Riassunto Reti di calcolatori 2

by diverse fonti

Si ringrazia Zazzu

Indice generale

Applicazioni

L'Architettura di Riferimento in questione è quella *Client-Server*. Il primo usufruisce della risorsa, il secondo la fornisce. Per risorsa si può intendere dati (documenti, film, musica) o servizi (prenotazioni, acquisti, servizi bancari).

Un'altra architettura possibile è quella *Client-Server in Cascata*, a tre livelli:
- l'utente della risorsa (il sistema con cui interagisce l'utente umano, es. web browser)
- il punto di accesso della risorsa (che media tra utente e gestore, es. web server)
- il fornitore della risorsa (sistema che gestisce la risorsa, es. dbms)

Il punto di accesso alla risorsa deve essere facilmente accessibile, gradevole ed usabile, sicuro, e in grado di integrare e selezionare fra più fornitori.

Altre architetture…

Nella *Pull And Push Technology* il server "cerca" il client e il client "cerca" il server.

Rilevanti sono le architetture dei *proxy* e del *caching*.

Importante è anche il bilanciamento del carico, sia tra client e server, che fra più server.

I protocolli utilizzati direttamente dal programmatore sono i seguenti:
- per il trasferimento di dati punto-punto, *http*, *ftp*, *tcp* e *udp*;
- per l'invocazione di procedure remote, *rpc* (remote procedure call) e *rmi*

I linguaggi utilizzati sono C, C++ (socket, per trasferire byte – protocollo usato *rpc*) e java (socket per trasferire qualsiasi oggetto e primitive per protocolli applicativi, come http – protocollo usato *rmi*).

Tecnologia *CORBA*.
Il mondo è fatto di oggetti che interagiscono. *CORBA* (common object request broker architecture) consente di vedere applicazioni scritte in linguaggi diversi su piattaforme diverse, in modo unificato, come oggetti usabili in vari linguaggi. Un analogo in ambiente Microsoft è *dcom* (distributed component object model).

Tecnologia *HTML dinamico*.
È il server a produrre pagine html da inviare al client, anche attraverso programmi esterni.
- Tecnologia *cgi* (common gateway interface, in qualunque linguaggio)
- Servlet java (codice java già compilato)

Esempi di tecnologie "interne" al web server sono jsp, asp e php, che consistono in linguaggi di programmazione immersi ("embedded") nel codice html.
Inoltre ci sono architetture che permettono al server di inviare pagine html "quasi standard", che contengono codice che viene eseguito dal client, come *javascript*, *applet*, *active X*, rispettivamente del codice simile a java interpretato dal browser, del codice java già compilato scaricato a parte ed eseguito da una macchina virtuale lanciata dal browser, e delle applicazioni per windows scaricate e installate al volo sulla macchina client.

FTP – File Transfer Protocol

FTP sta per File Transfer Protocol ed è un protocollo standard per il trasferimento di file. Gli obiettivi sono di incoraggiare l'uso di computer remoti, promuovere la condivisione di file, rendere l'utente indipendente dai Sistemi Operativi e ovviamente trasferire dati efficientemente.

La tecnica utilizzata è quella di instaurare due connessioni su *tcp*, una per i comandi ed una per i dati. Gli *user-pi* e *server-pi* sono gli interpreti del protocollo sul lato client e server, gli *user-dtp* e *server-dtp* sono i processi che giacciono sulle macchine client e server.

Architettura.
Lo user-pi instaura una connessione di controllo, tramite comandi che specificano i parametri per la connessione dati (data port, transfer mode, representation type, structure) e la natura dell'operazione (store, retrieve, append, delete). *In pratica lo user-pi si comporta come client del server-pi*. La porta sulla quale connettersi è data (porta 21). La porta dello user-pi che richiede la connessione è arbitraria. Tale connessione è quella che trasferisce i comandi al server.
Dopodiché il server-dtp instaura una connessione con lo user-dtp per il trasferimento di dati, *e si comporta come client dello user-dtp*. La porta del server che richiede la connessione è data (porta 20), mentre quella sulla quale connettersi l'ha scelta lo user (arbitrariamente) passandola come parametro nei comandi, con il comando DATA PORT. In pratica significa "su questa porta lo user è in ascolto". Una volta che lo user-pi comunica una data porta, lo user-dtp deve ascoltare su quella porta, sulla quale il server-dtp effettuerà una connessione.
Per ogni trasferimento di file o lista di file, viene instaurata una nuova connessione. A chiudere la connessione è la macchina di origine del file, che in pratica dichiara che il file è terminato (se il file si sposta dallo user al server, chiude lo user, e viceversa).
La data port non è necessariamente sulla stessa macchina che inizia la connessione. In questo caso lo user-pi deve forzare l'ascolto sull'ulteriore host.
A tradurre i possibili formati delle varie rappresentazioni di file ci pensa *ftp*. Sono supportati ASCII, EBCDIC, file strutturati in record, binario.
Ecco alcuni comandi fondamentali *ftp*, alcuni per il controllo d'accesso, altri per il trasferimento file:
- USER (user name): identificazione dell'utente;
- PASS (password): password dell'utente;
- CWD (change working directory): cambia la directory corrente;
- SMNT (structure mount): mount di un file system diverso da quello corrente;
- QUIT (logout): chiusura;
- PORT (data port): port da utilizzare nella connessione dati (la quale sarà in ascolto). Come argomento c'è sia l'indirizzo IP che la porta;
- PASV (passive): chiede al server di ascoltare una data port aspettando una connessione (per il trasferimento su una macchina diversa);
- TYPE (representation type): tipo di dati da trasferire.

Ecco invece i codici di risposta. Ogni comando è seguito da un codice di risposta. Ogni codice di risposta ha 3 cifre. La prima dice se la risposta è positiva, negativa o incompleta; la seconda indica la tipologia:
- 1yz: risposta preliminare positiva;
- 2yz: risposta positiva, completamento dell'azione prevista dal comando, possibilità di inviare una nuova richiesta;
- 3yz: risposta intermedia positiva, necessità di ulteriori informazioni;
- 4yz: risposta negativa transitoria;

- 5yz: risposta negativa permanente;
- x3y: risposta ai comandi USER e PASS di autenticazione.

Ed ora ecco uno spunto di riflessione su ftp *e sulla sua architettura. È possibile evitare il comando data port?* Il server potrebbe connettersi sulla porta che il client sta usando per la connessione comandi. Infatti la porta con cui il server effettua la connessione è la 20, e quella sulla quale ha ascoltato la connessione comandi è la 21, e sono diverse, e una connessione tcp univoca è identificata dalla coppia di porte, quindi la connessione fra la porta 20 del server e la porta della connessione comandi del client è *diversa* da quella fra la porta 21 del server e la porta della connessione comandi del client. Dato che il server *conosce* la porta con cui il client ha effettuato la connessione comandi, potrebbe usarla per effettuare la connessione dati dalla propria porta 20. Però la cosa non è possibile, perché, ad esempio, se il client richiede due LS al server in rapida successione, il primo andrebbe a buon fine, ma poi il tcp lato server entra in stato di attesa per 2*msl, e la coppia di porte 20-“porta del client” non può più essere usata per 240 secondi, e quindi il secondo comando LS fallirebbe (msl sta per maximum segment lifetime. L’attesa del doppio di questo tempo è una caratteristica del protocollo). Nella realtà invece, ogni comando LS comunica una differente data port. La data port è scelta spesso in maniera automatica.

Progettazione di un Servizio di Rete – La Posta Elettronica

Ci sono due domande che possiamo porci per quanto concerne la progettazione di un servizio di rete: di quante e quali fasi è composta e quali sono le scelte di progetto coinvolte, quando hanno luogo e in base a quali principi. Le risposte a queste domande possono essere utili alla sintesi di un nuovo servizio, all'analisi di un servizio già esistente, o in fase di documentazione, per organizzare in maniera sistematica la descrizione di un servizio.
Come esempio è stato scelto il servizio di posta elettronica, in uso da oltre 20 anni e ben consolidato.
Le fasi da seguire sono di seguito riassunte:

- Analisi dei Requisiti, nella quale si definiscono le caratteristiche del servizio, le specifiche, le necessità, le priorità e gli aspetti critici;
- Specifica delle Primitive di Servizio, durante la quale si definisce l'interfaccia dettagliata del servizio (le primitive che esso mette a disposizione);
- Definizione dell'Architettura e delle Operazioni, in cui si definiscono le applicazioni e i servizi ausiliari coinvolti nella gestione, gli ambienti di supporto, le operazioni e i flussi di informazione;
- Definizione dei Protocolli, nella quale definisco la struttura del PDU e le procedure del protocollo per ogni tipologia di comunicazione.

Esaminiamo tutti questi aspetti nell'esempio scelto della Posta Elettronica.

I requisiti che un servizio di posta elettronica deve soddisfare sono i seguenti:
- gestione dei messaggi in partenza
- spedizione di messaggi a uno o più destinatari
- ricezione di messaggi
- gestione messaggi ricevuti
- preservazione della privatezza dei dati
- sicurezza della consegna
- si ammette anche una consegna differita nel tempo
- configurazione non onerosa da parte degli utenti finali
- interfaccia utente elementare ed intuitiva

Ed ora passiamo alle primitive di servizio dettagliate, riferite ai soli requisiti di gestione dei messaggi in partenza ed in arrivo.

Per i messaggi in partenza:
- composizione di un messaggio
- memorizzazione di un messaggio
- cancellazione di un messaggio
- caricamento di un messaggio

Per la posta in arrivo:
- lettura dei messaggi
- memorizzazione di un messaggio
- cancellazione di un messaggio
- stampa di un messaggio

Infine c'è bisogno ovviamente delle seguenti primitive:
- spedizione di un messaggio
- ricezione dei messaggi per l'utente

Una prima ipotesi di architettura è quella in cui c'è una Connessione Diretta tra mittente e destinatario. È semplice da implementare, il messaggio viene recapitato in tempo reale, e c'è la sicurezza della consegna. D'altra parte però il mittente non può mandare il proprio messaggio se il destinatario non ha lanciato il suo processo o ha il computer spento, inoltre il mittente deve ricordare il nome della macchina del destinatario e non è chiaro come si possa autenticare il destinatario.

In una seconda ipotesi di architettura si potrebbe pensare che il destinatario abbia a disposizione un server di ricezione dei messaggi. In questo modo i processi di invio e ricezione di un messaggio sono disaccoppiati, il ricevente può essere autenticato dal server, il mittente deve ricordare solo il dominio del destinatario. Lo svantaggio è che il server può essere guasto o impegnato, e il mittente può chiudere il suo programma prima che il messaggio sia stato recapitato.

L'architettura definitiva è quella in cui non solo il destinatario ma anche il mittente ha a disposizione un server di invio della posta. Inoltre, per far fronte ai problemi del server del destinatario sovraccarico, si prevedono dei server ausiliari del dominio del destinatario.

Definiamo ora le Applicazioni coinvolte. Esse sono di due tipi:
MUA (Mail User Agent): detto anche *Mailer*, è il programma con cui interagisce l'utente. Implementa l'interfaccia per l'utente, viene eseguito quando si vuole accedere al servizio, e lo si può interrompere dopo aver acceduto alle primitive che si intende utilizzare.
MTA (Mail Transmission Agent): è un'applicazione che fa da intermediaria nel processo di trasmissione del messaggio dalla sorgente alla destinazione. Il servizio è offerto in maniera il più possibile continuativa e stabile nel tempo.

Le macchine che ospitano gli *MTA* possono essere *Outgoing Mail Server* (a cui fa diretto riferimento il *MUA* per l'inoltro della posta), *Incoming Mail Server* (dai quali il *MUA* ritira la posta, spesso coincidenti con il Mail eXchanger primario del dominio) o *Mail eXchanger* (incaricati di ricevere la posta per un certo dominio, la cui lista è definita nel DNS del dominio).

In questo modo, il MUA è facilmente configurabile, quindi si richiede la specifica dell'Outgoing Mail Server e dell'Incoming Mail Server.

È necessario definire, anche a più livelli di dettaglio successivi, come ogni primitiva possa essere evasa tramite una successione di operazioni eseguite dalle applicazioni coinvolte. Ad esempio, una volta stabilito che tutte le primitive relative a invio e ricezione dei messaggi, si può stabilire che richiedendo il salvataggio di una mail, il MUA deve salvare il testo della mail nel file system locale, dopodiché si può passare a stabilire quali sono le operazioni da fare più in dettaglio.

Di seguito sono descritte le azioni per evadere la primitiva "Spedizione di un Messaggio".

- il MUA trasmette il messaggio al suo Outgoing Mail Server
- il Mail Server chiede al DNS locale la lista dei Mail eXchanger in ordine di priorità, cerca di spedire la mail ad uno di essi. Se sono tutti occupati riprova a intervalli regolari. Se non riesce entro tre giorni, notifica al MUA il fallimento.
- Un Mail eXchanger secondario cerca a intervalli di tempo regolari di trasmettere il messaggio al Mail eXchanger primario.

Di seguito sono esposti i servizi ausiliari utilizzati:
- DNS (Domain Name System), necessario all'Outgoing Mail Server per ottenere la lista dei Mail eXchanger del dominio di destinazione;
- X-Server, sistema per la gestione di un interfaccia grafica a finestre comoda all'utente;

- NFS (Network File System), sistema per il file system distribuito che serve all'Incoming Mail Server per accedere ai file delle email sul Mail eXchanger primario (se i due non coincidono).

Si possono distinguere due tipi di Flussi di Informazione.
Il primo è quello che si ha fra il MUA che spedisce il messaggio e l'Outgoing Mail Server, e tra i vari MTA che si passano il messaggio fino a farlo arrivare al Mail eXchanger primario del dominio del destinatario. La conversazione è fra un client che intende spedire uno o più messaggi e un server in grado di accettarli.
Il secondo è quello che si ha tra un MUA che vuole scaricare i messaggi destinati a sé e l'Incoming Mail Server che li conserva. È necessario poter leggere un messaggio senza perderlo, o cancellarlo senza trasferirlo.

Definizione della Comunicazione
Per la diversità fra le due diverse tipologie di flussi di informazione, conviene istituire due differenti sistemi di comunicazione, necessari a trasportare una quantità non definibile a priori di dati. Si definiscono così due protocolli, uno per trasferire messaggi dal MUA fino al server di posta in uscita (smtp), ed uno per scaricare messaggi dal server di posta in entrata fino al proprio MUA (ad esempio pop3), entrambi che viaggiano sul canale *TCP*.

Per definire un protocollo occorre definire la struttura del *PDU* (*Protocol Data Unit*), ovvero il formato delle informazioni trasmesse, siano esse richieste del client o risposte del server; le *procedure* d'uso del protocollo, quali regole sintattiche ed effetto dei comandi; e gli *stati* che assumono le applicazioni a seguito dei comandi emessi e delle risposte ricevute, nonché i comandi ammessi all'interno di ogni stato.

Il Protocollo SMTP ha un PDU fatto in questo modo:
- *HELO <dominio>*: comando di apertura che inizia qualsiasi conversazione in smtp
- *MAIL FROM: <mittente>*: inizia il trasferimento di una mail per uno o più destinatari
- *RCPT TO:<destinatario>*: individua un destinatario. In caso di più destinatari il comando è reiterato. È sottinteso che se il destinatario non appartiene al dominio locale si richiede all'applicazione ricevente di fare da intermediaria verso i Mail eXchanger del dominio del destinatario
- *DATA*: il ricevente tratterà le linee seguenti come facenti parte della mail, finché non riceverà una combinazione speciale di caratteri (un punto ad inizio linea seguito da un "a capo")
- *RSET*: il trasferimento corrente viene abortito (il comando non viene riconosciuto se inserito all'interno di una mail)
- *VRFY <destinatario>*: richiede al ricevente di verificare che il destinatario esista
- *EXPN <alias>*: richiede al ricevente di confermare che l'alias di posta specificato corrisponda ad una lista di utenti (eventualmente ritornata)
- *HELP*: mostra i comandi possibili
- *HELP <comando>*: mostra un piccolo help sul comando specificato
- *NOOP*: non fa nulla, richiede solo una risposta
- *QUIT*: termina la connessione, entrambe le parti chiudono la connessione

Spendiamo ora qualche parola sul protocollo POP3.
I comandi sono composti da 4 lettere ciascuno, le risposte del server sono precedute da +OK se positive, o da –ERR se negative.
Ci sono USER e PASS per autenticarsi, STAT per le statistiche, LIST per le informazioni sui messaggi sul server, RETR per visualizzare i messaggi, DELE per cancellarli, NOOP per non fare niente, RSET per resettare e QUIT per interrompere la conversazione.

Pagine Web Dinamiche: Scripting e CGI

A differenza della richiesta da parte di un client di una pagina web statica, nella quale il server semplicemente individua la risorsa nel suo file system e la manda al client, con le pagine web dinamiche il funzionamento è leggermente più complesso. Tale tecnologia permette di mostrare su Web l'output di programmi, generando "al volo" (on-the-fly) pagine web in base alla richiesta ricevuta. Il contenuto di tali pagine quindi può cambiare di volta in volta in base a vari parametri (soprattutto quelli trasmessi dall'utente stesso).
Le due tecnologie possibili sono lo scripting e la chiamata a processi esterni.
Il primo è il caso in cui nelle pagine web del server è immerso del codice di programmazione in un qualche linguaggio, che viene interpretato ed eseguito dal server prima di inviare la pagina al client, e che contribuisce a creare la pagina stessa.
Esempi sono asp (standard Microsoft che utilizza Active-X scripting, generalmente VBscript o javascript), jsp (che embedda invece codice java) e php (più recente, molto diffuso, simile al Perl o al C).
Semplicemente tale codice immerso nella pagina html serve a generare parti della pagina html stessa.
L'altra tecnologia è il cosiddetto *cgi* (*Common Gateway Interface*), che si pone da interfaccia fra il web-server e il programma applicativo. Determina le regole per il passaggio dei parametri di input dal server web all'applicazione, la raccolta dell'output e l'invio di questo al client.
Il server riconosce una chiamata al cgi dal fatto che la risorsa richiesta si trova in una precisa cartella (spesso /cgi-bin). Spesso gli eseguibili di un web-server sono contenuti in una cartella apposita /cgi-bin esterna agli alberi dei vari siti.
L'input di una cgi è una sequenza di valori o di coppie nome-valore. I dati vengono mandati al server attraverso pacchetti get o post.
Sia i pacchetti get che i post possono essere mandati tramite una form (con attributo method settato rispettivamente su get o su post). Inoltre i get possono essere mandati specificando i parametri nell'url.
Il pacchetto post è indicato per grandi quantità di parametri, e con l'uso delle form. Il pacchetto get è indicato per poche quantità di parametri solo testuali.

I parametri vengono passati al cgi in tre modi:
- come argomenti della linea di comando (nel caso di un link diretto al programma cgi – il programma risale ad essi tramite opportune variabili specifiche, es. argc e argv[]);
- come variabili d'ambiente (nel caso di una form con metodo get o la specifica di una query string nell'url – il programma risale ad essi parsando la query-string);
- come standard input (nel caso di una form con metodo post. Con post i parametri non sono sull'url, ma "in fondo al pacchetto" – il programma risale ad essi parsando l'input della query-string carattere per carattere).

La risposta del cgi program al server è mandata direttamente al client se comincia per *nph* (*non parsed header*), altrimenti viene premesso l'header del pacchetto http dal server e il gateway program aggiunge la riga Content-type: type/subtype seguita da una riga vuota e il resto del pacchetto http.

Un gateway program può essere scritto in qualsiasi linguaggio di programmazione, ogni consultazione prevede il lancio di un nuovo processo sul server. Questo potrebbe essere troppo oneroso, inoltre c'è un potenziale pericolo alla sicurezza, perché i programmi vengono lanciati dal processo server, con tutti i privilegi di quest'ultimo. Comunque sia *cgi* resta uno standard *de facto*.

Socket

Una Socket è un'astrazione software che rappresenta il punto in cui un processo applicativo accede ad una connessione (ad esempio TCP o UDP) tramite una data porta. Come una presa elettrica serve a collegare uno strumento ad una risorsa che è la corrente, così una socket è una presa che serve a collegare un'applicazione ad una data risorsa che è la comunicazione attraverso una rete (al posto della corrente elettrica passano pacchetti del protocollo di trasporto). Le socket nascondono al programmatore tutti i dettagli del livello di trasporto, e rappresentano uno strumento per il colloquio tra processi che risiedono su macchine diverse, diventando così lo strumento base per la realizzazione di servizi di rete.

Le fasi di una connessione TCP sono le seguenti:
- dichiarazione, in cui la macchina dichiara al Sistema Operativo che si intende effettuare una connessione;
- apetura della connessione, in cui il server specifica una porta sulla quale è in ascolto e attende che un client si connetta, mentre il client esegue un tentativo di connessione specificando indirizzo e porta del server;
- scambio dati bidirezionale, ovvero sia in trasmissione che in ricezione per entrambe le macchine;
- chiusura della connessione.

La socket dunque è nient'altro che una primitiva del protocollo di trasporto che serve a creare uno strumento astratto da usare per stabilire la connessione.

In particolare, le primitive TCP per i server sono le seguenti:
socket (crea la socket)
bind (lega la socket ad una porta sulla quale si vuole ascoltare)
listen (mette la socket in ascolto)
accept (accetta una data connessione da un client)
read/write (comunica col client)
Ovviamente le *accept* vengono sempre dopo le *listen*, le *listen* vengono sempre dopo le *bind*, e sulle socket di tipo listening dei server non si può fare una *connect* (primitiva riservata alle socket dei client), né tanto meno una *write* prima di una *read*.

Le primitive TCP per i client sono appunto queste:
socket (crea la socket)
(bind) (lega la socket di un server – non la propria – ad una porta)
connect (si connette ad un server)
read/write (scambio dati)

Di seguito un esempio in linguaggio C di un client e di un server che creano le socket per la comunicazione fra loro. Si fa riferimento ad una macchina sommatore (193.204.161.8) che ascolta sulla porta 2000 e che, ricevendo una sequenza di numeri interi, restituisce ogni volta la somma di ogni numero che riceve con tutti i precedenti. Il programma client si chiama addiziona e accetta come parametri l'indirizzo del server e la porta. Il programma server si chiama sommatore e accetta come parametro la porta sulla quale ascoltare.

Il programma client è strutturato come segue:
- dichiarazione e inizializzazione

- dichiarazione delle strutture necessarie
- verifica della correttezza dei dati di input
- richiesta di una socket al SO

- instaurazione della connessione
- scambio dei dati
- chiusura

```
#include <stdio.h>              /* per usare input-output */
#include <sys/socket>           /* per usare socket */
#include <stdlib.h>
#include <errno.h>              /* gestione degli errori */
#include <netinet/in.h>
#include <netdb.h>

main(int argc, char** argv)
{
    int sock;                   /* descrittore del socket */
    struct sockaddr_in server;
    struct hostent *hp;
    char input[256];

    if(argc!=3) {
       printf("uso: %s <host> <numero-della-porta>\n", argv[0]);
       exit(1);
       }
    /* chiedo al resolver di tradurre il nome in indirizzo ip */
    /* se argv[1] è già un indirizzo, allora hp viene semplicemente aggiornato con l'indirizzo fornito */

    hp = gethostbyname(argv[1]);

    if (hp==NULL) {
       printf("%s: l'host %s è sconosciuto.\n",argv[0],argv[1])
       exit(1);
       }

    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
       printf("client: errore %s nella creazione del socket\n",strerror(errno));
       exit(1);
       }

    /* inizializzazione della struttura server */
    server.sin_family = AF_INET;
    bcopy(hp->h_addr, &server.sin_addr, hp->h_length);
    server.sin_port = htons(atoi(argv[2]));

    /* connessione */
    if(connect( sock, (struct sockaddr *)&server, sizeof(server)) < 0 ) {
       printf("client: errore %s durante la connect\n", strerror(errno));
       exit(1);
       }
    printf("client: connesso a %s, porta %d\n", argv[1], ntohs(server.sin_port));

    /* SCAMBIO DATI E CHIUSURA */
    printf("client: num. o 'quit'? ");
    scanf("%s",&input);
    while( strcmp(input,"quit") != 0) {
       char result[256];
       if ( write(sock, (char *)&input, strlen(input) ) < 0 ) {
               printf("errore %s durante la write\n", strerror(errno));
               exit(1);
               }
       if ( read(sock, (char *)&result, sizeof(result)) < 0 ) {
               printf("errore %s durante la read\n", strerror(errno));
               exit(1);
               }
       printf("client: ricevo dal server %s\n", result);
       printf("client: num. o 'quit'? ");
       scanf("%s",&input);
       }
    close(sock);
    printf("client: ho chiuso il socket\n");
}
/* fine della funzione main */
```

<u>Il programma server è strutturato come segue</u>:

- dichiarazioni ed inizializzazione
- dichiarazione delle strutture necessarie
- verifica della correttezza dell'input
- richiesta di una socket al SO

- inizializzazione della socket
- attesa della connessione
- scambio dati
- chiusura della connessione (ri-attesa della connessione e così via)

```
#include <stdio.h>      /* per usare input-output */
#include <stdlib.h>     /* per EXIT_FAILURE */
#include <sys/socket.h> /* per usare socket */
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <errno.h>      /* gestione degli errori */

main(int argc, char** argv)
{
    int sock;           /* socket di ascolto */
    int sockmsg;                /* socket di dialogo */
    struct sockaddr_in server;

    /* verifica la correttezza dell'input */
    if ( argc !=2 ) {
       printf("uso: %s <numero-della-porta>\n", argv[0]);
       exit(EXIT_FAILURE);
    }

    /* chiedi al SO una socket */
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if ( sock < 0 ) {
       printf("server %d: errore %s nel creare la socket\n", getpid(), strerror(errno));
       exit(1);
    }

    /* inizializza la struttura server */
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = htons(atoi(argv[1]));

    if ( bind(sock, (struct sockaddr *)&server , sizeof(server)) ) {
       printf("server %d: bind fallita\n", getpid());
       exit(EXIT_FAILURE);
    }

    printf("server %d: rispondo sulla porta %d\n", getpid(), ntohs(server.sin_port));

    if ( listen(sock, 4) < 0 ) {
       printf("server %d: errore %s nella listen\n", getpid(), strerror(errno));
       exit(EXIT_FAILURE);
    }

    /* attesa della connessione */
    while(1) {
       int totale=0, len;
       char input[256];

       if( (sockmsg = accept(sock, 0, 0)) < 0 ) {
               printf("server %d: errore %s nella accept\n", getpid(), strerror(errno));
               exit(1);
       }
       printf("server %d: accettata una nuova connessione\n", getpid());

       /* scambio dati */
       while (1) {
               char message[256];

               len = read( sockmsg, input, sizeof(input) );
               if (len == 0) break;
               input[len]='\0';
               printf("server %d: arrivato il numero: %s\n", getpid(), input);

               totale = totale + atoi(input); /* esegui la somma */
               printf("server %d: invio del totale %d\n", getpid(), totale);
               sprintf(message, "%d", totale); /* prepara messaggio */
               write(sockmsg, message, sizeof(message));
       }

       /* chiusura della connessione */
       close(sockmsg);
       prinf("server %d: ho chiuso la socket di dialogo\n", getpid());
    } /* fine del ciclo infinito del server */
    close(sock); /* inutile: non arriverò mai qui! */
    printf("server %d: ho chiuso la socket di ascolto\n", getpid());
}
```

Si ricapitola di seguito in breve, in pseudolinguaggio, i programmi del lato client e server.

Il Client.
Le variabili dichiarate:
- SOCK di tipo int (la socket)
- SERVER di tipo struct sockaddr_in (il server)
- HP di tipo struct hostent* (l'host del server, con vari campi)
- INPUT di tipo char[256] o stringa (l'input)
- RESULT di tipo char[256] o stringa (l'output di volta in volta)

I passi:
- HP inizializzato con l'host del server dato come parametro
- SOCK inizializzata con la primitiva *SOCKET*
- SERVER inizializzato grazie ad HP e alla porta data come parametro
- *CONNECT* (la socket SOCK al server SERVER)
- finchè non quitti:
    - INPUT inizializzato (dall'utente)
    - *WRITE* (con la socket SOCK l'input INPUT)
    - *READ* (con la socket SOCK il RESULT aspettando che si inizializzi) (Bloccante)
- *CLOSE* (la socket SOCK)

Il Server.
Le variabili dichiarate:
- SOCK di tipo int (la socket di ascolto)
- SOCKMSG di tipo int (la socket di dialogo)
- SERVER di tipo struct sockaddr_in (il server)
- TOTALE di tipo int (l'output di volta in volta)
- LEN di tipo int (la lunghezza dell'input)
- INPUT di tipo char[256] o stringa (l'input)
- MESSAGE di tipo char[256] o stringa (il messaggio del server)

I passi:
- SOCK inizializzata con la primitiva *SOCKET*
- SERVER inizializzato con la porta data come parametro (indirizzo qualsiasi esso sia)
- *BIND* (la socket SOCK al server SERVER su quella porta)
- *LISTEN* (la socket SOCK che diventa listening, con massima coda un ToT es. 4)
- ciclo infinito:
    - TOTALE inizializzato a 0
    - SOCKMSG inizializzata aspettando una *ACCEPT* (in base alla socket SOCK) (Bloccante)
    - ciclo scambio di dati:
        - LEN inizializzata con *READ* (con la socket SOCKMSG l'input INPUT così inizializzato) (Bloccante)
        - calcolo totale aggiornato
        - *WRITE* (con la socket SOCKMSG il messaggio MESSAGE inizializzato con TOTALE)
    - *CLOSE* (la socket SOCKMSG)
- eventualmente *CLOSE* (la socket SOCK) (più che altro l'utente esce dal programma)

La funzione "FORK"
Adesso, sempre a proposito delle socket, vediamo un server tcp che fa uso della funzione *FORK*.
Grazie all'uso della funzione *FORK*, al server possono connettersi più client in parallelo, e il server li serve tutti quanti senza dover attendere la fine di una connessione per iniziare la successiva.
La semantica della funzione *FORK* è la seguente: quando si esegue una fork(), il processo attuale che la sta eseguendo si sdoppia creando un processo gemello di sé stesso in tutto e per tutto, compresa ogni variabile e suo valore, tranne che per la variabile restituita dal metodo fork() stesso:

un intero che, nel processo cosiddetto “padre” assume come valore quello del *pid* del processo figlio così generato, mentre nel processo figlio assume valore 0.
Grazie a questa funzione, il processo inizialmente unico del server genera un processo figlio per ogni connessione ad un client da servire. Grazie al Sistema Operativo, essendo vari processi, vengono eseguiti in parallelo e quindi il server può servire contemporaneamente ogni client.
Ci sono però da fare delle osservazioni importanti: dopo una fork, il processo padre e il processo figlio, essendo in realtà l’uno la copia dell’altro, hanno gli stessi privilegi su ogni variabile, comprese la socket di ascolto e quella di dialogo (entrambi posseggono un riferimento alla socket di ascolto e a quella di dialogo). Le operazioni eseguite dai due processi sulle due socket potrebbero interferire, inoltre il livello di trasporto non invia il pacchetto *fin* se tutti i processi che condividono la stessa socket non hanno eseguito una *close*. Ciò significa che se il processo padre non chiude la socket di dialogo, allora se la chiude il figlio non basta a chiudere la connessione, il client rimane appeso e il processo padre potrebbe pure bloccarsi se eccede il massimo numero di “file” che può tenere aperti. Allo stesso modo, se il processo figlio non chiude la socket di ascolto, allora non basta che la chiude il padre: la connessione non si chiude e la porta rimane bloccata.
Per ovviare a questo problema basta fare una cosa semplice: assegnare la propria responsabilità ad ogni processo e disfare ciascun processo delle responsabilità non proprie: il processo padre si occupa di attendere connessioni, i processi figli di servire i client. Quindi subito dopo una fork, il processo padre deve disfarsi (fare una *close*) della socket di dialogo (e continuare così ad attendere altre connessioni “in santa pace, tanto se la vede il processo figlio”), mentre il processo figlio deve disfarsi della socket di ascolto ereditata dal padre (facendo una *close*) così può limitarsi a servire il client e chiudere la propria socket di dialogo, che tanto dal padre è già stata chiusa, così come, quando l’utente interromperà il processo server, eseguendo una *close* sulla socket di ascolto, le copie di quella socket saranno già state chiuse da tutti i processi figli.
Ecco in breve un esempio del codice.

```
if ( fork() == 0 ) {
        printf(“figlio %d: servo io il client\n”, getpid() );
        close(sock); { il figlio chiude la socket di ascolto }

                /* QUI CI VA _SCAMBIO_DATI_ → _CHIUSURA_CONNESSIONE_ */

        exit(0);
} else {
        close(sockmsg); /* il padre chiude la socket di dialogo */
}

                /* TORNA A _ATTESA_DELLA_CONNESSIONE_ */
```

Client e Server UDP
Ora invece guardiamo da vicino come funzionano gli stessi client e server a livello di trasporto su un altro protocollo: UDP (trasporto a Datagramma). E vediamo come vengono usate in questo caso le socket.

Il programma Client è strutturato come segue:
- dichiarazioni ed inizializzazione
- richiesta di una socket al SO
- inizializzazione della socket
- invio del messaggio al server e chiusura

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>

#define MY_PORT_ID      6089 /* la mia porta */
#define SERVER_PORT_ID  6090 /* la porta del server */
#define SERV_HOST_ADDR  “128.119.40.186”
```

```
main() {
	int sock, retcode;
	struct sockaddr_in my_addr;
	struct sockaddr_in server_addr;
	char msg[12];    /* conterrà il messaggio da inviare */

	sock = socket(AF_INET, SOCK_DGRAM, 0);
	if( sock < 0 ) {
		printf("client: errore %s nel creare la socket\n", strerror(errno));
		exit(1);
	}

	printf("client: eseguo la bind\n");

	bzero((char *) &my_addr, sizeof(my_addr));
	my_addr.sin_family = AF_INET;
	my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
	my_addr.sin_port = htons(MY_PORT_ID);

	if ( (bind(sock, (struct sockaddr *) &my_addr, sizeof(my_addr)) < 0) ) {
		printf("client: errore %s nella bind\n", strerror(errno));
		exit(1);
	}

	/* invia il messaggio al server e chiudi */
	bzero((char *) &server_addr, sizeof(server_addr));
	server_addr.sin_family = AF_INET;
	server_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
	server_addr.sin_port = htons(SERVER_PORT_ID);

	printf("client: invio il messaggio al server\n");
	sprintf(msg, "hello world");
	retcode = sendto(sock,msg,12,0,(struct sockaddr *) &server_addr,sizeof(server_addr));

	if (retcode <= -1) {
		printf("client: errore %s nella sendto\n", strerror(errno));
		exit(1);
	}

	close(sock);
}
```

Il programma Server è strutturato come segue:
- dichiarazioni e inizializzazione
- richiesta al SO di una socket
- inizializzazione della socket
- ricevi messaggio dal client e chiudi

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>

#define MY_PORT_ID      6090

main() {
	int sock, nread, addrlen;
	struct sockaddr_in my_addr;
	struct sockaddr_in client_addr;
	char msg[50];

	/****** RICHIESTA DELLA SOCKET E BIND UGUALE A QUELLA PER IL PROGRAMMA CLIENT ******/

	printf("server: attendo il messaggio dal client\n");
	addrlen = sizeof(client_addr);
	nread = recvfrom(sock, msg, 50,0, (struct sockaddr *) &client_addr, &addrlen);

	if (nread < 0) {
		printf("server: errore %s nella recvfrom\n", strerror(errno));
		exit(1);
	}

	printf("server: sono arrivati %d bytes\n", nread);
	if (nread > 0) printf("server: messaggio=%.11s\n",msg);
	close(sock);
}
```

A differenza di TCP, con il protocollo UDP le primitive che servono sono *recvfrom* per il server e *sendto* per il client.

Controllo di Congestione

Un problema importante nell'ambito delle reti di calcolatori è il calo delle prestazioni quando nella rete corrono troppi pacchetti. Inoltre quando i router sono troppo sotto pressione, tendono a perdere pacchetti.
Le cause possono essere molteplici. Spesso dipende dalle linee occupate o lente, o da processori troppo lenti. Aumentare le dimensioni dei buffer ai capi dei calcolatori non basta: anche con buffer infiniti, può scadere il time out per i pacchetti che si trovano in coda da troppo tempo e vengono inviati i duplicati, incrementando così il traffico sulla rete.

Prima di passare a descrivere in dettaglio il problema, è opportuno distinguere tra "controllo di congestione" e "controllo di flusso". Il primo è un controllo che si fa sul traffico "nella rete" (ad esempio, due PC *uguali* che si scambiano informazioni attraverso una rete soggetta a *traffico intenso*), il secondo è un controllo che si fa su una comunicazione "punto-punto" (ad esempio un *supercalcolatore* che trasferisce 10 Gbit/sec ad un PC normale: per quanto la rete sia *libera* il supercalcolatore deve ogni tanto fermarsi).

Un dilemma cui gli ISP devono far fronte è la scelta fra Overprovisioning e Controllo. Con l'overprovisioning si aumenta semplicemente la banda a disposizione in modo tale che il traffico diminuisca, con il controllo invece si assume un gruppo di ingegneria specializzato che supervisioni e gestisca il traffico nella rete, attraverso varie ottimizzazioni. Si dà il caso che, spesso e volentieri, la banda costi meno degli ingegneri.
Il rapporto fra la velocità degli accessi e quello delle backbone cambia ciclicamente: alla fine degli anni '70 la velocità degli accessi era superiore alla velocità dei backbone, mentre negli anni '80 era il contrario, e oggi la velocità degli accessi è di nuovo in forte crescita.

Per controllare il traffico, si può operare a diversi livelli dello strato ISO-OSI.
- trasporto: ritrasmissione, scelta dei timeout (che è l'aspetto più critico), controllo di congestione e di flusso;
- rete: scelta di circuito virtuale o datagramma, accodamento dei pacchetti, politica di servizio, scarto, instradamento (routing), tempo di vita dei pacchetti;
- data link: ritrasmissione, gestione dei fuori sequenza, acknowledgement, controllo di flusso.

Le tecniche di controllo possono essere di due tipi, così come si ha nella teoria del controllo (tipica dell'automatica): ciclo aperto (*open loop*) e ciclo chiuso (*closed loop*). Nella prima tipologia si definiscono a priori tutte le politiche, ad esempio quali pacchetti scartare, dove farlo, e come selezionare il traffico accettato. Nella seconda tipologia, secondo il principio di retroazione, le decisioni vengono prese in funzione di un monitoraggio del traffico stesso, trasmettendo le informazioni ottenute sull'entità del traffico ai punti in cui è possibile prendere tali decisioni. In questo caso il monitoraggio può essere effettuato in base a vari criteri: percentuale di pacchetti scartati, lunghezza media delle code, numero di pacchetti in timeout, ritardo medio dei pacchetti, varianza del ritardo.
In entrambi i casi, le tipologie di intervento sono semplicemente due: o aumentare le risorse, o diminuire il carico.
A loro volta, le tecniche di open loop e closed loop si dividono in sottocategorie. Le tecniche open loop possono essere caratterizzate da azioni prese alla sorgente o lontane dalla sorgente. Le tecniche closed loop possono essere a retroazione implicita (per cui la sorgente effettua il monitoraggio e autonomamente rileva la congestione) o a retroazione esplicita (la sorgente viene avvertita esplicitamente da chi effettua il monitoraggio).

Tecniche Open Loop

Uno dei metodi più intuitivi per evitare congestioni (dicendo “evitare”, si parla di tecniche open loop) è “rimodellare” il traffico alla sorgente, dal momento che una delle maggiori cause di congestione è il fatto che il traffico si presenta “bursty”, cioè “a raffiche”. Il modo migliore per ovviare a questo problema, è forzare i calcolatori a trasmettere in modo uniforme, piuttosto che “sparare” grandi quantità di dati in poco tempo per poi diventare inattivi per lunghi momenti.
L’algoritmo più famoso che risolve questo problema è il *leaky bucket*. L’idea è quella suggerita dal nome stesso: un secchio bucato in fondo. Per quanto venga riempito da sopra con grandi fiotti d’acqua o con poche gocce, quella che restituisce dal foro di sotto è una portata d’acqua costante e regolata (a seconda della larghezza del foro). Quando il secchio diventa pieno, l’acqua in più si perde. La stessa idea viene implementata su un’interfaccia che disaccoppia il calcolatore dalla rete. Il calcolatore mette in coda i pacchetti con qualunque distribuzione temporale, ma l’interfaccia ha un tempo di servizio costante per inviare i pacchetti sulla rete. In questo modo si appiattiscono i burst, riducendo così le possibilità di una congestione. La variante più usata è quella che al posto degli X pacchetti alla volta, fa passare dal “foro” un tot di byte alla volta.
Una variante ancora migliore è quella del *token bucket*. Con questo metodo, il secchio invece che di pacchetti (o byte) è riempito con dei “token”. Quando arrivano pacchetti (o byte), essi “spendono” un tot di token per passare. Se non ci sono abbastanza token nel secchio, i pacchetti sono segnati come non-conformi (e si adottano diverse iniziative sui pacchetti non-conformi). Ogni ΔT stabilito, il secchio si ricarica con un tot di token (c’è comunque un massimo numero di token che può contenere il secchio). Come il leaky bucket, il token bucket funziona meglio se orientato ai byte. Grazie al token bucket, a un calcolatore che resta inattivo per molto tempo è permesso di non “sprecare” questo tempo, accumulando token sulla propria interfaccia (non più della capacità del secchio, comunque), in modo tale che gli sia permesso spendere quel tempo accumulato con un piccolo burst (prima di tornare ad essere costretto a inviare in modo regolarizzato, in funzione del tempo di ricarica dei token sul bucket).

Un’altra tecnica open loop per non congestionare la rete è contrattare il flusso prima di instradare i pacchetti. L’host mittente, prima di stabilire una connessione o di spedire datagrammi, propone alla rete alcuni parametri, che possono essere accettati o contrattati.

Ancora nell’ambito dell’open loop, possono essere stabiliti dei criteri di accodamento dei pacchetti sui router di rete. Possibili criteri possono essere *fifo* (che però non discrimina fra sorgenti di traffico, risultando non equo. Una singola sorgente potrebbe acquisire una percentuale illimitata di banda a scapito delle altre), *fifo con priorità* (si può fare uso del campo “type of service” per discriminare fra più code), o *accodamento equo* (una coda per ogni coppia sorgente-destinazione, con politica “round-robin” sulle diverse code).

Inoltre, se non si riesce a scongiurare la congestione con una gestione ottimizzata delle risorse, non resta che passare alle maniere forti: scartare pacchetti. I router possono decidere di scartare pacchetti se soggetti a traffico eccessivo. Quali pacchetti scartare, dipende dall’applicazione. In una applicazione ftp conviene scartare i pacchetti più nuovi, cosicché eventuali ripetizioni siano più corte possibili (criterio *wine*, il vecchio è preferito al nuovo). In una applicazione multimediale conviene scartare i pacchetti più vecchi in favore di quelli più nuovi, che sono più importanti (criterio *milk*, il nuovo è preferito al vecchio). Altro fattore che può influenzare lo scarto è l’eventuale priorità dei pacchetti (secondo vari criteri, anche economici).

Tecniche Closed Loop

In generale le tecniche Closed Loop consistono nel rilevare man mano la situazione del traffico e comportarsi di conseguenza, avvisando la sorgente. Questo si può fare mandando pacchetti speciali dal punto di rivelazione (il che però può addirittura incrementare il traffico), oppure facendo uso di campi speciali nei pacchetti, o infine mandando pacchetti di esplorazione periodici (come gli elicotteri che esplorano un tratto di autostrada per avvertire le stazioni radio del traffico). La tempistica è un fattore molto importante: azioni repentine provocano oscillazioni, mentre azioni troppo lente risultano inutili.

Il controllo di congestione tcp è un tipico esempio di tecnica closed loop end-to-end con feedback binario implicito. In poche parole tcp si accorge della congestione notando pacchetti scartati, e decide se bisogna agire oppure no. Se decide che c'è congestione, non instaura più circuiti virtuali finché la congestione non scompare. Una variante è la potatura temporanea dei rami della rete che sono congestionati, permettendo così l'instaurazione di altri circuiti virtuali solo fuori dalle aree congestionate.

Un altro metodo è il seguente: ogni router effettua un monitoraggio delle proprie linee, verificando a intervalli regolari se una linea è eccessivamente utilizzata, grazie ad un opportuno valore u ricavato come segue:
$u_{nuovo} = a\ u_{vecchio} + (1 – a)\ f$
con u il valore da controllare, che se supera una certa soglia significa che la linea è congestionata;
f è un valore binario che vale 1 se la linea è utilizzata in quel momento e 0 altrimenti;
a è una costante decisa a priori tra 0 e 1, che più è alta, più il cambiamento di u risulta graduale.
Se u supera la soglia stabilita, la linea del router entra in stato di warning, e si applicano i provvedimenti. Uno di questi provvedimenti può essere l'invio dei cosiddetti *choke packets*. Viene inviato un pacchetto speciale detto appunto choke packet alla sorgente, marcandolo in modo che non generi altri pacchetti choke packet lungo il percorso. Quando la sorgente riceve un choke packet fa due cose: dimezza il traffico (la decrescita è quindi esponenziale) e ignora altri choke packet per un dato intervallo di tempo. Se quindi, dopo questo tempo, arriva un altro choke packet, la sorgente continua a dimezzare, altrimenti fa ricrescere il traffico, ma in questo caso molto più gradualmente, per non rischiare oscillazioni.
Un problema di questa tecnica è che è possibile far sì che le sorgenti non rispettino i choke packets.
Un altro provvedimento diverso dai choke packets che i router possono prendere se una propria linea supera il valore di soglia è il cosiddetto *early random drop*, ovvero scartare qualche pacchetto a caso con una certa probabilità di scarto, prima che la situazione diventi troppo pesante. Tale tecnica, accoppiata con tcp, è molto efficace per il controllo di congestione.

Alcune osservazioni. Lo standard ISO stabilisce che il livello migliore per il controllo di congestione dovrebbe essere il 3 (lo strato network), ma attualmente TCP (trasporto, livello 4) è lo strumento fondamentale per tale controllo. Ciò nondimeno, anche lo strato network contribuisce con la gestione delle code sui router. Inoltre, molte delle tecniche descritte finora, possono essere applicate anche al livello 3. Comunque sia:
OSI indica il livello 3 perché
- le congestioni avvengono *nella rete*,
- il livello 3 dovrebbe mostrare al livello 4 un servizio nel quale questi problemi sono trasparenti;
TCP/IP indica il livello 4 perché
- per una scelta fondamentale, si vuole mantenere il livello 3 più semplice possibile, per ragioni di efficienza e di mercato aperto.

Tecniche di trasporto

Esaminiamo ora le tecniche per il trasporto usate in TCP. In particolare l'instaurazione di connessioni (con il metodo di Tomlinson e sue varianti), il tcp per i servizi interattivi (riscontri ritardati e algoritmo di Nagle), il tcp per flussi di grandi quantità di dati (passando in esame la finestra di controllo di flusso, l'algoritmo di Clark, la finestra di controllo di congestione, slow start e congestion avoidance), il timeout e la ritrasmissione (misura del ritardo roundtrip e algoritmo di Karn), il fast recovery e il fast retransmit. La precedente è solo una breve panoramica, ora vedremo in dettaglio ciascuna di queste caratteristiche di tcp, ma prima è opportuno fare delle premesse.
Tcp è stato implementato su varie piattaforme e le implementazioni, sebbene rispettino lo standard, differiscono in molti aspetti, anche importanti. Esempi di alcune implementazioni sono 4.3 BSD Tahoe (1988), 4.3 BSD Reno (1990, quella a cui si farà riferimento) e 4.4 BSD (1993). BSD sta per *Berkeley Software Distrubution*, ambiente di sviluppo Unix originato all'università di Berkeley, in California.

Instaurazione di Connessioni – come scegliere il numero iniziale di sequenza (ISN) delle unità dati

Uno dei principali problemi cui un protocollo di trasporto deve far fronte è il fatto che un eventuale ritardo nella spedizione dei pacchetti, crea dei duplicati degli stessi pacchetti che possono essere molto pericolosi. Ad esempio, in un servizio di prelievo bancario, se ci sono dei duplicati è possibile che venga effettuato un secondo prelievo non richiesto a causa dei duplicati su una stessa connessione.
A questo problema si ovvia numerando le unità dati (siano esse byte o pacchetti, ma nel seguito per semplicità si farà riferimento sempre a pacchetti). In questo modo pacchetti relativi a connessioni differenti hanno numerazioni differenti, anche se le connessioni differenti hanno lo stesso identificatore di connessione (ricordiamo che una connessione è identificata dalla coppia ip/porta su entrambe le macchine). Tale numerazione si implementa attraverso i cosiddetti *numeri iniziali di sequenza ISN (Initial Sequence Number)*. In pratica si utilizza un ISN diverso per ogni connessione.
Per assegnare gli ISN si fa uso del semplice orologio di cui è munito qualsiasi computer. Si prendono i k bit meno significativi del valore dell'orologio e con tale valore si marca il primo pacchetto di una certa connessione. In questo modo ciascuna connessione numera le proprie tpdu a partire da un numero diverso. Esiste in questo modo una relazione lineare tra tempo e numero di partenza di una connessione. Si noti che k deve essere abbastanza grande, perché le vecchie tpdu devono già essere state eliminate quando la stringa di k bit assume di nuovo gli stessi valori.

Analizziamo ora il seguente problema: se un calcolatore si ferma (andando in crash) e poi riparte, non sa quali erano i numeri assegnati ai pacchetti spediti prima del crash, e così rischia di riusare gli stessi numeri (non si parla di numeri iniziali, bensì di semplici numeri di sequenza dei pacchetti, 1, 2, 3, 4 e così via).
A questo problema si potrebbe pensare ad una soluzione del genere: quando riparte, il calcolatore aspetta un tempo pari al tempo massimo di sopravvivenza di un pacchetto nella rete prima di ricominciare a spedire pacchetti, in modo da essere sicuro che tutti i pacchetti che aveva spedito siano morti. Questa soluzione è inefficiente perché tale tempo può essere anche piuttosto lungo (e si potrebbero creare sequenze di pacchetti con numeri mancanti).
La soluzione migliore invece è la seguente: un numero di sequenza di pacchetto non può essere utilizzato per un certo periodo T che precede l'istante in cui quel numero è potenzialmente utilizzabile come numero iniziale di sequenza. Questo costringe i pacchetti o a seguire un inoltro ottimale (invio del pacchetto 1 all'istante 1, del 2 all'istante 2, del 3 all'istante 3, ecc.) o tutt'al più il pacchetto N, se non all'istante N, viene inoltrato a un istante che segue N e non nei T secondi

prima. Da notare che essendo proibito inviare il pacchetto N solo in quei T secondi, in effetti quel pacchetto *può* essere inviato in un istante che precede N: basta che T sia abbastanza piccolo.
Se mettiamo sull'ascissa di un grafico il tempo, e sull'ordinata i numeri assegnabili ai pacchetti, possiamo notare una striscia diagonale che rappresenta la "zona proibita". Dato che i numeri iniziali di sequenza possibili (raffigurati sull'ascissa) sono modulari, la "zona proibita" si replica ogni certo periodo (vedere grafici sulle dispense, pag. 6-8).
Dato che è possibile che dopo un avvio lento i pacchetti comincino ad essere spediti molto velocemente, è opportuno avere un orologio abbastanza veloce (per permettere al più presto i vari numeri di sequenza per i pacchetti). Ricordiamo che dato che la zona proibita si replica, è possibile entrarvi da sinistra quando l'orologio torna a 0, quindi è opportuno prendere valori di k molto alti, come già detto in precedenza.
L'rfc 793 (ovvero lo standard TCP) prevede l'uso di un contatore (non di un orologio) a 32 bit, con un incremento ogni 4 microsecondi. Nella pratica si fa uso di un contatore che si incrementa di 64000 ogni mezzo secondo e ad ogni nuova connessione. Il contatore è inizializzato a 1 allo startup, e si ha un "quiet time" di 2*msl (maximum segment lifetime, tempo di attesa caratteristico del protocollo dopo ogni connessione).

<u>Tcp per servizi interattivi – obiettivo: ridurre il numero di pacchetti spediti</u>

Premettiamo che tcp possiede un certo grado di libertà sui pacchetti che gestisce: il mittente può ad esempio aspettare che i dati si accumulino prima di inviarli, e il destinatario può aspettare che i dati si accumulino prima di riscontrarli (ad eccezione dei dati urgenti). Inoltre le prestazioni di tcp possono essere largamente influenzate dalle politiche di trasmissione adottate.

Vediamo ora come si comporta tcp con telnet (protocollo applicativo utilizzato per connettersi con un host ad un server remoto, spesso su linea di comando). Consideriamo una connessione telnet ad un editor (di testo) online.
Caso Peggiore
- viene premuto un tasto
- il carattere da telnet arriva a tcp
- tcp prepara un segmento di 21 byte (20 di tcp header + 1 byte)
- ip lo incapsula in un pacchetto di 41 byte (21 per i dati tcp e 20 di header ip) e lo invia
- il ricevente riscontra subito con un pacchetto di 40 byte (20 tcp header e 20 ip header)
- quando l'editor legge il byte, tcp invia una variazione della finestra di un byte (incapsulato in un pacchetto ip da 40 byte)
- quando l'editor processa il byte, invia l'eco (incapsulato in un pacchetto ip di 41 byte).

Per digitare un carattere sono così stati utilizzati 4 segment e 162 byte ip.
La tecnica adottata per risolvere questo tipo di problemi è di inserire un ritardo dai 200 ai 500 ms degli ack e delle notifiche di variazione della finestra, sperando di poter trasmettere altri dati. Così se l'editor ha un tempo di eco di circa 200 ms il riscontro consta complessivamente di 41 byte ip, e i tempi risultano dimezzati (sì, GdB dice proprio "i tempi", a me sembra siano i byte ad essere dimezzati). In ogni caso però il mittente spedisce 41 byte per ogni carattere. In sostanza il problema è che se per ogni carattere serve un header tcp e un header ip vengono sprecati 40 byte alla volta ogni volta che si vuole spedire 1 byte. A risolvere questo problema subentra l'algoritmo di Nagle.
Algoritmo di Nagle
- Se i dati arrivano a tcp da trasmettere un byte alla volta, si invia il primo e si memorizzano i successivi in un buffer;
- i byte memorizzati nel buffer sono spediti in un unico segment solo quando il primo byte è stato riscontrato, dopo la spedizione si ricomincia ad alimentare il buffer fino al prossimo riscontro, e così via;

- in questo modo, se l'utente digita velocemente e la rete è relativamente lenta, ogni segment può trasportare molti byte;
- se si arriva a saturare la finestra o si raggiunge la massima dimensione di un segmento, viene effettuato comunque l'invio;
- si tratta di un algoritmo ampiamente usato, ma disabilitato in applicazioni x-windows per evitare movimenti inconsulti del cursore del mouse.

Dato che le spedizioni avvengono ogni volta che arriva un ack, date spedizioni sono tanto più frequenti quanto più la rete è veloce. Quindi gli effetti dell'algoritmo si notano di più nelle reti geografiche, caratterizzate da ritardi consistenti, più che nelle reti locali più veloci.

Tcp per flussi di grandi quantità di dati – il gioco delle finestre scorrevoli

La finestra di controllo di flusso è una tecnica tcp per controllare il flusso di byte da un calcolatore ad un altro. In breve, ogni calcolatore sa quanto spazio ha ancora disponibile sul buffer, che non è altro che *RcvBuffer – (LastByteReceived – LastByteRead)*, e lo comunica in un campo speciale di 16 bit dell'header tcp degli ACK che manda alla sorgente dei dati. Questo campo, detto appunto *window*, specifica il numero di byte che il ricevitore può accettare, a partire da quello specificato nel campo ACK (che è il prossimo del quale è in attesa).
Ad esempio, mettiamo che il mittente vuole mandare il pacchetto 12, ma gli arriva dal ricevente un ACK del pacchetto 5 in cui dice che la finestra è di 4. 12-5 = 7 > 4, quindi il mittente non manda ancora il pacchetto 12 perché sembra che la finestra sia già più che chiusa. Poi gli arriva il riscontro del pacchetto 6 in cui dice che la finestra è sempre di 4, quindi il mittente attende ancora. Poi gli arriva il riscontro del pacchetto 7 e la finestra si è aperta anche un po': ora è 7. Il mittente deve mandare il 12, e ora lo fa (12-7=5<7). E può mandare anche il 13 e il 14 (infatti una finestra di 7 comunicata sull'ACK del 7 significa che dal pacchetto 7 in poi possono essere spediti 7 pacchetti, cioè possono essere spediti il 7, l'8, il 9, il 10, l'11, il 12, il 13 e il 14). Però attenzione, perché nel frattempo potrebbero arrivare altre notizie "dal fronte" che comunicano un'altra variazione della finestra scorrevole.
Uno dei problemi delle finestre di controllo di flusso è che per quanto grandi siano i blocchi di dati inviati, l'applicazione ricevente potrebbe usarli un byte alla volta. Ogni volta che legge un byte, potrebbe inviare un segmento tcp per indicare che la finestra si è allargata di un byte, e il mittente potrebbe spedire questo byte, e così via per tutta la durata della trasmissione, con enorme overhead. Questa è definita *silly window syndrome*. Abbiamo già visto come l'algoritmo di Nagle risolve il problema se è il mittente a causare tale sindrome. Se invece è causata dal ricevente, si adotta la soluzione di Clark, che stabilisce che le variazioni della finestra vanno comunicate esclusivamente se questa raggiunge metà dell'ampiezza massima o la massima dimensione del segmento negoziata all'inizio della connessione (mss = maximum segment size). Come si può vedere, i problemi affrontati da Nagle e Clark sono complementari.

Si sa che tcp contribuisce largamente al controllo di congestione con accorte politiche di trasmissione. Tcp avverte la congestione quando nota che i timeout di molti pacchetti scadono prima che questi vengano riscontrati. Così, tcp mette a disposizione una seconda finestra scorrevole, adibita al controllo di congestione.
Quando tcp capisce che la rete è congestionata, riduce l'ampiezza della finestra di controllo di congestione. Dal punto di vista di un processo tcp, se la finestra di controllo di flusso è un limite che mi manda il mio interlocutore (attraverso gli ACK, per questo c'è il campo window nell'header tcp), la finestra di controllo di congestione è una propria autolimitazione che si impongono i mittenti (e quindi non essendoci niente da comunicare, non c'è un campo simile nell'header tcp). Quindi, per capire quanti byte può spedire, un processo mittente tcp utilizza la finestra di controllo di flusso comunicata dalla destinazione, e la finestra di controllo di congestione gestita da sé stesso. La finestra che viene usata per spedire, ovviamente, è la più piccola fra le due.

Algoritmo Slow-Start
Con l'algoritmo Slow-Start, la finestra di controllo di congestione è inizializzata a mms (maximum segment size, la dimensione massima del segmento concordata a inizio connessione). Il mittente manda tutti quei dati(sempre che la finestra di flusso lo permetta), e se arrivano i riscontri in tempo, la finestra di congestione aumenta delle dimensioni di tutti i dati mandati (in pratica raddoppia). Quindi ripete il procedimento: manda tutti i dati permessi e se riceve tutti i riscontri per tempo, la finestra raddoppia di nuovo. Se invece arrivano qualche timeout, allora la finestra dimezza e si ferma, perché considera quello il valore buono per non congestionare la rete. Ad esempio, se mms vale 1 KB, all'inizio il mittente può mandare 1KB. Lo manda, arrivano tutti i riscontri, allora la finestra diventa 2KB. Il mittente manda 2KB, arriva il riscontro del primo pacchetto da 1KB e la finestra aumenta a 3, arriva il secondo riscontro e la finestra aumenta a 4 KB (quindi è raddoppiata). Stesso procedimento, una volta arrivati tutti e quattro i riscontri, la finestra è arrivata a 8 KB. Parte il blocco con tutti e otto i pacchetti, e stavolta abbiamo un timeout prima che arrivino tutti i riscontri. La finestra allora si risetta a 4 KB (così secondo Tanenbaum. Secondo le dispense di GdB invece la finestra viene riportata a mss, e dovrebbe ripartire quindi da 1 KB appena riceve il primo timeout. Bah, l'importante è che comunque la finestra si riduce per sopprimere la congestione in atto).
L'algoritmo viene chiamato "avvio lento" perché parte piano piano, però in realtà poi cresce esponenzialmente!

Algoritmo Slow-Start con variante Congestion Avoidance
Questa variante è ancora più efficace. Si stabilisce un valore di soglia, all'inizio pari di solito a 64 KB, e la finestra di congestione si inizializza al solito a mss. Però con questa tecnica, la finestra, ogni volta che arrivano tutti gli ack senza timeout, ingrandisce esponenzialmente *soltanto* fino al valore di soglia attuale, dopodiché, ingrandisce linearmente (di mms ogni volta). Appena vede un timeout, tcp fa ripartire da capo la finestra (riportandola a mss), e setta la soglia alla metà del valore che aveva raggiunto la finestra.
(Differenza tra Tanenbaum e GdB: Tanenbaum parla del "blocco" di pacchetti, che una volta riscontrato tutto quanto, ovvero tutti i pacchetti, la finestra raddoppia se siamo sotto la soglia, o aumenta di mss se siamo sopra. GdB parla del singolo pacchetto, che se riscontrato fa aumentare la finestra di mss se sotto la soglia o di $mss^2/cwnd$ se sopra la soglia. In sostanza però, se si fanno i calcoli, è la stessa identica cosa.)
Esempio: soglia iniziale a 64 KB e finestra che parte da mss=1 KB. Parte 1 pacchetto da 1 KB (supponiamo che tutti i pacchetti che partono siano massimizzati per semplicità), viene riscontrato, la finestra sale a 2 KB. Partono i due pacchetti, vengono riscontrati, la finestra sale a 4 KB. Partono i quattro pacchetti, vengono riscontrati, la finestra sale a 8 KB. Partono gli 8 pacchetti, vengono riscontrati (wow bella libera questa rete), la finestra sale a 16 KB. Partono i 16 pacchetti, stavolta la rete non gliela fa, e arrivano i primi timeout: la soglia da 64 cade a 8 KB perché a quanto pare è quella la capacità della rete, e la finestra ricomincia da capo da 1 KB. Parte il pacchetto, viene riscontrato e la finestra arriva a 2 KB. Partono i due pacchetti (che palle di nuovo!), vengono riscontrati, e la finestra arriva a 4 KB. Partono i quattro pacchetti, vengono riscontrati, e la finestra arriva a 8 KB, attenzione, ora ci troviamo oltre la soglia, da qui in poi dobbiamo andarci cauti. Partono allora gli 8 pacchetti, vengono riscontrati, bene, ora la finestra aumenta a 9 KB. Partono i nove pacchetti, vengono riscontrati, la finestra aumenta a 10 KB (qui andiamo piano piano), partono i dieci pacchetti, vengono riscontrati, la finestra aumenta 11 KB. Partono gli undici pacchetti, vengono riscontrati, la finestra aumenta a 12 KB. Partono i dodici pacchetti, di nuovo il timeout! La finestra riparte da capo e la soglia si setta a 6 KB. E così via.
(In realtà GdB ci dice anche che quando la soglia si setta alla metà della finestra corrente, in realtà si setta "alla metà del minimo fra la finestra di controllo di congestione e la finestra di controllo di flusso corrente", il che mi sembra giusto, e comunque non meno di 2mss, il che mi sembra anche

giusto dato che altrimenti non si permetterebbe la prima fase di crescita esponenziale della finestra di controllo di congestione.)
Questo algoritmo consta quindi di due parti: la prima parte, della crescita esponenziale, è lo Slow Start. La seconda, quando la finestra cresce linearmente, è detta di Congestion Avoidance, perché "sta attento" ad evitare la congestione, piuttosto che ad accorgersi che c'è e porvi rimedio. Questa è una caratteristica del *tcp tahoe*.

Timeout e ritrasmissione – strategie cruciali

Il protocollo tcp è basato sul sistema di timeout e ritrasmissione dei segmenti. Ogni volta che viene spedito un segmento, parte un timeout relativo a quel segmento, e se il segmento non viene riscontrato entro il tempo limite, allora viene ritrasmesso. La scelta del timeout è cruciale: se troppo breve, si rischia di congestionare la rete, se troppo lungo si perde in efficienza.
È risaputo che è molto più difficile scegliere un timeout di trasporto piuttosto che uno di data link: infatti, dato un link fisico, il tempo di arrivo di un ack su quel link è facilmente predicibile, quindi basta scegliere un timeout di data link leggermente maggiore del valore atteso. La varianza del tempo di ack sulla rete, invece, è molto maggiore. E la stessa varianza può cambiare continuamente. Di conseguenza, il timeout tcp viene cambiato continuamente.

Algoritmo di Jacobson
- Per ogni connessione viene gestita la variabile rtt (roundtrip time), che misura la stima corrente del tempo per ricevere l'ack dal destinatario.
- quando viene inviato un segment che viene riscontrato in tempo, si misura questo tempo M, e viene aggiornato il nuovo rtt nel seguente modo:
$rtt_{nuovo} = \alpha\ rtt_{vecchio} + (1 - \alpha)\ M$
dove α è una costante che più è alta più rende graduale la variazione di rtt. Di solito viene impostato a 0,875.
- il timeout corrente è sempre β rtt.
In passato veniva impostato β=2, poi si è osservato che un valore costante non era efficace, e così è nata una variante secondo la quale β dipende dalla deviazione standard della distribuzione del tempo di ack. Dato che questa si può misurare velocemente con la deviazione media, a calcoli fatti si ha che la deviazione media D si calcola così:
$D_{nuovo} = \alpha\ D_{vecchio} + (1 - \alpha)\ |\ rtt - M\ |$
E varie implementazioni di tcp utilizzano timeout = rtt + 4D

Algoritmo di Karn
Uno dei problemi che sorgono con questo modo di fare è che quando c'è ritrasmissione di segment, se arriva un ack non è chiaro a quale istanza di trasmissione esso si riferisca. Ciò ovviamente può influenzare il calcolo di rtt. L'Algoritmo di Karn impone che per i segment ritrasmessi, rtt non venga modificato, e che il timeout sia duplicato ogni volta che un segment non riesce a passare, fino a che non arriva correttamente.

Fast Recovery e Fast Retransmit – tahoe e soprattutto reno

Qualche breve speculazione finale sulle tecniche di trasporto.
Un aspetto ancora inesplorato di tcp è il fatto che tcp deve immediatamente generare un ack (ack duplicato) quando gli arriva un pacchetto fuori sequenza. Questo tipo di ack non può essere ritardato. Quando viene ricevuto un ack duplicato non è chiaro se dipenda da un pacchetto perso o da un pacchetto fuori sequenza. Nel primo caso comunque, ci si aspetta che arrivino vari ack duplicati con lo stesso numero; nel secondo caso arriverà invece un ack duplicato con un numero di byte superiore. In sostanza se un pacchetto arriva fuori sequenza, significa che qualche pacchetto

precedente si è perso, per questo si avverte repentinamente la sorgente. Esempio: il mittente spedisce il pacchetto 1 e riceve un ack, poi spedisce il pacchetto 2 e riceve un ack, poi spedisce il pacchetto 3 e riceve un ack, il pacchetto 4 si perde lungo la rete, quindi al destinatario arriva il pacchetto 5, che manda subito ack duplicati del pacchetto 3 alla sorgente, per dire che il prossimo pacchetto che si aspetta è il 4 (il tutto mentre magari continua a ricevere i pacchetti 6, 7, 8…).

Algoritmo Fast Retransmit
Il segment che sembra essersi perso viene ritrasmesso immediatamente, senza attendere il timeout.

Algoritmo Fast Recovery
Se il segment viene ritrasmesso senza aspettare il timeout, non si fa ripartire slow start perché ci si aspetta che si sia perso un solo segment, e che non ci sia una congestione rilevata, quindi sarebbe una misura troppo drastica far ripartire slow start.

In pratica Fast Retransmit e Fast Recovery vengono usati assieme. Quando si riceve il terzo ack duplicato, si modifica la soglia come per congestion avoidance (viene portata alla metà della finestra minima tra quella di flusso e quella di congestione), si ritrasmette il segment mancante e si fa salire la finestra di flusso in modo lineare, facendola ricominciare, se non dall'inizio, dal valore della soglia. Questa è una caratteristica di tcp reno.

Ancora su TCP

Questa sezione aiuta a comprendere meglio tcp e ad esaminare alcuni suoi interessanti aspetti.
Supponiamo che tcp abbia una conoscenza completa dello stato della rete che i suoi pacchetti devono attraversare. Qual è il miglior comportamento che tcp possa avere e come può ottimizzare il suo throughput? Ricordiamo che in ogni caso la conoscenza della rete da parte di tcp è molto limitata: come può allora tcp eseguire efficientemente i propri doveri?

<u>Tcp con conoscenza completa</u>

Supponiamo che: 1) il routing sia stabile, 2) tcp sappia tutto delle caratteristiche del percorso dalla sorgente alla destinazione, e 3) non ci sia traffico.
Supponiamo che pc1 debba mandare un tot di pacchetti al pc2, e che sappia la situazione di tutta la rete che c'è nel mezzo, cioè ritardo e velocità di ciascuna linea, e capacità del buffer di ciascuno dei router, nonché la finestra di controllo annunciata dal pc2. Sapendo tutto ciò, è ovvio che a pc1 convenga scegliere una velocità di trasmissione che sia la minore fra le velocità di linea di tutte le linee, in modo da non intasare i router. Sapendo la dimensione dei propri pacchetti, in base alla velocità calcolata in precedenza, potrebbe mandare un pacchetto solo ogni certo istante, in modo da rispettare quella velocità. Tuttavia, tcp può davvero *scegliere arbitrariamente* l'istante per mandare un pacchetto? Sfortunatamente no (il sistema operativo ha altri processi da gestire, quindi spesso genera *interrupt*, e questi altri processi potrebbero essere altre connessioni tcp attive). Come può allora il "tcp con conoscenza completa" riuscire a spedire un pacchetto ogni tot secondi, almeno approssimativamente? La risposta è che può usare gli ack generati dai suoi stessi pacchetti come una sorta di autosincronizzazione.

<u>Comportamento di autosincronizzazione di tcp</u>

Si può calcolare che se tcp si autosincronizza con gli ack ricevuti, si ritrova a spedire alla velocità minima fra quelle delle linee attraversate. Ad esempio, se la finestra di flusso è (supponiamo costante) di 10 pacchetti, e il pc1 deve spedire un grosso burst, prima spedirà i 10 pacchetti, e poi

attenderà l'ack del primo per spedire l'undicesimo, l'ack del secondo per spedire il dodicesimo e così via, spedendo dieci pacchetti alla volta alla velocità del link con velocità minima.
Supponiamo ora che tcp stesso possa scegliere un valore ottimale per la finestra di controllo di flusso W (che supponiamo sia fisso per semplicità). Si ha che con un valore troppo piccolo si ha una pausa ogni volta che finisce un burst, mentre con un valore troppo alto si rischia di sovraccaricare le code dei router attraversati.
Si calcola allora che il valore migliore di W è
$W = D \cdot \mu$
Dove D è il roundtrip delay time, e μ è il valore di velocità più piccolo fra i link attraversati. W è chiamato anche *prodotto banda-latenza* ed è una delle principali caratteristiche di tcp.
In pratica la finestra in questione non è la finestra a scorrimento di controllo di flusso, bensì un'altra finestra, che questo nostro tcp ideale (con la premessa di poter conoscere tutto della rete) utilizza per *autosincronizzarsi* e ottimizzare il proprio dovere. In realtà, dalle dispense si intuisce che tale finestra è proprio la finestra cwnd di controllo di congestione (la stessa dell'algoritmo slow-start). E per far ciò, il valore di tale finestra dovrebbe essere settato al prodotto banda-latenza della connessione tcp.

Tcp reale in una rete stabile

Dato che nella realtà tcp non sa nulla della rete e delle sue componenti, per avvicinarsi al tcp ideale c'è bisogno di ottenere in qualche modo una stima della finestra per l'autosincronizzazione. L'unica cosa che tcp "sa" è se i pacchetti si perdono oppure no (grazie agli ack e i timeout).
Una possibile soluzione è la seguente: inizializzare la detta finestra W a un valore basso, ad esempio 1 mss (maximum segment size), e incrementarlo finché non ci sono pacchetti persi. L'incremento dovrebbe essere veloce in modo tale da approssimare il miglior valore di W entro poco tempo. Si ottiene così l'esatto comportamento di slow-start senza congestion avoidance. Ignorando la finestra di controllo di flusso comunicata dal ricevente, il mittente incrementerebbe la propria finestra in modo lineare, e si accorgerebbe che diventa troppo grande solo quando comincia a perdere pacchetti. Come ben sappiamo, tcp può fare di meglio, se unisce a slow-start il metodo di congestion avoidance. In questo modo può raggiungere molto più velocemente la finestra corretta, unendo i benefici dell'incremento esponenziale con quelli dell'incremento lineare.

Tcp reale su una rete che cambia nel tempo

In una rete che cambia nel tempo, c'è bisogno di mantenere la finestra aggiornata, tenendo conto che la banda disponibile e lo spazio libero nelle code dei router attraversati variano continuamente. Perciò è necessario monitorare lo stato della rete.

Tipico comportamento di tcp reno (slow-start + sawtooth)

Il comportamento di tcp reno è il seguente:
- all'inizio si applica slow-start per identificare velocemente la banda disponibile
- a regime, si applica fast-rentransmit e fast-recovery: fast-retransmit dice che quando si ricevono 3 ack duplicati di un pacchetto si rimanda immediatamente quel pacchetto, dopodiché bisogna aspettare che arrivino tutti gli ack che non erano arrivati (perché la destinazione aveva cominciato a mandare duplicati). Fast-recovery dice che si manda direttamente l'ack dell'ultimo pacchetto che era arrivato. Comunque sia, se arrivano questi ack, la soglia (treshold) non cambia e la finestra di controllo di congestione riparte da quel valore con congestion avoidance. Se gli ack non arrivano si va in timeout, e quindi si riparte con slow-start, azzerando la finestra cwnd e dimezzando la soglia.

Questo modo di fare è chiamato *aimd* (*additive increase moltiplicative decrease*), perchè la crescita della finestra quando non vengono persi pacchetti è lineare, mentre se ci sono pacchetti persi viene dimezzata la soglia.

Importante: si dimostra che la tecnica aimd porta alla stabilità. Poniamo caso infatti che ci siano due connessioni attive, l'una che utilizza più banda dell'altra, ed entrambe sono nel caratteristico stato *sawtooth* (a dente di tigre, o qualcosa del genere), in cui la banda cresce linearmente e poi ricade sulla soglia ogni volta che viene perso un pacchetto (congestion avoidance). Si dimostra che man mano che vanno avanti in questo modo, la connessione che ha banda più alta tenta a cederne un po' all'altra connessione, tendendo ad uno stato di parità, raggiunto il quale non viene lasciato più. Unica postilla (ma non indifferente) è che tali connessioni abbiano mss ed rtt uguale.

Per finire, vediamo come si comporta la rete con le varie tecniche diverse:
- additional increase additional decrease: c'è stabilità ma disparità, perché le due connessioni tendono a mantenere esattamente i loro throughput iniziali;
- multiplicative increase multiplicative decrease: in sostanza accade la stessa cosa di aiad, solo leggermente più instabile;
- multiplicative increase additional decrease: impraticabile, assolutamente instabile e tende ad assegnare sempre più banda alla connessione che ne ha di più.

Algoritmi di Instradamento per l'infrastruttura di Rete Fissa

Nelle reti di calcolatori assumono un importante ruolo gli algoritmi che sono usati dai router per instradare i pacchetti. Tali algoritmi si distinguono per le loro "qualità".
L'efficienza fa sì che un router non perda tutto il proprio tempo a calcolare le tabelle anziché instradare i pacchetti (che è la cosa più importante).
La robustezza e l'adattabilità rendono il routing resistente ai guasti e adattabile alle modifiche che una rete può subire mentre si trova a regime.
L'ottimalità garantisce che i cammini scelti siano sempre i migliori, in base a un qualche criterio.
La stabilità, fattore molto importante, deve garantire che le tabelle non cambino se non ci sono modifiche sostanziali alla rete, e che se avvengono tali modifiche, i valori delle tabelle tendano rapidamente ad un nuovo instradamento stabile.
L'equità stabilisce che nessun nodo deve essere privilegiato o danneggiato rispetto ad altri.

La scelta di un algoritmo non è facile, dal momento che le qualità, spesso e volentieri, sono contrastanti. Ad esempio massimizzare l'utilizzo delle linee va in contrasto con il ridurre il ritardo dei pacchetti. I criteri di ottimalità devono essere misurabili, e quindi abbastanza semplici: un criterio che tenga conto del traffico diventa enormemente più complicato da monitorare. Inoltre le cpu e le memorie sui router sono spesso insufficienti davanti alla complessità delle reti attuali.

Tutti gli algoritmi di instradamento possono distinguersi in due tipi: statici e dinamici. I primi sono indipendenti dalla topologia, i secondi variano a seconda della topologia o del traffico istantaneo. Tipicamente si fa uso di routing statico quando si ha a che fare con zone periferiche minori, mentre per le grandi zone centrali si fa ricorso al routing dinamico.

Uno degli algoritmi di routing statico, forse il più semplice, è il fixed directory routing: ogni nodo della rete dispone di una tabella che indica con quale linea raggiungere ciascun altro nodo. Tale tabella è compilata dall'amministratore. Una variante "quasi-statica" consiste nel prevedere più alternative, da scegliere in base a qualche criterio condizionale di priorità (caratteristica di *ibm sna*). Questa tecnica è utilizzata spesso nelle reti piccole e non magliate.

La tecnica di flooding è una tecnica di routing statico che prevede che il pacchetto, ogni volta che raggiunge un IS (Intermediate System, o Router), viene rispedito su ogni linea di quell'IS tranne quella da cui è arrivato. In questo modo si assicura che il pacchetto arrivi sicuramente a destinazione, ed è robusto in quanto anche a fronte di un guasto il pacchetto arriverà sicuramente passando da altri percorsi alternativi (dato che li attraversa tutti). La tecnica però carica pesantemente la rete. Per questo esistono varie tecniche di "selective flooding", che permettono di decidere se mandare un pacchetto in flood oppure no. Si può ritrasmettere il pacchetto solo su linee selezionate, o si possono dotare i pacchetti di un'età in modo da scartare i pacchetti più vecchi, o si può fare in modo che i router scartino i pacchetti già ricevuti una prima volta (perché rimandarli in flooding a questo punto sarebbe inutile). Per far ciò occorrono però memorie estese sui router e identificatori di sequenza per i pacchetti.

Per quanto riguarda le tecniche dinamiche, possiamo cominciare a distinguere le tre tipologie di routing isolato, centralizzato e distribuito.

Nel routing isolato ogni IS calcola in modo indipendente le proprie tabelle. Nella versione *hot potato* che ha scopo puramente teorico, ogni router consegna il pacchetto sulla linea con coda più breve. La versione più utilizzata è invece il *backward learnings*: ogni router "capisce" su quale

linea mandare pacchetti destinati ad un determinato nodo “perché” su quella linea gli sono arrivati pacchetti da quel nodo, e quindi dice ‘ah quel nodo sta da questa parte, bene lo aggiorno sulla mia tabella’. Si può migliorare con l’aggiunta di un campo di costo nei pacchetti, da incrementare ad ogni attraversamento di IS, in modo che ogni IS può decidere su quale linea mandare il pacchetto in funzione del costo migliore. Per far fronte ai guasti, le tabelle devono mantenersi aggiornate: ogni entry ha un tempo di vita che scema ogni certo periodo. Se arriva a 0 la entry viene cancellata, se invece arriva di nuovo un pacchetto da quella direzione, la entry viene confermata. Senza questa caratteristica, l’algoritmo imparerebbe solo le migliorie e non i peggioramenti. Quando la destinazione è ignota, si applica il flooding. Per evitare la generazione di cicli infiniti, spesso si associa questo algoritmo al calcolo di uno *spanning tree* (un albero sotto-insieme del grafo della rete, che ne unisca tutti i nodi).

Nel routing centralizzato invece, il compito di creare le tabelle non è dato agli IS singoli ma ad un unico centro di calcolo chiamato *routing control center* (*RCC*), che conosce la topologia della rete grazie ai messaggi di informazioni che periodicamente riceve da ogni nodo (l’RCC è collegato con tutti i nodi della rete), calcola le tabelle migliori per ogni nodo, e le spedisce a ciascuno di essi. Non è un buon metodo a causa del traffico intenso che gira intorno all’RCC, a causa dell’efficienza essendo l’RCC un collo di bottiglia per il routing e a causa dell’affidabilità per i pericoli che si corrono in caso di malfunzionamento sull’RCC.

Il routing distribuito è una fusione fra i due metodi precedenti: ogni router invia le proprie informazioni come nel routing centralizzato, ma non ad un RCC: agli altri router. E quindi ogni router riceve molte informazioni e si calcola le proprie tabelle come nel routing isolato. Quindi c’è equità: ogni router è pari a tutti gli altri, e tutti i router “si aiutano” a vicenda. Questi tipi di algoritmo sono quelli che hanno avuto maggior successo e sono più usati nell’ambito delle reti, soprattutto ad alto livello. Le tecniche più famose sono *link state* e *distance vector*. Prima di esaminarle in dettaglio, anticipiamo solamente che nella prima (link state, stato dei collegamenti) i router danno informazioni sui propri vicini (sul proprio stato dei collegamenti) a tutto il resto della rete, nella seconda i router danno informazioni sulla propria conoscenza della rete (delle tabelle che riassumono com’è la rete “secondo il router”) a tutti i propri vicini.

Algoritmi Distance Vector

In distance vector bisogna distinguere fra un vettore delle distanze e la tabella di routing, concetti molto simili, ma alla fin fine separati. Il vettore delle distanze è un vettore chiave → valore costruito da ciascun router in cui le chiavi sono tutti gli altri nodi della rete e i valori sono i costi stimati per raggiungere tali nodi (quindi è una tabella con il nome del router e due attributi di cui uno chiave). Tali vettori delle distanze vengono trasmessi da ogni router ai propri vicini, che li utlilizzano per aggiornare le proprie tabelle, che sono tabelle col nome del router e due attributi: una chiave che è il nodo da raggiungere, e la linea da utilizzare, scelta in base a tutti i vettori delle distanze ricevuti. Quindi, dal punto di vista di un IS funziona così: l’IS riceve n vettori delle distanze da ognuno dei suoi n vicini, dopodiché si crea una tabella di routing che ad ogni nodo della rete fa corrispondere la linea migliore in funzione dei vettori ricevuti. Ribadiamo che i vettori delle distanze contengono la chiave che è il nodo da raggiungere e il valore che è il costo per raggiungerlo: non serve che venga specificato anche la linea con cui lo si raggiunge, a quella ci penserà quello stesso router quando il pacchetto arriverà a lui, dato che avrà la propria tabella di routing anch’esso.
L’algoritmo è dinamico al massimo: ogni volta che si ricevono nuovi distance vector si ricalcola la propria tabella di routing. Ogni volta che viene calcolata una nuova tabella di routing si rispediscono i distance vector ai propri vicini. Quindi oscilla continuamente. Con reti piccole e

senza troppi cambiamenti significativi, risulta abbastanza stabile, ma in caso di reti con 1000 nodi o più si nota che l'algoritmo converge troppo lentamente.
Le metriche che potrebbero essere adottate sono molteplici (il numero di hop, il ritardo calcolato, il costo, la lunghezza della coda, o addirittura in modo casuale).

L'algoritmo Distance Vector è anche detto Ford-Fulkerson distribuito o Bellman-Ford, poiché basato sull'algoritmo di Bellman.
Senza entrare nei dettagli (pare che bisogna però saperli i dettagli, e quindi c'è da studiarsi bene il lemma delle distanze e le quattro proprietà, e relative dimostrazioni), ecco brevemente descritto l'algoritmo.
- Abbiamo un grafo con n vertici e archi orientati che rappresenta la rete, il vertice 1 rappresenta la destinazione, A(i) è l'insieme dei vertici j per cui c'è un arco orientato uscente (i, j), A(1) è vuoto e a_{ij} è la metrica dell'arco (i, j). La lunghezza del cammino è la somma delle metriche.
- Lavoriamo nell'ipotesi che per ogni nodo i esiste un cammino da i ad 1 (connettività) e che ogni ciclo ha lunghezza positiva
- Il sistema di equazioni di Bellman stabilisce semplicemente che ogni cammino ottimo da un nodo i a 1 è pari alla somma minima tra la metrica dell'arco uscente su ogni proprio vicino e il cammino ottimo da quel vicino a destinazione (semplice distance vector), e ovviamente il cammino minimo dalla destinazione alla destinazione è 0.
- Alla k-esima iterazione (istante k), il cammino ottimo "attuale" è il minimo della somma tra la metrica dell'arco uscente su ogni proprio vicino e il cammino ottimo per quel vicino all'istante precedente k-1 (molto ovvio insomma). I valori all'istante iniziale k=0 possono essere valori interi qualunque, o anche infinito. L'algoritmo termina quando per tutti i nodi si raggiunge la stabilità, ovvero i cammini ottimi restano quelli per tutti gli istanti.
- Per calcolare il valore attualmente ottimo per ogni nodo, si utilizza il *lemma delle distanze*. Questo dice che il valore ottimo all'istante k per ogni nodo è il valore minimo fra le somme tra la lunghezza minima del cammino da quel nodo a ciascun altro nodo j con esattamente k archi e il valore all'istante 0 di quel nodo j. A parole sembra complicato, ma a scrivere la formula risulta meglio comprensibile. La dimostrazione di tale teorema si fa per induzione. Caso base: valore per il nodo i all'istante 1. Dato che k=1 si contano solo i vicini (gli unici per cui sia possibile un cammino di esattamente un arco), quindi risulta proprio la definizione del valore ottimo all'istante k data in precedenza. Per il caso induttivo la dimostrazione è più complessa. Si dimostra che se vale per k vale pure per k+1. Si prende la formula per k+1, si utilizza l'ipotesi che valga per k e si dimostra che l'uguaglianza è corretta. Il valore del cammino minimo da i a 1 con al massimo k archi (chiamato w) si scinde nella somma di tutti gli archi, poi si porta il primo arco fuori dal *min*, quindi restano k archi e si accorpano nel secondo termine dell'uguaglianza che vale per k (con numerazione diversa), si utilizza a questo punto l'ipotesi induttiva e resta un termine che per la definizione del valore ottimo all'istante k-esimo rende vera l'uguaglianza cercata. Ancora, a parole si capisce ben poco, ma le formule sono già scritte sulle dispense (e su dei fogli sui quali ho preso appunti, e forse questo testo può risultare utile se si hanno le formule sotto mano, ma dico "forse"!).
Si dimostrano quindi, grazie al lemma delle distanze, le quattro proprietà fondamentali che mostrano come sia possibile usare tale lemma per calcolare i cammini ottimi da ogni nodo a destinazione.

a) Ogni cammino minimo da i a 1 ha al più n-1 archi (infatti se ne ha di più c'è almeno un ciclo, ma dato che tutti i cicli sono positivi, possiamo rimuoverlo e ottenere un cammino con minor costo);
b) Per ogni insieme di condizioni iniziali, l'algoritmo termina dopo k iterazioni dando la soluzione ottima (infatti, con i cicli positivi, per k abbastanza grande tutti i w diventano infinito e i minimi da calcolare possono solo essere quelli che comprendono i w fino a 1, perché sono definiti in maniera diversa, e corrispondono proprio ai cammini minimi cercati);

c) Se le condizioni iniziali sono tutte sopravvalutate rispetto a quelle ottime, allora l'algoritmo termina al più in m*+1 iterazioni, dove m* è il massimo tra i massimi numeri di archi in un cammino da ciascun nodo a i (infatti se le condizioni iniziali sono tutte impostate a infinito – tranne per il nodo destinazione – i valori ottimi correnti, per il lemma, possono essere solo presi con j=1 e quindi sono gli stessi valori ottimi se k≥m*);
d) I cammini minimi sono l'unica soluzione delle equazioni di Bellman (infatti se iniziamo l'algoritmo con le soluzioni delle equazioni, l'algoritmo termina in un solo passo e dà le soluzioni ottime per la proprietà b. Questo significa che non trova mai soluzioni più ottime di quelle ottime, e che quindi quelle sono proprio le vere soluzioni ottime).

Parliamo ora di un altro problema degli algoritmi basati sul vettore delle distanze: il *conto all'infinito*. Secondo questo noto problema, con un algoritmo basato sul vettore delle distanze, in una rete, le buone notizie si propagano rapidamente, mentre le cattive notizie lo fanno molto lentamente.
Infatti, immaginiamo una rete composta da 4 macchine A, B, C e D in fila, la prima delle quali era guasta (le distanze verso di lei sono all'inizio infinite per tutte le altre). La macchina A poi viene riparata. B si accorge di poter raggiungere A in 1 hop. All'istante successivo B ha mandato il proprio distance vectore a C, che quindi, dato che sa di poter raggiungere B in 1 hop, ora sa anche che può raggiungere A in 2 hop. Similmente, all'istante successivo D saprà di poter raggiungere A in 3 hop.
Ora vediamo il caso contrario: a un certo punto A si rompe di nuovo. B in qualche modo si accorge che A non è più raggiungibile come "vicino". Però nota che C può raggiungere A in 2 hop, quindi decide di poter raggiungere A in 3 hop passando da C (*perchè non sa che il percorso di C passa da lui stesso!!!*). Quindi abbiamo che ora B crede di poter raggiungere A in 3 hop, C in 2 e D in 3. Al periodo successivo, arrivano di nuovo i vettori delle distanze e C si accorge che entrambi i suoi vicini raggiungono A con 3 hop, quindi decide di poterlo raggiungere con 4 hop passando da uno dei due. Abbiamo B con 3, C con 4 e D con 3. Successivamente sia B che D decidono di passare da C e raggiungere A con 5 hop. E così via, come mostrato nella tabella seguente:

3	4	3
5	4	5
5	6	5
7	6	7
7	8	7

Eccetera
Come si può notare, solo molto lentamente le macchine si accorgono che A si trova sempre più lontano, e in teoria loro crederanno sempre di poter raggiungere A in quel numero di hop. In pratica ovviamente avranno già scelto un'altra linea che in qualche modo raggiunge A sicuramente in meno tempo, ma non la sceglieranno finché il valore registrato continuerà ad essere migliore. In pratica non si arriva ad infinito, ma si sceglie un valore abbastanza alto che rappresenti l'infinito, in modo da far smettere alle stupide macchine di contare inutilmente. Il valore che di solito si attribuisce a infinito è quello del cammino più lungo più 1.

Valutiamo l'efficienza di Distance Vector. Possiamo usare come criterio uno dei seguenti:
- contare il numero di passi che impiega l'algoritmo a convergere alla stabilità,
- contare il numero di pacchetti che vengono scambiati sulla rete fino a farla convergere,
- valutare per ciascun router il tempo che impiega a convergere la propria tabella alla stabilità.

Supponiamo che nella rete ci siano N nodi ed M link, e che la metrica sia relativa al numero di hop. Tra i nodi più lontani ci saranno al massimo N-1 link (lunghezza massima del cammino più lungo). Una buona notizia si propaga su tutta la rete in al più N-1 passi (basta tenere in conto l'esempio precedente). Attribuendo alla "variabile" infinito il valore N, allora una cattiva notizia si propaga su

tutta la rete in al più N-1 passi. Quindi N-1 è il numero di passi che impiega l'algoritmo a convergere.
Se vogliamo esaminare invece il tempo che impiegano i router a far convergere la propria tabella alla stabilità, dobbiamo fare le seguenti osservazioni. Un router, ogni passo, scandisce al più M distance vector (più di M linee il router non può avere). Ogni tabella ha al più N righe. Quindi il tempo speso dal router ad ogni passo è O(NM). Dato che la convergenza avviene nel caso peggiore in O(N) passi, ogni router impiega un tempo $O(N^2M)$ per convergere alla stabilità.

Algoritmi Link State Packet

Negli algoritmi basati sullo stato dei collegamenti (Link State), ogni IS ha una mappa completa della rete, costruita in base a dei pacchetti che gli arrivano da ogni altro IS. Tali pacchetti, chiamati link state packets, sono spediti appunto da ogni altro IS e contengono informazioni sui vicini di quegli IS. Questi pacchetti sono spediti in selective flooding attraversando tutta la rete.
In pratica, ogni router deve scoprire i propri vicini e relativi indirizzi. Misurare il costo per raggiungerli e creare un pacchetto lsp che contiene tutte le informazioni raccolte. Inviare tale pacchetto sulla rete. Quindi ciascun router, in base a tutti i pacchetti che riceve, elabora il percorso più breve verso ogni altro router.
Grazie ad alcune tecniche, ogni IS ha un database in cui mantiene il lsp *più recente* per ogni nodo della rete. Quindi nel caso di Link State, la collaborazione fra i router non serve a calcolare direttamente le tabelle di instradamento, ma serve a tenere aggiornata la mappa della rete. Questi algoritmi sono molto efficaci ed efficienti, sono usati in molte reti tcp/ip e ipx, possono gestire reti con anche oltre 10.000 nodi e convergono rapidamente alla stabilità.
Dopo aver ricevuto i pacchetti Link State, ogni nodo calcola la propria tabella di instradamento con un semplicissimo algoritmo di Dijkstra (riassunto in pillole di seguito):
- inizializza l'insieme dei nodi per cui è già stata calcolata la distanza e si inizializzano le distanze dei nodi con i costi degli archi da esso a tutti gli altri;
- si sceglie un nodo in V-S tale che D sia minimo, si aggiunge ad S e si aggiornano le distanze di ogni nodo rimanente minimizzando tra distanze attuali e somma della distanza del nodo corrente più l'arco che li congiunge, iterando questo procedimento.

L'efficienza è presto detta. Supponiamo che ci siano N nodi ed M link. Un'implementazione semplice richiede $O(N^2+M)=O(N^2)$. Se la rete è sparsa, cioè M=O(N), conviene un'implementazione più sofisticata, con complessità O(N log N + M). Tali valori risultano ricavabili dallo studio di complessità dell'algoritmo di Dijkstra, ma in questa sede si risparmiano le dimostrazioni.
Ogni router dunque ha un proprio cosiddetto "shortest path spanning tree", che è uno spanning tree che rappresenta i percorsi secondo i quali si hanno le distanze più corte trovate. Ogni router ha il proprio e in generale sono tutti differenti. Ogni router sceglie la linea sulla quale instradare il pacchetto secondo il proprio spanning tree.

Analizziamo la struttura dei router lsp: un router può ricevere un pacchetto sul proprio cosiddetto "receive process". Se tale pacchetto è destinato ad altri, lo manda sul "forwarding process" e viene risputato sulla rete. Se invece il pacchetto è destinato al router, allora è un pacchetto di gestione (magari un lsp), e viene destinato ai protocolli superiori di gestione dell'algoritmo.
In particolare, il pacchetto può essere di *neighbor greetings* se magari un nodo adiacente non era già noto e vuole farsi conoscere dal resto della rete.
Oppure può essere un lsp e allora viene ritrasmesso solo se giudicato il più recente (ovvero se va a sostituire l'lsp che era precedentemente nel database). Tale controllo si applica grazie a numeri di sequenza sui pacchetti. Ogni volta che un IS genera un nuovo pacchetto lsp vi applica un numero di

sequenza maggiore (come se ad esempio il nodo *pippo* dicesse ‘questo è il mio pacchetto lsp versione 5.0!’ Quando arriva a un router, il router magari scopre che aveva pippo 4.0 e dice ‘ah bene, questo è più aggiornato’ e lo sostituisce).
Se il pacchetto è più vecchio di quello posseduto, lo si rispedisce al mittente, per informarlo che sta emettendo pacchetti lsp obsoleti.

Ultima osservazione: una lan si presta molto male ad essere modellata come un grafo, e per questo Link State si adatta poco alle lan. Per questo la lan spesso si modella come un unico pseudo-nodo in corrispondenza con uno dei router sulla lan (detto “designato”), a cui sono collegati tutti gli altri sistemi. Gli IS in questo modo vedono la lan come un unico nodo con collegamenti punto-punto a tutti gli altri sistemi ad essa interconnessi.

Protocolli di instradamento e la rete Internet

Passiamo ora a descrivere quali sono i principali protocolli di routing utilizzati e di quali algoritmi fanno uso. Prima però, facciamo una breve introduzione.
Ogni protocollo di routing serve a generare sui router intermedi (o IS) una tabella di routing. Le entry di queste tabelle sono costituite da una chiave che indica la rete da raggiungere e quindi il gateway da cui passare per raggiungere tale rete e l'interfaccia sulla quale spedire i pacchetti destinati a tale rete (nonché un campo owner che specifica quale protocollo è applicato sulla rete successiva, o almeno così si intuisce dalla prima pagina della dispensa di GdB).
Le reti sono spesso suddivise in subnet, ma soprattutto, esse sono raggruppate nei cosiddetti Autonomous System, o più semplicemente AS. Gli AS sono grandi raggruppamenti di reti gestiti da un'unica entità amministrativa. Gli AS sono identificati da un numero a 16 bit univoco a livello mondiale. I router che instradano all'interno di un AS si chiamano *interior router*, quelli che instradano (anche) fra AS si chiamano *exterior router*. Quindi i protocolli con i quali gli interior router si scambiano informazioni si chiamano *igp* (*interior gateway protocol*), mentre i protocolli usati dagli exterior router prendono il nome di *egp* (*exterior gateway protocol*). All'interno di un singolo AS si utilizza di norma un unico igp.

Rip

Routing Information Protocol è un igp introdotto nel 1982 associato con tcp/ip e molto diffuso nelle reti fra pc. Fa uso di distance vector con invio del vettore delle distanze ogni 30 secondi. Soffre di varie pecche, come ad esempio il fatto che sia molto lento a convergere e che subisce il problema del conto all'infinito. Infatti viene usato solamente per reti piccole. L'unica metrica considerata è il numero di hop, e il valore massimo di hop permessi è tipicamente 15.
I pacchetti rip sono spediti in broadcast sulle porte sulle quali il router è connesso, e contengono il comando, la versione di rip usata (attualmente la più diffusa è la 2) e la descrizione delle subnet in base a indirizzo ip e subnet mask. Gli update sono inviati sì ogni 30 secondi ma con un piccolo margine di errore voluto, per evitare che la rete si sincronizzi normalmente e che ogni 30 secondi i router calcolino solo la tabella di instradamento.
Un prefisso su cui non arrivano informazioni aggiornate entro un certo tempo viene considerato non più raggiungibile.

Igrp

Interior gateway routing protocol è un igp della cisco del 1985, disponibile solo so router cisco, che fa uso di distance vector con una sofisticata metrica che combina ritardo, banda, affidabilità della linea, lunghezza massima del pacchetto e carico. Inoltre permette il multipath routing (suddivisione del carico su più linee), che spesso dà prestazioni migliori che non inviare sempre tutto sulla linea migliore, ignorando magari la seconda migliore.

Ospf

Open Shortest Path First è un igp per tcp/ip molto recente, basato sullo stato dei collegamenti, che ha sostituito in gran parte rip perché molto stabile e robusto, capace di gestire una moltitudine di router. La O di Open Literature indica che il protocollo è non proprietario. La rete viene rappresentata con un grafo, le lan vengono considerate servite da un unico nodo scelto come router designato. Secondo la definizione, il protocollo instrada anche in funzione del tipo di servizio (type of service) (sul Tanenbaum però dice che non è più usata questa caratteristica). Il routing ospf è

organizzato in maniera gerarchica: ogni AS è diviso in aree che contengono reti contigue. La topologia interna di un area è invisibile alle altre aree. Un'area dell'as è detta dorsale (backbone), e la sua caratteristica è che comunica con ogni altra area, è l'unica area con cui comunicano le aree isolate, ed è l'unica area che comunica via egp con gli altri as (ovvero con le dorsali degli altri as).
Ospf prevede quattro tipi di router, non mutuamente esclusivi, ovvero ogni router può essere di uno o più di questi tipi.
- interno, se tutte le reti connesse appartengono alla stessa area;
- di confine d'area, se collega più aree attraverso la dorsale;
- di dorsale, se si trova sulla dorsale;
- as boundary o di confine di as, se collega l'as con un altro as.

Egp

Exterior Gateway Protocol è stato il primo egp ad essere diffusamente usato, nel 1984 (da non confondere egp come protocollo, con egp come "tipo" di protocollo). Si applicava solo su topologie ad albero e fa uso di un algoritmo simile a distance vector ma solo con indicazione di raggiungibilità.

Bgp

Bgp è il protocollo egp più diffuso attualmente nella rete Internet. Border Gateway Protocol, pensato proprio per sostituire il vecchio egp. Esso fa uso di distance vector con una variante detta *path vector*, ovvero con informazioni complete sui cammini. In questo modo si evita il conto all'infinito, perché ogni router sa se il cammino che gli sta comunicando un vicino passa o no per sé stesso.
Bgp è nato soprattutto per tenere conto delle politiche, perché gestendo il traffico fra as, la politica diventa un fattore fondamentale. Certi as, essendo gestiti da talune società, potrebbero non volere che il traffico di qualche as concorrente passi attraverso di loro, potrebbero voler proibire traffico da e/o verso altri as, oppure potrebbero voler vendere il diritto di transitare attraverso di essi. I vincoli politici sono gestiti esplicitamente dagli amministratori. Le comunicazioni fra i router bgp avvengono attraverso lo strato di trasporto (porta 179) per ragioni di affidabilità.
Due router bgp si scambiano informazioni durante quella che viene chiamata "sessione" (*session*). Tali router vengono detti *peerers*, o anche si dice che l'uno *fa peering* con l'altro.
Per comunicare fra gli AS si fa uso del meccanismo di "ridistribuzione". La ridistribuzione BGP → IGP permette ai router di un AS di conoscere le rotte ip di un altro AS perché sono state apprese da un router di confine tramite bgp. Se però tali rotte sono davvero tante, si sconsiglia tale "ridistribuzione" e si invia solamente la "rotta di default", ovvero 0.0.0.0/0. È possibile distribuire verso i router network rotte apprese da processi di routing interno, da configurazioni di routing statico e da processi di routing esterni.
Bgp è formato in oltre da un input policy engine e da un output policy engine, che in base a dei criteri specificati, passano da un processo di decisione che decide se accettare certe rotte oppure no, e se farle uscire verso quali as e quali no.
Gli as secondo bgp si suddividono in single-homed (un solo punto di accesso a Internet), multi-homed (più punti di accesso a internet) e transit-network (grandi as che si pongono da tramite fra gli as piccoli).

Qualche ultima parola la si spende per descrivere brevemente uno dei progetti più importanti degli ultimi anni per la nostra nazione: la rete garr. Tale rete è nata alla fine degli anni '80 ed è la rete scientifica nazionale creata dall'Istituto Nazionale di Fisica Nucleare (INFN) e fornisce connettività ad oltre 300 Università ed Enti di Ricerca italiani.

Indirizzamento privato e NAT

Il *nat* è una delle più diffuse soluzioni ad uno dei maggiori problemi di internet: la carenza di indirizzi ip. Se ad un ISP è assegnato un certo range di indirizzi ip pubblici da distribuire ai propri clienti, e i clienti degli isp sono tutte aziende che richiedono un ip per ogni propria macchina connessa ad internet, o anche famiglie che richiedono un adsl casalinga ma vogliono connettere il pc di ogni componente della famiglia, gli indirizzi ip finiscono rapidamente.
La soluzione trovata è abbastanza semplice, ma bisogna partire dal basso. Innanzitutto, le reti di molte aziende o società, spesso non hanno neanche bisogno di essere connesse ad internet, ma basta che siano connesse tra loro le macchine di questa rete, che vogliono sfruttare il protocollo ip pur senza dover essere immesse nella rete mondiale. Quindi sono state appositamente create delle allocazioni di spazi di indirizzamento "privati". Ecco come: sono stati presi tre range speciali di indirizzi ip, ed è stato decretato che tali indirizzi ip *non possono* viaggiare sulla rete internet, perché sono riservati a queste reti. Questi range sono:
10.0.0.0/8 (da 10.0.0.0 a 10.255.255.255)
172.16.0.0/12 (da 172.16.0.0 a 172.31.255.255)
192.168.0.0/16 (da 192.168.0.0 a 192.168.255.255)
Con il primo tipo è possibile creare reti di 16 milioni di pc, con la seconda di un milione, con la terza di 65.536 pc. In pratica questi range di indirizzi ip sono *replicati* su ogni rete che ne ha bisogno. Questo risolve anche il problema della carenza di indirizzi ip: infatti, tutte le reti che hanno bisogno di essere interconnesse tra di loro, fanno uso, per i loro host, sempre di questi stessi range di indirizzi, detti privati, e quindi non si mangiano altri indirizzi pubblici. Il problema arriva quando queste reti si connettono ad internet: ci sarebbero tante macchine con lo stesso indirizzo 10.0.0.1 (cosa che effettivamente accade), tante macchine con lo stesso indirizzo 192.168.1.5, eccetera. È qui che viene in aiuto nat. In realtà, gli indirizzi privati, sono riconosciuti solamente all'interno della rete privata, ma quando un pacchetto esce fuori dal cosiddetto "realm" privato, un dispositivo nat cambia, traducendolo, l'indirizzo ip sorgente privato con quello pubblico reale.
Un esempio per capire meglio: un ISP assegna ad un'azienda un certo range di indirizzi, ad esempio 193.10.2.0/24, quindi questa azienda può connettere 254 host ad internet. C'è il router aziendale connesso all'isp, e al router aziendale è connessa una lan con tutti i pc. Se i pc fossero meno di 255, nat non servirebbe e l'azienda potrebbe richiedere che venga assegnato un singolo ip pubblico a ciascuna delle proprie macchine. Invece mettiamo che i pc di questa azienda siano migliaia (è una grossa azienda). Allora la soluzione è questa: all'interno dell'azienda, viene creata una rete privata del tipo 192.168.0.0/16, e vengono dati indirizzi ip in base a questa rete, e viene connesso al router principale dell'azienda un dispositivo nat (*network address translation*) (spesso i dispositivi nat sono all'interno del router stesso). Quando i pc comunicano all'interno stesso dell'azienda, lo fanno normalmente: l'azienda è una piccola internet a sé stante senza problemi, con i suoi indirizzi privati che all'interno di essa sono univoci e quindi è come se fossero pubblici, quindi l'azienda fa uso di internet protocol normalmente, può avere i suoi router interni normali, eccetera. Quando invece un pc dell'azienda, ad esempio 192.168.1.1, deve spedire un pacchetto fuori la rete aziendale, questo passa per il router con il nat. Il router nat allora fa una traduzione di indirizzi: questo ha una propria tabella di traduzione nat, indicizzata dagli indirizzi ip assegnati all'azienda (193.10.2.1, 193.10.2.2, 193.10.2.3, e così via). Cosa fa? Prende il primo indirizzo ip "libero", cioè una chiave della tabella a cui non è associato ancora nessun valore, ad esempio 193.10.2.10, gli associa come valore l'indirizzo della macchina che ha spedito il pacchetto (quindi 193.10.2.10 → 192.168.1.1), e cambia, all'interno di quel pacchetto, il campo source address con la chiave utilizzata. In questo modo, nella rete non viaggerà un pacchetto con un source address privato (in quanto vietato: in internet gli indirizzi ip devono essere univoci), ma viaggerà un pacchetto con un indirizzo pubblico permesso. Il pc che lo riceverà, saprà che dovrà rispondere *a quell*'indirizzo. La risposta quindi avrà

un campo destination address del tipo 193.10.2.10 che è sempre l'indirizzo pubblico permesso all'azienda, che il router ha usato per tradurre l'indirizzo privato. Quando il pacchetto di risposta arriverà al router nat dell'azienda (perché ci arriva, dato che gli algoritmi di routing sanno che la rete 193.10.2.0/24 si raggiunge da quel router), il router prenderà l'indirizzo ip di destinazione all'interno del pacchetto, cercherà sulla propria tabella a quale indirizzo ip privato corrisponde, farà la traduzione inversa nel campo del pacchetto e instraderà il pacchetto nella rete interna privata (magari se non serve più cancellerà anche la entry nella tabella: in fondo la quantità di pc dell'azienda che possono connettersi attivamente ad internet è limitata proprio dal range assegnato all'azienda). Così si ritroverà di nuovo un pacchetto all'interno della lan privata dell'azienda con destinazione 192.168.1.1 che troverà facilmente la macchina a cui è destinato.
C'è una variante molto importante di nat, chiamata *pat* (*port address translation*) o *napt* (*network address port translation*). Poniamo caso che l'azienda non sia un'azienda ma un capofamiglia che chiede ad un ISP una connessione adsl ad internet a casa sua, e che vuole connettere il computer proprio e di ciascuno dei propri tre figli a internet. L'isp dice "mi dispiace ma io ti do soltanto un indirizzo ip pubblico, tu comprati un router nat e arrangiati". Quindi abbiamo il router principale, e questi vari pc collegati, che fanno uso di una rete privata casalinga (ad esempio del tipo 10.0.0.0/24). Dato che il router principale ha disponibile *esclusivamente un* indirizzo ip pubblico, secondo la precedente tecnica di nat semplice è possibile connettere soltanto un pc alla volta ad internet (sebbene all'interno della lan stessa i pc possono fare uso del protocollo ip come vogliono, ma non è certo questo che voleva il capofamiglia). Il nat con traduzione di porta utilizza le porte tcp e udp, approfittando del fatto che sono i protocolli di trasporto più usati all'interno di internet. La tabella di nat sul router, stavolta, non è indicizzata dagli ip pubblici assegnati al router, ma da semplici indici (ad esempio da 1 a 65536), e tali indici stavolta non andranno a riempire il campo address del pacchetto ip, bensì il campo port del pacchetto tcp. Esempio: il pc 10.0.0.1 invia un pacchetto fuori dalla lan casalinga. Stavolta dobbiamo tener conto anche di tcp. Diciamo che questo pc sta cercando di connettersi ad un server su una certa porta del server, con la propria porta 4747. Quindi la source del pc prima di arrivare al router nat è 10.0.0.1:4747. Il pacchetto arriva al router nat. Per quanto riguarda la traduzione di indirizzo ip non c'è scelta: l'indirizzo assegnato è uno solo, quindi ci mette quello (ad esempio 198.113.65.42). Però prende una entry libera della tabella, mettiamo che sia la 1, e fa la traduzione sul campo port del pacchetto tcp: il pacchetto in uscita avrà all'interno la pdu tcp con campo port 1, mentre sulla tabella viene memorizzato che il valore 1 corrisponde all'indirizzo 10.0.0.1 con porta 4747 (attenzione: le chiavi della tabella quindi sono dei numeri che vengono usati come numeri di porta, ma i valori sono indirizzo ip + numero di porta vera). Quando arriva la risposta dal server, arriva cercando 198.113.65.42:1. Trova il router casalingo che cerca 1 sulla propria tabella, e capisce a chi è destinato quel pacchetto, quindi instrada nella lan casalinga un pacchetto con destinazione 10.0.0.1:4747. In questo modo, se anche il pc 10.0.0.2 vuole stabilire una connessione con qualcuno usando la porta 4747, nella tabella nat verrà memorizzata una entry con valore 2 (perché l'1 è già occupato) e quindi entrambi i pc possono usare la stessa porta senza rischio di fraintendimento, perché quando arriva un pacchetto destinato alla porta 2, il router nat sa che questo è per la porta 4747 del 10.0.0.2.
Ricordiamo che sia nel caso di nat che di pat, ad ogni traduzione vengono ricalcolati l'header checksum dei pacchetti ip e tcp. Osserviamo che in caso di nat semplice, potrebbe non essere necessario aprire il pacchetto ip perché non c'è bisogno di andare a toccare il pacchetto tcp.

Nat e pat si portano appresso tantissimi difetti, e hanno attirato l'odio di molte persone un po' per questo e un po' perché rifugiandosi in una soluzione temporanea ad un problema grave (l'esaurimento graduale degli indirizzi ip) si riduce il desiderio di implementare la soluzione reale, cioè IPv6. I problemi di nat sono i seguenti:
- viola il principio end-to-end sancito da rfc 1958, che stabilisce che internet è una rete orientata alle connessioni, e che quindi nessun dispositivo intermedio dovrebbe mantenere traccia della connessione tra un punto ed un altro;

- viola il principio secondo il quale ogni macchina deve essere identificata da un univoco indirizzo ip;
- nascono molti problemi nelle reti peer-to-peer: infatti una macchina in una rete privata non può comunicare con un'altra macchina in un'altra rete privata neanche se fanno uso entrambi di nat (a meno che non siano esperti programmatori che riescano ad accedere alla tabella di nat del proprio router);
- è compromesso il funzionamento dei protocolli di livello applicativo che menzionano gli indirizzi ip in input (come ad esempio ftp);
- ne risente anche la frammentazione dei pacchetti ip durante la traduzione;
- nasce confusione anche sui DNS, che devono far corrispondere i nomi interni e i nomi esterni delle macchine interne rispettivamente agli indirizzi del private realm e del public realm;
- infine le prestazioni dei router si deteriorano notevolmente quando si fa uso di nat.

CIDR

CIDR sta per *Classless Inter-Domain Routing*, ovvero Routing in domini senza classi. All'inizio, gli indirizzi di internet erano suddivisi in tre classi di reti possibili, A, B e C. Le prime con 16 milioni di host, le seconde con 16000 host, le terza da 256 host. In sostanza molte società chiedevano una rete di classe B perché prevedevano di collegare ad internet qualche centinaio dei propri computer. In realtà si scoprì che 1) molte di queste società non collegavano poi più di 50 computer e 2) anche se avessero oltrepassato la soglia di 256 (e quindi non potessero comunque chiedere una rete di classe C), magari non ne connettevano più di 300-400. In questo modo migliaia e migliaia di indirizzi ip si stavano sprecando, tornando così al problema dell'esaurimento degli indirizzi.
Cidr è un'altra soluzione a questo problema. Come dice il nome stesso, cidr ha tolto di mezzo le classi. L'idea di fondo è assengnare gli indirizzi ip rimanenti in blocchi di dimensioni variabili senza tener conto delle classi. Ad esempio, se un'azienda chiede 2000 indirizzi ip, gli viene assegnato un blocco di 2048 indirizzi (allineati), con una subnet mask adeguata. In questo caso, essendo $2048 = 2^{11}$, serviranno 11 bit per identificare il pc all'interno della rete, e quindi la rete sarà una /32-11 = /21, con subnet mask seguente:
11111111 11111111 11111000 00000000.
La maschera di sottorete serve ai router. Quando arriva un pacchetto su un router, viene estrapolato l'indirizzo ip di destinazione e ne viene fatto un AND con ogni maschera nella tabella di routing (indicizzata dalle reti, ciascuna affiancata da una subnet mask e dall'interfaccia sulla quale far uscire i pacchetti destinati a quella rete). Se il risultato di IP destination address AND subnetmask = la rete di quella subnet mask, allora il pacchetto è destinato a quella rete. Se risulta destinato a più reti, si prende quella più specifica (con il numero di barra più alto). In pratica la maschera di sottorete indica la lunghezza del prefisso comune a tutti gli ip di quella sottorete. Nell'esempio precedente, la rete N aveva un certo prefisso comune a tutti gli indirizzi, e tale prefisso era lungo 21 bit. Quando arriva un pacchetto, arrivato a quella entry il router quindi prende i primi 21 bit e vede se sono uguali a quelli della rete (se affianco alla chiave della tabella, ovvero la subnet, ci fosse /21 anziché 255.255.248.0 magari risulterebbe più facile comprendere).
Quindi non c'è più bisogno, come si faceva prima, di capire prima a quale classe appartiene l'indirizzo del pacchetto in entrata e poi cercare la rete sulla tabella di quella classe, all'interno delle tabelle del router. Questa tecnica non si usa più, grazie a CIDR, ma si usa la tecnica dell'AND con le subnet mask.
Un'altra caratteristica di cidr è l'*accorpamento* (o *aggregazione*) di indirizzi ip su una tabella di routing. Se il router si accorge che più subnet specifiche e *contigue* (ovvero formate da indirizzi ip successivi gli uni agli altri) collidono sulla stessa interfaccia, devono accorpare automaticamente le varie entries in un'unica entry di una subnet più generale, cioè col numero della barra della netmask minore. Esempio, in una tabella di routing ci sono le entries:
194.254.64.0 255.255.255.0 eth3
194.254.65.0 255.255.255.0 eth3
194.254.66.0 255.255.255.0 eth3
194.254.67.0 255.255.255.0 eth3.
Queste possono essere accorpate nella unica entry:
194.254.64.0 255.255.252.0 eth3.
Infatti, se le vediamo in binario, diventano:
11000010 11111110 01000000 00000000 /24
11000010 11111110 01000001 00000000 /24
11000010 11111110 01000010 00000000 /24
11000010 11111110 01000011 00000000 /24

I primi 22 bit di tutte e 4 le reti sono sempre uguali, i 2 bit successivi determinano una delle quattro possibilità, quindi identificare prima una delle quattro reti con i 2 bit dopo i primi 22, e poi il pc con gli 8 bit successivi equivale a identificare direttamente il pc con 10 bit dopo i primi 22, quindi considerare le quattro reti come un'unica rete è la stessa cosa, ammesso che rispetto al router esse si trovino tutte sul dominio di collisione della stessa interfaccia.
Si ricorda che il subnetting (distinzione dei bit per la rete da quelli per l'host grazie al subnet mask) esisteva già prima di cidr, ed è usato all'interno delle società che già hanno richiesto un alto range di indirizzi ip per poterli gestire a propria volontà finché gliene avanzano. Cidr fa semplicemente uso di molti dei concetti del subnetting.
Uno dei benefici di cidr è che, grazie all'aggregazione sulle tabelle, si risparmia spazio in tali tabelle che altrimenti esploderebbero, tante sono le net e le subnet da tenere in considerazione, soprattutto sui router delle backbone (dorsali), che non hanno una rotta di default (infatti sono anche dette zone "default-free") e che devono mantenere una entry per ogni rete o sottorete. Nat, cidr e subnetting sono solo soluzioni a breve e medio termine per far fronte al problema dell'esplosione delle tabelle di routing e all'esaurimento degli indirizzi ip. La soluzione a lungo termine è il protocollo IPv6, che aumenta di gran lunga lo spazio di indirizzamento, lo rende più veloce grazie ad un header semplificato e razionalizza la distribuzione degli indirizzi.
Nell'aggregazione bisogna fare attenzione all'allineamento. Si possono aggregare solamente rotte che sono contigue, che sono destinate alla stessa interfaccia e che abbiano in comune tutti i primi n bit sui quali si vuole aggregarle (per questo conviene sempre controllare in binario la lista delle reti da aggregare).
Un altro principio fondamentale di cidr è l'*allocazione*. Secondo tale principio, le reti devono essere allocate in modo tale da rendere più probabile l'aggregazione. Spesso tale allocazione è geografica. Infatti viene fatta una suddivisione in macroregioni. Ad esempio, l'Europa è caratterizzata da reti che cominciano per 194.x.y.z e 195.x.y.z, il Nord America dalle 198.x.y.z e 199.x.y.z, il Centro e Sud America dalle 200.x.y.z e 201.x.y.z e l'Asia e il Pacifico dalle 202.x.y.z e 203.x.y.z. In questo modo è probabile che tutte le 65536 reti di classe C europee figurino nelle tabelle di instradamento dei router americani con una singola riga (194.0.0.0 /8) (se per l'Europa abbiamo 194 e 195 probabilmente va bene anche /7, comunque sulle dispense dice /8).

Spanning Tree

Il bridge è un dispositivo simile ad un repeater che opera allo strato data link e che instrada i frame ricevuti mediante tabelle indicizzate sugli indirizzi MAC o tramite flooding su tutte le proprie porte tranne quella di ricezione del frame. In una LAN, per far fronte a problemi di robustezza, si possono creare percorsi multipli ridondanti, ma in questo modo si rischia che i frame finiscano in cicli infiniti (problema classico del flooding). Per ovviare a tale problema si ricorre agli algoritmi per il calcolo degli spanning tree.
Questi algoritmi individuano una topologia ad albero (quindi con assenza di cicli) in una rete con un bridge.
L'algoritmo utilizza lo scambio di bpdu (bridge protocol data units), pacchetti dello strato data link.
Le caratteristiche dell'algoritmo sono la creazione dell'albero in poco tempo e la limitazione del disservizio e del consumo di banda per le bpdu. L'algoritmo è *distribuito* su tutti i bridge della rete.

L'amministratore assegna una priorità a ciascun bridge e a ciascuna porta.
Ogni bridge ha un identificatore [priorità o indirizzo mac].
La configurazione dell'albero avviene ponendo le porte in più in blocking state.

L'algoritmo di spanning tree funziona in questo modo:
- si elegge il root bridge, eleggendo quello con l'id minore;
- si elegge la root port per ogni bridge, ovvero la porta più conveniente per connettersi al root bridge;
- si seleziona il bridge *designato* per ogni LAN, ovvero il bridge destinato a interconnettere la lan con il root bridge. La porta attraverso la quale avviene l'interconnessione viene chiamata porta *designata*. Il root bridge è l'unico ad avere le porte tutte designate e nessuna root pot;
- le porte non root e non designated vengono messe in blocking.

Spesso le porte si scelgono secondo dei parametri. Per ogni porta è definito un path cost, che indica il costo di attraversamento della porta, definito dall'amministratore. Dopodiché, il root path cost per una porta è il costo totale del percorso per raggiungere il root bridge, e viene usato per scegliere le root port e le designated port (tra le non-root port).
Le bpdu sono pacchetti che servono per realizzare il protocollo. Il root bridge genera automaticamente bpdu in multicast a tutti i bridge della lan. Quando viene ricevuta una bpdu su un bridge, vengono aggiornati i parametri e la bpdu viene ritrasmessa sulle designated port.
L'elezione del root bridge avviene all'inizio, quando tutti i bridge continuano a comunicare di voler essere root bridge, comunicando il proprio identificatore. Quando un bridge riceve un id più basso, smette di considerarsi root bridge, finché non ne rimane solo uno.
Le designated port cambiano proprio in base ai parametri che vengono osservati quando arrivano le bpdu, quindi ogni bridge si accorge immediatamente di un cambiamento nella topologia, e lo spanning tree viene immediatamente aggiornato, perché il root che ha notato il cambiamento, lo comunica al root bridge con un altro bpdu apposito.

Particolari del protocollo a parte, l'algoritmo è molto semplice, converge rapidamente e tiene costantemente sott'occhio i cambiamenti della topologia della rete. (Per una descrizione più dettagliata vedere la dispensa 0160-esercitazione-sta-02.pdf che spiega alla perfezione l'algoritmo di Spanning Tree secondo GdB.)

Vlan

Vlan sta per Virtual LAN. Per cominciare a capire a cosa serve una rete virtuale, prendiamo un caso d'esempio. Poniamo che un'azienda abbia a disposizione un solo switch a cui collega tutti i propri pc, e che abbia due diversi dipartimenti, e vuole dividere la rete in due lan separate per motivi di sicurezza o anche solo per dividere il traffico broadcast. Una possibile soluzione sarebbe comprare un nuovo switch e usare i due switch per le due lan separate. Tale soluzione è poco flessibile, poco adattabile ai cambiamenti e poco economica. Si potrebbe risolvere invece creando due reti virtuali con l'unico bridge a disposizione.

L'idea è la definizione di una topologia logica indipendente da quella fisica, ovvero realizzare diverse lan virtuali sullo stesso switch, tenendo separato il traffico di ciascuna vlan da quello delle altre. Le vlan sono configurate dall'amministratore. Possono essere configurate semplicemente in funzione della porta (certe porte appartengono ad una vlan, altre ad un'altra, eccetera), oppure anche del contenuto dei pacchetti.
Una vlan denota in particolare un certo insieme di pacchetti che transita per lo switch (ad esempio "tutti i pacchetti che entrano dalla porta 1, dalla porta 3, dalla porta 4 e dalla porta 7") e un pacchetto può appartenere ad una sola vlan (le vlan sono mutuamente esclusive).
Le regole che descrivono una vlan sono specificate in un linguaggio che dipende dal costruttore dello switch e attribuiscono ciascun pacchetto ad una vlan, cosicché l'insieme dei pacchetti che entrano nello switch viene partizionato fra le varie vlan. Se per uno switch l'amministratore non specifica nessuna regola di partizione tutti i pacchetti vengono attribuiti ad una stessa vlan di default.
Una vlan si configura settando per essa un insieme di *ingress port*, ovvero specificando che bisogna considerare solo i pacchetti che entrano da determinate porte. Di questi pacchetti, si specificano regole secondo le quali solo certi pacchetti sono considerati (ad esempio solo quelli destinati a certi indirizzi MAC, o quelli che hanno a bordo il protocollo ip, eccetera), e una volta fatta la cernita, possono essere mandati fuori dallo switch solo da certe porte, dette *egress port*.
Un pacchetto è *associato* ad una vlan nell'istante in cui entra in uno switch. Ad ogni vlan è associato un intero tra 1 e 4094 detto *vlan id* (la lan di default ha sempre lan id 1), ma è possibile anche dare un nome di 32 caratteri, che spesso è più facile da ricordare.
Quando uno switch riceve un pacchetto, associa tale pacchetto alla sua vlan, e individua la porta attraverso la quale trasmettere il pacchetto, attraverso il *filtering database*.
Gli switch possono operare in due modalità alternative, dette IVL (Independent Vlan Learning) e SVL (Shared Vlan Learning). Nella prima, il filtering database è separato per ogni vlan, nella seconda c'è un filtering database condiviso fra tutte le vlan (alcuni switch possono operare solo in SVL). Un filtering database SVL è una tabella condivisa che associa ad ogni host della lan la porta con la quale raggiungerlo e la vlan associata. Un filtering database IVL associa ad ogni host della vlan la porta con la quale raggiungerlo.
Quando arriva un pacchetto, il filtering database controlla se c'è una porta dalla quale far uscire il pacchetto. Se non ne individua nessuna, trasmette in broadcast su tutte le egress port della vlan alla quale è associato il pacchetto. Se invece la ricerca ha successo, si controlla che la porta di uscita sia una egress port della vlan del pacchetto. Se lo è viene spedito su quella porta, altrimenti viene scartato.
L'insieme delle egress port di una vlan può essere diverso dall'insieme delle ingress port. In questo caso le vlan si dicono essere *asimmetriche*. Ad esempio, un'azienda può volere che la propria vlan principale che comprende i router aziendali possa comunicare con tutte le altre vlan, mentre le vlan di dipartimento possano comunicare soltanto con la vlan principale e non fra di loro.

Quando si hanno vlan asimmetriche, conviene lavorare in modalità SVL. Infatti, quando una vlan non trova nel proprio filtering database IVL il mac di un pc che invia solo sull'altra vlan, manda il pacchetto in broadcast su ogni sua egress port, inviando quindi non solo al destinatario ma anche ad altri, degradando le prestazioni della rete. Con SVL i pacchetti vengono inviati invece solamente al destinatario.

Vlan in reti con più switch e 802.1Q

È fondamentale poter definire le vlan anche nel caso più complesso (e comune) di più di uno switch per realizzare un insieme di lan logiche "sopra" un'infrastruttura lan fisica. Immaginiamo il caso semplice di due switch, ciascuno connesso a entrambi i dipartimenti "vendite" e "progettazione" di un'azienda. Si potrebbe voler interconnettere vendite del primo switch con vendite del secondo e la stessa cosa per progettazione. Ma in questo modo si creerebbe un ciclo, che l'algoritmo spanning tree 802.1D risolverebbe bloccando una delle porte.
A questo punto viene in aiuto lo standard 802.1Q emesso dallo standard IEEE. Tale standard prevede che un link tra due switch possa essere dichiarato *trunk 1q*. Nei link di tipo trunk 1q possono transitare pacchetti di varie vlan che, per essere distinti, sono etichettati dallo switch trasmittente con l'id della vlan cui appartengono. Il tag è un campo addizionale della pdu di livello data-link, che richiede 4 byte in più al formato pdu originale (81-00 + 2 byte di tag). Quando uno switch riceve un pacchetto con un tag da un link trunk 1q lo attribuisce alla vlan cui appartiene e gli toglie il tag. Le porte trunk 1q partecipano a tutte le vlan come egress per default.
L'amministratore può specificare se i pacchetti in uscita da una egress port di una vlan devono essere taggati oppure no. Quando una porta riceve un pacchetto, se è taggato viene attribuito a quella vlan, altrimenti viene attribuito alla vlan secondo le regole definite per quella porta. È anche possibile configurare una porta in modo tale che scarti i pacchetti a seconda che siano taggati oppure no. Detto ciò, una porta si definisce essere *access* se invia e riceve solo pacchetti non taggati, *trunk* se riceve e invia solo pacchetti taggati o *ibrida* se fa entrambe le cose.

Spanning Tree Multipli e 802.1S

Le vlan hanno un grande successo su tutte le reti di piccole, medie e grandi dimensioni. Sui trunk 1q possono decidere di far transitare solo poche vlan anziché tutte quante (per motivi che possono essere di sicurezza o quant'altro).
Anche nelle reti con vlan è necessario attivare il calcolo degli spanning tree per individuare una topologia priva di cicli. Se un trunk 1q fa passare tutte le vlan, non rischia di disconnettere delle vlan, ma se si sceglie di evitare che qualche vlan passi dal trunk 1q, è possibile che quelle vlan vengano disconnesse. Anche in caso di trunk 1q completi, per motivi di ottimizzazione e bilanciamento lo spanning tree tradizionale potrebbe non essere una scelta ottimale; inoltre, potrebbe essere utile bloccare alcuni collegamenti solo per certe vlan.
Lo standard IEEE 802.1S prevede la presenza contemporanea di più istanze di spanning tree sulla stessa lan. L'associazione di ciascuna istanza ad una vlan può risolvere i problemi elencati. Una porta in blocking per una vlan non necessariamente lo è anche per l'altra, così c'è meno spreco di risorse, inoltre, calcolando uno spanning tree per ogni vlan, non c'è rischio che una di esse sia disconnessa.
L'architettura di 802.1s prevede il raggruppamento degli switch in aree chiamate *regioni*. Le regioni SST (Single Spanning Tree) comprendono tutti gli switch che non conoscono 802.1s e che quindi non utilizzano Spanning Tree Multipli (in una rete ci può essere una sola regione SST). Le regioni MST comprendono solo switch che conoscono e utilizzano 802.1s. All'interno di ciascuna delle regioni del secondo tipo, gli switch possono utilizzare più istanze dello spanning tree (MSTI, Multiple Spanning Tree Instance). Ogni istanza è identificata da un numero fra 1 e 4094 (MSTID),

e per ogni istanza interna a una regione viene eletto un MSTI Regional Root Bridge, analogo al root bridge di 802.1d.
Per ogni istanza di spanning tree è definita una diversa priorità dei bridge e un diverso costo di attraversamento e priorità per ogni porta. Le bpdu di ogni istanza viaggiano in modo completamente trasparente per ogni altra istanza. I vantaggi, ricordiamo, sono un ottimizzazione delle risorse e un migliore bilanciamento del carico (nell'ipotesi di lavorare con più vlan).
Dal punto di vista del livello 3, i router possono avere porte configurabili come trunk 1q (e quindi instradare i pacchetti su una singola interfaccia), oppure instradare su due interfacce separate per le due vlan.
Tipiche sono le configurazioni MSTP a stella e ad anello.
La pecca di MSTP è che le bdpu viaggiano nella rete senza nessun riferimento ai tag trasportati sui trunk link. In sostanza, 802.1s ha bisogno di sapere a quale vlan appartiene ogni link, perché non tiene conto dei trunk.
L'interazione tra le regioni SST e MST viene realizzata considerando ciascuna regione MST come un singolo switch chiamato CIST Regional Root e utilizzando un'istanza globale di spanning tree detta appunto CIST (*Common Instance Spanning Tree*). Quindi ogni regione MST ha un'istanza MSTI associata ad una o più vlan, con MSTI Regional Root Bridge per ogni istanza, e un'istanza comune di spanning tree con un CIST Regional Root Bridge.