TRANSIZIONE DA HTTP/1.1 AD HTTP/2

Uno dei motivi della transizione è stato l'**head line blocking**
Infatti nel caso del pipeline si dovevano conservare le risposte per **rispettare** l'ordinamento **FIFO**
Con HTTP/2 sono stati introdotti meccanismi per risolvere questo problema, utilizzando un protocollo intermedio costruito al di sopra di TCP

HTTP/2 NON ha cambiato la **semantica** (cioè il significato dei campi di intestazione), ma cambia **leggermente** la **sintassi** del messaggio
L'introduzione più importante è stato il concetto di **stream**

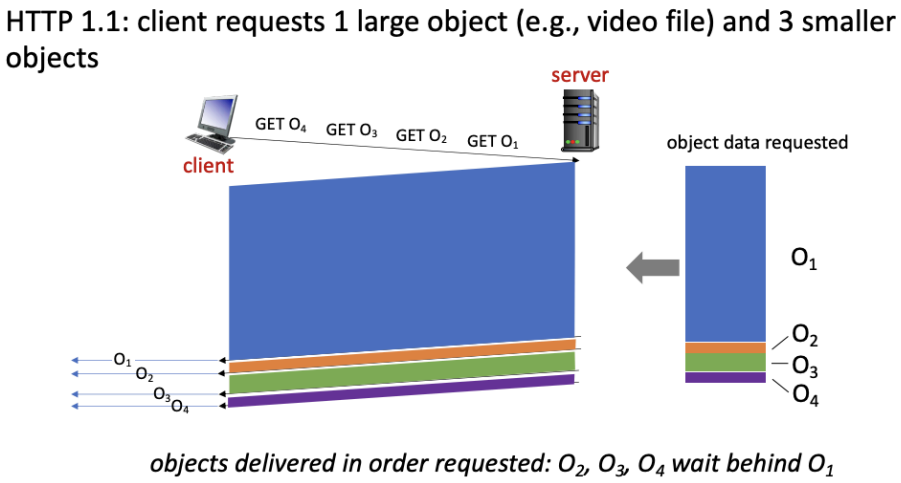
Inoltre è stata introdotta la **compressione** dell'**header** che consente di ridurre la quantità dei byte scambiati

Un'altra introduzione importante è stato il **server push**, cioè una tecnica che permette al server di anticipare le richieste

Con HTTP/1.1 nel caso in cui arrivassero richieste in parallelo, anche se elaborate bisognava garantire l'ordinamento FIFO
Invece HTTP/2 ha introdotto un meccanismo di **framing** che consente di inviare i contenuti in maniera **concorrente**
Quindi è stato introdotto un modello di concorrenza all'interno del protocollo

Se riuscissimo a frammentare un messaggio, allora potremmo inviare un frammento alla volta

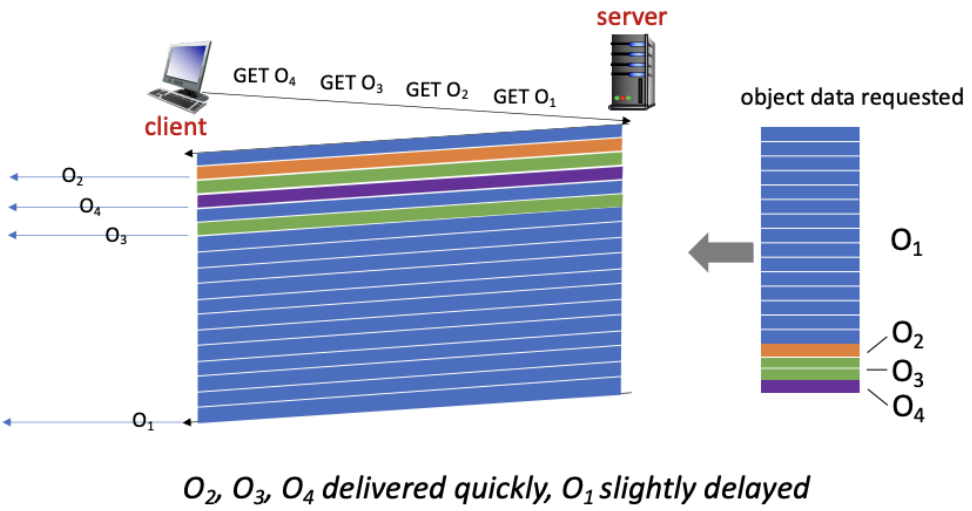
MITIGATING HOL (Head Of Line) BLOCKING



Con HTTP/1.1 gli oggetti richiesti dovevano essere inviati in **ordine**
Ciò significa che per spedire secondo, terzo e quarto (che sono più piccoli), bisognava **attendere** la spedizione del primo (**effetto convoglio**)

Invece con **HTTP/2** c'è la possibilità di dividere il messaggio in **frame**

ATTENZIONE: sono frame a livello **applicativo**, quindi da **NON** confondere con quelli del livello datalink



O_2, O_3, O_4 delivered quickly, O_1 slightly delayed

Con HTTP/2 ogni stream viene utilizzato per uno scambio

In pratica **connessione** e **stream** hanno la stessa relazione che c'è tra **processi** e **thread**
Infatti ogni stream può essere associato ad un thread

Per cui ad **ogni connessione** è possibile associare **più stream**
Questi stream però sono **indipendenti** tra loro e questa è la manifestazione del modello concorrente
Una conseguenza di ciò è che c'è un **interliving arbitrario**

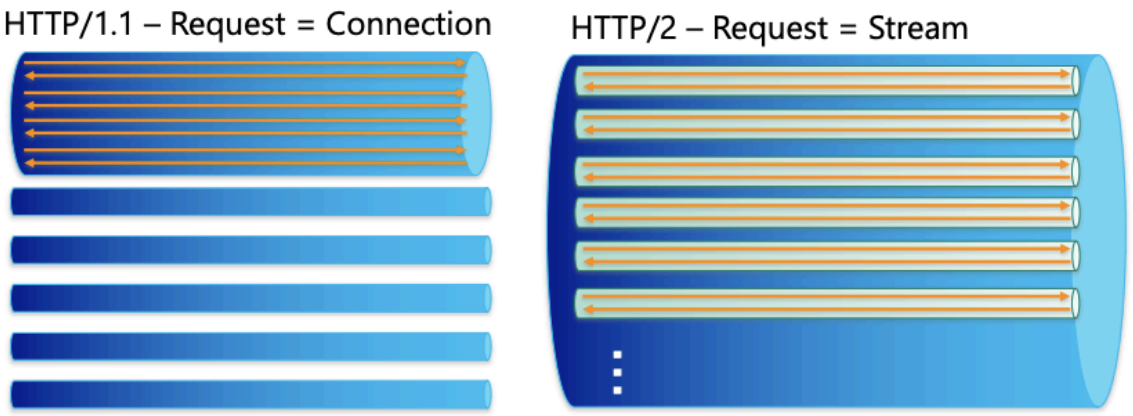
Ogni stream è utilizzato per **uno scambio**, cioè ad ogni richiesta o risposta è associato **uno e un solo stream**

Inoltre è possibile che ci sia un frame utilizzato per trasferire solo l'intestazione e altri stream che trasferiscono i dati

La frammentazione è utile per permettere l'utilizzo della risorsa, in questo caso della **connessione**, anche ad altri

Lezione 8 12/10/2023
Abbiamo visto che HTTP/2 NON ha introdotto variazioni semantiche, ma solo alcune differenze dal punto di vista sintattico
Inoltre il formato dei messaggi non è più testo ma **binario**

In HTTP/1.1 avvenivano più interazioni richiesta-risposta
Invece con HTTP/2 **una stessa connessione ospita degli stream**



Ad **ogni stream è associata una richiesta o risposta**

Gli stream sono **indipendenti** tra loro, ciò significa che ci si trova in un modello concorrente
Per questo motivo l'**ordine in ricezione può essere arbitrario**, quindi un web server può rispondere ad una richiesta non **appena ha terminato l'elaborazione**

HTTP/2 utilizza sempre TCP come protocollo al livello di trasporto, ma è come se venisse costruito un altro protocollo intermedio che utilizza i frame
In pratica, lo stream di byte che viene trasferito NON è più illimitato, ma all'interno degli stream possono essere individuati dei **frame**

Inoltre, non si ha la certezza che dei frame consecutivi facciano parte dello stesso stream

Si può realizzare l'**interleaving**, cioè i **frame** possono essere **inviati appena pronti**

HTTP/1.1 aveva il problema dell'head line blocking
Con il framing, lo stream viene frammentato e quindi possiamo individuare a quale stream afferisce ogni blocco
Esiste un'intestazione che fa capire a quale stream appartiene un blocco

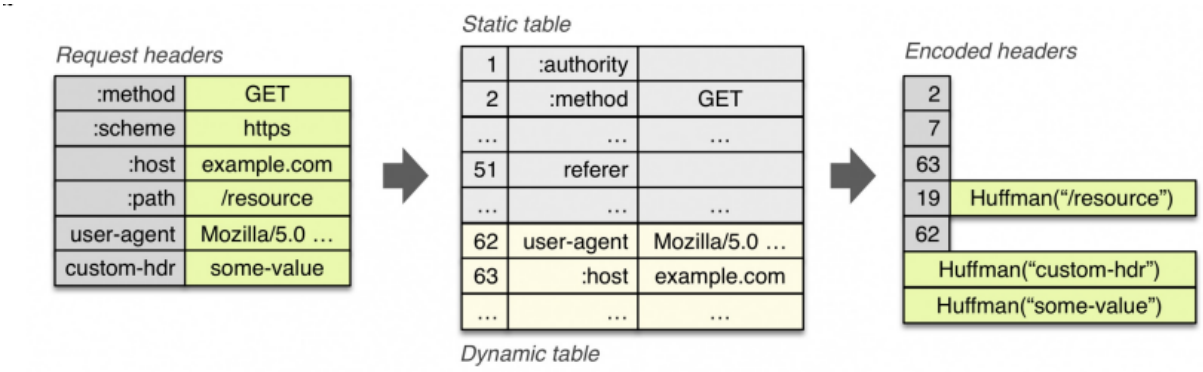
Il canale **FIFO** in HTTP/2 è **bidirezionale (full duplex)**

Un messaggio HTTP per consentire l'interleaving, cioè l'**alternanza**, frammenta la richiesta dividendo header e dati (questi a loro volta vengono divisi in più frame)

COMPRESSIONE

La tecnica utilizzata da HTTP/2 è detta **HPACK**

In pratica HTTP/2 utilizza una **tabella statica** con **61** codici in cui **ai campi dell'intestazione** del messaggio sono **associati i vari numeri**
Ciò è utile perché le intestazioni dei vari messaggi hanno più o meno sempre la stessa forma



Tuttavia la tabella prevede anche una **parte dinamica**

Ad esempio, user-agent e host si trovano nella parete dinamica

La parte dinamica contiene le informazioni specifiche di una conversazione

In generale viene utilizzata la compressione Huffman per le stringhe

La **tabella dinamica cessa** di esistere appena **termina la conversazione**

SERVER PUSH

È un meccanismo introdotto in HTTP/2

In pratica, in HTTP/2 il **parsing** del file HTML viene fatto interamente **lato server**
L'elaborazione è più costosa, ma spesso permette di evitare la ricezione di richieste inutili

Il server dirà su quali stream invierà i riferimenti ad altri oggetti che si trovano nel file HTML
Si dice che il server fa **PUSH PROMISE**

Se il client decide di non volere le push promise, allora viene resettato lo stream
Ad esempio questo può capitare se il contenuto è già presente sul client

Questa tecnica cerca di ridurre l'RTT

COMPATIBILITÀ

Per avere una comunicazione in HTTP/2 avvengono vari step

Inizialmente il client contatta il server in HTTP/1.1 perché non può sapere se il server sa comunicare in HTTP/2
Dopodiché il client **chiede l'upgrade** ad HTTP/2, così se il server è in grado di comunicare, avverrà lo switching al nuovo protocollo e il server si predisporrà per ricevere frame

Quindi ci sono due possibilità di risposta

```
GET / HTTP/1.1
Host: server.example.com
Connection: Upgrade, HTTP2-Settings
Upgrade: h2c
HTTP2-Settings: <base64url encoding of HTTP/2 SETTINGS payload>
```

```
HTTP/1.1 200 OK
Content-Length: 243
Content-Type: text/html

...
```

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: h2c

[ HTTP/2 connection ...
```

101 è il codice informativo che ci dice che il server ha accettato la richiesta di cambio protocollo

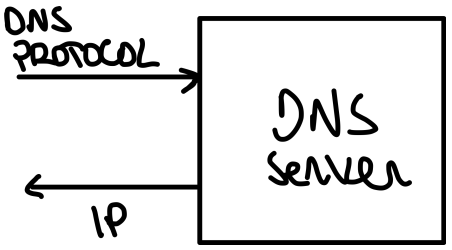
DNS (Domain Name System)

Quando navighiamo su internet, per accedere ad un sito web, utilizziamo www.unisannio.it e non l'indirizzo IP perché è più semplice da ricordare
Tuttavia è necessario che l'**URL** venga **convertito** in un **indirizzo IP**

Quest'operazione è quella che compie un **DNS**

Un **DNS** è un **database** al quale vengono fatte delle richieste e **risponderà con indirizzi IP**
In pratica gli viene passato l'**URL** e come output produrrà l'**indirizzo IP** corrispondente

Un **DNS** è un server che ospita un **database** che, a sua volta, ospita le **coppie nome simbolico-indirizzo IP**
Questi database sono implementati in una gerarchia di server detti **name server**



Lo "scambio" avviene mediante un protocollo

L'obiettivo del protocollo è la **risoluzione** di **nomi** in **indirizzi IP**

È un servizio ausiliario implementato in Internet

Esempio: browser che contatta un server

Il **browser** NON contatta **direttamente** il **server**, ma prima viene effettuata una richiesta al **server DNS**
In pratica, il browser al suo interno ospita un altro client, ovvero un **client DNS**, il quale ha il compito di chiedere al server DNS l'indirizzo IP da contattare
Una volta ottenuto l'IP, il browser si connetterà al server utilizzando il protocollo TCP

Nomi principali = nomi canonici

Esempio: un DNS consente di individuare un mail server affinché avvenga il corretto scambio di mail

Quando viene effettuata una richiesta DNS, NON è detto che la risposta sarà sempre uguale alle volte precedenti, ma dipende da vari fattori come, ad esempio, la posizione geografica dalla quale si effettua la richiesta, dal carico del server, ecc.

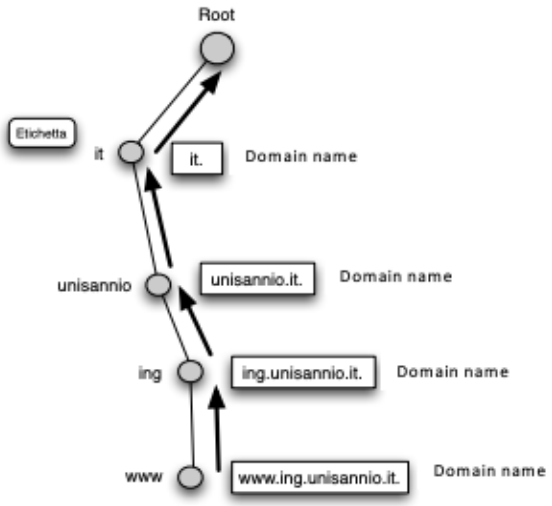
Un server DNS deve avere tanta memoria perché deve gestire tante richieste
Inoltre è anche importante il luogo in cui viene posizionato un server DNS

DNS NAME STRUCTURE

Per assicurare nomi unici, è necessario avere un **name space**
I nomi sono organizzati in modo **gerarchico** (è possibile utilizzare un albero)

ESEMPIO:

Il nome che ci consente di individuare un host è un nome che viene costruito navigando l'albero
L'etichetta di **ROOT** è **vuota**
La **gestione** dei nomi è **decentralizzata**, cioè se ad esempio **it** decide di assegnare un'etichetta a **unisannio**, allora quest'ultimo può assegnare qualsiasi nome esso voglia
Ciò significa che chi gestisce un dominio NON gestisce anche i suoi sottodomini (e quindi anche i name space sottostanti)



Questo è un esempio di albero

L'etichetta sotto la radice è detta **TLD (Top Level Domain)**
La gestione dei nomi dei TLD è assegnata all'ICANN

La deresponsabilizzazione NON dà problemi di **unicità**, perché è l'etichetta del TLD a garantirla

DOMAIN NAMES

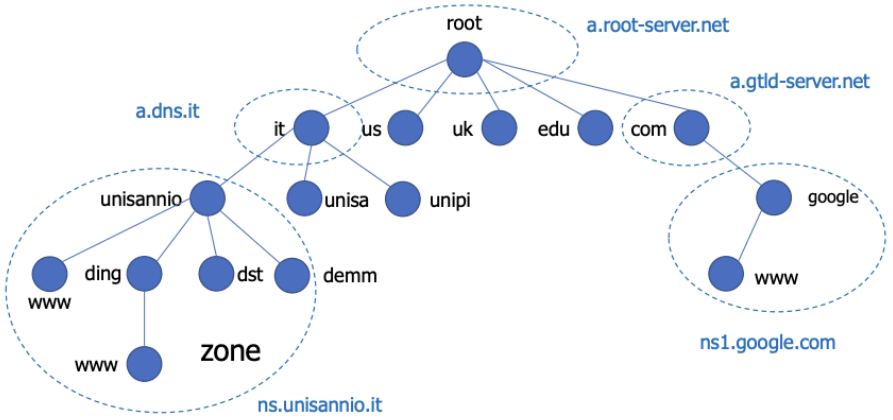
Sono di due tipi:

- **FQDN (Fully Qualified Domain Names)**, cioè il nome completo dalla foglia (o nodo interno) fino alla radice. Esempio: www.ding.unisannio.it
- **PQDN (Partially Qualified Domain Names)**, cioè una sequenza di etichette che non termina con una stringa vuota (nodo root). Esempio: www.ding, questo ha senso solo all'interno dell'organizzazione

Un dominio può essere scomposto in sottodomini, ma è una responsabilità dell'organizzazione

I domini DNS sono strutture logiche
Non sono necessariamente associati alla posizione geografica delle organizzazioni
Inoltre un dominio DNS di un'organizzazione può includere anche più reti

DAL NAME SPACE AI NAME SERVER

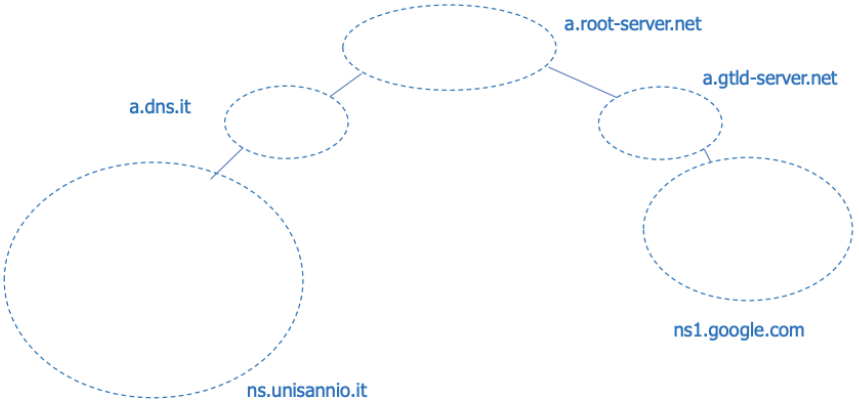


I **name server** vanno a **partizionare** lo **spazio dei nomi**
Ogni partizione è assegnata ad un server diverso
I link (linee blu) indicano i riferimenti al server che gestisce il dominio superiore

"com" e "it" sono TLD e ognuno ha il proprio server

NAME SPACE VS NAME SERVERS

Nella slide ci sono i nomi dei server



Con whois.com si possono vedere informazioni relative ad un server
Esempi: server root, server TLD, se è autorevole, ecc

Per arrivare al nodo **foglia**, **analizziamo** l'**FQDN** da **destra** a **sinistra**, cioè partiamo dal nodo root

Sono **12** le **organizzazioni** che gestiscono i **13 root server**
Però i server fisici NON sono **13** perché altrimenti sarebbero facilmente sovraccaricati

Per cui ogni organizzazione prevede **N istanze** per ogni server

Per le nostre interrogazioni dobbiamo chiederci qual è il punto di ingresso al server che contattiamo
L'**entry point** verso il **DNS** sarà il punto più vicino, cioè il **name server locale**
Il server locale poi interrogherà il root main server e questo saprà qual è il dominio
Se il dominio che si vuole raggiungere è lo stesso di quello di partenza, NON è necessario transitare per il root server

I **server DNS** in un dominio si dividono in due categorie:

- **DNS primario:** server su cui opera l'amministratore dell'organizzazione
- **DNS secondario:** possiede un backup dei dati del DNS primario e serve per evitare sovraccarichi oppure semplicemente come backup in caso di fault del primario

In pratica, il database del DNS primario viene replicato nei file di zona dei DNS secondari

I browser fanno anche da client DNS, anche se questi sono nascosti perché spesso vengono implementati mediante chiamate di funzioni

Il **codice** del **client** che effettua la **risoluzione dei nomi** è detto **resolver**
Si tratta di funzioni di libreria messe a disposizione dai vari linguaggi

Esempio: **dig**
È un client DNS utilizzabile mediante il terminale
Per fare l'interrogazione il comando è: ***dig A www.unisannio.it***
La **risposta** conterrà l'**FQDN** e l'**indirizzo IP** di quell'FQDN oltre ad altre informazioni come:

- 86400, cioè il tempo di vita dato che l'informazione può essere mantenuta in cache
- IN = internet
- A = alias
- aa = autentity answer, cioè la risposta è data da un server autorevole

Lezione 9 17/10/2023
Abbiamo visto che per definire i **server DNS**, si parte dallo **spazio dei nomi**, il quale ha una precisa struttura

Il nodo **root** è un'etichetta **vuota**
Sotto la radice ci sono i **TLD**

Un'organizzazione che chiede il sottodominio è responsabile di mantenere l'unicità dei nomi

I name server possono controllare più domini
Il **link** tra i server indica che un main server ha conoscenza del name server immediatamente successivo

Link: **conoscenza** che i **name server** hanno **uno** dell'**altro** e servono per navigare lo spazio dei server per arrivare al server **autorevole**, cioè quello che possiede l'informazione richiesta

Quando si naviga su Internet, il **primo server DNS** che viene contattato è il **LOCAL NAME SERVER**
È locale rispetto a chi effettua l'interrogazione

Un server è **AUTOREVOLE** se possiede la **copia autentica** dell'informazione richiesta

Quando si parla di zona si intende la zona geografica che il server DNS gestisce
Nel file di zona di un server DNS troviamo le risorse disponibili in quel dominio (o sottodominio)

FILE DI ZONA: **database** gestito da un **server DNS**

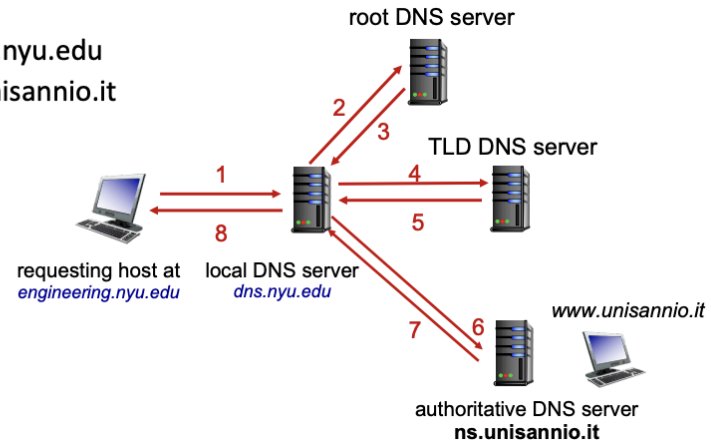
Esistono due metodi per la risoluzione dei nomi

INTERROGAZIONE ITERATIVA

Example: host at engineering.nyu.edu
wants IP address for www.unisannio.it

Iterated query:

- contacted server replies with name of server to contact
- "I don't know this name, but ask this server"



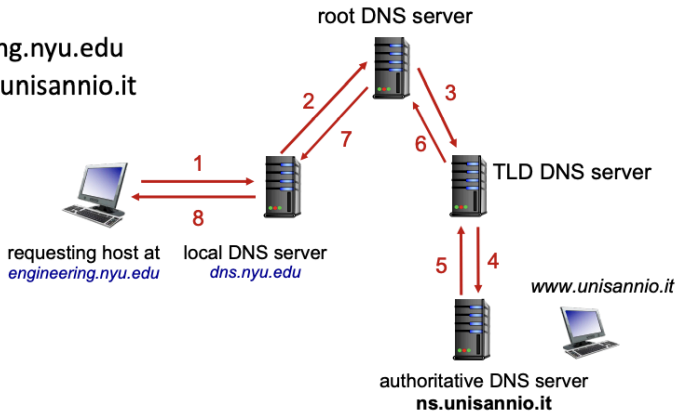
Si parte dal **client**, andiamo al **name server locale** e questo se non possiede l'informazione richiesta, **interroga** il root **DNS** e riceverà o l'IP corrispondente all'**FQDN** oppure l'indirizzo del prossimo DNS da contattare
Al termine delle interrogazioni, si ottiene l'indirizzo corrispondente all'**FQDN richiesto**

INTERROGAZIONE RICORSIVA

Example: host at engineering.nyu.edu
wants IP address for www.unisannio.it

Recursive query:

- puts burden of name resolution on contacted name server
- heavy load at upper levels of hierarchy?



L'**host** effettua la richiesta **interrogando** il **name server locale** (è sempre il punto di accesso)
Se NON possiede l'informazione, il **name server locale interroga** il root DNS
Quest'ultimo però NON risponderà con l'**indirizzo del prossimo server DNS** da contattare, ma risponderà con l'**indirizzo richiesto**

L'indirizzo IP risale fino al server locale

Con questo metodo, si **lascia traccia dell'informazione** richiesta negli altri name server

Lo **svantaggio** è che il **root server** deve mantenere un'informazione di stato, cioè deve ricordare **da chi è arrivata la richiesta**
Quindi si va a **sovraccaricare** facilmente il **server root** perché tutte le richieste partono da lì

Quindi se la richiesta prevede di interrogare il server root, conviene utilizzare il metodo iterativo

I **name server** sono organizzati per mantenere le informazioni in **cache**
Bisogna però assicurare la **consistenza** e ciò viene fatto con un parametro, cioè il **tempo di vita**
Questo parametro ci dice **per quanto tempo** è **valida** l'**informazione** che si trova in **cache** e si trova all'interno del file di zona

DNS RECORD

Un **record** di una **risorsa** possiede 4 informazioni:

- **name**
- **value**
- **type**
- **tempo di vita (ttl)**

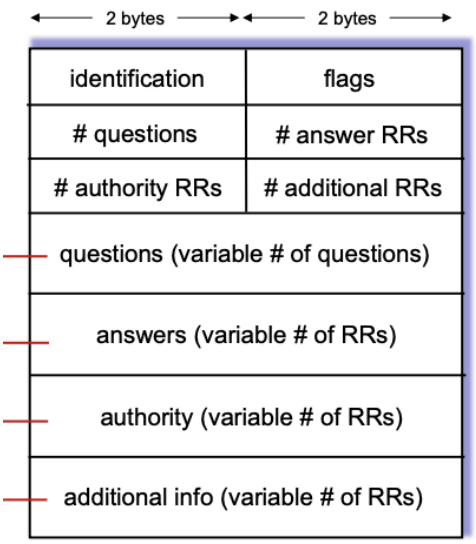
In base al valore che assume il tipo, i parametri avranno valori diversi:

- **type = A** → nome = FQDN (host name), value = indirizzo IP; è il tipo di record più diffuso
- **type = NS** → nome = un dominio, value = name server ; in pratica con il nome non individuiamo una risorsa specifica, ma solo dove opera, mentre con il value indichiamo il name server che opera in quel dominio
- **type = MX** → nome = nome di un dominio, value = il nome di un server mail SMTP; in pratica chiediamo informazioni relative ad un particolare name server, ad esempio per sapere chi gestisce la posta elettronica in un determinato dominio
- **type = CNAME** → nome = alias di un nome canonico, value = nome canonico;

NS e MX effettuano richieste per servizi specifici

DNS PROTOCOL

Le interazioni che avvengono sono di tipo **richiesta-risposta** (si tratta sempre di client-server)
I messaggi di **richiesta** e quelli di **risposta** hanno lo **stesso** formato



La **prima word** è formata da due campi, quindi **2 byte ciascuno**

Il **primo** campo è l'**identificatore di messaggio** che permette di **associare** i messaggi di risposta a quelli di richiesta

I **flag** indicano se si tratta di un messaggio di richiesta o di risposta, se la risposta è autorevole e se si desidera effettuare una richiesta ricorsiva o meno

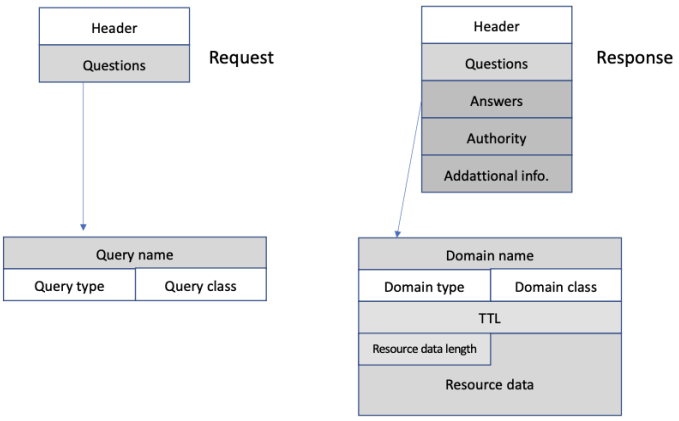
La seconda e terza word contengono i numeri di record

Il campo **questions** conterrà il **record** dell'interrogazione, cioè **nome** e **tipo** della query
Answers conterrà i **valori** corrispondenti all'interrogazione
Authority avrà informazioni relative ai **record gestiti** da server **autorevoli**
Additional info conterrà varie informazioni che **permettono** di **evitare** di fare **ulteriori query**

Esempio: una richiesta di tipo **NS** ci restituirà il nome del server (**FQDN**), quindi in **additional info** potrebbe esserci l'**indirizzo IP** del web server, altrimenti poi bisognerebbe fare una richiesta di tipo A

La struttura del messaggio è uguale per richiesta e risposta, ma un **messaggio** di **richiesta conterrà meno informazioni**

Il **numero** che sarà contenuto da **question** sarà **sicuramente >1**, ma il valore **massimo** sarà 2^{16}



Il valore del campo **query name** **dipende** dal valore di **query type**
Esempio: se in query type c'è A, allora in query name ci sarà l'host

Il campo **answer** del **messaggio di risposta** contiene varie informazioni:

- **tipo** e **classe** del dominio
- il **tempo di vita**
- campo per indicare la **lunghezza** dei **dati** della **risorsa**, dato che la risorsa può avere dimensioni variabili
- campo per i **dati**

Anche il campo **query name** ha **lunghezza variabile**, ma NON viene specificata perché si legge fin quando non si trova lo **0** (cioè il nodo **root**)

Esempio: www.unisannio.it
3 caratteri per www, 9 per unisannio, 2 per it e poi c'è la stringa vuota del nodo root (0 byte), quindi nel campo query name troveremo: 3www9unisannio2it0

È un approccio simile al chunked

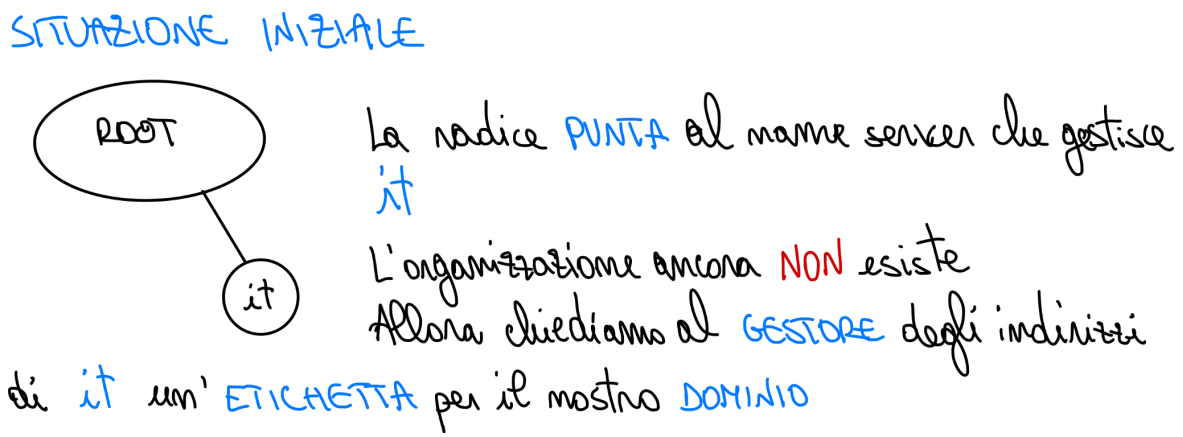
I nomi di dominio sono soggetti a determinate restrizioni: per esempio ogni parte del nome (quella cioè limitata dai punti nel nome) non può superare i 63 caratteri e il nome complessivo non può superare i 255 caratteri.

COME CREARE UN'ORGANIZZAZIONE

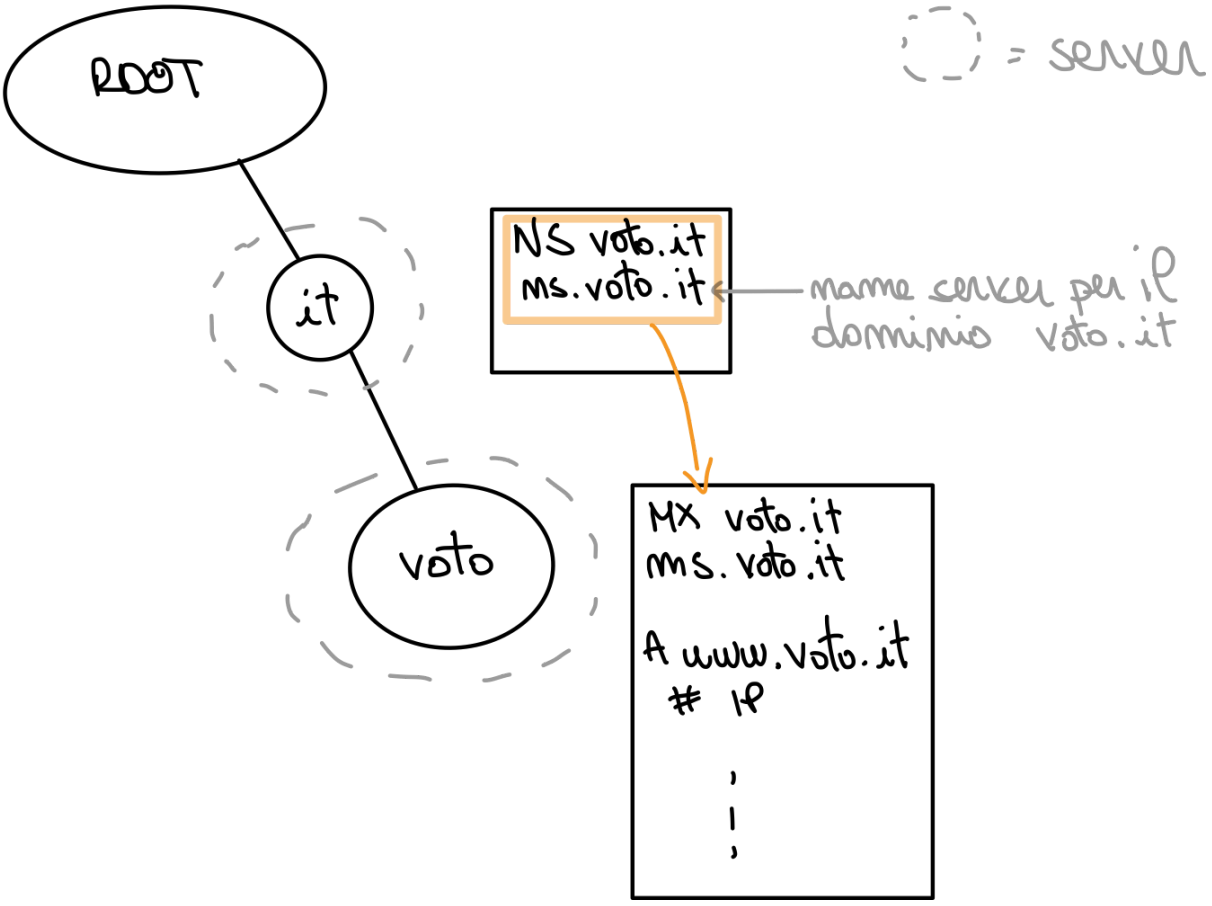
Per creare un'organizzazione bisogna effettuare una richiesta al **DNS registrar**
Nel **server TLD** il **registrar** inserisce **due record** per l'organizzazione, uno **NS** e uno **A**
La **terna NS contiene** il **nome** del **dominio** dell'**organizzazione** e il **name server**
Invece la **terna A** contiene il **name server** e l'**indirizzo IP**

Come gestori dell'organizzazione dobbiamo dar vita al name server, cioè **popolare** il **file di zona** del nostro web server
Sicuramente nel file di zona ci saranno record NS per il nostro server, MX per il mail server e tanti record di tipo A

Chi gestisce il **TLD** NON sa cosa c'è nel **file di zona** del nostro name server, ma sa solo **chi è** il **name server sottostante**



Esempio: chiedo al registro di creare l'etichetta "voto"



Di record di tipo A ce ne possono essere molteplici

Il **registrar** prima di assegnare l'**etichetta**, **controlla** che questa sia **libera**

dig e nslookup sono due client DNS pensati per fare debug delle interrogazioni DNS
Solitamente i client DNS sono integrati in altri client mediante delle librerie, dette resolver

Esempio: dig A www.unisannio.it
QUERY = 1, ANSWER = 1, AUTHORITY = 3, ADDITIONAL = 3
flags: qr rd ra

rd = ricorsione desiderabile
ra = ricorsione accettata
qr = indica se il messaggio è di richiesta o risposta

Nel messaggio di risposta viene ripetuta tutta la richiesta

Nella sezione AUTHORITY ci sono le informazioni sull'organizzazione e notiamo che si sono 3 main server:

unisannio.it.	3600	IN	NS	ns.unisannio.it.
unisannio.it.	3600	IN	NS	dns2.unisannio.it.
unisannio.it.	3600	IN	NS	ns1.garr.net.

Di questi 3 FQDN vediamo gli indirizzi IP nella sezione ADDITIONAL
I server locali non è detto che siano i più vicini

Esempio: dig A www.google.it @ns1.google.com
Con la notazione forziamo di arrivare al web server autorevole
Senza la notazione ci saranno restituiti i contenuti presenti in cache

E-MAIL

Sono 3 i maggiori componenti che operano nell'ambito mail:

- **user agent:** è il client di posta elettronica, spesso oggi si trova già implementato nei browser
- **mail server**
- **protocollo SMTP** (Simple Mail Transfer Protocol): utilizzato per la spedizione di un messaggio di posta elettronica

Per la **ricezione** esiste un **altro protocollo**

All'interno dei **mail server** troviamo:

- **mailbox:** è lo spazio nel server di posta elettronica destinatario utilizzato per contenere i messaggi in arrivo
- **coda dei messaggi:** quando viene inviata una mail, questa non viene subito spedita, ma finisce nella message queue, cioè un buffer che contiene le mail da inviare

Ogni **server** può essere **destinatario** o di **transito**
Un **server** è di **transito** se NON **possiede** la **mailbox** del **destinatario** e quindi effettuerà il **routing** del messaggio

Quando inviamo una mail, affidiamo il messaggio al server locale, o di uscita, affinché la mail arrivi a destinazione

Come fa il **mail server** del nostro provider a **raggiungere** un **altro** mail server?
Utilizza il **dominio** che si trova **dopo** la **@**
In pratica il mail server del nostro provider **interroga** il **server DNS** chiedendo chi è il **mail server relativo** al **dominio** dopo la **@**
Quando un server deve consegnare un messaggio, diventa client dell'altro

Il protocollo utilizzato per l'**invio di mail** è il **TCP**
Per cui si crea una connessione TCP e avviene l'handshaking

Esempio: nc mailgw.unisannio.it 25
Con questo comando contattiamo un server in ascolto alla porta 25 della macchina corrispondente all'FQDN

SAMPLE SMTP INTERACTION

```
S: 220 hamburger.edu

C: MAIL FROM: <alice@crepes.fr>
S: 250 alice@crepes.fr... Sender ok
C: RCPT TO: <bob@hamburger.edu>
S: 250 bob@hamburger.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Do you like ketchup?
C: How about pickles?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection
```

SMTP è un protocollo nato prima del Web

Quando si trova una riga con solo un punto (“.”), il messaggio termina

La stessa connessione può essere utilizzata per inviare più messaggi

Con QUIT viene chiusa la connessione

Rispetto ad HTTP, **SMTP** è più **conversazionale**
Infatti i messaggi HTTP sono più articolati in quanto sono presenti vari attributi

SMTP è un protocollo che prevede un **dialogo basato** su **stringhe**

La posta elettronica prevede più protocolli
Infatti per **recuperare** il **messaggio** dalla mailbox e **visualizzarlo** occorre un altro **protocollo**

I più diffusi sono **IMAP** e **POP3**
La differenza tra i due è che il **primo** è **stateful**,mentre il **secondo** è **stateless**
Quindi con **IMAP** per ogni client viene lasciata traccia della sessione, cioè ci sarà un **checkpoint** che indica fin dove è arrivata la lettura

Quando utilizziamo un browser web, utilizziamo HTTP per accedere ad un'interfaccia grafica di un server web che contiene la logica per accedere al mail server

Quindi utilizziamo un **browser** per **contattare** il **server web** che **gestisce** la **posta**, cioè facciamo una **richiesta HTTP**
Il **server web** a sua volta **contatterà** un **server DNS** per capire chi è il server al quale deve ricevere la mail
Dopodiché la **comunicazione** avviene mediante il **protocollo SMTP** e il messaggio viene **recuperato** mediante il protocollo **IMAP**

Browser, HTTP, server web è la tripla che permette di inviare mail evitando di utilizzare un client di posta elettronica puro

In sintesi gli step che vengono fatti sono:

1. **interazione HTTP;**
2. **interrogazione DNS;**
3. **comunicazione SMTP;**
4. **recupero del messaggio IMAP;**

Lezione 10 19/10/2023

Ora ci orientiamo alla programmazione del livello applicativo, in particolare come scrivere codice per il ruolo client e per quello server

Le applicazioni sono composte da più programmi, ognuno con il proprio main
Quindi più eseguibili

Vediamo come dar vita ad un'interfaccia di programmazione

PRIMITIVE DI COMUNICAZIONE

Per realizzare un'applicazione abbiamo bisogno di due **primitive**:

- **send (destinatario, &messaggio)**
- **receive (destination address, &messaggio)**

Queste due primitive si sviluppano diversamente in base al protocollo del livello di trasporto adottato

La primitiva **receive** è **bloccante**, cioè **blocca** l'**esecuzione** del programma fin quando non arriva il messaggio
Infatti se il **messaggio** NON è ancora **disponibile**, receive o restituisce un messaggio di errore oppure il processo invocante viene sospeso

SEMANTICA BLOCCANTE: meccanismo che garantisce sincronizzazione tra processi, in questo caso client e server

La comunicazione client-server è un classico esempio di programmazione concorrente

Dato che in **rete** la **comunicazione** avviene **tra due processi**, essa è **intrinsecamente concorrente**

Quando si sviluppa un'applicazione è necessario definire il protocollo

L'API di riferimento che vedremo è quella del linguaggio C

CLIENT

```
void main() {
  struct message m1, m2;
  <message m1 is prepared>
  send(SERVER, &m1);
  receive(CLIENT, &m2);
  <message m2 is processed>
}
```

CLIENT è un indirizzo locale al client

&m2 è l'area di memoria dove ci aspettiamo di trovare il messaggio in arrivo dal server

Un codice client, ovviamente, avrà codice più complesso di questo perché saranno implementate altre funzionalità (ad esempio per presentazione, I/O)

receive sospende l'esecuzione fin quando NON arriva il messaggio

SERVER

```
void main() {
  struct message m1, m2;
  while(1) {
    receive(SERVER, &m1);
    <application protocol:
       m1 is processed to produce m2>
    send(m1.source, &m2);
  }
}
```

Questo codice è utilizzato da qualunque server

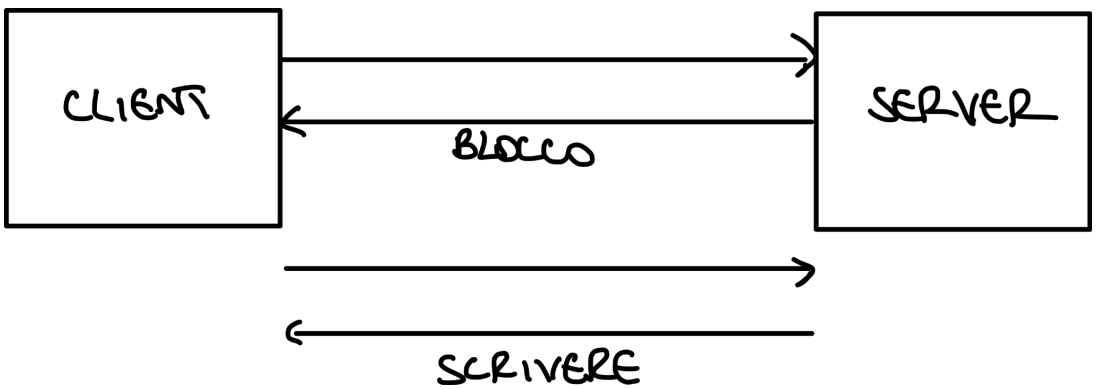
C'è un ciclo while sempre vero perché il server deve essere sempre in esecuzione

Le **primitive**, rispetto al codice **client**, sono in ordine **inverso**

In **send** il primo parametro è m1.source perché è **necessario parametrizzare** il codice, altrimenti diventa complicato utilizzare il codice per qualsiasi client

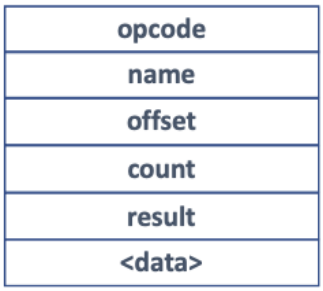
Il primo programma ad essere mandato in esecuzione è il server, perché è quello che mette a disposizione il servizio
Ogni volta però va stabilito il formato di m1 ed m2

Ciò che vogliamo realizzare è un'applicazione che funzioni da **FILE SERVER**
Supponiamo di voler **copiare** un **file**



Dobbiamo effettuare una **lettura** e una **scrittura** di un **blocco**
Dato che stiamo lavorando in rete, NON è possibile **caricare tutto** il **file**, ma per aiutare il livello di trasporto, **lavoriamo** a **blocchi**

Ci saranno quindi ripetute operazioni di lettura e scrittura
Il server deve quindi mettere a disposizione queste due operazioni



opcode: comando; consente di specificare al server l'operazione che il client vuole eseguire
Può anche essere un numero che è associato al tipo di operazione

Bisogna specificare anche **quali file** bisogna leggere
Inoltre, poiché leggiamo a blocchi, è necessario indicare **da dove iniziare** a **leggere**, cioè all'inizio la lettura parte da byte 0, ma alle iterazioni successive NON si leggerà più dal byte 0

Con **count** indichiamo il **numero massimo** di **byte** che si vogliono/possono leggere
Con **result** indichiamo il **numero** di **byte effettivamente** letti

I dati saranno presenti nel messaggio di risposta della lettura

Example: client

```
int copy(char *src, char *dst) {
  …
  int pos = 0;
  do {
    m1.opcode = READ;
    m1.offset = pos;
    m1.count  = BUF_SIZE;
    strcpy(&m1.name, src);
    send(FILE_SERVER, &m1);
    receive(CLIENT, &m1);
    m1.opcode = WRITE;
    m1.offset = pos;
    m1.count  = m1.result;
    strcpy(&m1.name, dst);
    send(FILE_SERVER, &m1);
    receive(CLIENT, &m1);
    pos += m1.result;
  } while (m1.result > 0);
```

strcpy ha come parametro *&m1.name* perché *src* è un indirizzo di memoria, per cui se non ci fosse **&**, nel campo name del messaggio troveremo l'indirizzo di memoria

A *m1.count* viene assegnato *m1.result* perché NON è certo che siano stati ricevuti BUF_SIZE byte, ma può essere anche un numero minore

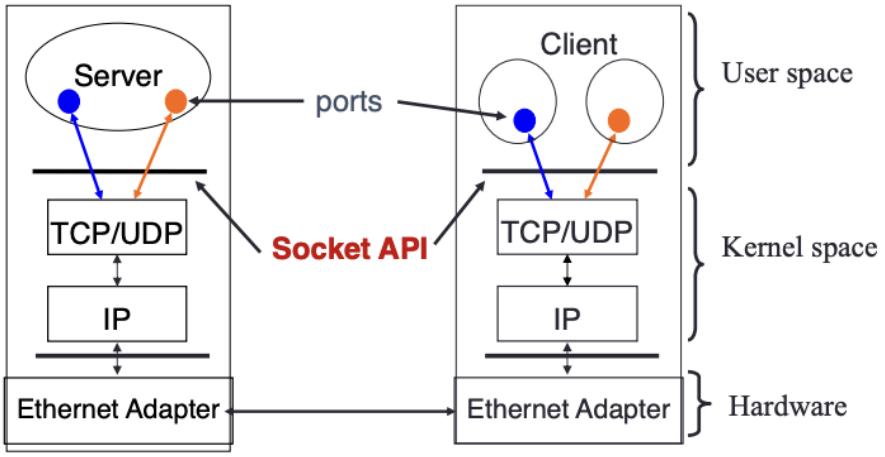
Example: server

```
void main() {
  struct message m1, m2;
  int r;
  while(1) {
    receive(FILE_SERVER, &m1);
    switch(m1.opcode) {
      case CREATE: r = do_create(&m1, &m2); break;
      case READ:   r = do_read(&m1, &m2); break;
      case WRITE:  r = do_write(&m1, &m2); break;
      case DELETE: r = do_delete(&m1, &m2); break;
      default:     r = ERROR;
    }
    m2.result = r;
    send(m1.source, &m2);
  }
}
```

Il server invece utilizza il costrutto **switch** perché a seconda del contenuto di *opcode* eseguirà una funzione diversa

SOCKET PROGRAMMING

Passiamo ad un'interfaccia reale
Ci collochiamo al di sopra del livello di trasporto



Al livello applicativo, le **socket** sono gestite con un **descrittore** di **file** che permette di fare le operazioni di read e write verso la scheda di rete

```
int fd;         /* descrittore di socket */
if ((fd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        exit(1);
}                                          // vers. C
```

La **funzione socket** è molto simile alla funzione **open** per i **file**

Questa funzione consente di iniziare la creazione di una socket, cioè un end point dello schema di comunicazione

PF_INET (Protocol Favorite Internet): specifica l'utilizzo dell'architettura TCP/IP
Il secondo parametro specifica il tipo di servizio che si vuole, cioè se si vuole un canale di comunicazione a stream (TCP) o a datagram (UDP). Infatti può assumere due valori:

- **SOCK_STREAM:** seleziona il modello stream-oriented
- **SOCK_DGRAM:** seleziona il modello datagram-oriented

Il terzo parametro a 0 indica che si utilizza il protocollo di default per il modello di comunicazione

NUMERI DI PORTA

I numeri di porta sono utilizzati per **identificare** entità (**processi**) sugli **host**

Esempi: la porta di **default** di HTTP è 80, di DNS è 53

Il **numero di porta** è espresso con **16 bit** (il numero massimo è 65535)
Infatti nei segmenti del livello di trasporto ci sarà un campo da 16 bit dedicato

I numeri di porta si dividono in 3 range:

- **0-1023** sono detti **ben noti** e sono utilizzati dai primi server creati
- **1024-49151** sono **registrati**
- **49152-65535** sono detti **dinamici** o **privati**

Le porte **dinamiche** sono utilizzate tipicamente dai **client** ed è l'informazione che riempie il campo port nei segmenti
In un sistema UNIX è possibile vedere quali sono le porte ben note e registrate accedendo alla directory *etc/services*

Vediamo come si utilizzano i numeri di porta nel codice
Sappiamo che un **end point** deve avere un **indirizzo di trasporto**, cioè la coppia indirizzo **IP-numero di porta**

```
/* Socket address structure*/
struct sockaddr_in {
  u_char  sin_family;          /* Address family*/
  u_short sin_port;            /* Port number */
                               /* Network byte order*/
  struct in_addr sin_addr;     /* Internet address*/

  char    sin_zero[8];         /* not used */
};
```

L'indirizzo di una socket è una **struct** e prevede due campi **unsigned**:

- **sin_port (16 bit):** è proprio il numero di porta
- **sin_family**

sin_zero può essere utilizzata per ospitare indirizzi di lunghezza diversa, ad esempio IPV6

Un end point NON è raggiungibile se non si conosce l'indirizzo di trasporto

fd è detto **DESCRITTORE DI SOCKET** ed è locale al programma
Quindi per raggiungere l'end point dall'esterno bisogna conoscere l'indirizzo di trasporto, mentre il valore di fd NON è significativo

L'end point viene inizializzato in maniera incrementale

PROBLEMA DELLA PRESENTAZIONE

Due processi potrebbero **rappresentare** gli **interi** in **modo diverso** (esempio: Little o Big Endian)
Per cui potrebbe esserci un problema con i numeri di porta

union è un tipo in C che consente di vedere la memoria in maniera diversa

Esempio:
Abbiamo due macchine, una utilizza una rappresentazione Little e l'altra Big e supponiamo che la prima invii un messaggio di richiesta alla seconda
Dato che l'indirizzo di trasporto deve essere utilizzato nella richiesta, l'ordine di invio dei pacchetti sarà con una logica Little
Quando i pacchetti arriveranno alla macchina Big, questa non riuscirà a trovare l'end point perché interpreterà l'indirizzo di trasporto nell'ordine contrario rispetto a quello inviato

Per cui è necessario introdurre funzioni di **conversione** che si riferiscono ad una **rappresentazione comune**
La rappresentazione **comune** scelta al livello di rete è quella Big Endian

Ci sono due famiglie di **funzioni** di conversione:

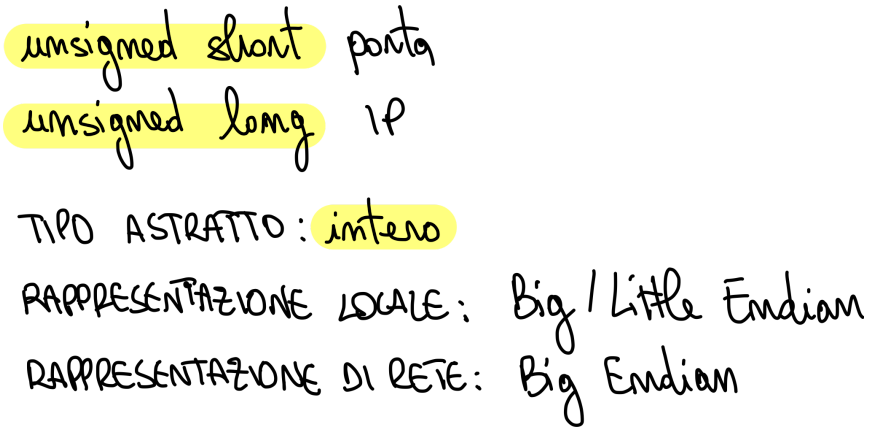
- **hton** (host to network): effettua la conversione da rappresentazione **locale** a quella di **rete**
- **ntoh** (network to host): effettua la conversione da rappresentazione di **rete** a quella **locale**

Queste due funzioni hanno due varianti:

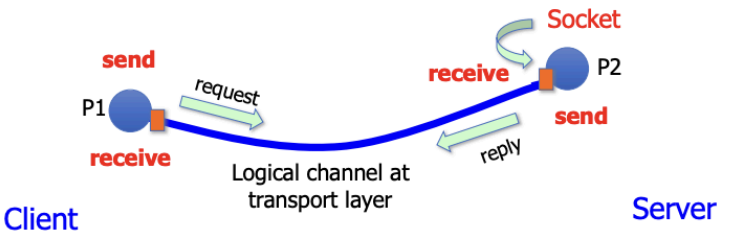
- se terminano con **"s" (short)** allora convertono i numeri di porta perché sono short int
- se terminano con **"l" (long)** allora convertono gli indirizzi IP

Per cui in totale abbiamo 4 funzioni

In sintesi:



Client/server paradigm with sockets



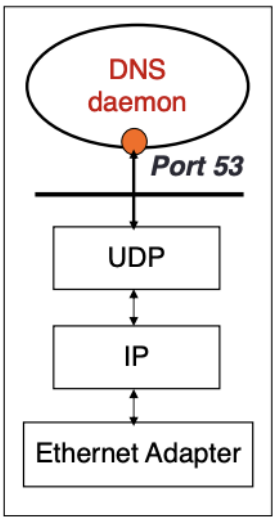

```
void main() {
  int s = socket(…);
  …
  struct message m1, m2;
  <message m1 is prepared>
  send(s, SERVER, &m1);
  receive(s, CLIENT, &m2);
  <message m2 is processed>
}
```

```
void main() {
  int s = socket(…);
  …
  struct message m1, m2;
  while(1) {
    receive(s, SERVER, &m1);
    <application protocol:
       m1 is processed to produce m2>
    send(s, m1.source, &m2);
  }
}
```

Sia client che server invocano la **funzione socket**
Quindi anche send e receive avranno come parametro l'intero restituito da socket, il quale è significativo solo localmente

MODELLO DI COMUNICAZIONE DATAGRAM-ORIENTED

Un esempio di questo modello è un server DNS



Quindi supponiamo di volere realizzare un server DNS (porta 53) che utilizza UDP

La funzione socket avrà come secondo parametro **SOCK_DGRAM**
Dopo aver creato la socket, dobbiamo **assegnare** l'indirizzo di trasporto alla socket (è un'operazione necessaria per il server)
Ciò viene fatto mediante la funzione **bind()**

```
int fd;                     /* socket descriptor */
struct sockaddr_in srv;     /* used by bind() */

/* create a socket */

/* bind: Internet address family*/
srv.sin_family = AF_INET;

/* bind: specifies 53 as port number */
srv.sin_port = htons(53);

/* bind: the address to use for the socket - INADDR_ANY
specifies whatever address of the host */
srv.sin_addr.s_addr = htonl(INADDR_ANY);

if(bind(fd, (struct sockaddr*) &srv, sizeof(srv)) < 0){
      perror("bind"); exit(1);
```

srv è una struct utilizzata per l'indirizzo di trasporto

Però prima di assegnare il numero di porta e l'indirizzo IP, bisogna convertire tutto in Big Endian

In particolare, per l'indirizzo IP alla funzione viene passato come parametro la costante INADDR_ANY perché il server deve funzionare con qualsiasi indirizzo

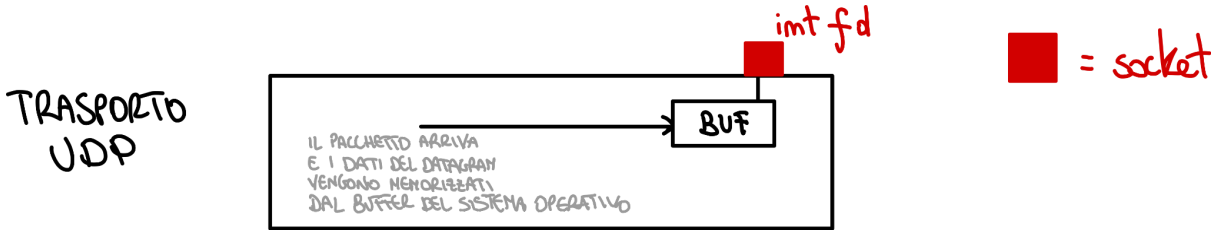
La funzione bind ritorna un int

recvfrom()

La funzione utilizzata per ricevere datagram è **recvfrom()** ed è una primitiva **bloccante**
Questa funzione prevede diversi parametri:

- **descrittore di socket** da utilizzare per leggere
- **area di memoria** dove allocare ciò che viene letto
- **dimensione del buffer**
- **&cli:** indirizzo di trasporto del client, viene inviato perché il server non lo conosce se non dopo che arrivi la richiesta
- **dimensione** di **cli**

Questa primitiva è invocata dal server
È la primitiva attraverso cui il processo server copia il contenuto del buffer di sistema nel buffer del livello applicativo
In particolare, vengono copiati massimo 512 byte alla volta



In pratica, con **recvfrom** si **chiede** che **ciò** che è **memorizzato** nel **buffer** di **sistema sia copiato** nel **buffer applicativo**, il quale è allocato **dinamicamente**

Con recvfrom NON si prendono i dati dalla rete

È una system call
Primitiva = system call
Si può intervenire ai livelli più bassi per modificare un po' il funzionamento, rendendola ad esempio NON bloccante, quindi asincrona

sendto()

Ci troviamo lato client, anche se è utilizzata anche lato server
Lato client il bind NON serve perché il numero di porta viene assegnato automaticamente alla socket
sendto() è la primitiva che consente di inviare un messaggio

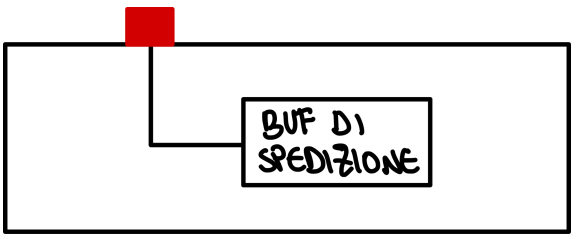
```
int fd;                              /* socket descriptor */
struct sockaddr_in srv;              /* used by sendto() */
char buf[512];

/* 1) create a socket */
/* sendto: send data to IP address "128.2.35.50" port 53 */
srv.sin_family = AF_INET;
srv.sin_port = htons(53);
srv.sin_addr.s_addr = inet_addr("128.2.35.50");

nbytes = sendto(fd, buf, sizeof(buf), 0 /* flags */,
                (struct sockaddr*) &srv, sizeof(srv));
if(nbytes < 0) {
      perror("sendto");    exit(1);
}
```

Questa volta, il parametro **&srv** conterrà l'indirizzo di trasporto del server

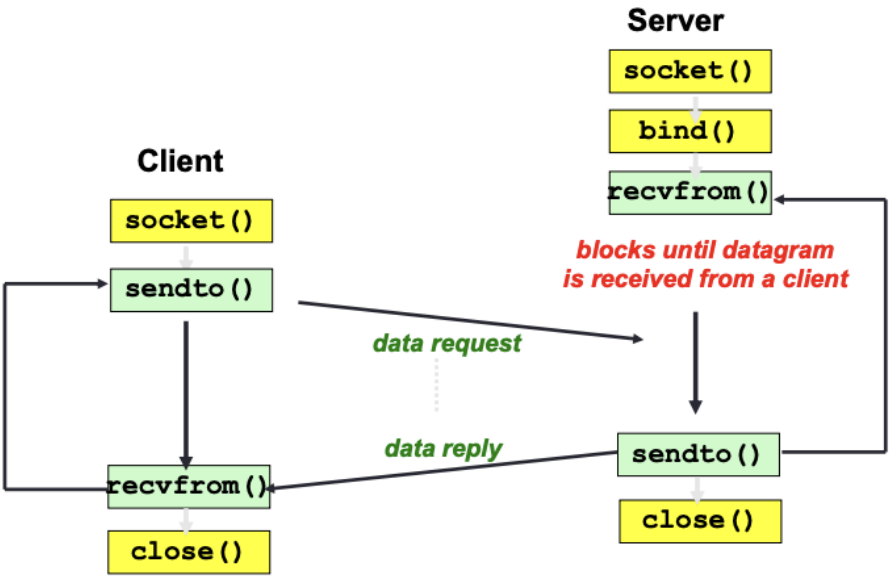
sendto copia i dati dal buffer dato come parametro al buffer di spedizione di sistema associato alla socket



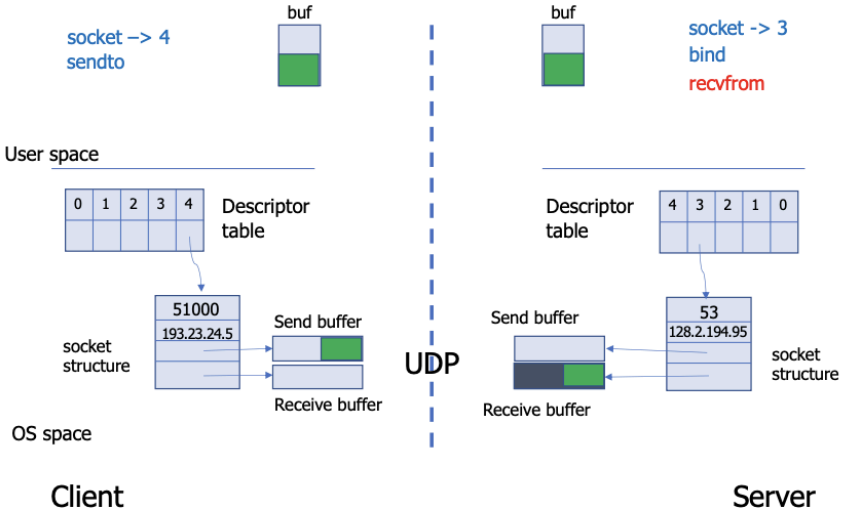
I dati da spedire saranno copiati nel buffer di spedizione e poi sarà responsabilità del livello di trasporto trasferire i dati

Le socket hanno **due buffer**: uno per la **spedizione** e uno per la **ricezione**

Questo modello prevede una comunicazione one-to-many (**punto-multipunto**), cioè da una socket è possibile comunicare con più socket
Ciò è possibile perché le socket NON sono accoppiate, ma libere
Per questo motivo ad ogni spedizione è necessario indicare a chi si invia il messaggio



Per terminare la comunicazione, viene invocata **close()** (proprio come per i file)



Esiste una **tabella di descrittori di socket** (come accade per i file)
È una tabella di **indirezione** per evitare che l'utente possa fare danni
Infatti per ogni entry della tabella c'è un puntatore che punta ad una struttura socket, la quale contiene l'indirizzo di trasporto e i puntatori al buffer di ricezione e spedizione

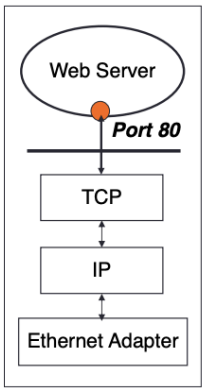
Lezione 11 23/10/2023

Lab 1

Lezione 12 24/10/2023

MODELLO DI COMUNICAZIONE STREAM-ORIENTED

Il protocollo di trasporto che utilizziamo è TCP perché supporta gli stream
L'API che vediamo è differente rispetto a quella orientata ai datagram



Example: a Web server

The operations to receive client requests and to serve them

La funzione **socket** ha lo stesso prototipo di quella utilizzata in UDP, ma differisce il secondo parametro
Infatti ora utilizziamo **SOCK_STREAM**

La funzione socket viene invocata sia dal client che dal server

Il server invoca anche **bind** perché anche in questo caso bisogna assegnare l'indirizzo di trasporto all'end point

```
int fd;                          /* socket descriptor */
struct sockaddr_in srv;          /* socket address */


/* 1) create a socket */

/* initialization of a socket address */
srv.sin_family = AF_INET; /* Internet address family */
srv.sin_port = htons(80); /* port number */
srv.sin_addr.s_addr = htonl(INADDR_ANY);
/* the address to use for the socket - INADDR_ANY specifies
   whatever address of the host */

if(bind(fd, (struct sockaddr*) &srv, sizeof(srv)) < 0) {
   perror("bind"); exit(1);
}      /* binds the socket with descriptor 'fd' to port 80*/
```

La struct **sockaddr_in** contiene l'indirizzo di trasporto

Però un client NON può ancora contattare il server

Vediamo le novità rispetto al modello orientato ai datagram

listen()

```
int fd;                     /* socket descriptor */
struct sockaddr_in srv;     /* socket address */

/* 1) create a socket */
/* 2) bind the socket to a port */

if(listen(fd, 5) < 0) {
      perror("listen");
      exit(1);
}
```

Questa funzione si trova **solo** nel **server**
Bisognare fare in modo che una socket di un server si connetta ad una di un client

Listen caratterizza la socket come **socket** di **tipo server**
Quindi ci sono due tipi di socket, infatti in Java ci sono due classi apposite

Listen prevede **due parametri**:

- **la socket**
- **la dimensione della coda** delle richieste che arrivano alla socket passata come primo parametro

fd NON è una socket utilizzata per lo scambio di dati, ma è utilizzata solo per attivare la connessione

5 invece indica che nella coda ci possono essere al massimo 5 richieste di connessione in attesa

Dopo l'esecuzione della listen è possibile ricevere richieste ed accettarle

accept()

Viene invocata dopo listen

```
newfd = accept(fd, (struct sockaddr*) &cli, &cli_len);
if(newfd < 0) {
      perror("accept");   exit(1);
}
```

Riceve come **parametri** la **socket** di **ascolto** e **restituisce** un **intero** che identifica un **altro descrittore di socket**, quindi una nuova socket finalizzata alla comunicazione
Questa nuova socket viene creata sulla base della socket identificata da fd

Il descrittore per la nuova socket sarà newfd

Inoltre l'altro parametro è l'indirizzo di trasporto, identificato dalla struct sockaddr

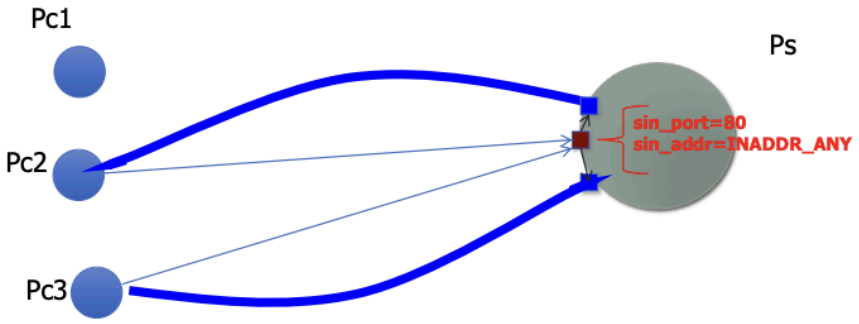
Piccolo OT: i passaggi di parametri in C, avvengono solo per valore e NON per riferimento
cli viene passato per copia, quindi passando il puntatore è come se avessimo un'area condivisa tra chiamante e chiamato

accept si comporta più o meno come receivefrom
Infatti è **bloccante**, quindi si rimane in attesa che un client richieda la connessione

accept può anche trovarsi in un **ciclo** perché si possono servire più client

La **socket** di **ascolto** è **una sola**
Quando un client effettua una richiesta di connessione, viene **creata** una **nuova socket** che permette di comunicare solo con il client che ha fatto richiesta

accept è come se facesse una funzione di dequeue



Le frecce blu sottili indicano le richieste per **l'attivazione** della **connessione**
Invece quelle blu chunky sono i **collegamenti** per il **dialogo**

Dato che NON siamo in concorrenza, se il server sta servendo un client e un nuovo client fa una richiesta, questa andrà in coda
La coda si satura se non viene eseguita l'accept

Quindi per eseguire la funzione accept NON interessa sapere chi è il client (cioè l'indirizzo di trasporto del mittente) perché è sufficiente utilizzare newfd

L'indirizzo del client può essere utile quando si vuole implementare una politica di sicurezza (ad esempio una black list)

Dopo l'accept il server deve fare una **lettura**

read()
È la funzione per effettuare una **lettura**
La read è della stessa API utilizzata per i file per la lettura **sequenziale**, perché il canale è FIFO

```
char buf[512];              /* used by read() */
int nbytes;                 /* used by read() */

/* 1) create a socket */
/* 2) bind the socket to a port */
/* 3) listen on the socket */
/* 4) accept connections */

if((nbytes = read(newfd, buf, sizeof(buf))) < 0) {
      perror("read"); exit(1);
}
```

Ora abbiamo un **buffer** di **512 byte**

La read **restituisce** un **intero** che corrisponde al numero di byte che leggerà
NON c'è il concetto di pacchetto
Sicuramente il numero di byte che leggerà sarà <512 perché ha come parametro sizeof(buf)
Infatti, spesso è necessario utilizzare la read all'interno di cicli

La read è una funzione bloccante

connect()

Permette di **connettere** un **client ad** un **server**

```
int fd;                     /* socket descriptor */
struct sockaddr_in srv;     /* used by connect() */

/* create a socket */

/* connect: Internet address family */
srv.sin_family = AF_INET;

/* connect: specifies the port number to contact */
srv.sin_port = htons(80);

/* connect: specifies the IP address to contact */
srv.sin_addr.s_addr = inet_addr("128.2.35.50");

if(connect(fd, (struct sockaddr*) &srv, sizeof(srv)) < 0) {
      perror("connect"); exit(1);
}
```

L'indirizzo di trasporto riguarda il server da contattare e viene utilizzato per chiedere l'attivazione della connessione

inet_addr converte l'indirizzo IP in un intero a 32 bit

connect riceve come parametri la **socket locale** (fd) e la **socket remota** (srv)

Dopo la **connect** sarà realizzato un **dialogo 3-way handshake** tra la componente client e server

Creare una connessione a livello di trasporto significa lasciare su un end point traccia dell'altro e viceversa
Le strutture dati utilizzate per creare le socket hanno tutte le informazioni necessarie a realizzare il dialogo
newfd è associata ad un indirizzo locale e ad uno di trasporto

Se il server non invoca listen, allora la connect fallisce

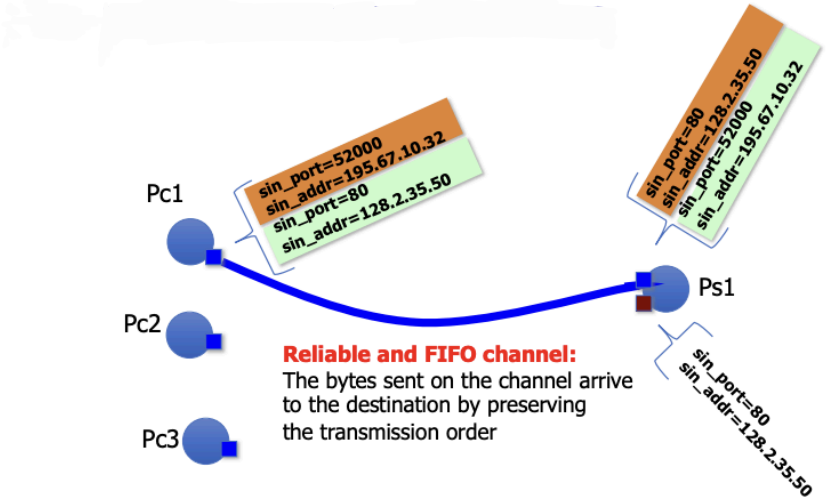
write()
La funzione per inviare un messaggio è la write

Le operazioni di lettura e scrittura sono disaccoppiate
Sappiamo che l'ordine di scritture viene rispettato

Esempio: un client scrive 512 byte non è detto che il server legge tutti i 512 byte in una sola volta, ma potrebbero essere necessarie più read

Ciò accade perché il buffer quando viene veicolato, questo viene frammentato
Per questo motivo l'applicazione che riceve deve sapere quanti byte leggere

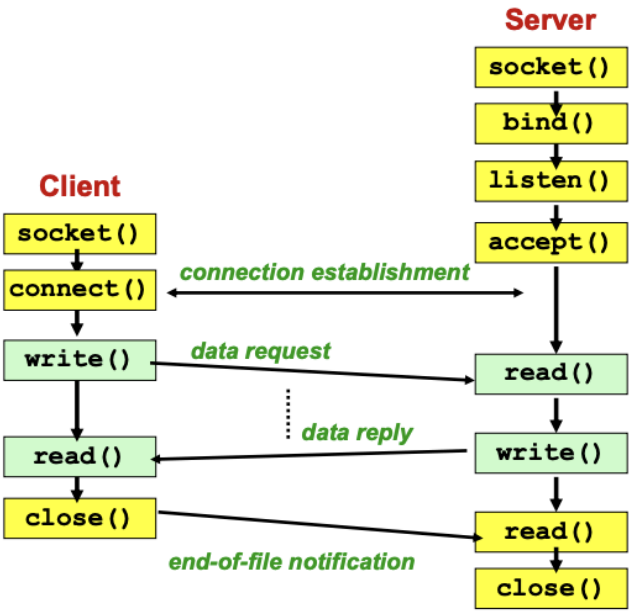
POINT-TO-POINT COMMUNICATION



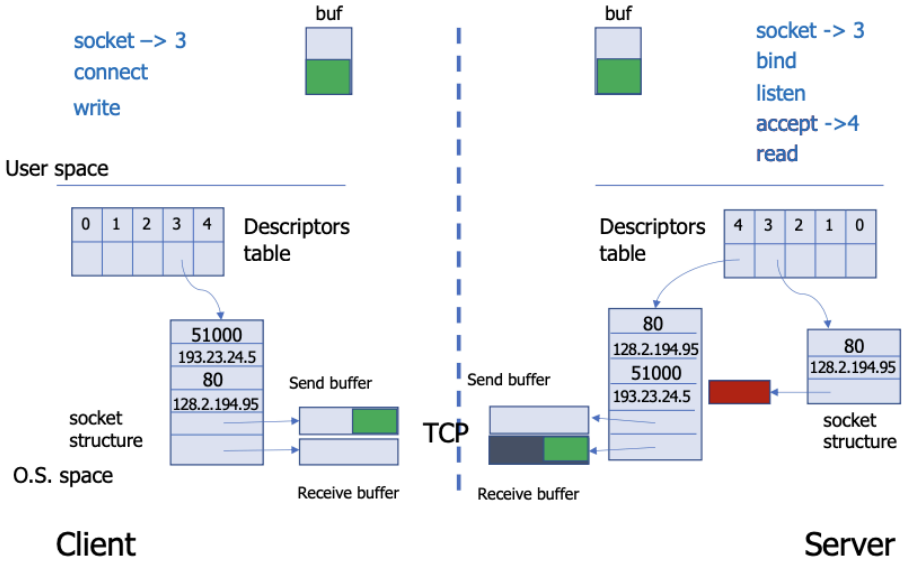
Nella socket troviamo 2 indirizzi di trasporto:

- **lato server:**
  - in **verde** l'**indirizzo** di **trasporto** del **client**
  - in **arancione** l'**indirizzo** di **trasporto locale**
- **lato client:** i colori sono invertiti

Se due socket di un client sono sulla **stessa macchina**, l'**IP** sarà **uguale**, ma **cambierà** il **numero di porta**, per cui il server può identificare univocamente un client



Quando il server invoca **close**, dobbiamo **invocarla** su **newfd** perché se si invocasse su **fd**, NON sarebbe più possibile **contattare** il **server** perché si **chiuderebbe** la **socket** della **connessione**



La socket del server ha la coda degli indirizzi, creata con la listen
Attraverso la connet il TCP scambia i segmenti per attivare la connessione

La funzione **write copia** il **contenuto** del buffer del **livello applicativo** in quello di sistema (come sendto)

Il server invia al client quante volte è stato contattato
La risposta deve contenere una stringa o se ottimizzata anche degli interi

Vediamo codice Server.c

```
/* Create a socket */
sd = socket(PF_INET, SOCK_STREAM, 0);
if (sd < 0) {
  fprintf(stderr, "socket creation failed\n");
  exit(1);
}
/* Bind a socket address to the socket */
if (bind(sd, (struct sockaddr *)&sad, sizeof(sad)) < 0) {
   fprintf(stderr,"bind failed\n");
   exit(1);
}
/* Create the listen queue with the specified QLEN */
if (listen(sd, QLEN) < 0) {
   fprintf(stderr,"listen failed\n");
   exit(1);
}
```

Queste sono funzioni specifiche del server

L'API opera sulla stessa tabella dei descrittori dei file

```
   while (1) {
     alen = sizeof(cad);
     if ( (sd2=accept(sd, (struct sockaddr *)&cad, &cadlen)) < 0)
 {
        fprintf(stderr, "accept failed\n");
        exit(1);
     }
     visits++;
     sprintf(buf,"This server has been contacted %d
              time%s\n", visits, visits==1 ? "." : "s.");
     write(sd2, buf, strlen(buf));
     close (sd2);
   }
}
```

sd2 sarà il newfd degli esempi di prima

Nel client facciamo prima una read perché la connect può essere utilizzata per segnalare un contatto, quindi è come se la connect facesse una write

Lato server NON viene fatta la read iniziale perché la deleghiamo ad accept

```
if (connect(sd, (struct sockaddr *)&sad, sizeof(sad)) < 0) {
  fprintf(stderr,"connect failed\n");
  exit(1);
}
/* This client does not send a request message. Connect is sufficient*/
/* Read data from the socket and write them to the standard output */
n = read(sd, buf, sizeof(buf));
while (n > 0) {
  write(1, buf, n);
  n = read(sd, buf, sizeof(buf));
}
/* close the socket */
close(sd);
/* the program terminates */
exit(0);
}
```

Dopo la prima read nel client si fa un controllo

n == 0 quando ci sarà un **fine stream**

Nel server NON viene indicato quando fermare la lettura perché altrimenti bisognerebbe mandare un altro messaggio prima di inviare la stringa
HTTP invia Content-Length, ma in questo caso noi non lo facciamo

Funziona perché dopo che il server invia la stringa chiude lo stream
Quindi **n == 0** sarà la lettura dell'**end of stream**

Ciò che stiamo implementando è quello che fa HTTP 1.1 con connessioni non permanenti

La **write** nel client **scrive** il contenuto di buf sullo **stdout**, mentre il 3° parametro è la quantità di byte letti