host differenti comunicano tramite scambio di mesasggi.
- **Processo client**: processo che dà inizio alla comunicazione.
- **Processo server**: processo che attende di essere contattato.
- **Socket**: Il processo comunica tramite un socket, equiparabile a una porta, mette in comunicazione il livello del processo applicativo e il livello trasporto.
- **API**: Application Programming Interface, definisce come un'applicazione accede ai servizi di trasporto.

  Le applicazioni con architettura P2P hanno sia processi client che server.

Il progettista di rete può scegliere il protocollo di trasporto e alcuni parametri a livello di trasporto.

Per identificare il processo ricevemente bisogna avere due informazioni:

- **Indirizzo dell'host**: spcificati dal loro **indirizzo IP**, un numero di 32 bit che identfica univocamente l'host.
- **Identificatore del processo ricevente sull'host di destinazione**: la sua **socket**, il **numero di porta di destinazione** svolge questo compito.

2.1.3 Servizi di trasporto disponibili per le applicazioni

2.1.4 Servizi di trasporto offerti da Internet

Ci sono a disposizione due protocolli di trasporto per le applicazioni di internet:

- **TCP**: prevede una connessione e un trasporto affidabile dei dati.
  Servizio orientato alla connessione (connection-oriented service): fa in modo che client e server si scambino informazioni di controllo a livello di trasporto prima che i messaggi a livello di appllicazione comincino a fluire, questa procedura è denominata **handshaking**, pre-allerta client e server per lo scambio di messaggi.
  Dopo la fase di *handshaking* si dice che esiste una **connessione TCP** tra le socket di client e host, una connessione di tipo *full-duplex* (i processi possono scambiare messaggi contemporaneamente). *Servizio di trasferimento affidabile (reliable data transfer service)*: il protocollo TCP garantisce ai processi il flusso di dati senza errori.
  TCP evita la congestione della rete, strozzando il flusso quando è eccessivo.

- **UDP**: protocollo minimale, è "senza connessione" (non ha bisogno di *handshaking*), ciò non garantisce l'affidabile della connessione. Non ha un meccanismo di gestione della congestione, manda il flusso di dati al livello di rete a qualsiasi velocità.

2.1.5 Protocolli a livello di applicazione

2.1.6 Applicazioni di rete trattate in questo libro

2.2 Web e HTTP

Il **Web** è *on demand*, ciò significa che si può avere quello che si vuole quando si vuole.

2.2.1 Panoramica di HTTP

HTTP (*HyperText Transfer Protocol*) è un protocollo a livello di applicazione che utilizza **TCP** come protocollo di trasporto, implementato a due programmi, client e server. **Pagina web**: documento costituito da oggetti, un **oggetto** è un file indirizzabile tramite un **URL**. In un *URL* ci sono due componenti: il nome dell'host e il percorso dell'oggetto. I **browser** implementano il protocollo *HTTP* lato client. I **server web** implementano il protocollo *HTTP* lato server (Apache è un esempio), sarà sempre attivo con un indirizzo IP statico. *HTTP* è un protocollo **senza memoria di stato** (stateless protocol).

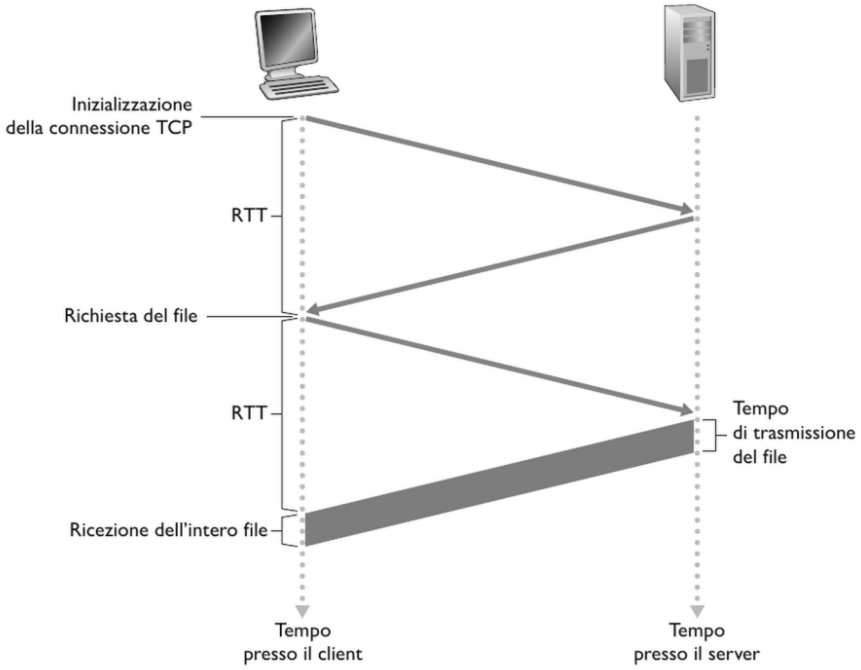
2.2.2 Connesioni persistenti e non persistenti

Si possono creare due tipi di connessioni:

- **non persistenti**: ogni coppia richiesta e risposta deve essere inviata su una connessione TCP *separata*, deve essere unica per ogni coppia.
- **persistenti**: tutte le comunicazioni sono mandate sulla stessa connessione TCP (scelta di default per HTTP).

HTTP con connessioni non persistenti

Nelle *connessioni non persistenti TCP* ogni connessione viene chiusa dopo l'invio dell'oggetto da parte del server, quindi ogni connessione trasporterà un solo messaggio di richiesta e solo uno di risposta. Esistono browser che possono aprire più connessioni TCP parallelamente per velocizzare.
Round-trip time (RTT): tempo impiegato da un pacchetto per viaggiare dal client al server e tornare al client. Include i ritardi di propagazione, di accodamento nei router e nei commutatori intermedi e di elaborazione del pacchetto.



Quando un utente clicca su un collegamento ipertestuale il browser inizializza una connessione TCP con il web server, iniziando così un **handshake a tre vie** (three-way handshake): il client invia un piccolo segmento TCP al server e il server manda una conferma sempre con un piccolo segmento TCP, il cliente manda una conferma di ritorno al server. Con queste prime due operazioni di handshake a tre vie calcoliamo il *RTT*.
Il client ora invia un messaggio di richiesta HTTP insieme alla conferma di avvenuta ricezione (*acknowledgement - ACK*), a messaggio ricevuto dal server il server procede a inviare il file al client. La richiesta-risposta consuma un altro

RTT. Il tempo di risposta totale sarà di 2 RTT più tempo di trasmissione del file dal server al client.

HTTP con connessioni persistenti

Nelle connessioni persistenti il server lascia aperta la connessione dopo la prima coppia di richiesta-risposta col client, tutte le altre coppie verranno trasmesse sulla stessa connessione. Una delle caratteristiche di queste connessioni è il *pipelining*, la capacità di poter effettuare delle richieste senza aspettare la risposta delle richieste in corso. La connessione si chiuderà dopo un lasso di tempo configurato in cui la connessione è stata inattiva.

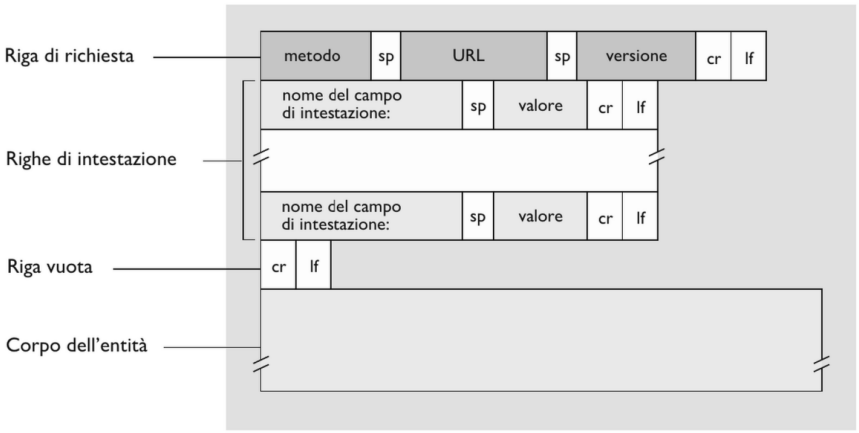
2.2.3 Formato dei messaggi HTTP

Esistono due formati di messaggi HTTP per richiesta e risposta.

Messaggio di richiesta HTTP

```
GET /somedir/page.html HTTP/1.1
Host: www.someurl.com
Connection: close
User-agent: Mozilla/5.0
Accept-language: fr
```

Questa è una richiesta HTTP, può avere un numero indefinito di righe, la riga fondamentale è la prima che è la **riga di richiesta**, le succesive sono **righe di intestazione**. La riga di richiesta ha 3 campi: metodo (GET, POST, DATA, PUT, DELETE), l'URL e la versione di HTTP.
GET è il metodo più utilizzato nel web, si usa per richiedere un oggetto tramite l'URL. Notiamo la riga "Connection: close", con questa riga il browser comunica al server che non si deve occupare di connessioni eprsistenti, deve chiudere la connessione dopo aver iniviato l'oggetto.
Alla fine del richiesta troviamo un *corpo* del messaggio, vuoto in caso di metodo GET, utilizzato in caso di metodo POST.
Il metodo **POST** viene utilizzato per mandare form compilati dell'utente al server, si può utilizzare anche il metodo GET per questo scopo ma includendo questi dati nell'URL della pagina richiesta.
Il metodo **HEAD** viene utilizzato dagli sviluppatori, è come il metodo *GET* ma si riceve solo la risposta HTTP, senza ricevere l'oggetto.
Il metodo **PUT** viene utilizzato per caricare dal client dei file sul server.
Il metodo **DELETE** viene utilizzato per cancellare file sul server (spesso disabilitato per ragioni di sicurezza insieme al metodo PUT).

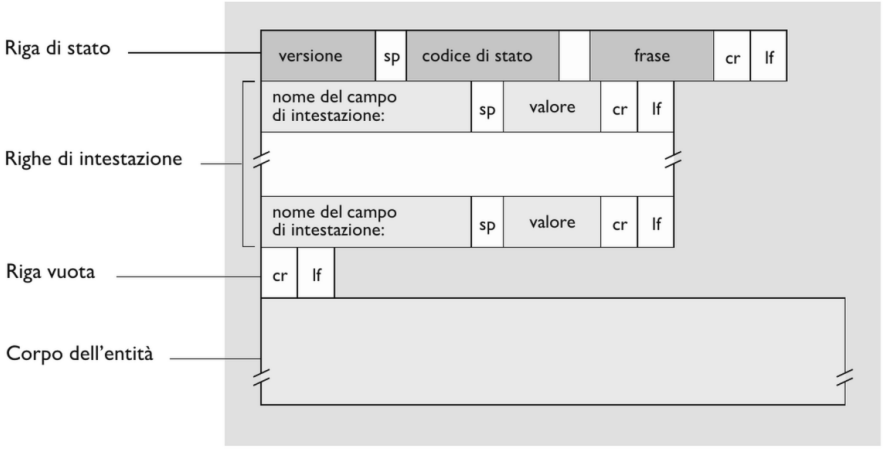


Messaggio di risposta HTTP

HTTP/1.1 200 OK Connection: close
Date: Thu, 18 Aug 2015 15:44:04 GMT
Server: Apache/2.2.3 (CentOS)
Last-Modified: Tue, 18 Aug 2015 15:11:03 GMT
Content-Lenght: 6821
Content-Type: text/html
(data data data data ....)

Abbiamo: una **riga di stato** iniziale (contenente 3 campi, versione di protocollo, codice di stato HTTP e il corrispettivo messaggio), sei **righe di intestazione** e il **corpo dell'oggetto** finale (fulcro del messaggio, contiene l'oggetto richiesto).
Esistono vari codici di stato, questi i più comuni:

- **200 - OK**: la richiesta ha avuto successo e in risposta si invia l'informazione.
- **301 - Moved Permanently**: l'oggetto richiesto è stato trasferito in modo permanente; il nuovo URL è specificato nell'intestazione Location: del messaggio di risposta. Il client recupererà automaticamente il nuovo URL.
- **400 - Bad Request**: si tratta di un codice di errore generico che indica che la richiesta non è stata compresa dal server.
- **404 - Not Found**: il documento richiesto non esiste sul server.
- **505 - HTTP Version Not Supported**: il server non dispone della versione di protocollo HTTP richiesta.



2.2.4 Cookie

HTTP è un protocollo *stateless*, un elemento utile per i webserver sono i **cookie**, un identificativo per l'utente che mantiene le informazioni sul server, per esempio l'*autenticazione*. È formato da 4 componenti, tra cui:

- Riga di intestazione nel messaggio di risposta HTTP (Set-cookie: numero identificativo)
- Riga di intestazione nel messaggio di richeista HTTP (Cookie: numero identificativo)
- File cookie, mantenuto sul sistema terminale dell'utente e gestito dal browser del client
- Database sul webserver che mantiene l'identificativo dei cookie

  I cookie possono anche essere usati per creare un livello di sessione utente al di sopra di HTTP che è privo di stato.

2.2.5 Web caching (proxy server)

2.3 Posta elettronica

Mezzo di comunicazione asincrono, tre componenti principali: gli **user agent** (o agenti utente), i **mail server** (server di posta) e il **protocollo SMTP (Simple Mail Transfer Protocol)**. L'*user agent* invia il messaggio al proprio *mail server* (il distributore del servizio) che invierà la mail al *mail server* del destinatario. Componenti della posta elettronica:

- **casella di posta**: contenitore dei messaggi in arrivo, collocata in un *mail server*.

- **Coda di messaggi**: mail che devono arrivare al destinatario.
- **Protocollo SMTP** (Simple Mail Transfer Protocol): regola la comunicazione tra i *mail server* e tra gli *user agent* e i proprio *mail server*. Principale protocollo a livello di applicazione per la posta elettronica, utilizza *TCP*.

  > Quando un server invia posta a un altro, agisce come client SMTP; quando invece la riceve, funziona come server SMTP.

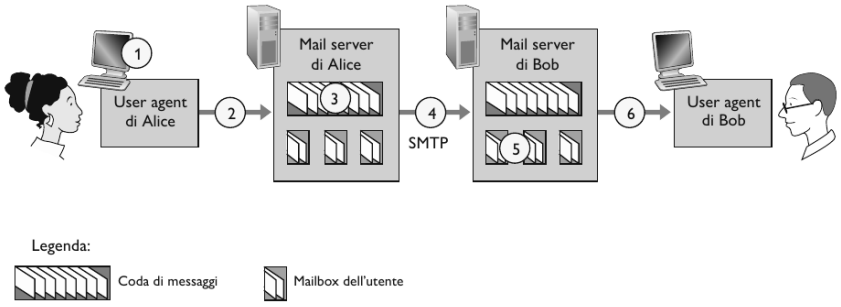
2.3.1 SMTP

Utilizzo di TCP, utilizza la **porta 25**, è un *protocollo con stato*, codificato in ASCII-7bit.
Si prevedono 3 fasi diverse:

- Prima fase: handshake
- Seconda fase: scambio effettivo dei messaggi
- Terza fase: chiusura della connessione

Scenario di funzionamento del protocollo

1. L'user agent mittente compone l'indirizzo mail del destinatario
2. L'user agent invia il messaggio al mail server del mittente
3. Il mail server mittente apre una connessione TCP con il mail server del destinatario sulla porta 25
4. Il client SMTP (mail server mittente) invia il messaggio tramite la connessione TCP e lo colloca nella coda dei messaggi
5. Il mail server del destinatorio invia il messaggio nella mail box
6. L'user agent destinatario può ora leggere il messaggio



Si utilizzano dei mail server poiché nel caso in cui le mail non possono essere consegnate in un preciso momento o ci sono errori, il mail server può fare vari tentativi (in base alla configurazione fatta) per consegnare il messaggio.

continuo su tablet, pc scarico. inserire schema + appunti.

2.3.2 Protocolli di accesso alla posta

2.4 DNS

Il **DNS** è un protocollo, ma in realtà è un sistema, *Domain Name System*. Il web è identificato da un nome simbolico, il nome dell'host, ma il modo univoco per identificare il server è il suo indirizzo ip (32 Byte). È un sistema distribuito, esistono vari server che conoscono l'associazione fra nome host e indirizzo IP.
È un protocollo a livello di applicazione, i vari servizi che offre sono:

- Traduzione degli hostname in indirizzi IP.
- Host aliasing, più nomi per lo stesso indirizzo ip (stesso host). Esiste un **nome canonico** della macchina che la identifica, esistono anche degli **alias** che sono altri nomi per la stessa macchina.
- Possibilità di gestire la posta elettronica in server diversi ma con lo stesso hostname (esempio mail@unipa.it e unipa.it).
- Possibilità di distribuire il carico, più macchine possono gestire lo stesso server, ci saranno pìù server (e più indirizzi IP) per un unico hostname.

Non dobbiamo centralizzare il DNS in un unico server per evitare un *single point of secure*, non è scalabile, scarsa manutenzione e difficoltà nel gestire il volume di traffico.

2.4.1 Gestione gerarchica DNS

Abbaimo un sistema gerarchico, più in alto abbiamo i **server radice**, sotto i **server TLD (top-level domain)** e infine i server **autoritativi (o di competenza)**.
I *server autoritativi* sono i server che posseggono le traduzioni, sono di società che posseggono host Internet, devono fornire i record DNS di pubblico dominio che mappano i nomi di tali host in indirizzi IP.
I *server TLD* sono responsabili dei domini ad alto livello (.com, .co.uk, .it...), vengono gestiti da nazioni o da aziende.
I *server radice* responsabile di tutto. Sono 13 nel mondo (numero limitato), verrà contattato da un **DNS locale**
Il client contatterà sempre il server radice, che darà indicazioni su dove trovare i server TLD interessati che diranno al client qual'è il server autoritativo responsabile per l'hostname scelto.

2.4.2 DNS locale

Una macchina fuori la gerarchia, che si occuperà di fare da client per le comunicazioni da il client reale e l'host radice, funzionamento di intermediario. Ogni ISP ha un **DNS locale** e lui opera da proxy, inoltra la query in una gerarchia di server D e lui opera da proxy, inoltra la query in una gerarchia di server DNS. Approccio con **expiring date** per il mantenimento delle informazioni.

Esempio di query iterativa

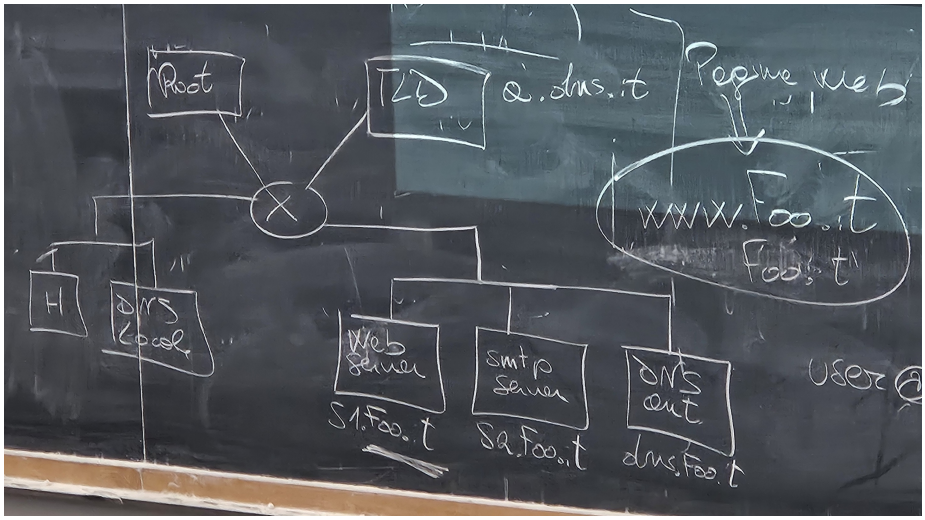
1. Il **client web richiedente** chiede al **client DNS proprio** (chiamata API del sistema) di tradurre un hostname.
2. Il **client DNS** parla col **server DNS locale**.
3. Il **server DNS locale** richiede informazioni al **server DNS radice**.
4. Il **server DNS radice** da informazioni sul **server TLD** che avrà l'informazione richiesta al **server DNS locale**.
5. Il **server DNS locale** richiede informazioni al **server DNS TLD**.
6. Il **server DNS TLD** darà informazioni sul **server di competenza** che conterrà l'informazione che ci interessa al **server DNS locale**.
7. Il **server DNS locale** richiede informazioni al **server di competenza**.
8. Il **server di competenza** darà l'IP corrispondente dell'hostname al **server DNS locale**.
9. Il **server DNS locale**, ricevendo l'informazione dal **server di competenza**, la manda al **client dns richidenete** che trasferirà l'informazione al **client web richiedente** che potrà accedere ora al **server web richiesto**.

2.4.3 Record DNS

Formato **RR (Record di risorsa)**: (name, value, type, ttl).
4 tipi di type:

- **Type=A**: name = nome host, value = indirizzo IP
- **Type=NS**: name = dominio, value = nome del server di competenza
- **Type=CNAME (canonical name)**: name = *nome alias* di un nome *nome canonico*, value = *nome canonico*
- **Type=MX**: value = nome del server di posta di *name*, name = nome server

Esempio popolazione DB server



DB del server Radice

Name	Value	Type
.it	a.dns.it	NS
a.dns.it	22.4.9.10	A

DB del server TLD

Name	Value	Type
foo.it	dns.foo.it	NS
dns.foo.it	147.163.2.1	A

DB del server DNS autoritativo

Name	Value	Type
www.foo.it	s1.foo.it	CNAME
foo.it	s1.foo.it	CNAME
s1.foo.it	143.163.2.2	A
foo.it	s2.foo.it	MX
s2.foo.it	143.163.2.3	A

2.4.4 Messaggi DNS

Il **protocollo DNS** mantiene lo stesso formato per le domande (query) e messaggi di risposta.

3 Livello di trasporto

3.1 Introduzione e servizi a livello di trasporto

Strumento che instaura una **connessione logica** tra i processi applicativi dei vari host, *non è un collegamento fisico*. Il **livello di trasporto** è in esecuzione sui **sistemi terminali**, non sui router.
Durante l'invio scinde i messaggi in vari segmenti, passandoli al livello inferiore (livello di rete), mentre durante la ricezione riassembla il messaggio.
Il **livello di rete** mette in comunicazione logica gli host.
Il **livello di trasporto** è un **deamon** che mette in comunicazione logica i processi, usa i servizi del livello di rete, aspetta dal livello applicativo il messaggio da inviare e a chi inviarlo.

3.1.1 Protocolli utilizzati

I due protocolli utilizzati sono **TCP** e **UDP**. Il protocollo TCP offre vari servizi, tra cui: controllo di congestione e controllo di flusso, questo garantirà affidabilità ma avrà ritardi nel trasporto.
Il protocollo UDP essendo non orientato alla connessione è meno affidabile, non avendo nemmno i controlli garantiti dal TCP, ma guadagna in velocità, non garantendo il corretto ordine di ricezione dei pacchetti e la ricezione di tutti i pacchetti.
Entrambi i protocolli non garantiscono servizi di *garanzia su ritardi* (visto che non possiamo prevedere il tempo di accodamento, il paccehtto potrebbe perdersi ed essere infinito bloccando tutta la connessione) e *garanzia su ampiezza di banda*.

3.2 Multiplexing e demultiplexing

L'operazione di **multiplexing** è l'operazione di invio, prende i dati da inviare dai vari processi, incapsula il paccehtto con l'intestazione e invia il pacchetto, prende un pacchetto e lo divide in sottopacchetti. L'operazione di **demultiplexing** è l'operazione di ricezione, prende i vari pacchetti, li ricompatta con le indicazioni dell'intestazioni e li manda alla **socket** (canale di comunicazione virtuale tra *livello applicazione* e *livello di trasporto*) corretta.
Nell'intestazione a livello di trasporto abbiamo bisogno di minimo: l'etichetta numerica della socket associata ai processi di mittente e destinatario, quindi il numero porta d'origine e di destinazione, il resto dei campi dipende dal protocollo scelto.
I protocolli **standard** hanno delle porte precise, motivo per cui già sappiamo qual è la porta di destinazione. L'host usa gli indirizzi IP e i numeri di porta per inviare i vari segmenti.

3.2.1 demultiplexing senza connessione (UDP)

Crea le socket con il numero di porta d'origine e di il numero di porta di destinazione.
(da completare)

3.2.2 demultiplexing orientato alla connessione (TCP)

Crea le socket indentificata da 2 parametri sia per host mittente sia per host destinatario, un host può supportare più socket TCP contemporaneamente.
Si possono creare thread web per gestire le socket, ogni thread del processo originale gestisce un client con una socket. La socket di benvenuto è la socket del server che attende la connessione dei vari client, una volta stabilita la connessione il client crea una socket con i suoi dati e i dati del server, infine il server crea una socket corretta con i dati suoi e del client, stabilendo la connessione tra i due host sulla socket appena creata.
(da completare)

3.3 Trasporto senza connessione: UDP

Protocollo senza connessione, i segmenti (NON SONO DATAGRAMMI) UDP (User Datagram Protocol) possono essere perduti o consegnati in ordine errato. L'intestazione UDP è formato da numero porta orgine, numero porta di destinazione, lunghezza in byte del segmento UDP con intestazione e il checksum (aggiunge bit alla fine per controllare viene corrotto il pacchetto), tutti tasselli da 16 bit, totale di 8 Byte. UDP viene usato nei protocolli **DNS** e **SNMP**, viene utilizzato nelle applicazioni multimediali.

Checksum UDP

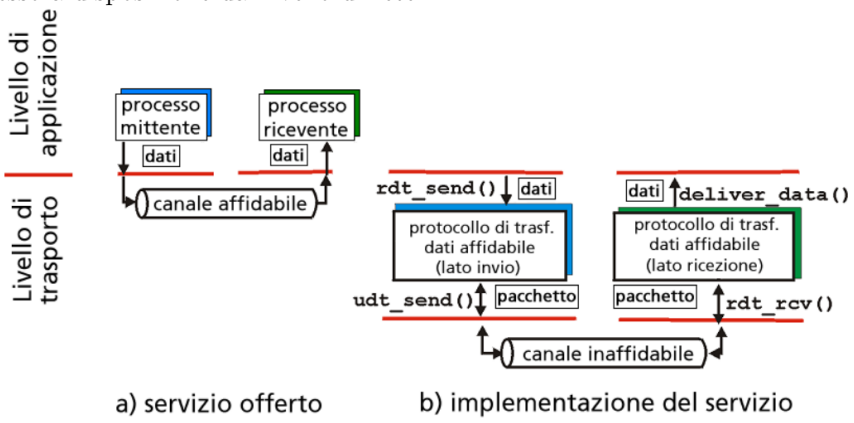
Server a rilevare gli errori nel segmento trasmesso, controlla se ci sono bit alternati nella checksum confrontandolo con il checksum prima del trasporto.
Operazioni del mittente:

- Somma tutte le parole (tutti i campi presenti nel campo UDP, compreso di intestazione) tradotte in binari, nel caso in cui ci sia un riporto (17 bit) lo sommo al bit meno significativo, quindi il primo bit viene sommato al diciasettesimo bit, così che ora il pacchetto è lungo 16 bit.
- La checksum è il complemento a 1 della somma (gli 1 diventano 0 e gli 0 diventano 1)
- Il client calcola il checksum e lo mette nell'intestazione
- L'host calcola il checksum e lo controlla con quello dentro l'intestazione, se rileva una discrepanza scarta il pacchetto

Errori multipli possono annullare bit corrotti, essendo somma binaria, se due bit opposti si corrompono il risultato non cambia.

3.4 Principi del trasferimento dati affidabile

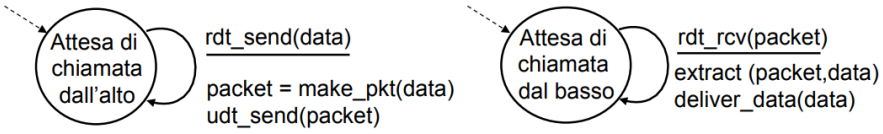
Il servizio che offre il livello di trasporto è di un **canale affidabile**, ma l'implementazione del servizio utilizza un **canale inaffidabile** realizzato dal **livello di rete**, il livello di trasporto deve realizzare il collegamento e rendere affidabile il canale messo a disposizione dal livello di rete.



Rdt1.0: Mondo ideale

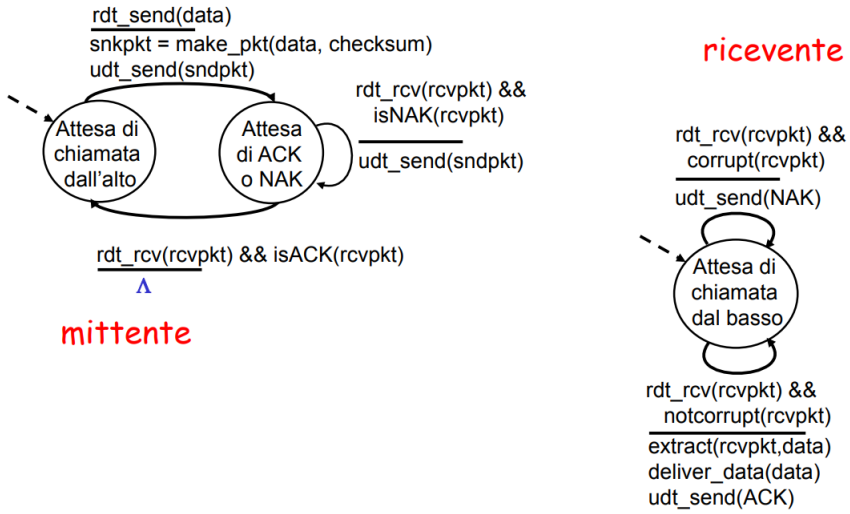
In un mondo ideale dove il livello rete offre un **canale affidabile**, il livello di trasporto esegue solo queste operazioni:

- L'host riceve i dati da inviare e il destinatario dal livello applicativo
- Crea i pacchetti da inviare
- Invia i dati al destinatario tramite il livello di rete
- ...
- Il client riceve i dati dal livello di rete
- Estrae i dati
- Invia i dati al livello applicativo



Rdt2.0: canale con errori nei bit

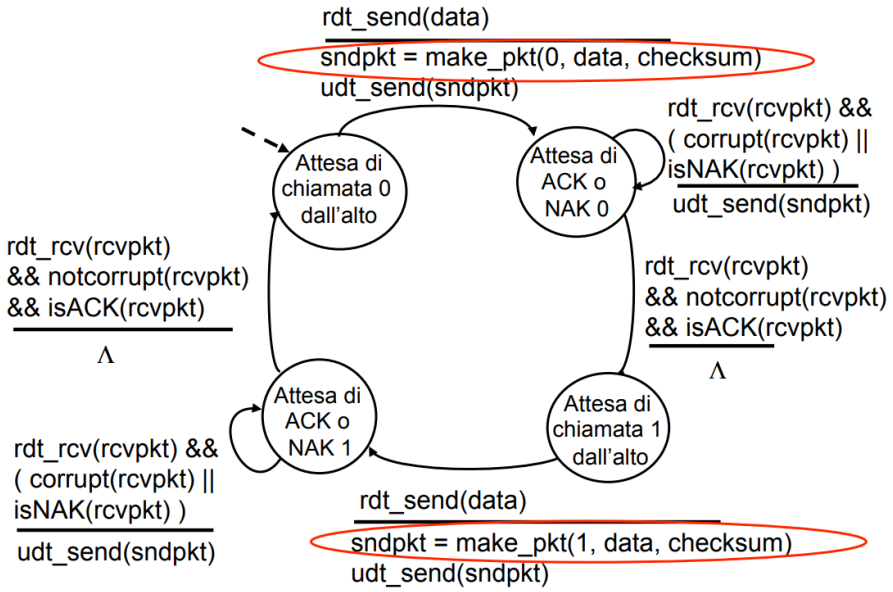
Il *livello di trasporto* riceve i dati dal *livello applicativo*, crea i pacchetti e aggiunge all'intestazione la checksum a ogni pacchetto. Invia il pacchetto al destinatario che ricalcolerà il checksum e controllerà se è corretto, nel caso in cui sia corretto manderà un messaggio di **ACK** (conferma di ricezione) al mittente e manderà il pacchetto al *livello applicativo* del destinatario altrimenti manderà un messaggio di **NAK** (notifica pacchetto corrotto) al mittente che dovrà rimandare lo stesso pacchetto prima di procedere a inviare i restanti.



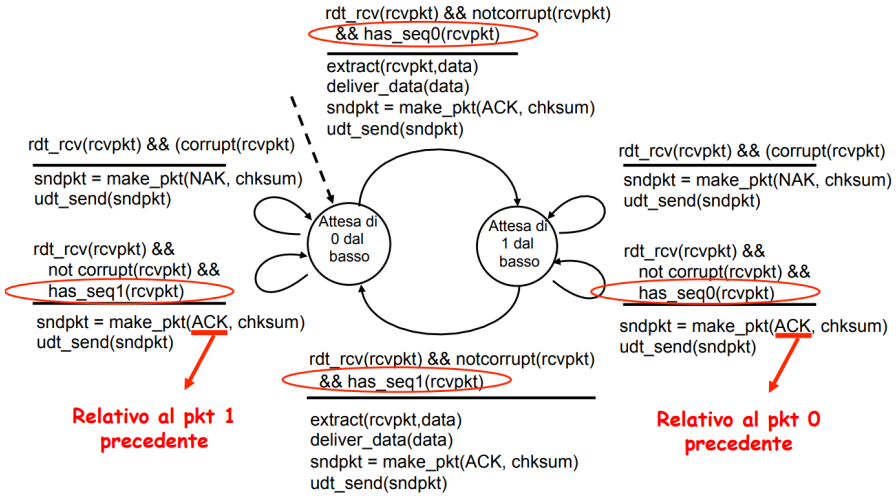
Rdt2.1: il mittente gestisce gli ACK o NAK alterati

Un problema di questo metodo è il caso in cui i pacchetti di ACK o NAK vengano corrotti, quindi il mittente non sa risposta corretta del destinatario, ritrasmettere è un opzione ma si possono essere dei **duplicati**. Dobbiamo risolvere il problema dei *duplicati*, aggiungiamo il **numero di sequenza** a ogni pacchetto, quindi il ricevente scarterà il pacchetto duplicato nel caso in cui il mittente rimandi lo stesso capito anche avendo mandando un ACK avendo già memorizzato il *numero di sequenza*.

Schema automa a stati finiti mittente:

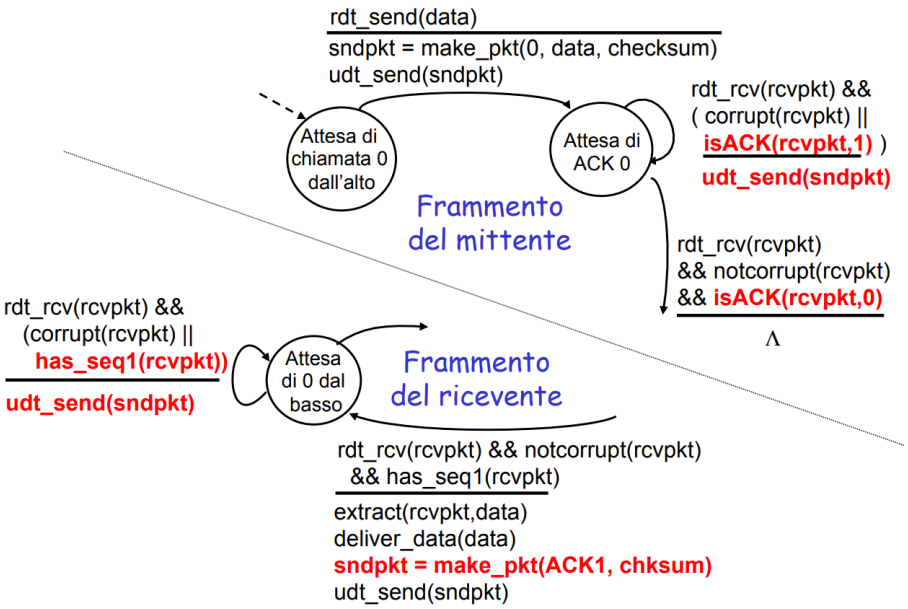


Schema automa a stati finiti ricevente:



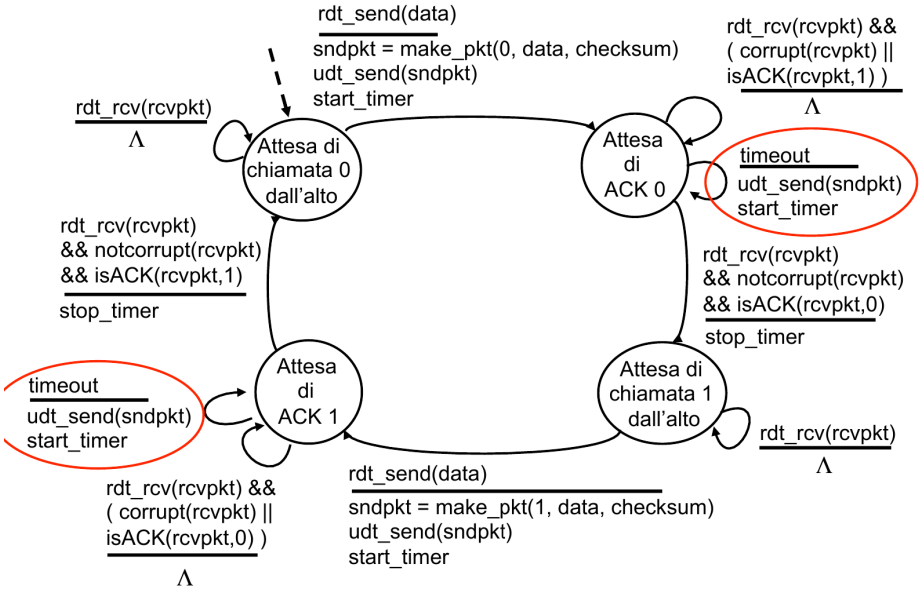
RDT2.2: Protocollo senza NAK

Si utilizza il numero di sequenza opposto al numero di sequenza del pacchetto che stiamo visualizzando come *NAK*, se inivio il pacchetto con **numero di sequenza = 0** e il destinatario non capisce, manderà come messaggio un **ACK con numero di sequenza 1**, darà al mittente un ACK con un altro numero di sequenza come *NAK*.



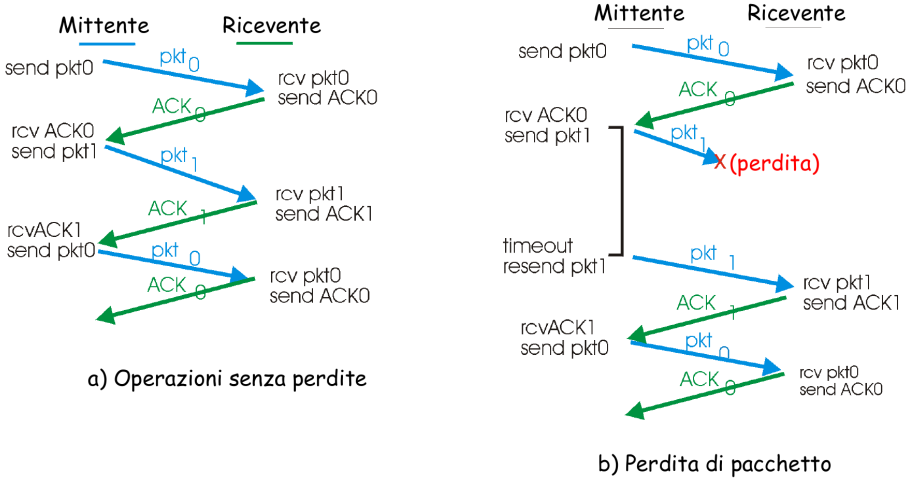
RDT3.0: canali con errori e perdite

Si aggiunge un timer di attesa per la ricezione di un **ACK**, così in caso di pacchetto perso il mittente ritrasmetterà il pacchetto, nel caso in cui sia solo il ritardo il mittente invierà il pacchetto ma il duplicato verrà gestito tramite i numeri di sequenza.

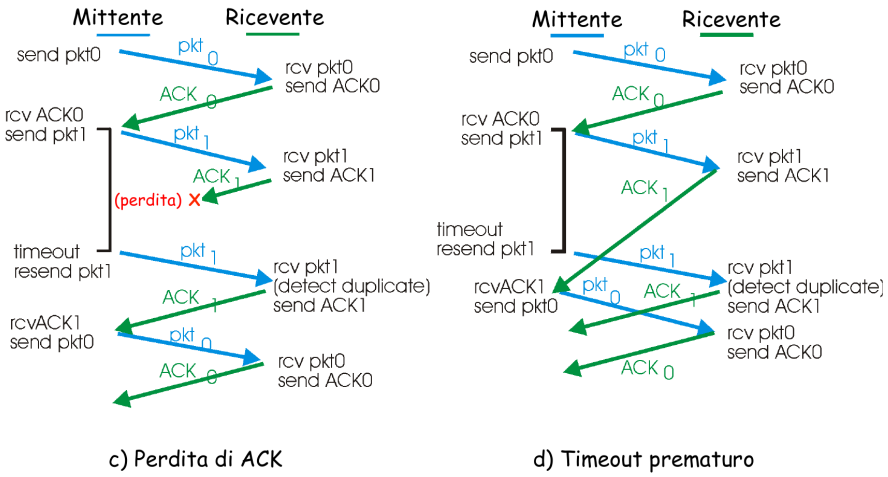


l'automa a stati finiti del ricevente è uguale a quello del **RTD2.2**

RDT3.0: Perdita di pacchetto



RDT3.0: Perdita di ACK



Il **RDT3.0** utilizza un algoritmo **STOP and WAIT** ma è molto lento, utilizziamo le **pipeline** per velocizzare il sistema.

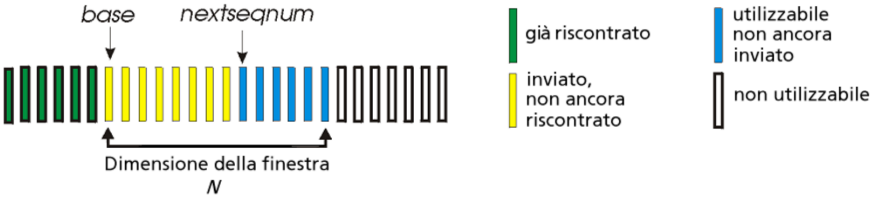
3.4.1 Protocolli con pipeline

Utilizzeremo due tipi di meccanismi, sono opposti come filosofia

Go-back-N

- Il mittente può avere fino a N pacchetti senza ACK in pipeline
- IL ricevente invia solo **ACK cumulativi**, non dà l'ACK di un pacchetto se c'è un gap
- Il mittente ha un timer per il più vecchio pacchetto senza ACK, se scade il time ritrasmette tutti i pacchetti senza ACK

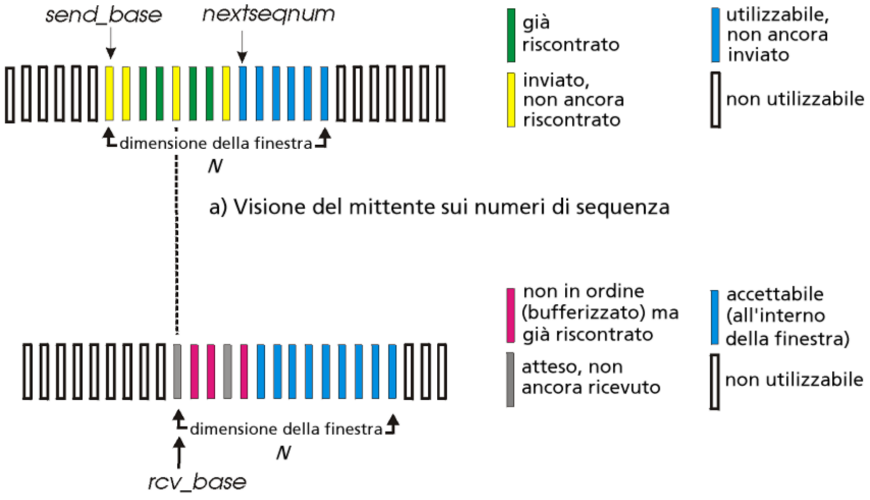
L'ACK sarà con il numero di sequenza dell'ultimo pacchetto arrivato correttamente e in ordine, il mittente può continuare a mandare altri pacchetti, ma verranno rifiutati dal destinatario che manderà l'ACK con l'ultimo pacchetto arrivato oridinato.



La finestra contiene N pacchetti inviati di cui ancora non è arrivato un riscontro.
nextseqnum: prossimo pacchetto da inviare

Selective repeat

asd

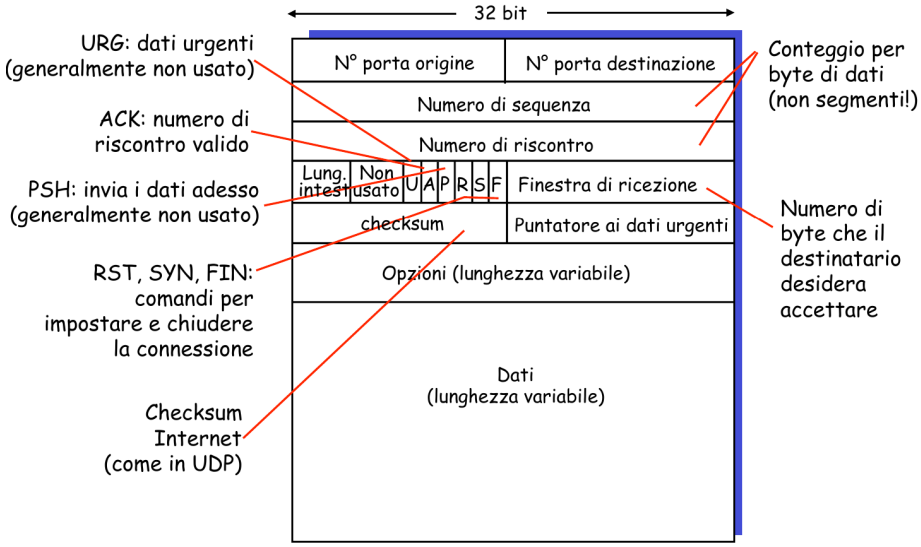


THROUGHPUT: tasso di occupazione medio con cui i dati vengono trasmessi sul collegamento, rapporto tra tutti i dati trasmessi (anche più volte)
GOODPUT: tasso con cui il livello applicativo di destinazione vede arrivare i dati utili, rapporto tra i dati utili e il tempo di trasmissione
Il *throughput* è sempre maggiore del *goodput*, solo nel caso ideale saranno uguale.

3.5 TCP: trasporto orientato alla connessione

È un tipo di connesione **punto-punto**, tra mittente e destinatrio, **full duplex**, abbiamo un flusso di dati bidirezionale, e **orientato alla connessione**, *handshaking a tre vie*, ha un flusso di byte affidabile, arrivano nella sequenza corretta. I dati vengono mandati in **pipeline**, attraverso un meccanismo *sliding window*, abbiamo un **buffer d'inivio** e un **buffer di ricezione**, così che i paccehtti che arrivano fuori ordine vengono conservati e riordinati successivamente.

3.5.1 Struttura dei segmenti



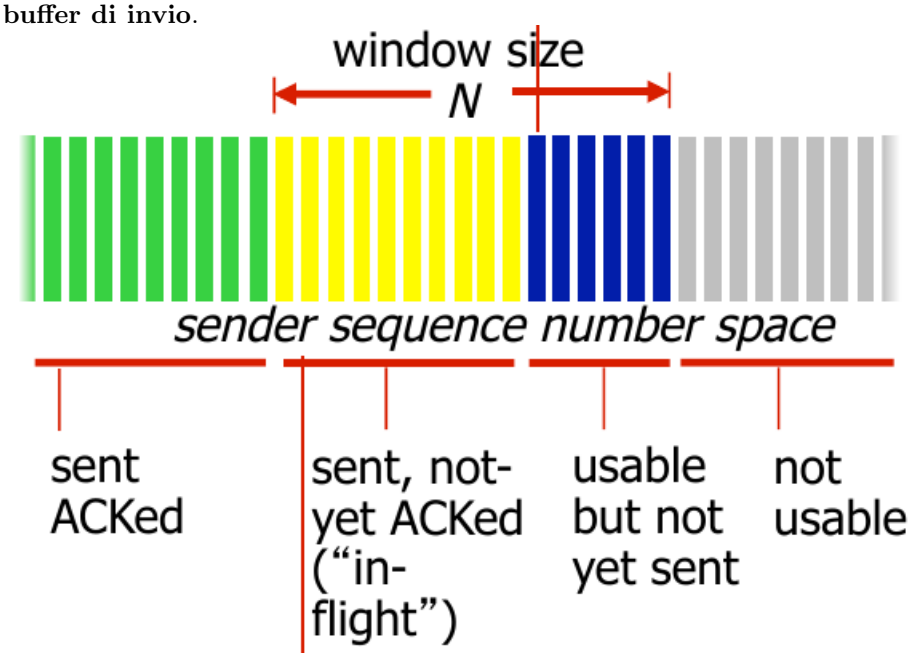
L'OVERHEAD minimo del *TCP* è di **20 Byte**.

- Prima riga - uguale alla struttura del protocollo *UDP*, serve per il multiplexing.
- Seconda riga - **Numero di sequenza**: Posizione del primo Byte all'interno del payload (= MSS, Maximum Segment Size)
- Terza riga - **Numero di riscontro**: Il numero di riscontro sarà il primo byte del segmento successivo a quello arrivato

- Quarta riga - **BIT DI FLAG**: U (dati urgenti), A (**ACK**), P (PUSH), R (RST), S (SYN), F (FIN), gli ultimi tre sono comandi per impostare e chiudere la connessione
- Quarta riga - **Lunghezza intestazione**: nel protocollo UDP è sempre 8 byte, qui no, verrà specificato pacchetto per pacchetto
- Quarta riga - **Finestra di ricezione**: Da non confondere con la *sliding window*, dice quanto spazio si ha a disposizione per ricevere i dati
- Quinta riga - **checksum**: grandezza di 16 bit, calcolata come la checksum dell'UDP
- Quinta riga - **Puntatore ai dati urgenti**: Nel caso in cui la flag U sia attiva ci sarà il puntatore alla memoria per quei dati
- Sesta riga - **Opzioni**: varie ed eventuali

3.5.2 Gestione numeri di sequenza e riscontro del TCP

Il *numero di sequenza* è il primo byte del segmento nel payload, dipende dal mittente e da come gestisce la memoria del proprio **buffer di invio**.
Il *numero di riscontro* utilizza un *ACK cumulativo*, sarà il numero del prossimo dato che vuole ricevere in ordine, quindi sarà il primo byte del segmento che vorrà ricevere, quindi del successivo all'utltimo correttamente ricevuto e immagazzinato, dipende dal destinatario e da come gestisce la memoria del proprio **buffer di invio**.



3.5.3 Gestione del timer nel TCP

Il problema principale è stimare correttamente la durata del timer utilizzando la **media mobile esponenziale ponderata**. La formula è la seguente:

$$\text{EstimatedRTT}(t) = (1 - \alpha) \cdot \text{EstimatedRTT}(t-1) + \alpha \cdot \text{SampleRTT}(t)$$

dove solitamente $\alpha = 0.125$.
Bisogna calcolare la *deviazione standard* del RTT:

$$\text{DevRTT}(t) = (1 - \beta) \cdot \text{DevRTT}(t-1) + \beta \cdot |\text{SampleRTT} - \text{EstimatedRTT}|$$

dove solitamente $\beta = 0.25$.

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \cdot \text{DevRTT}$$

3.5.4 Trasferimento dati affidabile del TCP

Eventi del mittente

Crea un segmento con il numero di sequenza, sarà il primo byte del segmento nel payload, avvia il timer. In caso di timeout o di ACK duplicati, il protocollo TCP ritrasmetterà il pacchetto, riavvando il timer. Controlla gli ACK ricevuti, aggiorno ciò che è stato ricevuto e avvio il time nel caso in cui dovessi completare segmenti già inviati.

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum
loop (sempre) {
  switch (evento)

  evento: i dati ricevuti dall'applicazione superiore
    creano il segmento TCP con numero di sequenza
       NextSeqNum
    if (il timer attualmente non funziona)
      avvia il timer
    passa il segmento a IP
    NextSeqNum = NextSeqNum + lunghezza(dati)

  evento: timeout del timer
    ritrasmetti il segmento non ancora riscontrato con
        il piu' piccolo numero di sequenza

  evento: ACK ricevuto con valore del campo ACK y
    if (y > SendBase) {
      SendBase = y
      if (esistono timer non attualmente riscontrati)
        avvia timer
    }
} /* fine loop */
```

Quando si ritrasmette il pacchetto, il prodotocollo raddoppia il tempo di timeout al riavvio, nel caso di altra ritrassmissione raddoppierà il valore dell'ultimo timeout usato (quindi già raddoppiato).

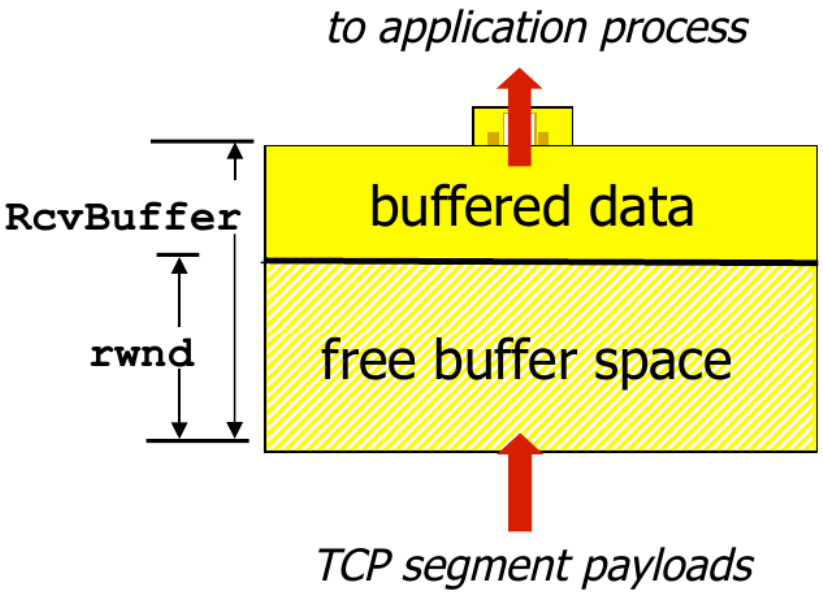
Algoritmo della ritrassmissione rapida

Quando si arriva a 3 ACK duplicati si effettua una ritrasmissione rapida, prima che scada il timer, il timer non viene spento. Nel caso in cui nel frattempo sono arrivati i pacchetti successivi, il destinatario, al momento in cui riceverà il pacchetto che era andato perso e per cui ha mandato 3 ACK duplicati, invierà l'ACK del successivo dell'ultimo pacchetto ricevuto (sono stati bufferizzati nel mentre che aspettava quel pacchetto).

```
evento:  ACK ricevuto, con valore del campo ACK pari a
    y
  if (y > SendBase) {
    SendBase = y
  if (esistono attualmente segmenti non ancora
     riscontrati)
  avvia il timer
  } else {
    incrementa il numero di ACK duplicati ricevuti per
        y
    if (numero di ACK duplicati ricevuti per y = 3) {
      rispedisci il segmento con numero di sequenza y
  }
```

3.5.5 Controllo del flusso

Ricordiamoci che il protocollo *TCP* inizializza delle zone di memoria per il *buffer di ricezione* e *buffer di invio*, il mittente non deve sovraccaricare il buffer del destinatario. Mittente e destinatario comunicano continuante quanto spazio hanno libero nei vari buffer. Il valore di **RcvWindow** verrà inserito all'interno dei segmenti, il mittente limita i dati non riscontati *RcvWindow*, così che non vengano inviati dati che verranno sicuramente persi. **RcvBuffer** funzione per la creazione della socket.

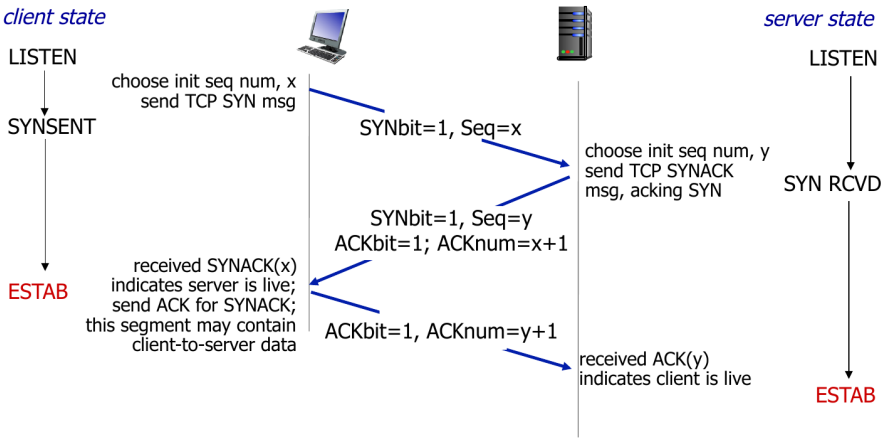


receiver-side buffering

Gestione della connessione: Handshake a tre vie

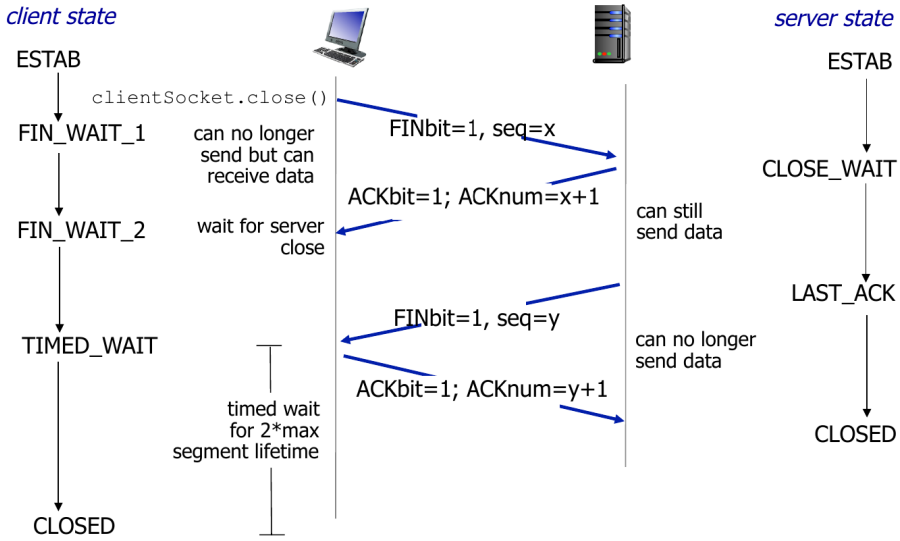
Si stabilisce una connessione tra mittente e destinatario, si mettono d'accordo e inizializzano dei dati come *numero di sequenza* e i *buffer*. L'inizializzazione della connessione avviene tramite l'**Handshake a tre vie**:

- Passo 1: il client invia un segmento SYN al server
  specifica il numero di sequenza iniziale
  nessun dato
- Passo 2: il server riceve SYN e risponde con un segmento SYNACK
  il server alloca i buffer
  specifica il numero di sequenza iniziale del server
- Passo 3: il client riceve SYNACK e risponde con un segmento ACK, che può contenere dati



Per chiudere una connessione abbiamo 4 passi:

- Passo 1: il *client* invia un segmento di controllo FIN al server.
- Passo 2: il *server* riceve il segmento FIN e risponde con un ACK e invia un FIN.
- Passo 3: il *client* riceve FIN e risponde con un ACK. inizia l'attesa temporizzata - risponde con un ACK ai FIN che riceve
- Passo 4: il *server* riceve un ACK. La connessione viene chiusa.



3.6 Principi del controllo di congestione

La **congestione** è il blocco della rete per via di un numero di dati elevato mandati a una velocità elevata che la *rete* non riesce a gestirli. Ciò causa pacchetti smarriti (overflow nel buffer) e lunghi ritardi (di accodamento nei buffer).

Scenario 1

Scenario 2

I buffer hanno dimensione *finita*

Scenario 3

Scenario 4

3.7 Controllo di congestione

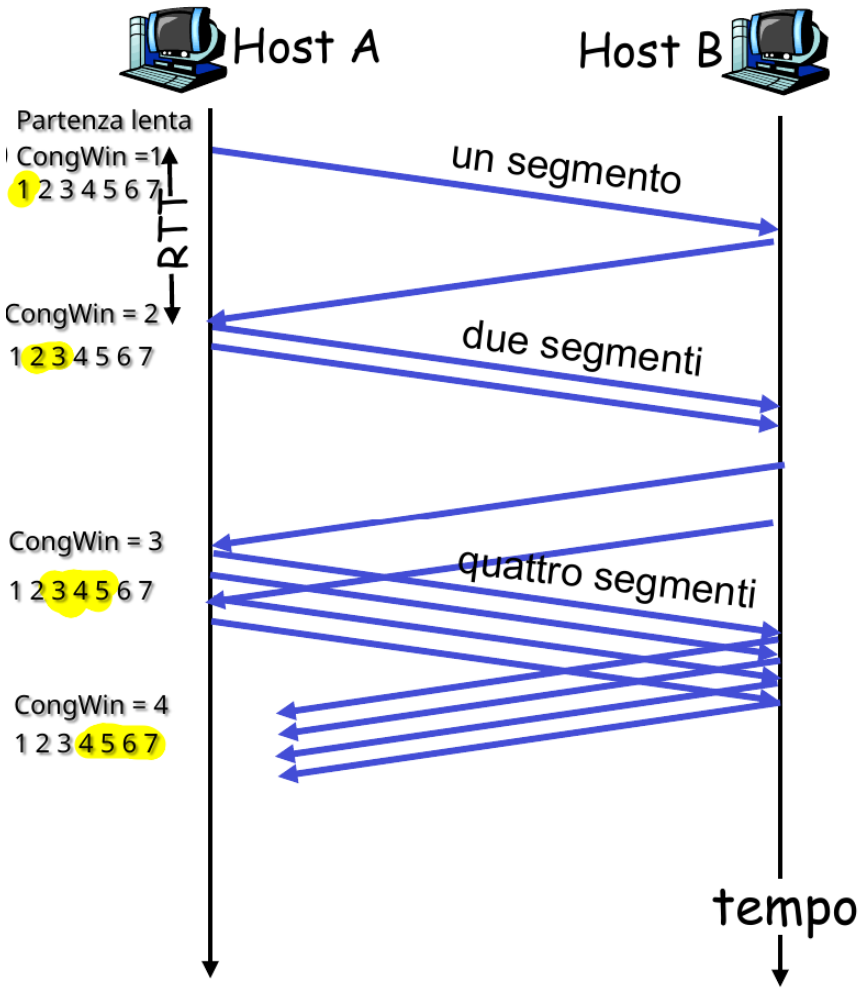
Bisogna limitare la trasmissione, si effettua mdiante una "finestra di congestione" (CongWin), cioè una funzione dinamina della congestione.

$$\text{Frequenza d'invio} = \frac{\text{CongWin}}{\text{RTT}} \text{byte/sec}$$

Definiamola come una misura a spanne, non precisa.

Il mittente si accorge della congestione tramite il *timeout* o il *triplice ACK duplicato*. Il mittente riduce la *frequenza d'invio* dopo essersi accorto, con tre meccanismi:

1. **Partenza lenta**: Stabilita la connessione si manda un solo pacchetto. All'inizio la velocità di trasmissione è molto lenta, poi crescerà a livello esponenziale finché non si verifica un evento di perdita, raddoppiamo la *finestra di congestione* dopo ogni RTT (ogni volta che torna un ACK). La *partenza lenta* progredisce fino a un valore di soglia deciso dai progettisti.

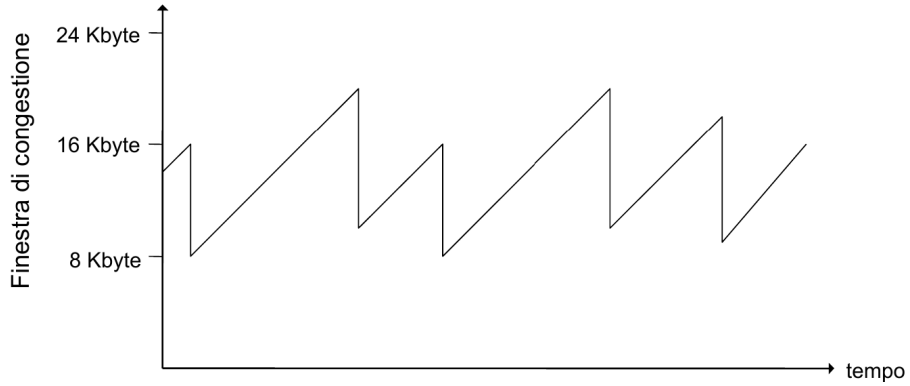


Nel caso di *triplice ACK duplicato* si passa all'algoritmo successivo per una crescita più lenta, impostando però:

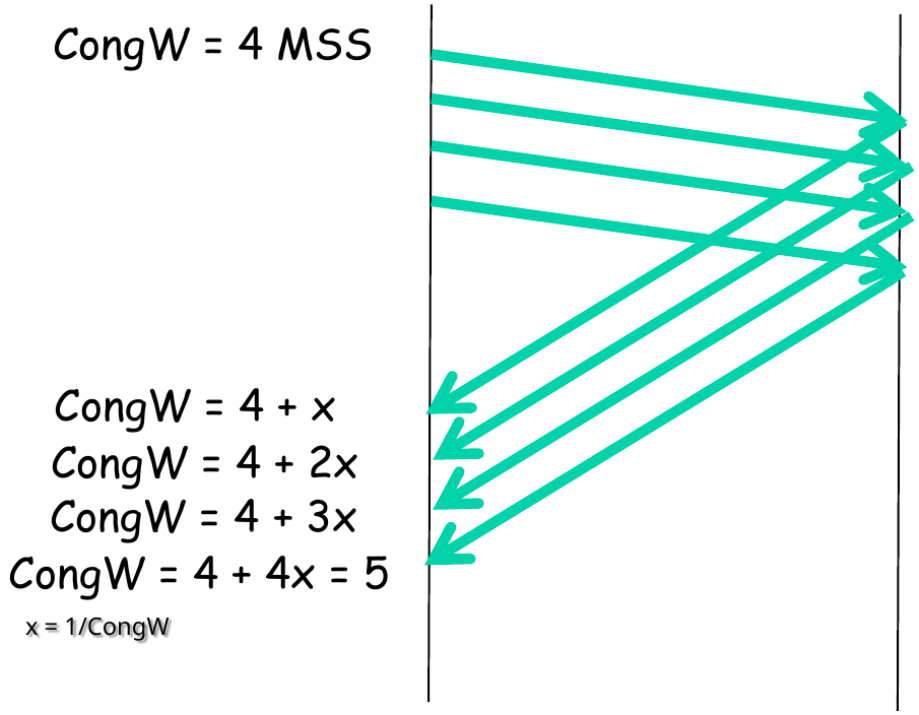
$$\text{valore di soglia} = \frac{\text{CongWin}}{2}$$

$$\text{CongWin} = \text{valore di soglia}$$

2. **AIMD**: Incremento additivo, decremento moltiplicativo in italiano. **Fase a regime**, dopo la partenza lenta. La crescita della *finestra* continua in maniera lineare di **1 MSS** dopo ogni *RTT*, nel caso di errori la *finestra* viene **dimezzata**. Questo è il suo andamento:



Il suo funzionamento è così diagrammato:



negli esercizi, in caso di perdita di *ACK*, arrotondiamo a +1 la frazione, solo per comodità. Nella realtà, essendo in byte, si fa il calcolo e si mantiene quel valore, non ci sono problemi in caso di perdità di ACK poiché l'ACK cumulativo conferma pure il pacchetto perso.

3. **Reazione agli eventi di perdita**: Dividiamo i casi. Nel caso in cui si verifica il *timeout* resettiamo la *finestra* a 1, tornando così alla *partenza lenta*, il valore di soglia viene impostato a

$$\text{Valore di soglia} = \frac{\text{CongWin}}{2}$$

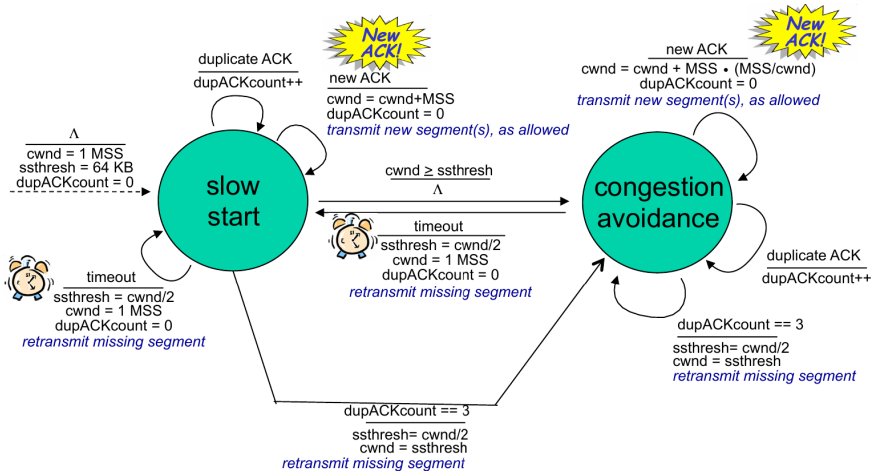
$$\text{CongWin} = 1$$

Nel caso in cui abbaimo una perdita (**Fast recovery**), *triplice ACK duplicato*, imposto

$$\text{CongWin} = \frac{\text{CongWin}}{2}$$

$$\text{Valore di soglia} = \text{CongWin}$$

Diagramma riassuntivo



3.7.1 Throughput TCP

3.8 Programmazione delle socket

Le **socket** mettono in comunicazione *livello applicativo* con il *livello di trasporto*. Esistono due tipi di *socket*: **UDP** e **TCP**.

Il *server* crea una socket col comando `socket()`, inizialmente crea la `socket di benvenuto`, quindi si crea una *socket* senza parametri, bisogna comunicarglieli successivament. Il *server* aggancia i parametri alla *socket* precedentemente creata tramite il comando `bind()`. Mettiamo il *server* in stato di ascolto tramite il comando `listen()`.

Il *client* crea una socket con i parametri del *server*, tramite il comando `socket()`. Il *client* aggancia i 4 parametri alla *socket* precedentemente creata con il comando `bind()`. Il *client* si connette al server mediante il comando `connect()`, il *server* dall'altra parte accetterà la connessione sulla *socket specifica* col comando `accept()`, e manda il messaggio **SYNACK**. Il *client* manda il messaggio al *server* mediante il comando `send()` che verrà ricevuto dal comando `recv()` dal *server*, mandando in risposta tramite `send()` l'**ACK**, che verrà ricevuto dal *client* tramite `recv()`.

Si usa il comando `close()` per chiudere la connessione, possono mandarlo sia *server* che *client*, mandando il messaggio di **FIN**, ricevendo in risposta un **FINACK**.

`socket()`

Creazione della *socket*: `int s_listen = socket(family, type, protocol);`
family: AF_INET specifica IPV4
type: SOCK_STREAM, SOCK_DGRAM
protocol: 0 (pseudo, IP).
Avremo un identificativo della socket come ritorno della funzione.

`bind()`

Aggancio dei parametri alla *socket* precedentemente creata vuota:
`bind(s_listen = localAdd, AddLength);`
Si specifica la porta su cui mettersi in ascolto.
s_listen: identificatore della *socket*
localAdd: di tipo **sockaddr_in**, una struttura già definita.
AddLength: lunghezza della variabile localAdd

```
struct sockaddr_in {
u_char sin_len; // length of address
u_char sin_family; // family of
   address
u_short sin_port; // protocol port num
struct in_addr sin_addr; // IP Addr
char sin_zero[8]; // set to zero, used
   for padding
};
```

Definisco `struct sockaddr_in sockAdd;` Imposto la famiglia: `sockAdd.sin_family = AF_INET;` Per impostare l'indirizzo IPV4 abbiamo due modi:

1. Specifichiamo l'indirizzo da ascoltare: `inet_pton(AF_INET, \127.0.0.1", &sockAdd.sin_addr.s_addr);`
2. Ascolta da tutti gli indirizzi locali (in questo caso): `sockAdd.sin_addr.s_addr = htonl(INADDR_ANY);`

Imposto la porta: `sockAdd.sin_port = htons(9999);`

`listen()`

`int status = listen(s_listen, queuelength);`
Risultato: -1 errore, 0 ok
s_listen: riferimento alla socket
queuelength: numero di client che possono stare in attesa.
È una funzione **non bloccante**, ritorna immediatamente un valore.

`accept()`

`int s_new = accept(s_listen, &clientAddress, &AddLength);` s_new: nuova socket per la comucazione con il client, fino ad adesso abbiamo usato una *socket di benvenuto*.
s_listen: riferimento alla vecchia socket di benvenuto
clientAddress: riferimento alla struttura sockAddr_in con l'indirizzo del client.
AddLength: dimensione della variabile clientAddress.
È una funzione **bloccante**, ritorna un valore solo quando riceve una richiesta di connessione e quindi un **SYN**.

`send()`

`int send(int s_new, const void *buf, int len, int flags);` s_new: descrittore della socket
buf: puntatore al buffer
len: dimensione del buffer
flags: da impostare a 0

`recv()`

`int recv(int s_new, void *buf, int len, unsigned int flags);` Simile alla send
buf: conterrà i dati da ricevere

`fork()`

Funzione della libreria di C, **biforca** il *processo*, creando **processo padre** e **processo figlio**, processi identici con stesse variabili. Il *processo figlio* gestirà le connessioni con il *client* mentre col *processo padre* gestiamo il *listening*, rimandendo in `accept()`, ogni volta che gli arriverà una nuova richiesta per una socket faremo il `fork()` del processo padre, affidando al figlio la socket appena creata.