Introduzione

lunedì 31 luglio 2023 19:08

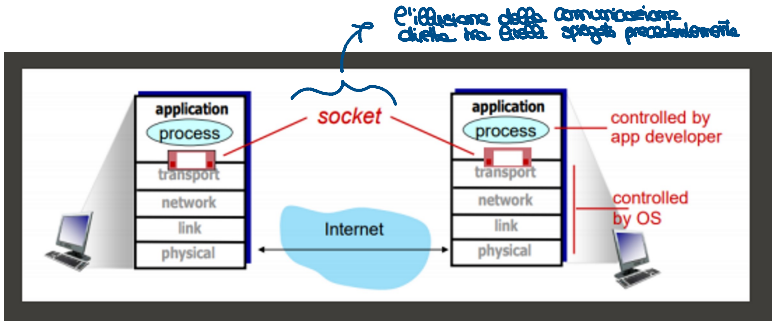
Sono programmi eseguiti su sistemi periferici che comunicano fra loro tramite una rete. Per uno sviluppatore di applicazioni l'**architettura di rete** è fissata e fornisce uno specifico insieme di servizi, il compito dello sviluppatore è quello di sviluppare l'architettura dell'applicazione stabilendo la sua organizzazione sui vari sistemi periferici, le due principali architetture di rete sono:

- **Architettura Client-Server**, abbiamo un host sempre attivo (server) il quale risponde alle richieste di servizio di molti altri host(client). Un' esempio sono le applicazioni web come i browser, c'è un web server sempre attivo che risponde alle richieste dei browser eseguiti sui client.
  In questa architettura i client non interagiscono in modo diretto tra di loro, questo comporta l'uso di **data center** che ospitano gli host(server) creando un potente server virtuale --> aumentano i costi;
- **Architettura Peer-to-Peer(P2P)**, l'infrastruttura in questa architettura non richiede (o comunque minimamente) l'uso di data center, questo perché viene sfruttata la **comunicazione diretta tra coppie di host, chiamate peer**. I peer, quindi essendo host, si tratta di sistemi periferici controllati dagli utenti; ogni peer funge sia da client che da server questo rende intrinseca la scalabilità, inoltre non essendoci una specifica infrastruttura server da mantenere è un organizzazione economicamente conveniente, tuttavia ci sono problemi di sicurezza, prestazioni e affidabilità. Un esempio di applicazione web P2P è BitTorrent.

L'indirizzo IP del server è statico, non cambia mai

Processi

Processi che comunicano in esecuzione su diversi host vengono chiamati **processi comunicanti**. Sappiamo che un processo è un programma in esecuzione che in uno stesso sistema comunicano usando un approccio interprocess communication(code di messaggi, memoria condivisa, ecc.).
L'approccio è all'incirca simile tra processi in diversi host, questi comunicano tramite uno scambio di messaggi su una rete. **Ogni coppia di processi comunicanti è composta da un processo client e un processo server**: il processo client è il processo che avvia sempre la comunicazione, ovvero contatta l'altro processo per richiedere un servizio all'inizio della sessione di comunicazione, il processo server risponde erogando un servizio e rilanciando un ulteriore messaggio al client laddove è previsto.
Questo meccanismo di **scambio di messaggi** tra processi comunicanti è possibile **attraverso un interfaccia software detta socket**, *come funziona?* Supponiamo che la nostra socket sia una porta, un processo invia il messaggio aprendo la sua porta presumendo che ci sia un "corriere" che trasporterà questo messaggio attraverso la rete fino alla porta del destinatario…



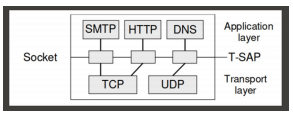
La socket quindi è un'interfaccia tra il livello applicativo e il livello di trasporto, il progettista esercita un controllo totale sul livello applicativo ma poco controllo su quello di trasporto

Uno stesso host può avere piu' "porte", *come indentifichiamo quindi l'host e il processo dell'host destinatario?*
Per identificare un processo ricevente ci servono due informazioni:

1. **Indirizzo dell'host** a cui è destinato il messaggio, tramite *INDIRIZZI IP*, ovvero un numero di 32 bit che identifica univocamente un sistema periferico collegato in rete;
2. **Indentificatore del processo ricevente sull'host di destinazione**, ovvero ogni processo ha una socket diversa e su un host potrebbero essere in esecuzione piu' applicazioni di rete.

Si identifica una socket tramite un **numero di porta**, questi fino al numero 1024 sono riservati e standardizzati, esistono delle convenzioni, ad esempio i web server (HTTP, HTTPS) sono identificati dalla porta 80 e il server di posta (SMTP) è identificato dalla porta 25.
L'URL ha la possibilità di specificare una porta che deve essere > 1024 perché quelle prima abbiamo detto che sono riservate e non avrei quindi i privilegi di amministratore di sistema per usarle.
Riassumendo il client per comunicare con il server si presenta con [indirizzo mittente, porta mittente], [indirizzo destinatario, porta destinatario].

Il mittente invia il messaggio (tramite la socket) e successivamente è il protocollo a livello di trasporto che ha la responsabilità di consegnare i messaggi alla socket del processo ricevente. Le socket sono quindi lo strumento attraverso il quale il livello applicativo dialoga con il livello di trasporto tramite delle primitive di 3 tipi: **specifiche client, specifiche server, specifiche client-server** e queste differiscono a seconda del protocollo a livello di trasporto che utilizziamo.



Servizi di trasporto disponibili per le applicazioni

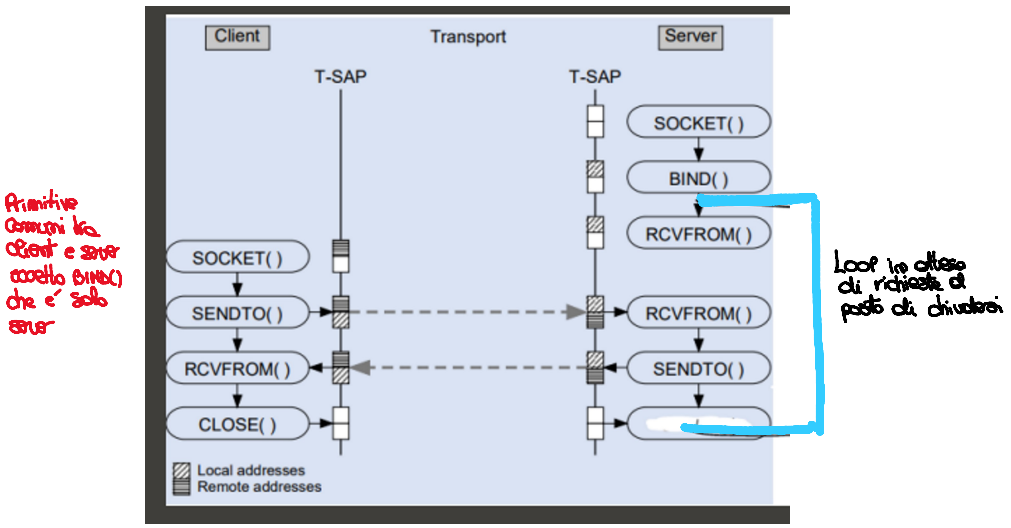
Internet mette a disposizione diversi tipi di protocollo con lo stesso risultato finale ma con raggiungimento differente, possiamo classificarli in 4 dimensioni:

1. **Data Integrity - Trasferimento dati affidabile**: quando il buffer di un router è pieno, potrebbe capitare di perdere un pacchetto (*packet loss*), questo potrebbe causare gravi conseguenze in applicazioni come quella la posta elettronica. Vi è quindi la necessità di garantire che i bit del pacchetto vengano consegnati corretti e completi, si parla di **servizio di trasferimento affidabile** di un protocollo.
   Non in tutte le applicazioni serve tale servizio, un applicazione streaming di musica ad esempio tollera una "perdita", si chiamano **applicazioni loss-tolerant reliable;**
2. **Throughput:** abbiamo visto precedentemente che è la velocità con cui il processo mittente può inviare i singoli bit del pacchetto al processo ricevente, un tipo di servizio che può fornire un protocollo di trasporto è quello di assicurare un throughput stabile e garantito di r bps, interessa ad applicazioni come quelle di telefonia internet che deve codificare la voce a N kbps e inviarli alla stessa velocità. Questo tipo di applicazioni sono dette **bandwith-sensible applications**, le applicazioni opposte invece sono dette **applicazioni elastiche**;
3. **Sicurezza:** un protocollo a livello di trasporto può fornire ad una applicazione piu' servizi di sicurezza, come ad esempio

cifrare un messaggio all'invio che il destinatario dovrà poi decifrare.

Due protocolli di trasporto: UDP

UDP è un protocollo di trasporto minimalista, è senza connessione quindi non c'è **handshaking** tra client e server (*procedura con cui il processo client e server si scambiano informazioni di controllo a livello transport prima che i messaggi a livello applicativo cominciano a fluire)*, questo rende inaffidabile il servizio di trasferimento dati UDP: quando un processo invia un messaggio tramite una socket UDP, tale protocollo, non garantisce che questo giunga a destinazione e se giunge non è detto nell'ordine giusto, in alcuni casi può avvenire anche la duplicazione dei pacchetti.
Nel protocollo UDP il mittente allega esplicitamente l'indirizzo IP di destinazione e il relativo numero di porta per ogni pacchetto e non ha un meccanismo di controllo della congestione.
Le primitive socket UDP sono: **SOCKET()** crea una nuova entry nella tabella dei socket/file aperti ed ha alcuni *paramentri* tra cui *quello che specifica se è una socket di tipo UDP o TCP*, **SENDTO()** system call che genera e invia il pacchetto alla specifica socket remota(destinatario), **BIND()** associa la coppia < *network_address , port_number* > alla struttura dati socket creata, **RECEIVEFROM()** systemcall bloccante, lo stato del processo è in attesa e smette di esserlo quando arriva il paccheto e infine **CLOSE()** rimuove l'entry della tabella dei socket/file aperti *(usata piu' che altro dai client mentre i server rimangono sempre attivi)*…

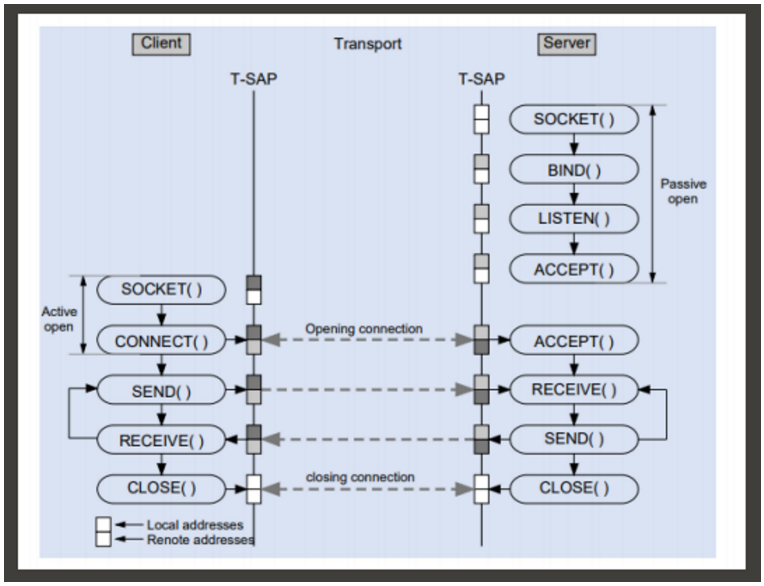


Due protocolli di trasporto: TCP

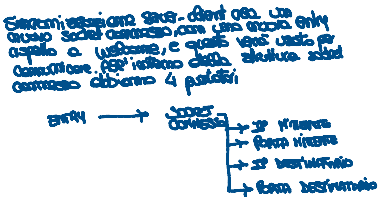
Il protocollo TCP è un servizio orientato alla connessione, ovvero **occorre stabilire una connessione prima di avviare lo scambio di messaggi**, un applicazione che lo invoca ottiene:

- **Servizio orientato alla connessione**, procedura **handshaking**, una volta stabilita la connessione tra client e server abbiamo due socket TCP connessi, questa connessione è **full duplex**, i due processi possono scambiarsi contemporaneamente messaggi e quando il client termina di inviare messaggi la connessione viene chiusa;
- **Servizio di trasferimento affidabile**, TCP trasporta dati senza errori, con il giusto ordine e senza duplicazioni *(se durante la trasmissione c'è stata qualche perdita il protocollo maschera queste perdite)* ed include anche un meccanismo di controllo della congestione, il quale effettua una "strozzatura" del processo d'invio (che sia client o server) quando il traffico in rete appare eccessivo.

Le primitive socket UDP sono: **SOCKET()** , **BLIND()** viste in UDP, **LISTEN()** system call lato server che imposta la ricezione delle richieste di connessione, dico al socket creato, lato server, che arriveranno a lui le richieste di connessione, **CONNECT()** system call lato client che usata per richiedere la connessione con il server, devo quindi specificare, quando la richiamo, indirizzo IP e porta del server, **ACCEPT()** system call lato server che permette alla socket (lato server) di passare in stato di wait…*ATTENZIONE: non passa lo stesso socket in stato di wait, ovvero il socket quando viene creato e istanziato come colui che riceve le richieste prende il nome di welcome socket, il risultato della sc ACCEPT() è quindi quello di creare un socket detto connesso*…**SEND()** è implementata sia dai client che server e serve per inviare pacchetti, **RECEIVE()** anche questa implementata ambo i lati, mette in stato di wait il processo in attesa di nuovi dati da ricevere, **CLOSE()** implementata ambo i lati con una differenza, lato client viene rimossa l'intera struttura socket, lato server abbiamo due socket (welcome e connesso), viene rimosso socket connesso, altrimenti rimuovendo pure welcome non potrei piu' richiedere una connessione al server, chiuderei l'intero servizio dell'applicazione.



Il socket a cui si connette il client per la prima volta è sempre il socket welcome, se la connessione ha successo viene generato un nuovo socket…



Due protocolli di trasporto: TCP, UDP e sicurezza

Nessuno dei due fornisce forme di cifratura dei dati, ne consegue che se ad esempio inoltro un pacchetto con una pwd IN CHIARO, questo viaggia nella rete e potrebbe venire intercettato da un'utente malevolo che così ottiene la pwd. Per intervenire su questo la comunità di internet ha sviluppato un elemento aggiuntivo che arricchisce il protocollo TCP chiamato **TLS** che permette di aggiungere servizi critici di sicurezza tra processi che comunicano tramite TCP, questo arricchimento è implementato a livello applicativo.
Un'altra soluzione sono i **TSL socket API**, in cui viene posto uno strato tra il livello applicativo e il livello di trasporto in cui i messaggi verranno cifrati.
Ultima soluzione è quella di implementare servizi critici di sicurezza a livello di rete.

Scelta tra TCP e UDP

Se si tollerano perdite possiamo pensare di usare UDP, altrimenti TCP.
La scelta di usare TCP per alcune applicazioni è esclusa perché richiede un tempo per stabilire la connessione…