Parliamo di Wireshark

Wireshark ha una modalità di funzionamento chiamata **promisqua**, ed in questa modalità di funzionamento non si comporta in modo "onesto", nel senso che non cattura solo i pacchetti destinati all'interfaccia, ma cattura anche tutti gli altri pacchetti.

Interfaccia

Possiamo selezionare l'interfaccia da analizzare ed iniziare la scansione; otterremo un risultato del genere:

🏁 0:15 lezione 4

Layer applicazione

Applicazioni e protocolli

L'obbiettivo è quello di realizzare delle **applicazioni distribuite** in rete, che sono tipicamente basate su processi distribuiti su macchine diverse e comunicanti tra loro. Lo scopo di questo livello è quello di specificare i protocolli che sono usati per lo scambio di questi messaggi.

Il livello delle applicazioni **non ci dice come deve essere fatta l'applicazione**, ma ci dice come deve essere **strutturato un messaggio** scambiato tra processi che caratterizzano quell'applicazione, **a tempo di esecuzione**.

Quindi ci aspettiamo che il protocollo del livello delle applicazioni specifichi il formato del messaggio, e poi, per ciascun campo del messaggio, il significato e valore di quel dato campo.

Paradigma di comunicazione

Per capire questo paradigma partiamo da un'analogia: **il comportamento di una rete telefonica**.

Quando usiamo una rete telefonica e vogliamo contattare un interlocutore, egli non sa che vogliamo contattarlo, ma il sistema telefonico è predisposto a ricevere telefonate (ad esempio il telefono squilla e l'utente risponde).

Una delle entità comunicanti si predispone a ricevere un contatto, collegandosi al software di rete affinchè possa ricevere messaggi di comunicazione o affinchè possa essere contattato per avviare il dialogo. Stiamo quindi dicendo che affinchè due entità possano comunicare, è necessario che una delle due entità in maniera preventiva si colleghi al software di rete e si ponga in ascolto per ricevere un contatto dall'altra entità.

Il paradigma del client/server

Quando una delle due entità si predispone alla ricezione di contatti da parte di altre entità, parliamo di paradigma **client-server**; in questo paradigma le unità comunicanti sono **asimmetriche**, nel senso che una delle entità si pone in attesa (passiva) di ricevere un contatto dall'altra entità. L'entità **passiva viene chiamata server**, mentre quella attiva, che promuove il contatto ed inizia la conversazione, viene chiamata **client**.

I due ruoli determinano anche **comportamenti diversi** da parte delle due entità, e quindi anche esigenze diverse dal punto di vista **dell'hardware**.

SERVER

Il server ha lo scopo di **erogare servizi**, e quindi di mettere a disposizione dei client dei servizi; gestisce delle **richieste provenienti dai client**, a cui risponderà fornendo degli output.

Poiché essi possono essere contattati da diversi client, i server hanno bisogno di sistemi operativi specifici, e sopratutto **hardware ad elevate prestazioni**; questo perchè all'aumentare del numero dei client la capacità di servizio da parte del server può andare degradando.

CLIENT

Per quanto riguarda i client, si tratta di programmi che vengono avviati dall'utente, e che hanno come scopo quello di chiedere al server dei servizi, e quindi inviare dei messaggi di richiesta per avere degli output; una volta che hanno realizzato la comunicazione con il server possono anche essere spenti, questo non succede dal lato server, che tende a rimanere sempre in esecuzione per soddisfare delle richieste **in ogni momento**.

Paradigma di interazione: richiesta-risposta

Il modello vincola i tipi di messaggi che dobbiamo definire, questo significa che per ogni applicazione (protocollo) che vogliamo definire, dobbiamo definire **due tipi di messaggi**.

Per realizzare questo scambio, e quindi per far inviare dal server un messaggio di richiesta al client, è necessario che il server sia identificabile all'interno della rete; per identificare un entità ad un qualsiasi livello, usiamo un indirizzo. Vogliamo identificare **un processo**, questo perchè è possibile che su un host vi siano in

esecuzione **più processi**: ad esempio un server web, server di posta elettronica, ecc.

E' necessario avere un meccanismo per identificare un processo, e non l'intero host. L'indirizzo di cui abbiamo bisogno, è l'indirizzo che viene usato nella comunicazione end to end, ovvero tra due processi. Questa comunicazione è realizzata dal **livello di trasporto**, e l'indirizzo a cui facciamo riferimento, è l'indirizzo **di trasporto**, che viene usato per la costruzione delle applicazioni. Potremmo anche usare un indirizzo **specifico del livello applicativo**, ed infatti molto spesso vengono usati i cosiddetti **nomi di dominio** per contattare un processo.
Per contattare un server si usa molto spesso un nome di dominio **(domain name)**, avente come prefisso **www.** ; ad esempio, **www.unisannio.it** è un esempio di dominio.

Addressing

Un indirizzo di trasporto è chiaramente l'indirizzo che ci consente di arrivare al processo con il quale vogliamo comunicare, e rappresenta l'indirizzo dell'estremità del canale di comunicazione che realizza la comunicazione tra due processi, e nel caso dell'architettura TCP/IP è composto da due parti:

- **Indirizzo IP** : indirizzo della macchina su cui il processo è in esecuzione.
- **Numero di porta** : consente di individuare su quella macchina uno specifico processo.

Si parte quindi dall'indirizzo IP e si aggiunge un'altra informazione che caratterizza un processo in esecuzione su una macchina. Possiamo dire che l'addressing ha una **struttura gerarchica**, perchè la prima parte ci consente di arrivare alla macchina, e l'altra al processo.

La composizione di questo indirizzo in questa maniera ci consente di recuperare i numeri di porta su ciascuna macchina, siccome non devono essere univoci in tutta la rete, ma solo su quella macchina: ad esempio il numero di porta 8080 che su una macchina ci permette di comunicare su uno specifico processo, può essere usato anche su un'altra macchina.

Primitive di comunicazione Client Server

Le primitive necessarie per realizzare un'applicazione in accordo con il paradigma client server, sono **molto semplici**, e sopratutto, **solo due**.

Abbiamo bisogno di una primitiva che ci consenta di **spedire messaggi** applicativi, ed una che ci consente di **ricevere messaggi**; entrambe le primitive sono usate sia da client che da server, in ordini diversi.

Nel momento in cui dobbiamo far comunicare due processi dobbiamo capire che stiamo operando in un contesto concorrente, ovvero abbiamo due entità in esecuzione che non è detto che siano perfettamente sincronizzate: ad esempio se una spedisce un messaggio non è detto che l'altra sia pronta a riceverlo. Abbiamo bisogno che i due processi si **sincronizzino**: la sincronizzazione viene tipicamente realizzata con una modalità di funzionamento della primitiva di ricezione (**receive(addr, &msg**), utilizzata in **modalità bloccante**: significa che l'invocazione da parte di un processo della **receive()** nel caso in cui non siano disponibili messaggi da fornire al processo invocante, il comportamento è quello di **sospendere l'esecuzione del processo** in attesa che il messaggio arrivi.

La **send(dest, &msg)** ci permette quindi di spedire un messaggio al processo destinatario; la **receive(addr, &msg)** ci dice da quale indirizzo si vuole ricevere il messaggio. Come appena detto, per evitare che il processo vada avanti quando il messaggio non è ancora arrivato, sospende il processo, in attesa che il messaggio arrivi. Questo metodo è particolarmente usato per **sincronizzare i due processi**,

in modo che il processo ricevente attendi l'invio del messaggio.

Ordine di utilizzo delle primitive

Entrambe le primitive vengono utilizzate da entrambi i tipi di processi, sia client che server; il client però le utilizza in un ordine diverso rispetto al server:

- **Client** : Il client utilizza prima la send e poi la receive
- **Server** : il server utilizza prima la receive e poi la send

Esempio: 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);
  }
}
```

Come possiamo vedere dal codice, abbiamo un comportamento **bloccante**, e questo è importante per non far eseguire al server delle operazioni, a meno che non abbia effettivamente ricevuto l'operazione da eseguire. Infatti, se il server non riceve correttamente il messaggio, non potrebbe estrarre **l'opcode** da eseguire, e quindi non potrebbe operare in modo corretto.

Il codice rimane quindi bloccato all'istruzione `receive(FILE_SERVER, &m1);` finchè non riceverà il messaggio.
Quando il messaggio sarà disponibile, il server lo estrae ed utilizza i campi del messaggio.

Esempio: client

```
int copy(char *src, char *dst){
  ...
  int pos = 0;
  do{
    m1. opcode = READ;  //l'opcode è un'operazione di tipo read
    m1.offset = pos;  // campo usato dal server per capire da quale posizione del file deve iniziare a leggere; inizialmente vale 0
    m1.count = BUF_SIZE; //inizializzato come il valore della grandezza del buffer; non avremo overflow

    strcpy(&m1.name, src); //specifichiamo il nome del file; a differenza dei casi precedenti, per assegnare il nome della variabile al campo m1.name, usiamo la funzione strcpy, perchè lavorando su macchine e processi diversi, lo spazio indirizzo dei vari processi potrebbero essere diversi, e non possiamo ragionare con dei puntatori.
    send(FILE_SERVER, &m1); //inviamo il messaggio m1 al server; usare la '&' vuol dire passare per indirizzo, evitando di effettuare la copia del messaggio
    receive(CLIENT, &m1); //subito dopo la spedizione c'è la ricezione. Dopo la spedizione il client va in attesa della risposta del messaggio da parte del server.

    //  se arriviamo a questo punto, vuol dire che il messaggio è arrivato correttamente; in questo caso il messaggio di risposta diventa un nuovo messaggio di richiesta:
```

```
    m1.opcode = WRITE;  // prepariamo il nuovo opcode
    m1.offset = pos;  // aggiorniamo l'offset
    m1.count = m1.result; // scriviamo una quantità di dati uguale a quella ricevuta effettivamente; questo perchè i byte ricevuti potrebbero essere minori rispetto alla dimensione di buffer.

    strcpy(&m1.name, dst);  // il contenuto non riguarda più il file sorgente, ma il file destinazione.

    send(FILE_SERVER, &m1); // il file viene spedito al server
    receive(CLIENT,&m1); // riceviamo il messaggio di ritorno che ci dice quanti byte sono stati effettivamente scritti.

    pos += m1.result;
  }while(m1.result > 0);
}
```

Il client analizzato utilizza le funzionalità messe a disposizione dal server per implementare un'operazione di **copia remota**, che consente di copiare un file sorgente operando da remoto.

Un aspetto importante è la struttura del messaggio:

I campi presenti all'interno del messaggio sono:

- **opcode:** il codice operativo, ovvero il codice dell'operazione che il client chiede al server
- **nome:** ovvero il nome del file
- **offset:** ovvero la posizione all'interno del file dalla quale si deve operare.
- **count:** il conteggio dei byte che vogliamo gestire in un'operazione
- **risultato:** risultato dell'operazione
- **<data>** i dati scambiati

Se torniamo al server, notiamo che viene utilizzato uno **switch** che, a seconda di un determinato **campo del messaggio** (appena visti), esegue una tra le diverse operazioni:

- **create**
- **read**
- **write**
- **delete**

Per ognuno degli opcodi previsti abbiamo quindi funzioni diverse; una volta eseguita l'operazione, **sfruttando altri campi** del messaggio, ad esempio il nome del file da creare, l'operazione **restituisce un risultato** che viene catturato dalla variabile **r** che verrà poi assegnato al campo **m2.result** del secondo messaggio da inviare come risposta al client.

🏁 0:54

Stiamo quindi facendo transitare i dati sulla rete sottoforma di blocchi di byte; dal punto di vista dell'efficienza di comunicazione, significa usare in maniera impropria la rete, infatti potremmo evitare questo scambio.

Sebbene questa applicazione funzioni correttamente, è particolarmente inefficiente: l'inefficienza deriva dal fatto che trasferiamo il blocco dal server verso il client per poi rispedirlo nuovamente al server; impegnamo la reta in maniera non utile, e potremmo evitare il trasferimento portando i byte letti direttamente nel file di destinazione con un'operazione da eseguire sul server, anche se questo significa **modificare le funzionalità del server**.

Riassumendo

In basso abbiamo una sorta di **template** che possiamo usare per la realizzazione del client e del server.

🏁1:24

Una API reale per la programmazione Client-Server

L'API da considerare caratterizza l'interfaccia tra il livello di trasporto ed il livello delle applicazioni nell'architettura TCP/IP; prende il nome di **socket interface** dove il concetto di socket è analogo a quello del file: rappresenta l'astrazione che useremo per la comunicazione così come si utilizzano i file per operare sul disco.

Una socket è un SAP, ovvero **service access point**; è un punto di accesso ai servizi messi a disposizione da un livello, ed in questo caso vogliamo accedere ai servizi messi a disposizione dal **livello di trasporto** (TCP/IP).

BSD Socket interface

E' un'interfaccia nata con UNIX, ed è così utilizzata che oggi viene impiegata anche a livello web.

Possiamo vedere l'interfaccia come un insieme di system call messe a disposizione dal sistema operativo usato per eseguire le operazioni. I nostri processi in esecuzione utilizzano quindi queste syscalls per chiedere al SO l'utilizzo del **dispositivo di rete**.

Socket

Per operare con le **socket** usiamo delle funzioni di sistema: una delle funzioni più importanti è proprio la funzione **socket()**, che ci serve per creare la socket stessa, ovvero un **endpoint** nel canale di comunicazione; un processo che invoca la funzione socket chiede al sistema operativo di creare un punto terminale del canale di comunicazione, che sarà usato secondo un particolare modello di servizio.

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

Il primo parametro ci permette di specificare **l'architettura di rete che vogliamo usare**, nel nostro caso la useremo principalmente con TCP/IP, ovvero PF_INET.

Il secondo parametro ci permette di specificare quale servizio di comunicazione vogliamo usare a partire dalla socket che stiamo creando: ci dice che tipo di comunicazione ci consentirà di realizzare l'endpoint che stiamo creando; possiamo usare due costanti:

- **SOCK_STREAM** : servizio orientato ai flussi
- **SOCK_DGRAM** : servizio orientato ai datagram

Se ad esempio scegliamo di impiegare SOCK_STREAM, stiamo dicendo di voler impiegare il protocollo TCP a livello di trasporto; se chiediamo di usare un modello basato su datagram, diciamo di voler impiegare il protocollo UDP.

L'ultimo parametro, che sarà sempre **0** nei nostri esercizi, viene usato per specificare esattamente il protocollo che vogliamo usare; 0 = default.

🏁 1:35 lezione 4

Il risultato dell'invocazione di **socket()** è un intero, che è il descrittore della socket che verrà usato dalle altre funzioni per poter comunicare.

Essenzialmente, con l'invocazione di **socket()** stiamo creando quella "parte" astratta cerchiata nell'immagine.

Numero di porta

Cerchiamo di capire come poter gestire gli indirizzi di trasporto usati per caratterizzare le socket; con l'invocazione della funzione socket creiamo un terminale di comunicazione, che è raggiungibile localmente al processo che l'ha creato, ma l'endpoint creato non sarà visibile **all'esterno** finchè non assegnamo a quell'endpoint un **indirizzo visibile dall'esterno**. Dobbiamo assegnare quindi un **indirizzo di trasporto** a quella determinata socket.

Il numero di porta ci permette di differenziare i processi che utilizzeranno i vari canali; per poter differenziare questi processi, utilizziamo un numero di porta, ovvero dei numeri che vanno da **zero a 65535** suddivisi in 3 intervalli:

1. **numeri di porta "ben noti"** : 0-1023; i primi 1024 numeri di porta sono riservati per i servizi fondamentali.
2. **Numeri di porta "registrati / riservati"** : 1024-49151; se creiamo un nuovo tipo di applicazione possiamo credere la registrazione di quel servizio associato ad un dato numero di porta.
3. **Numeri di porta dinamici o privati** : 49152-65535; sono quindi utilizzabili per scopi privati, e non sono assegnati a servizi specifici.

Indirizzo socket - punto di vista della programamzione

```
#include <netinet/in.h>

//internet address structure
struct in_addr{
  u_long s_addr;
};
```

```
//socket address structure
struct sockaddr_in{
  u_char sin_family;
  u_short sin_port;

  struct in_addr sin_addr;

  char sin_zero[8];
};
```

La struttura usata per rappresentare l'indirizzo di trasporto si chiama `sockaddr_in`, quindi è un indirizzo di socket, in particolare della famiglia **internet**.

Se siamo in grado di assegnare un indirizzo di trasporto a quel punto terminale del canale, un altro processo è in grado di contattare quell'endpoint:

- **sin_port = 80**
- **sin_addr = 128.2.194.95**

---

fine lezione 4