Sistemi Distribuiti

Architetture di elaboratori:

I sistemi di elaboratori possono dividersi in categorie:

1. SISD *simple instruction – single data* ← processore tradizionale (sistema centralizzato)
2. SIMD *simple instruction – multiple data* ← sistema vettoriale
3. MISD *multiple instruction – single data* --
4. MIMD *multiple instruction – multiple data* ← sistema multiprocessore (parallelo o distribuito)

sistemi vettoriali Sono sistemi nei quali i dati vengono forniti in vettori, la singola istruzione viene eseguita su tutti gli elementi del vettore

sistemi multiprocessore Sono sistemi in cui cooperano (o competono) processi su più CPU.

Architetturalmente si suddividono in:
- macchine multiprocessore es. CRAY, NCUBE,…
- sistemi multicomputer es. LAN, Internet,…

A seconda del tipo di gestione della memoria si hanno:
- sistemi a memoria condivisa ← i processi comunicano tramite porzioni di memoria condivisa
- sistemi a memoria distribuita ← i processi comunicano mediante l'invio di messaggi

Sessione:

Per sessione di lavoro si intende tutto l'insieme di operazioni che possono essere considerate facenti parte di un unico processo logico instaurato tra diverse applicazioni su computer remoti. Ad es:
- sessione al terminale
- file transfer
- trasferimento video real-time
- …

parametri che caratterizzano la trasmissione all'interno di una sessione:
- velocità e variabilità nell'arrivo dei messaggi
- durata della sessione
- lunghezza media dei messaggi
- ritardo massimo consentito
- affidabilità richiesta
- …

La comunicazione tra applicazioni su host remoti avviene mediante lo scambio di messaggi.

messaggio: Per messaggio possiamo intendere un insieme di informazioni o dati necessarie alla cooperazione (competizione) tra processi. Tale messaggio sarà verosimilmente dotato di headers (intestazioni) e subirà talune operazioni a seconda del tipo di dati contenuti (codifica, compressione, crittografia, frammentazione…).

modelli di distribuzione temporale dei messaggi all'interno di una sessione:

poissoniano: distribuzione statistica di Poisson

on-off: slots che possono (o meno) contenere messaggi →
- non si supera la capacità della linea (→ non si ha congestione);
- i campioni viaggiano con ritardo costante fissato;

Commutazione di circuito ⇔ commutazione di pacchetto

prima di iniziare una trasmissione deve allocarsi un intero cammino source→ receiver capace di permettere la trasmissione.
Se non è possibile allocare tutte le risorse necessarie, la trasmissione non ha inizio. Una volta allocate, le risorse verranno però mantenute –in modo dedicato- fino alla fine della connessione.

Attualmente la commutazione di circuito non è utilizzabile per le trasmissioni dati:
- costo delle linee eccessivo
- trasmissione dati con caratteristica di discontinuità molto accentuata (burst)

è un processo store-and-forward:
i pacchetti si muovono indipendentemente gli uni dagli altri (seguendo possibilmente percorsi differenti).

Conseguentemente si ha:
- migliore utilizzo delle linee (best-effort)
- nessuna garanzia sul mantenimento dell'ordine dei pacchetti all'arrivo

Varie soluzioni per lo smistamento dei pacchetti:
- *circuiti virtuali*
  non vengono allocate risorse ma tutti ipacchetti della stessa sessione seguono lo stesso percorso (-->stesso ritardo,...)
- *datagram puro*

Sia:
- λ: capacità di arrivo dei messaggi
- $1/\lambda$: tempo medio di arrivo dei messaggi (intervallo)
- χ: tempo medio di trasmissione di un messaggio
- $\underline{L}$: lunghezza media di un messaggio
- r: rate (velocità di trasferimento)

$\chi = \underline{L}/r$,

$\chi / (1/\lambda) = \lambda\chi$: tempo medio di occupazione del canale
$\lambda\chi \ll 1$ indica una cattiva utilizzazione del canale (telnet: $\lambda\chi = 0.01$)

Teoria della Complessità

Algoritmo: Procedura computazionale ben definita (specifica precisa ed inequivocabile) per la risoluzione di un problema. E' corretto se, per ogni istanza di ingresso, esso termina sul corretto output.

Parametri per la valutazione di un algoritmo distribuito:
- complessità spaziale (=occupazione di risorse)
- complessità temporale
- complessità di comunicazione (=numero di messaggi necessari)

dipendono da:
- **input size**: dimensione dei dati d'ingresso (indicata con uno o più valori)
- **running time**: numero di operazioni primitive (=passi) da eseguire su un particolare input, per la terminazione dell'algoritmo.

Indichiamo con T(n) la sommatoria dei costi delle singole istruzioni per il numero di volte cui l'istruzione viene eseguita.
Si possono considerare il caso migliore (*bcrt: best case running time*) e quello peggiore (*wcrt: worst case running time*) (a seconda dell'input).
Generalmente si prende in esame il caso peggiore per diverse ragioni:
- il *wcrt* costituisce un limite superiore per qualsiasi input
- il *wcrt* si presente (generalmente) molto spesso (es. ricerca in un database di un elemento non presente)
- il tempo del "caso medio" è spesso molto vicino al *wcrt.*

Ordine di crescita

Non è importante ricavare (volta per volta) il costo computazionale preciso dell'algoritmo, interessa infatti la velocità di crescita della complessità (tempo di esecuzione, numero di messaggi, risorse occupate,…) al variare delle dimensioni dell'input.
Si considera quindi soltanto l'ordine di grandezza della misura di complessità.
Definiamo:

- $\Theta(g(n)) = \{ f(n): \quad \exists c_1,c_2,n_0 \in \Re+ \rightarrow \quad 0 \le c_1 g(n) \le f(n) \le c_2 g(n), \quad \forall n \ge n_0 \}$
  → g(n) costituisce un limite asintoticamente stretto per f(n).
- $O(g(n)) = \{ f(n): \quad \exists c,n_0 \in \Re+ \rightarrow \quad 0 \le f(n) \le cg(n), \quad \forall n \ge n_0 \}$
  → g(n) costituisce un limite superiore asintotico per f(n)
- $\Omega(g(n)) = \{ f(n): \quad \exists c,n_0 \in \Re+ \rightarrow \quad 0 \le cg(n) \le f(n), \quad \forall n \ge n_0 \}$
  → g(n) costituisce un limite inferiore asintotico per f(n)

I problemi, a seconda dell'ordine degli algoritmi *migliori* di risoluzione, si classificano in:
- $\wp$ → problemi risolvibili con algoritmi di complessità di ordine polinomiale
- $\aleph \wp$ → problemi non risolvibili con algoritmi polinomiali (→esponenziali)

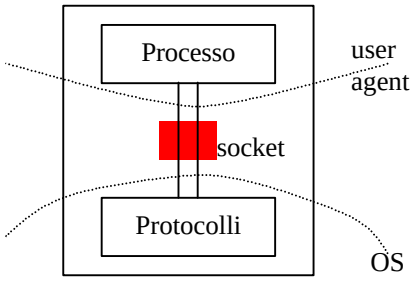
Metodi di sviluppo per applicazioni distribuite:

- **RPC** *Remote Procedure Call*
  Chiamata di funzione remota: permette di eseguire una funzione che si trova su una macchina remota.
  limiti: - durante l'esecuzione della funzione il programma resta in *waiting* → superabile by multitheading
  - problema del passaggio dei parametri, necessariamente per valore
- **MPI** *Message Passing Interface*
  Sistema che permette di creare processi che possono scambiarsi messaggi, si basa sulla PVM (Parallel Virtual Machine del SO). Alla creazione di un processo (mediante richiesta alla PVM), viene assegnato un identificativo (TID: Task IDentify); è mediante tale valore che è possibile comunicare (si utilizza il TID come un indirizzo univoco del processo, indirizzando alla PVM i messaggi).
  Primitive messe a disposizione della PVM:
  - send (TIDdest, Data);
  - receive (TIDsource);
  - multicast … → usata per il multicast
  - barrier … → usata per la sincronizzazione di gruppo
  - …

INTERNET
Applicazioni

Introduzione

L'applicazione distribuita è divisa in due parti: **client** (chi pone domande) e **server** (chi risponde)



Applicazione

analogamente ai files, le socket in UNIX sono foglie del file system:

attributi del file system UNIX (by ls –l):

```
_ ___ ___ ___
t user group all
```

r,w,x (read,write,execute)

type file: { d: directory; f: file; s: socket

socket

(letteralmente *presa telefonica*)

- L'indirizzo di un'applicazione è quello della socket non del processo (analogamente a quel che avviene in telefonia).
- L'applicazione prepara i dati e li fornisce alla socket (al limite suggerisce il tipo di protocollo), sarà questa ad occuparsi della trasmissione.
- E' compito della socket, smistare i dati in arrivo sul computer verso i vari processi residenti (l'host ha un unico indirizzo IP, ogni processo ha un suo numero di porta).
- E' possibile effettuare delle richieste di qualità del servizio (**QoS**: quality of service) a livello di applicazioni (relativamente, ad es., a data loss, band width, timing,…)

DNS Domain Name System

E' un'applicazione distribuita ma non viene utilizzata dagli utenti (è di servizio per tutte le altre applicazioni).
Si basa sullo UDP.

Funzionalità del DNS:

1. traduzione dei nomi simbolici (max 63 caratteri) in indirizzi IP
2. *alias* dei nomi
3. distribuzione del carico: Il DNS può tradurre un indirizzo logico in più indirizzi fisici diversi (ciclicamente) per distribuire il carico di lavoro (es. Yahoo)

Organizzazione gerarchica:

I DNS sono organizzati gerarchicamente. Quelli a livello più alto sono controllati da organizzazioni internazionali.
Ogni dominio ha un server DNS che mantiene l'autorità sullo stesso (*Name Server autoritative*).

una risposta (dal DNS) può essere *autoritativa*, se data dal DNS di competenza, o meno, se fornita da un DNS intermedio (che aveva l'indirizzo richiesto nella cache)

Modalità di lavoro del DNS in base alle richieste:

a. **ricorsiva**: ogni DNS inoltra (in modo trasparente) la richiesta verso i DNS competenti
b. **iterativa**: il DNS interrogato ritorna l'indirizzo del DNS che ne ha autorità.

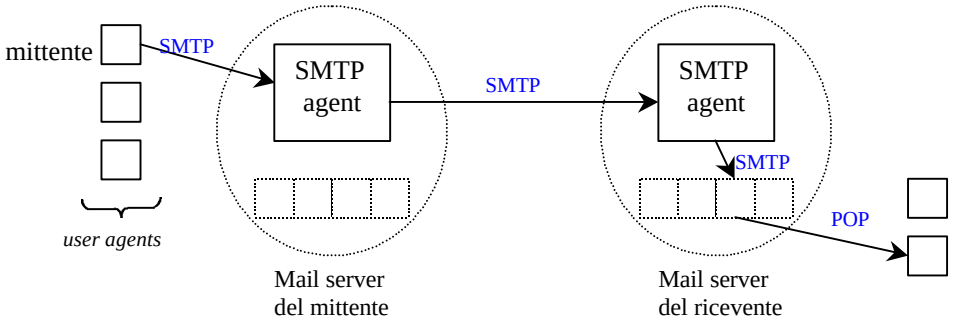
Record DNS (interni ai files di configurazione):

nome logico || IP address || tipo A: address || TTL (tempo di permanenza di un record nella cache di un DNS server)
Cname: per la risoluzione degli alias
…

Messaggi DNS:

16bit identificazione della query | 16bit flags | 16bit n°questions
16bit n°answers | 16bit n°authoriry | 16bit n°additional info
questions || answers || authority || additional info

EMail Posta elettronica



protocolli:

- **SMTP** Simple Mail Transfer Protocol (RFC 821)
  E' un protocollo di push (HTTP era di pull). L'utente spedisce le email all'agente SMTP più vicino. Questo si collega direttamente (senza passaggi intermedi) all'agente SMTP del destinatario, mediante TCP sulla porta 25.

  es.: sender: alice@crepes.fr → receiver: bob@hamburger.edu
  A: HELO crepes.fr
  B: 250 Hello crepes.fr, pleased to meet you
  …

  formato **MIME**: Permette all'SMTP di trasportare formati non ASCII.
  Effettua il mapping in ASCII di qualsiasi file, mediante due attributi:
  1. *content-type*: descrive cosa si sta' trasportando: tipo/ sottotipo; parametri
     es: text/ plain; charset="ISO8859-1"
     text/ html
     multipart ← dentro il messaggio ci sono porzioni di tipo diverso
  2. *content trasfert encoding*: specifica il metodo di codifica usato nel mapping.

  osservazioni:
  - anche l'SMTP server ricevente aggiunge all'email una propria intestazione (data,…)
  - è possibile forzare una gerarchizzazione negli SMTP server (per controllo spamming)

- **POP** Post Office Protocol
  Effettua lo scaricamento della posta dalla casella al computer, mediante TCP sulla porta 110.
  Ha due modalità: 1. scarica e lascia (download and keep)
  2. scarica e cancella (download and delete)

  I comandi principali sono:
  - USER -- per fornire userID+password
  - list -- elenca gli ID dei messaggi
  - retreive -- effettua il download
  - delete -- cancella la posta scaricata

  NB: Il POP3 mantiene uno stato interno (nella sessione con *delete* è necessario tener conto di quali messaggi devono essere cancellati (cancellazione alla fine!))

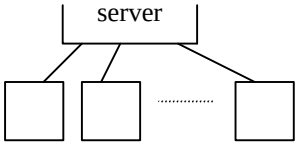
- **IMAP** Internet Mail Access Protocol (RFC 1720)
  Protocollo più complesso dei precedenti.
  Si possono settare diversi filtri sui mittenti, si può scegliere di scaricare solo i subject,…

  Si suddivide in quattro strati:
  1. non autenticato
  2. autenticato (necessita di userID+password)
  3. selected
  4. logout

WWW World Wide Web

protocollo:

- **HTTP** Hyper Text Transfer Protocol (1.0→RFC 1926)
  Protocollo banale (si tratta di un anonymusFTP semplificato) senza memorizzazione di stati interni; la potenza alle applicazioni Web viene data dal linguaggio HTML (Hyper Text Meta Language).



clients:
indipendentemente dall'architettura della macchina (Sun, PC, Apple,…)

Connessione:

- **non persistente** (HTTP 1.0):
  Il client instaura la connessione (TCP) e richiede la pagina. Il server invia la pagina e chiude la connessione. La pagina viene interpretata sul client, se vi sono oggetti correlati (es. immagini), si effettua un'ulteriore richiesta al server (una connessione TCP per ogni oggetto).
  NB: Risulta pesante per via dell'handshaking necessario al TCP.
- **persistente** (HTTP 1.1)
  Si suddivide in due tipologie:
  - *con pipelining*:
    Usa thread differenti (es. immagini caricate in parallelo)
  - *senza pipelining*:
    Non usa thread differenti (es. immagini caricate in serie)

Sintassi generale:

Richiesta Client→Server:
```
method URL version
headerfieldname:valore
…
```

Risposta Client←Server:
```
HTTP/versione   codice-risposta
date: xxxxxx                    -- data+ora invio dal server
server:Apache/1.3.0             -- versione server
last modified: xxxx             -- data ultima modifica
counter_lenght: xxxx            -- dimensione pagina
content_type: xxxx              -- tipo file: html, jpg,…
```

codice-risposta:
- 200: file found
- 3xx: object not found
- 4xx: errori!

accesso condizionato al Web: Nei siti con accesso condizionato, è il browser a memorizzare userID e password (dopo averle inserite la prima volta), ogni altra pagina richiesta viene visualizzata senza ulteriore richieste (anche se l'accesso è condizionato su ogni pagina) perché l'invio di userID e password viene effettuato dal browser in modo "trasparente" all'utente.

cache del browser:

- Prima di scaricare una pagina Web, si effettua un controllo sulla sua eventuale presenza nella cache del disco. In tal caso si confrontano le versioni della pagina (campo last modified) e si procede al download solo se la pagina sul server è più aggiornata di quella presente sulla cache.
- **PROXY** server:
  Funge da server nei confronti degli utenti e da client per il server vero e proprio. Rappresenta una cache condivisa da più utenti.
  . dà vantaggi se le pagine richieste non variano molto (traffico mirato su pochi siti)
  . sono possibili gerarchizzazioni di proxy o proxy distribuiti.

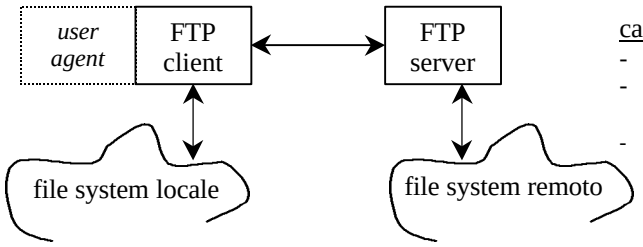
Ha tre comandi: GET, HEAD, POST

es: **GET** URL/namepage.html ↵↵
Permette il download delle pagine Web.
con la ver.1.1 è possibile specificare alcuni parametri:
- connection:close -- simula il comportamento della ver.1.0
- accept: text/html, image/gif,… -- seleziona gli oggetti che si vuole scaricare
- accept: language:xxx -- se il server dispone di più versioni della pagina in lingua diversa

Trasferimento files

protocollo:

- **FTP** File Transfert Protocol
  Il protocollo opera su due file-system differenti, memorizza stati interni;



caratteristiche:
- mantiene lo stato
- richiede userID+password della macchina remota (accesso limitato)
- usa il protocollo TCP;

Usa due canali diversi di comunicazione (comunicazione **out-band**):
porta 21 → comandi; porta 20 → files;

I comandi si dividono in tre categorie:
- -navigazione: cd, lcd, …
- -download: retr (=get)
- -upload: store (=put)

Programmazione di applicazioni mediante socket

La socket identifica una connessione → necessaria una quintupla:
1. protocollo usato
2. indirizzo client
3. processo client (= numero porta)
4. indirizzo server
5. processo server (=numero porta)

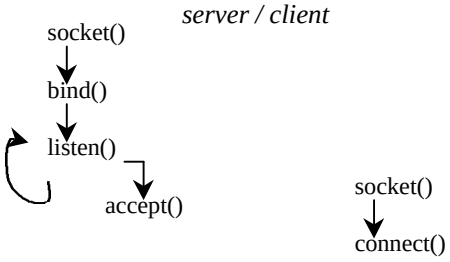
Primitive:

primitive server:
socket () -- creazione socket
bind () -- associazione socket –indirizzo
listen () -- thread in attesa
accept () -- accettazione della connessione

primitive client:
socket() -- creazione socket
bind () ← a carico del SO
connect () -- instaura la connessione

primitive comuni:
read (); write (); close (); shutdown ();

server / client



Esempi di programmazione delle socket:
server e client TCP ed UDP. Compito del server è quello di rispedire indietro i pacchetti che sono arrivati.

```
#include <stdio.h>, <sys/types.h>, <sys/socket.h>, <netinet/in.h>, <arpa/inet.h>
#define   SERV_UDP_PORT   6000
          SERV_TCP_PORT   6000
          SERV_HOST_NAME "147.163.3.5"
char *pname
```

porte libere (tra quelle non gestite dal sistema).
NB: Su una stessa porta possono stare in ascolto demoni di protocolli diversi (una socket è individuata da una quintupla, contenente anche il tipo di protocollo)

La funzione socket, per la creazione della socket, ha 3 parametri:
1. famiglia di protocolli: AF_INET ← costante che identifica la famiglia di tutti i protocolli Internet
2. categoria di protocolli ∈ alla famiglia: SOCK_STREAM ← identifica la famiglia di protocolli affidabili (stream)
   SOCK_DGRAM ← identifica la famiglia di protocolli non affidabili (datagram)
3. protocollo interno alla categoria: 0 ← indica il primo (ed unico, nel caso di TCP ed UDP) protocollo

la funzione bzero ha il compito di pulire (inserendovi 0), byte × byte, la variabile passata come parametro (puntatore).

la funzione htonl ha il compito di effettuare la conversione dell'argomento: host → network long int.
Il formato di destinazione è quello definito XDR (*eXternal Data Rappresentation*), indipendente dall'architettura della macchina.

la costante INADDR_ANY si riferisce ad uno (qualsiasi) degli indirizzi propri del server (se si lavora, ad es., con più schede Ethernet)

Server UDP

```
main (int argc, char *argv[ ])
{           int         sockFD;
            struct      sockaddr_in         serv_addr;
            pname=argv[0];

            if (sockFD=socket(AF_INET, SOCK_DGRAM,0)) < 0) then err_dump ("…");        ← creazione della socket

            bzero ( (char *)&serv_addr, sizeof(serv_addr) );
            serv_addr.sin_family = AF_INET;
            serv_addr.sin_addr.s_addr = htonl (INADDR_ANY);                            binding degli indirizzi
            serv_addr.sin_port = htons (SERV_UDP_PORT);
            if ( bind (sockFD, (struct sock_addr*) &serv_addr, sizeof(serv_addr)) < 0) then err_dump ("…");

            dg_echo (sockFD, &cli_addr, sizeof (cli_addr))                             ← listen ed azione!
}
```

Client UDP

```
main (int argc, char  *argv[ ])
{           int         sockFD;
            struct      sockaddr_in         cli_addr, serv_addr;
            pname=argv[0];

            bzero ( (char *)&serv:addr, sizeof(serv_addr));
            serv_addr.sin_family = AF_INET;                                    instanziazione indirizzo del server
            serv_addr.sin_addr.s_addr = inet_addr  (SERV_HOST_ADDR);
            serv_addr.sin_port = htons (SERV_UDP_PORT);


            if (sockFD=socket(AF_INET, SOCK_DGRAM,0)) < 0) then err_sys ("…");         ← crazione socket

            bzero ( (char *)&cli_addr, sizeof(cli_addr) );                             è necessario effettuare il binding degli indirizzi (UDP
            cli_addr.sin_family = AF_INET;                                             usa la trasmissione mediante datagram, è necessario
            cli_addr.sin_addr.s_addr = htonl (INADDR_ANY);                             mantenere l'indirizzo del mittente.
            cli_addr.sin_port = htons (0);                                             ← si usa una delle port disponibili
            if ( bind (sockFD,  (struct sockaddr *) cli_addr , sizeof(cli_addr)) < 0) then err_dump ("…");

            dg_cli (stdin, sockFD,  &serv_addr, sizeof (serv_addr));                   ← invio dal client mediante datagram
            close (sockFD);
            exit (0)
}
```

Server TCP

```
main (int argc, char *argv[ ])
{       int     sockFD, newsockFD, cliLen, childPID;
        struct  sockaddr_in     cli_addr, serv_addr;
        pname=argv[0];

        if (sockFD=socket(AF_INET, SOCK_STRAM,0)) < 0) then err_dump ("imp. aprire socket stream");   ← creazione della socket

        bzero ( (char *)&serv_addr, sizeof(serv_addr) );
        serv_addr.sin_family = AF_INET;
        serv_addr.sin_addr.s_addr = htonl (INADDR_ANY);                                  } binding degli indirizzi
        serv_addr.sin_port = htons (SERV_TCP_PORT);
        if ( bind (sockFD, (struct sock_addr*) &serv_addr, sizeof(serv_addr)) < 0) then err_dump ("…");

        listen (sockFD,5);                                              ← listen
                                                                          5 indica il numero massimo di connessioni gestibili parallelamente
                                                                          (dipende dalla potenza della macchina)
        while true do
        {
                cliLen=sizeof (cli_addr);
                newsockFD=accept (sockfd, (struct *sockaddr) &cli_addr, &cliLen);
                if newsockFD<0 then err_dump ("…");

                if ( (childPID=fork())<0) then err_dump ("…");          ← fork: padre resta in attesa, figlio serve la richiesta
                if (chidPID==0) then                                    processo figlio:
                {       close(sockFD);
                        str_echo (newsockFD);
                        exit (0);
                }
                close (newsockFD);
        }
}
```

Client TCP

```
main (int argc, char  *argv[ ])
{       int     sockFD;
        struct  sockaddr_in     cli_addr, serv_addr;
        pname=argv[0];

        bzero ( (char *)&serv:addr, sizeof(serv_addr));
        serv_addr.sin_family = AF_INET;                                  } instanziazione indirizzo del server
        serv_addr.sin_addr.s_addr = inet_addr  (SERV_HOST_ADDR);
        serv_addr.sin_port = htons (SERV_TCP_PORT);


        if (sockFD=socket(AF_INET, SOCK_DGRAM,0)) < 0) then err_sys ("…");      ← crazione socket

        il clinet ha un solo indirizzo (TCP è connection-oriented) → non si effettua il binding degli indirizzi (realizzato automaticamente dal SO)

        if (connect  (sockFD, (struct sockaddr*)&serv_addr, sizeof (serv_addr)) < 0) then err_sys ("…");

        str_cli (stdin, sockFD);                                        ← invio dal client mediante strem
        close (sockFD);
        exit (0)
}
```

INTERNET
Trasporto

Introduzione

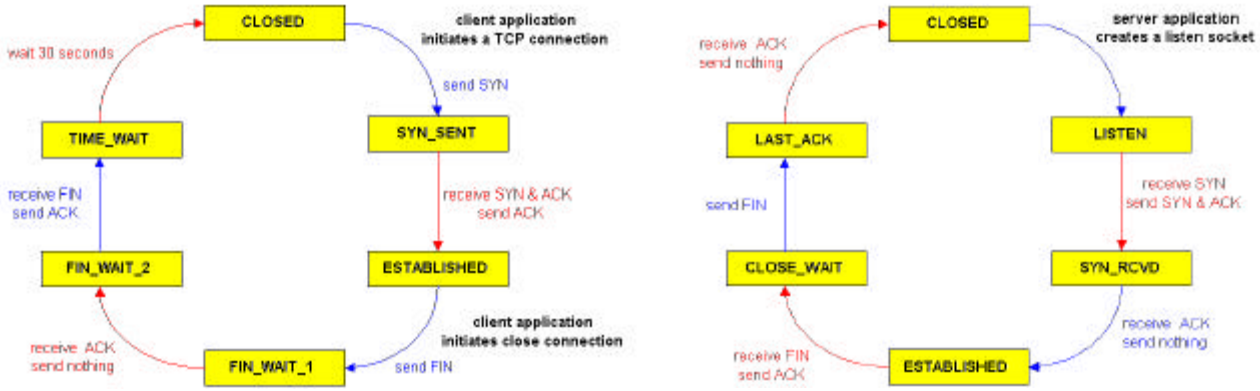
Livello di trasporto gestisce la trasmissione dati tra processi che stanno su host remoti (il livello Network, invece, si occupa della trasmissione tra gli host).
Il pacchetto, in questo livello, prende il nome di *segmento*.

Multiplexing e demultiplexing dei flussi

Il livello Network effettua il trasferimento dei dati tra due host, ogni host è identificato da un unico indirizzo IP.
Sull'host di destinazione, lo strato di trasporto riceve il segmento dallo strato di rete sottostante ed ha il compito di fornirne i dati alla relativa applicazione (attiva sull'host).
Ogni segmento dello strato di trasporto ha un campo che viene usato per individuare l'applicazione cui fare riferimento.
Il lavoro di smistare i dati contenuti nel segmento verso l'applicazione corretta è detto **demultiplexing**.
Il lavoro, all'host source, di riunire dati provenienti da applicazioni differenti, incapsularli in un segmento (con un header contenete informazioni necessarie al demultiplexing) e fornire tale segmento allo strato sottostante detto **multiplexing**.

UDP e TCP realizzano i demultiplexing e multiplexing inserendo nell'header del segmento i campi source port number e destination port number.

Connessione:



Protocolli:

- **UDP** User Datagram Protocol
  Datagram (*connection less*)

  funzionalità: - multiplexing/demultiplexing dei dati (su un'unica connessione di rete)

  utilizzo: UDP usato dal DNS (ovviamente), nel multicast (impossibile stabilire connessioni uno a molti), nelle applicazioni real-time (es: voice on IP), nelle informazioni di servizio di routing.

  formato pacchetto:

source:port (16)	dest:port (16)	lungh. dati (16)	checksum (16)

  ← header

  controllo errori: La checksum è calcolata sull'header UDP + l'header IP, come segue:
  Si sommano tutti i blocchi di 16bit e si complementa ad uno il risultato. Il ricevente effettuerà la somma di tutti i blocchi, checksum compreso, dovendo ottenere 0.

  Se si incontra un pacchetto danneggiato:
  1. viene ugualmente fornito all'applicazione, segnalandone la corruzione
  2. viene scartato

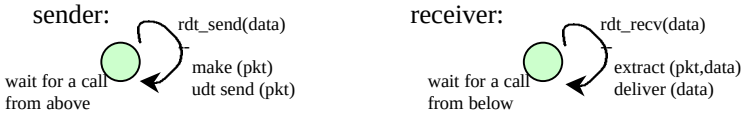
- **TCP** Transmission Control Protocol
  Orientato alla connessione (*connection oriented*) → three way handshaking

  funzionalità: - multiplexing/demultiplexing dei dati (su un'unica connessione di rete)
  - trasferimento affidabile dei dati
  - controllo di flusso
  - controllo della congestione

  costruzione: (rdt: reliable data transfert; udt: un-reliable data transfert)

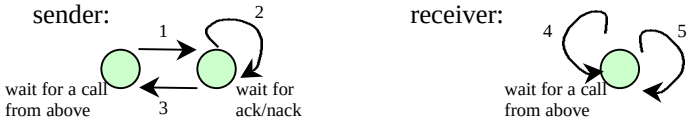
  - release 1.0: rete affidabile

    dsf: sender: receiver:

  - release 2.0: rete dà errori sul bit

    dsf: sender: receiver:

    1. rdt_send (D)
       compute (check)
       make (pkt,D,check)
       udt_send (pkt)

    2. rdt_rcv (pkt) &&
       isNACK (pkt) →
       udt_send (pkt)

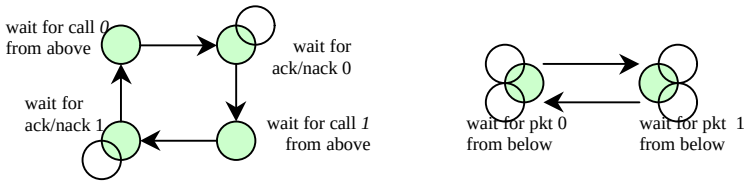
    3. rdt_rcv (pkt) &&
       isACK (pkt)

    4. rdt_rcv (pkt) &&
       NOTCorrupt (pkt) →
       extract (pkt,D)
       deliver (D)
       udt_send (ACK)

    5. rdt_rcv (pkt) &&
       Corrupt (pkt) →
       udt_send (NACK)

  - release 2.1:
    Se non viene riconoscuito ACK o NACK, si rinvia nuovamente il pkt, marcandolo però come copia (in modo che il receiver lo possa scartare se la ricezione precedente era anadata bene (aveva inviato un ACK))

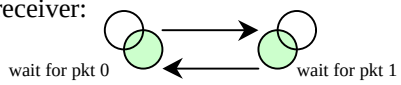
    dsf: sender: receiver:

  - release 2.2:
    Sostituiamo il segnale di NACK con l'ACK precedente (già costruito e memorizzato ancora nel buffer in out) es: accettiamo il pkt1 ed inviamo l'ack1; attendiamo il pkt0, giunge corrotto, inviamo nuovemente l'ack1!

    dsf: sender: receiver:

    simile al precedente

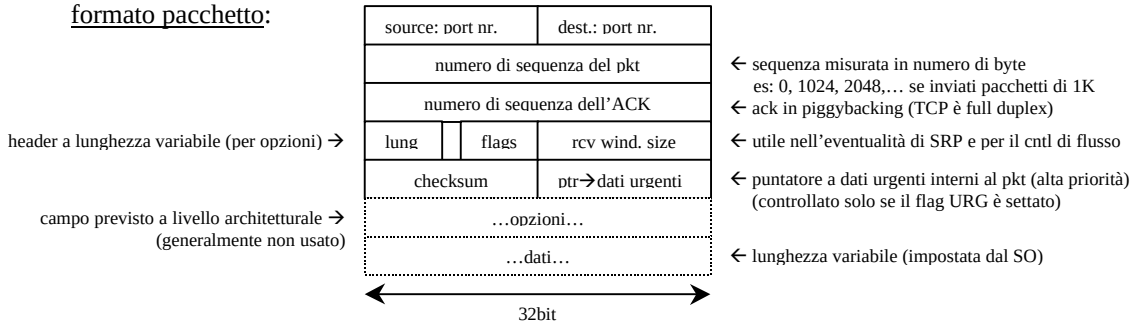
    

  - release 3.0: la rete può far perdere pacchetti (non solo danneggiarli)
    E' necessario introdurre dei numeri di sequenza (come prima) ed un timer (per l'attesa del riscontro dell'invio)
    → Si tratta dell'**ABP** (protocollo usato a livello data link), bassa efficienza

  - release 4.0:
    Nell'automa a stati finiti è prevista l'introduzione di variabili (→ è un vero e proprio programma).
    → Introduzione del *windowing* secondo la procedura **go-back-N**.
    NB: La RFC del TCP non specifica la struttura del receiver, in genere è privo di buffer (go-back-N). In alcune RFC proposte, si introduce un buffer al receiver, con possibilità di adottare strategie **selective-repeat**.

    eventi→azioni:
    - giunge un pacchetto in sequenza (atteso):
      Si attende un po' (500ms) per vedere se arriva il segmento successivo e mandare un **ack cumulativo** (immediatamente). Se non arriva, si invia l'ACK per il segmento in oggetto.
    - giunge un pacchetto fuori sequenza:
      Si rinvia (immediatamente) l'ultimo ACK inviato (ancora nel buffer).

formato pacchetto:



controllo di flusso: (relativo al problema dell'inondazione di dati sul receiver (receiver lento))

Si usa il campo rcv_window_size. Indica quanto spazio di memoria il receiver ha realmente a disposizione (alcuni pacchetti giunti, e già riscontrati, potrebbero non essere ancora stati prelevati dall'applicazione, continuando –quindi- ad occupare il buffer).

Se rcv_window_size và a zero e nessun pacchetto deve andare dal receiver al sender (ad es. ACK,..) allora il sender non ha modo di sapere se si è liberato spazio sul receiver (l'informazione rcv_window_size viaggia infatti nei pacchetti TCP). Per evitare il potenziale deadlock, il sender invia periodicamente (anche nel caso di rcv_window_size=0) un byte di "prova", il receiver dovrà riscontrarlo, trasmettendo quindi anche l'informazione sulla rcv_window_size)

controllo della congestione:

Principi generali

- Sui nodi terminali (a livello trasporto):
  Sui nodi terminali non è possibile avere una visione della rete (e della congestione in essa). Ci si basa solamente su informazioni dedotte (frequenza di perdita dei pacchetti, ritardi di ricezione,…)
- A livello di rete:
  Molto efficiente. Utilizzato in sistemi proprietari particolari ma proposto anche per Internet (RED). Quando la lunghezza delle code di smistamento (sui routers) supera una certa soglia, si possono avere due azioni:
  1. Si notifica all'indietro (router x router, fino al sender) lo stato di congestione.
  2. Lo stato di congestione viene notificato al ricevente (settando un opportuno flag sul pacchetto stesso).

Nel TCP:

Il TCP effettua il controllo della congestione tramite due variabili: **congestion window**, **threshold**.

Supponiamo che sul receiver non vi siano problemi di buffer che non dipendano dalla congestione (valore della rcv_window_size molto alto).

funzionamento: Il sender invia un pkt, se viene riscontrato in tempo (entro un timeout) si raddoppia la dimensione della finestra (congestion window). Si procede in questo modo fino a che la dimensione della finestra=threshold (che funge da limite superiore).
Da questo momento, se tutto continua a procedere bene (riscontri corretti e giunti entro il timeout), la dimensione della finestra aumenta di una unità per volta (aggiornando conseguentemente la threshold)).
Se si dovesse riscontrare un problema (ACK oltre il timeout,…) la dimensione della finestra torna immediatamente ad ampiezza 1 e la theshold viene aggiornata alla metà dell'ultimo valore raggiunto.

INTERNET
Network

Introduzione

Livello di rete si occupa del trasferimento dei dati tra host (quindi instradamento dei pacchetti, scelta della strada migliore, eventuale frammentazione dei pacchetti (a causa del **MTU** delle sottoreti attraversate),…)
Perché gli host possano essere raggiunti, devono essere distinguibili e quindi dotati di indirizzi univoci su tutta la rete.
Il pacchetto, in questo livello, prende il nome di *datagram*.

IP Internet Protocol

Indirizzamento:

L'assegnazione degli indirizzi Internet viene svolta da autorità internazionali dedicate, adesso suddivise in *dipartimenti* nazionali (in Italia è il CNUCE, Pisa).

- IPv4: dimensione: 4byte → oltre 4mld di host indirizzabili
  suddivisione in classi:

classe A (primo byte ∈ [0…127]):	assegnato il primo byte → 256 gruppi da 16mln di host
classe B (primo byte ∈ [128..191]):	assegnati i primi due byte → 65500 gruppi da 65500 host
classe C (primo byte ∈ [192..223]):	assegnati i primi 3 byte → 16mln di gruppi da 256 host
classe D (primo byte ∈ [224..225]):	utilizzati per il multicasting su Internet

NetMask

Serve per effettuare distinzioni in sottoreti nell'indirizzamento interno di una rete locale (con indirizzo di classe A,B o C).

Ad es: una rete cui è stato assegnato un indirizzo di classe B, può indirizzare 65536 host. Questi potrebbero essere suddivisi in sottoreti interne. Si potrebbero, ad es., creare 256 sottoreti di 256 host (in questo caso, dei due byte *liberi*, uno verrebbe assegnato *internamente* per l'individuazione della sottorete).

La NetMask effettua un mascheramento dei byte dell'indirizzo che non è possibile gestire per i singoli host. Nell'esempio, la NM vale 255-255-*255-0*.

NB: Risulta più comodo lavorare con i numeri binari (bit=1→assegnato all'identificazione della rete-sottorete; bit=0→identifica l'host)

- IPv6: dimensione: 16byte → oltre $3,4 * 10^{38}$ host indirizzabili

IPv4 IPv6

formato pacchetti:

- IPv4:

- versione	4bit	
- header lenght	4bit	
- type of service	8bit	pensato per suddividere traffici di tipo differente
- packet lenght	16bit	
- identifier	16bit	ID specifico del pacchetto (i frammenti hanno stesso ID)
- flags	3bit	flags; tra questi uno identifica l'ultimo frammento di un pacchetto
- fragment offset	13bit	offset del frammento all'interno del pacchetto originario (in byte)
- TTL	8bit	Time To Live
- upper layer prot.	8bit	identificativo del protocollo dello strato superiore (=cosa c'è nei dati?)
- header checksum	16bit	
- IP source addr.	32bit	
- IP destination addr.	32 bit	
- options	…	campo opzionale
- data	…	

- IPv6:

- versione	4bit	
- priority	4bit	
- flow label	24bit	
- payload lenght	16bit	lunghezza del campo dati
- next header	8bit	analogo al campo upper-layer-protocol nell'IPv4
- hop limit	8bit	analogo al campo TTL nell'IPv4
- IP source addr.	128bit	

Confronto IPv4 IPv6:

L'header del pacchetto IPv6 ha dimensione fissata (contrariamente a quello dell'IPv4) ed è molto snello (solo 40byte).
Nella versione 6 si introduce il concetto di **flusso di dati** (identificato dal campo flow-label): insieme di dati, da una sorgente ad un destinatario, su cui devono essere intraprese politiche uniche particolari (es. ritardo massimo fissato,…)

E' possibile assegnare priorità diverse (con il campo priority) ai diversi flussi (telnet, audio, video,…).
Si eliminano invece i concetti di:

- *segmentazione* → risolta attraverso il protocollo **ICMP** (Internet Control Message Pr.)
  si manda un messaggio indietro al sorgente, indicando l'MTU della sottorete attraversata → Il source trasmette pacchetti con la nuova dimensione.
- *checksum* → i controlli vengono delegati ai protocolli superiori (transport layer)
  il calcolo della checksum avverrebbe su ogni router (su ogni router avviene infatti una modifica al TTL e quindi all'header), rallentando troppo il loro lavoro.

L'idea di base dell'IPv6 è infatti quella di snellire il più possibile il peso computazionale a carico dei singoli router.
NB: IPv6 permette, oltre che l'unicast, il multicast ed il broadcast (possibili anche con la versione 4), anche l'**anycast** (=*uno tra questi*, utile -ad es.- per lo switching nei providers).

Passaggio IPv4 → IPv6:

Impossibile pensare ad un flag-day di passaggio istantaneo in tutto il mondo.
Si può quindi pensare ad una fase di transizione in cui le macchine possano leggere entrambe le versioni del protocollo IP.
La scelta tra le due versioni avverrebbe in fase di instaurazione della connessione, con il supporto dei DNS (si opta per la comunicazione IPv6 se sorgente e destinazione sono entrambe in grado di interpretare tali pacchetti).
Un problema potrebbe insorgere qualora tra source e destination ci fossero sottoreti intermedie non compatibili con l'IPv6.
In questo caso sarebbe necessario effettuare un **tunneling**
si incapsula il pacchetto IPv6 in uno IPv4; si attraversa la sottorete "scoperta" (mediante tunnel = rotta statica) e si estrae il pacchetto originario (IPv6) al raggiungimento di una sottorete idonea.

Routing

L'instradamento dei pacchetti viene effettuato sulla base di:
- indirizzo dell'host di destinazione
- conoscenza della rete

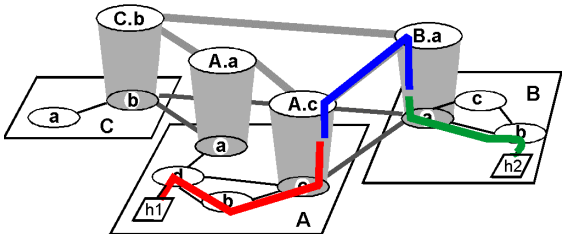
Gli algoritmi di routing assolvono al compito di fornire una visione delle rete e di individuare (sulla base di tale *mappa*) i percorsi *migliori* per l'instradamento (← si tratta quindi di **routines di servizio**)

Il routing può essere **statico** (→topologia della rete non cambia) o **dinamico**.

Routing gerarchico:

I router si dividono in:
- router interni
- router di bordo (gateway)

Per i routers di bordo si usano protocolli dedicati, come il BGP (border gateway protocol), basato sui path (non sui costi), con scelte politiche-economiche.



Si è introdotta la gerarchizzazione per due motivi:

scala: Con l'aumentare del numero dei nodi, il peso computazionale ed il numero di messaggi necessari al routing diviene eccessivo (e le tabelle avrebbero dimensioni enormi!).

autonomia amministrativa;

famiglie di protocolli di routing:

- *distance vector protocol*

  Protocolli distribuiti → in ogni nodo è necessaria conoscere solo i neighbors (e dei costi dei relativi link)
  Un protocollo di questa famiglia è il RIP.

  RIP: Routing Internet Protocol (basato sull'algoritmo di **Bellmann-Ford-Fulkensonn**)

  E' un protocollo iterativo, asincrono e distribuito.
  Ogni 30'' ciascun router comunica agli adiacenti il proprio *distance-vector*. Quando il router riceve il vettore delle distanze di un vicino, aggiorna (se si ottengono costi minori di quelli attualmente previsti) la propria tabella.
  Il confronto è del tipo: $costo_{A,i}$ $costo_{B,i} + distanza_{A-B}$;

  Tale algoritmo richiede:
  - potenza di calcolo: bassissima (solo somme e confronti)
  - numero di messaggi: molto elevato (propagazione continua delle tabelle)

  I vettore delle distanze (*distance-vector*) è del tipo:

elenco nodi rete	costo per il singolo nodo	vicino sul cammino min.

  inizializzato con le distanze dei soli adiacenti (gli altri nodi avranno distanza ∞), a regime dopo n-1 passi.

- *link state protocol*

  Sono protocolli centralizzati → necessitano della completa conoscenza della rete
  Un protocollo di questa famiglia è l'OSPF.

  OSPF: Open Short Path First (basato sull'algoritmo di **Dijkstra**)

  Per verificare l'ipotesi (conoscenza della topologia di tutta la rete), i router trasmettono in broadcast (a tutta la rete) le variazioni occorse (aggiunta nuovo link, cancellazione link,…).
  Il protocollo, basandosi sull'algoritmo di Dijakstra, è iterativo e centralizzato.

  NB: Dijakstra necessita di N passi (se N nodi). Dopo k passi, però, riesce a fornire il percorso migliore verso i k nodi più vicini.

- *Routing adattivo*

  Routing che si adatta alle condizioni (*variabili*) della rete, inclusa la congestione.
  Un protocollo di questo tipo è quello basato sulle *formiche*.

  Formiche:

  Informazioni iniziali: le formiche camminando rilasciano una sostanza (**feromone**) che attira altre formiche (proporzionalmente alla quantità); il feromone è volatile (diminuisce con il tempo).

  algoritmo:
  - le formiche esplorative(=capsule) si muovono inizialmente a caso, fino al raggiungimento della destinazione.
  - tornando indietro alla sorgente (simulando sorgente→destinazione ≡ formicaio→cibo) si effettua lo stesso percorso: il feromone si somma a quello precedente (aumentando di intensità). La strada più breve (in termini di tempo) ha più feromone (minore tempo→minore evaporazione).

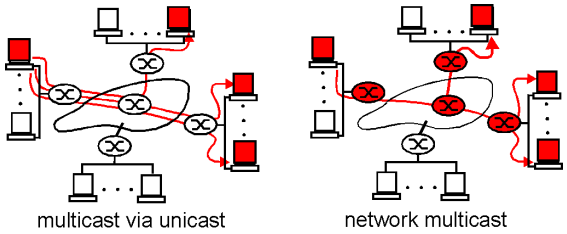
  I pacchetti (dati) seguono, verso la stessa destinazione, la strada con più feromone.
  Periodicamente ogni router invia le formiche esploratrici (per saggiare le mutate condizioni della rete).

Multicast

Tipologie di comunicazione su Internet:

- **Unicast** coinvolge solamente un host sorgente ed un host destinazione (comunicazione *punto-punto*).
- **Multicast** una singola operazione di trasmissione invia dati a molte destinazioni (comunicazione *uno-a-molti*).
  es: videoconferenza, streaming continuos media,…
- **Broadcast** una operazione di trasmissione invia i dati dalla sorgente a tutte le destinazioni (*uno-a-tutti*).
  es: TV, radio,…

Il Multicast può essere realizzato in due modi:



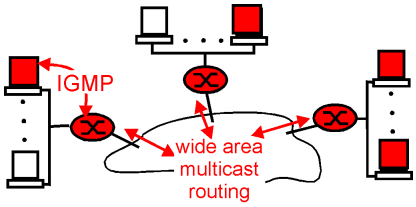
a. mediante connessioni punto-punto (sfruttando quindi l'unicast per realizzare una *simulazione* di multicast)

b. mediante una effettiva rete di multicast

L'utilizzo di una network multicast implica un uso molto più efficiente delle risorse (prevalentemente ampiezza di banda) ma necessita di uno specifico supporto a livello di rete (algoritmi di routing per multicast inseriti su tutti i router).

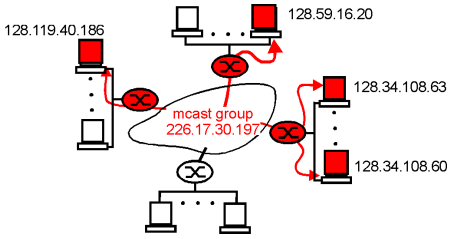
Protocolli:

- protocollo di gestione del multicast tra host finale e router di multicast: IGMP (unico)
- protocolli di multicast routing (tra routers della rete): molti (PIM, BGMP, MOSPF,…), manca una standardizzazione

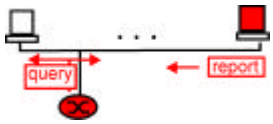


MBone (*Multicast Backbone*):
MBone è una rete virtuale sovrapposta a porzioni di Internet che supporta la trasmissione multicast di datagrams IP. Non è presente su tutti i router di Internet, ma è costituita da *isole* che supportano direttamente il multicast IP e che comunicano tramite IP tunneling.

Multicast group:



L'iscrizione di un host ad un ben determinato gruppo di multicast (identificato da un indirizzo IP di classe D) avviene mediante il protocollo **IGMP** (Internet Group Managment Protocol; RFC:1112):



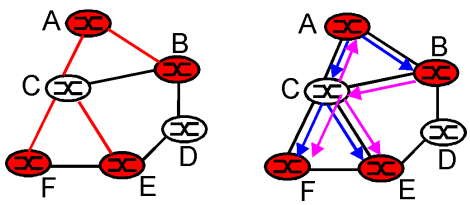
I router di MBone (quelli esterni, cui sono connessi degli host) inviano periodicamente una query di *inscrizione* al multicast group a tutti gli host. Questi possono rispondere, con un messaggio di report, se sono interessati al gruppo.
Se il router non riceve nessuna risposta, non si occupa più del gruppo di multicast (non fa più parte dei router di multicast per *quel* gruppo).

Multicast Routing

Scopo del Multicast Routing è quello di individuare un sottoinsieme dei nodi e degli archi della rete che permetta di connettere tutti quei router necessari a raggiungere i nodi del gruppo di multicast.

Gli algoritmi di Multicast Routing si suddividono in due gruppi:

- **Source-based trees**
  Si individua un albero di multicast per ogni sorgente (nodo del gruppo)
  algoritmi: - spanning trees
  - reverse path forwarding
- **Group-shared tree**
  Si crea un unico albero condiviso da tutti gli host del gruppo.
  algoritmi: - flooding
  - core-based tree
  - Steiner tree



1. **Flooding**

Quando un nodo della rete riceve un pacchetto indirizzato ad una destinazione di multicast, lo inoltra (se non è già stato precedentemente ricevuto ed inoltrato) verso tutte le sue interfacce tranne quella di provenienza.

tecniche possibili per stabilire se un pacchetto è già giunto sul nodo:

- sul nodo si mantiene una lista dei pacchetti che lo hanno attraversato di recente
- sul pacchetto si mantiene la lista dei nodi attraversati

vantaggi: semplice, robusto, non dipende da protocolli unicast, garantisce che il nodo inoltri una sola volta ogni pacchetto
svantaggi: non garantisce che il pacchetto giunga sul nodo una sola volta (sicuramente non più di una volta sul singolo link), dispendioso (in termini di memoria e di ampiezza di banda).

2. **Spanning trees**

Si marcano i link appartenenti all'albero di copertura (sottografo aciclico). Quando un pacchetto viene ricevuto, viene inoltrato su tutti i link marcati (tranne quello di provenienza).

vantaggi: semplice, efficiente
svantaggi: minimizza (se alberi costruiti con Dijkstra) ciascun path S→D, ma non il costo complessivo dell'albero. concentra il traffico sui link marcati lasciando (possibilmente) del tutto scarichi gli altri.

3. **Reverse path forwarding**

Rappresenta un miglioramento del flooding (ed è l'algoritmo attualmente usato su MBone). Quando il nodo riceve un pacchetto proveniente dalla sorgente S attraverso l'interfaccia del percorso più breve (cammino minimo *inverso*: nodo→S), allora lo inoltra verso tutte le interfacce tranne quella di provenienza; altrimenti il pacchetto viene scartato.

vantaggi: non si richiedono più risorse di quelle necessarie al routing unicast, cammini minimi tra S e D
svantaggi: differenti alberi di copertura per differenti sorgenti, problemi nel caso di link unidirezionali.

variante: **flood and prune**: Si troncano i rami inutili (che non portano a host interessati al gruppo di multicast), tenendo conto delle informazioni sugli appartenenti al gruppo.
La potatura avviene *a ritroso* (un messaggio viene inviato periodicamente su tutta la rete per *innescare* il processo di pruning).
Problema: se un nodo, di un ramo potato, volesse iscriversi al gruppo?

4. **Core-based tree**

Il "*core*" è un nodo della rete che svolge le funzioni di centro (pivot) per l'intero gruppo di multicast. L'albero di copertura è costituito da tutti i cammini minimi dai nodi del gruppo di multicast verso il *core*.

I nodi che vogliano inserirsi nel gruppo devono avanzare una richiesta (join message) al core. La richiesta, nel suo viaggio dal nodo al core, permette la costruzione dell'albero di copertura.

vantaggi: efficiente (nessuna inondazione), non dipende da protocolli unicast, minimizza i cammini nodo→S
svantaggi: l'albero non è ottimale, concentrazione di traffico sui link (condivisi).
problema principale: individuazione del nodo core.

5. **Steiner tree**

Individua l'albero di copertura (dei soli nodi di multicast) dal costo minimo globale (albero di copertura ottimo) → minimizza le risorse della rete piuttosto che garantire il percorso più breve verso le singole destinazioni.
Il problema della determinazione di tale albero di copertura è noto come **Steiner tree Problem**.

Procedure relative agli algoritmi prima descritti:

- da (2): Spanning trees:

Shortest Paths Tree

Si abbia un grafo non orientato, connesso, pesato: G=(V,E,ω); con $s \in V$ nodo sorgente, ω pesi non negativi, |V|=n, |E|=m

Problema: Determinare l'albero dei cammini minimi (=albero radicato in s che copre tutti i nodi , costituito dai cammini minimi dal nodo sorgente s ad ogni nodo della rete)

Algoritmo:

Dijkstra

E' un algoritmo centralizzato (prevede la conoscenza dell'intera topologia della rete).

```
X←s;                          → insieme dei nodi già analizzati completamente
d[s]←0;                       → vettore delle distanze
p[s]←ϕ                        → vettore dei nodi vicini (a quelli in esame) nel cammino minimo verso s.
for ∀u∈V-{s} do               → per tutti i nodi eccetto quello sorgente…
    d[u]←ω(s,u) or ∞          → inizializzazione distanza s—u: ∞ se non esiste link diretto
    p[u]←s or ϕ               → inizializzazione nodo più vicino verso s: ϕ se non esiste link diretto s—u
end for
while X≠V do                  → finché non tutti i nodi sono stati esaminati…
    u←nodo∈V-X: d[u] is minimum   → estrai il nodo (ancora da esaminare) che ha d[u] minimo
    X←X U {u}                 → aggiorna insieme dei nodi già esaminati
    for ∀ v∈V-X do            → per tutti i nodi ancora esterni all'albero…
        if (d[v]>d[u]+ω(v,u)) then   → se la distanza attuale di v dalla radice è maggiore di quella
          d[v]←d[u]+ω(v,u); p[v]←u;    ottenuta usando il nodo u come intermedio: aggiorna distanza
    end for
end while
```

correttezza: Ad ogni passo sono validi i seguenti invarianti:

1. $\forall u \in V$, d[u] è il costo del cammino minimo da s ad u che utilizza solo nodi di X (già esaminati)
2. $\forall u \in X$ e $v \notin X$, $d[u] \leq d[v]$

Per cui, alla terminazione dell'algoritmo (X=V), il vettore d contiene i costi dei cammini minimi.

running time: selezione nodo e aggiornamento vettore → $O(n^2)$
ordinamento pesi degli archi → O (m log m)
} $O(n^2 + m \log m)$

- da (2) Spanning trees:

Minimum Spanning Tree

Si abbia un grafo non orientato, connesso, pesato: G=(V,E,ω); con |V|=n, |E|=m, ω pesi non negativi.

Problema: Determinare l'albero di copertura (per tutti i nodi della rete) di costo *globale* minimo.

Algoritmi:

Prim complessità: $O(m + n \log n)$

```
Q←V;                          ← Q è coda di priorità di nodi non inclusi nell'albero
for ∀u∈Q do Key[u]← ∞         ← Key[v] è il costo minimo tra ∀ arco che connette v ad un nodo già incluso nell'albero
Key[s]←0;
p[s]←null;                    ← p[v] è il nodo del MST, cui v si connette con peso minore (Key[v])
while Q≠ϕ do                  ← finché non tutti i nodi sono stati esaminati…
    u←Extract_Min (Q)         ← estrai il nodo (non già del MST) che ha il link di peso minore verso l'albero
    for ∀v∈ Key[v]←ω(u,v) Adj[u] do   ← per ogni nodo adiacente ad u…
        if v∈Q & ω(u,v)<Key[v] then   ← se v∈Q giunge attualmente al MST con peso > ω(u,v)…
            p[v]←u                    ← v può connettersi al MST attraverso il nodo u
                              ← aggiornato il peso del link tra v e il MST.
    end for
end while
```

Kruskal complessità: O (m log m)

A←ϕ ← A è l'insieme degli archi di E già esaminati
for ∀ vertex v∈V do ← per ∀ nodo del grafo…
Make_Set(v) ← genera la foresta: inizialmente ogni nodo è un albero a sé.
ordina gli archi (∈E-A) con peso ω non decrescente
for ∀ edge (u,v)∈E-A do ← per ogni arco di E (ordinato con peso ω non decrescete)
if FindSet(u)≠FindSet(v) then ← FindSet(x) restituisce un leader (es. radice) dell'albero di appartenenza
A←A U {(u,v)} ←
Union(u,v) ← Union(x) combina due alberi in uno
end for
return A

- da (5) Steiner Tree:

Steiner Tree Problem
Determinare l'albero di copertura per un sottoinsieme (=gruppo di multicast) dei nodi della rete, di costo *globale* minimo. L'albero che se ne ricava è detto **Steiner Minimal Tree** (SMT).
Problema: Dato un grafo non orientato, connesso, pesato G=(V,E,ω) ed un insieme di nodi Z⊂V, determinate l'albero T_0 di copertura di Z in G di costo minimo (=SMT).
Valgono: |V|=n, |E|=m, ω:E→ℜ+, |Z|=p; Si v definisce **nodo di Steiner**, se v∈T_0 & v∉Z.

Il problema si riduce (**casi particolari**):
- alla costruzione dello *Shortest Path* (mediante algoritmo di Dijkstra) ← per p=|Z|=2;
- alla costruzione del *Minimum Spanning Tree* (algoritmi di Prim e Kruskal) ← per p=n (ossia Z≡V)

La costruzione di uno SMT è provato essere problema **ℵ℘ completo**, per cui:
- algoritmi esatti in tempi non polinomiali:
  Spanning Tree Enumeration Algorithm (**STEA**) $O(p^2 \times 2^{n-p} + n^3)$ performance: 1
- euristiche con tempi polinomiali e prestazioni garantite (performance = $c(T_H) / c(T_0)$):
  Pruned Dijkstra Heuristic (**PDH**) $O(n^2)$ performance: p
  Distance Network Heuristic (**DNH**) $O(p \times n^2)$ performance: 2-2/p
  Shortest Path Heuristic (**SPH**) $O(p \times n^2)$ performance: 2-2/p
  Kuskal-based SPH (**K-SPH**) $O(p \times n^2)$ performance: 2-2/p

Applicazione di queste tecniche agli attuali protocolli di Multicast Routing:

MOSPF: *Multicast Open Shortest Path First protocol*
Il protocollo MOSPF si basa sul protocollo di routing OSPF. Tutti i routers hanno, non solo, informazioni sulla completa topologia della rete (già con l'OSPF), ma conoscono i link che, dai router, vanno ad host interessati ai vari gruppi di multicast. Con queste informazioni è possibile costruire degli shortest path tree per ogni diverso gruppo di multicast.

PIM: *Protocol Independent Multicast*
Il protocollo PIM permette due differenti scenari di distribuzione di nodi di multicast: **dense mode** (i membri del multicast group sono densamente localizzati: alto rapporto tra nodi di multicast e nodi totali della rete), e **sparse mode** (i membri del gruppo sono dispersi: il numero dei router di multicast è molto piccolo rispetto la totalità dei nodi).
PIM Dense Mode utilizza una tecnica di *flood-and-prune reverse path forwarding.*
PIM Sparse Mode utilizza invece un approccio *center-based*.

Applicazioni Multimediali su rete

Dati Multimediali

Su Internet sono presenti 3 tipi di trasmissione dati:

1. flussi non continui	es. pagine Web
2. flussi continui non interattivi	es. visualizzazione film online
3. flussi continui interattivi	es. telefonata o videoconferenza

La rete può:

- *portare ritardi variabili*

  Ciò comporta problemi nella trasmissione di flussi continui (ritardi costanti possono essere sopportati nelle trasmissioni non interattive) → Per le trasmissioni multimediali non è quindi possibile l'utilizzo del TCP (megli perdere qualche pacchetto che generare nuovi ritardi (e molto variabili).

  soluzione possibile:

  E' possibile "nascondere" la variabilità del ritardo, se risulta noto un valore di ritardo massimo, con l'utilizzo di un buffer. Il buffer avrà dimensione tale da poter contenere tutti i dati che possono essere ricevuti in un tempo pari al ritardo massimo.

  Nel buffer si raccolgono i pacchetti, eventualmente si risenquenziano (se giunti in ordine errato), in modo da mantenere un ritardo costante (pari al massimo possibile).

- *perdere pacchetti*

  La frequenza di perdita dei pacchetti può essere diminuita utilizzando un metodo particolare di trasmissione:

  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ⟶ 1 5 9 13

  La perdita di pacchetti può essere gestita in modi diversi:

  a. ignorando il problema
  b. effettuando una ricostruzione dei pacchetti persi (mediante l'utilizzo di informazioni aggiuntive da trasmettere nei pacchetti) → è infatti impensabile una richiesta di ritrasmissione

Sono necessari, per il trattamento di dati multimediali,: - buffer
- numeri di sequenza nei pacchetti
- time stamp ← per avere una misura del ritardo

Qualità

La rete Internet è di tipo **best effort**: la rete mette a disposizione le sue risorse, nessuna garanzia viene però data alla qualità del servizio (ad es. i messaggi vengono instradati più velocemente possibile ma non c'è comunque garanzia sui tempi di ricezione)

Il problema può risolversi con l'introduzione di qualità del servizio stabilita a priori e garantita.

QoS (Quality of Service)

Le basi sono:
- la rete deve essere in grado di classificare i pacchetti
- devono pensarsi politiche di scheduling dei pacchetti adatte (in base al livello di qualità)
- le risorse della rete devono comunque essere utilizzate al massimo
- è necessaria l'introduzione di una *call admission* (richiesta di accesso alla qualità desiderata)

Servizi Integrati

Per Servizi Integrati su Internet si intendono tutti quei servizi (generalmente multimediali e di multicast) per i quali è necessaria una garanzia di qualità sull'intero percorso sender→routers→receivers.

caratteristiche:

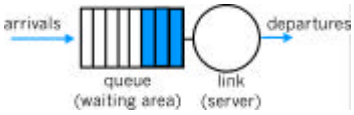
1. **Reserved Resources.** Le risorse (ampiezza di banda, buffers,…) sui routers vengono riservate ai vari flussi di dati (in genere previa richiesta da parte del receiver) per una sessione.
2. **Call Setup**. Una richiesta di sessione con determinati livelli di qualità (QoS contrattata) può essere accettata solo se è possibile garantire la qualità (ci si riferisce essenzialmente all'ampiezza di banda) richiesta, cioè se è possibile riservare le risorse necessarie lungo l'intero percorso.

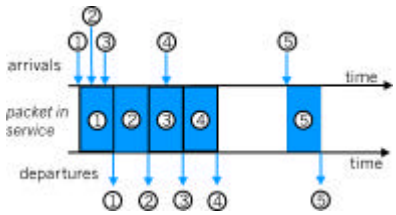
Scheduling

Politiche di scheduling nei routers per garantire la qualità del servizio

Esistono essenzialmente due tipi di scheduling per il trattamento di dati multimediali sui routers:

1. FCFS: è la politica più semplice (basso carico computazionale sul router); I flussi vengono trattati in modo uniforme.

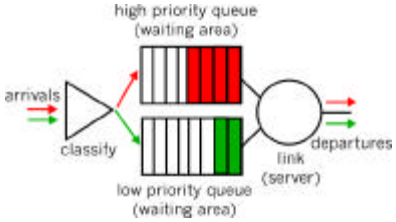


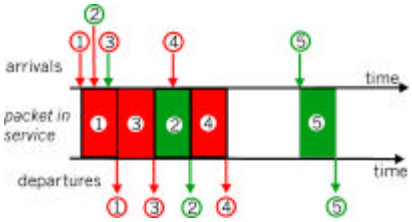


2. con priorità: permette di associare diversi livelli di priorità a diversi tipi di flussi di dati.

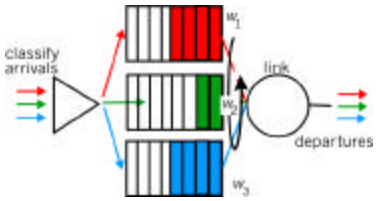
   Le politiche di scheduling, tra le diverse code di priorità, possono essere:

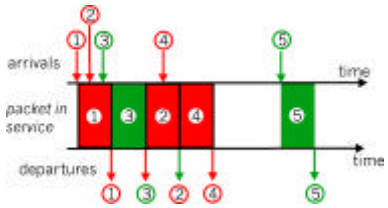
   a. Finché vi sono pacchetti nella coda a priorità più alta, non possono essere serviti pacchetti inseriti in altre code → possibile problema di *starvation* per i pacchetti nelle code a priorità inferiore





   b. WFQ (*round robin pesato*). Ad ogni ciclo vengono serviti pacchetti che stanno su tutte le code (il numero dei pacchetti esaminati per ogni coda, sarà proporzionale al suo livello di priorità).





   Ad ogni classe di priorità *i*, si assegna un peso w_i. Durante un ciclo WFQ, ai pacchetti della classe *i* si garantisce una frazione di servizio (che coinciderà con una frazione di ampiezza di banda) pari a $w_i/(\Sigma w_j)$, proporzionale al peso (la somma al denominatore è relativa ai pesi di tutte le classi di priorità). Per un link con *transmission rate* R, la classe *i* avrà un throughput di almeno $R \cdot w_i/(\Sigma w_j)$.

   Lo scheduling previsto è comunque **non-preemptive**.

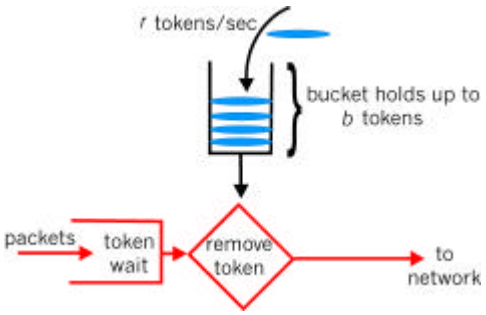
Policing

Politiche di policing nei routers, per garantire il rispetto dei limiti imposti dalla qualità del servizio concordato (mediante monitoraggio del traffico dei flussi).

Parametri di valutazione del flusso:

- averange rate: bit rate medio) ← difficile da effettuare e comunque poco significativo (in una trasmissione come quella di Internet, prevalentemente orientata ai burst)
- peak rate: bit rate misurato in un intervallo di tempo molto piccolo, dell'ordine dei secondi (o ms)
- burst size: misura dell'ampiezza massima di una raffica (burst: sequenza di pacchetti consecutivi sul link)

Meccanismo del **Leaky-bucket**



Permette la produzione (siamo sull'host sorgente) di flusso in maniera aderente alle specifiche di qualità concordate.

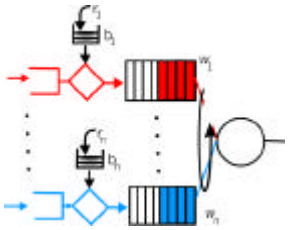
Il leaky-bucket (*secchio bucato*) si riempie a velocità costante, pari al pick rate che si deve mantenere nella trasmissione, ed ha dimensione pari al burst size massimo consentito.

Per ogni pacchetto che deve essere inviato, si toglie un token dal leaky bucket (altrimenti i token si accumulano). Se dovesse svuotarsi la leaky-bucket, non possono più inviarsi pacchetti (limite massimo consentito).

Stesso meccanismo (del leaky bucket) può essere utilizzato per il controllo (quindi sui routers, non sugli host) del mantenimento dei livelli di qualità prefissati.

Quando i pacchetti giungono in coda e la pila di token è vuota, allora i pacchetti vengono scartati (la pila si riempie sempre alla velocità costante pari al pick rate massimo consentito).

E' ovviamente possibile considerare livelli di priorità differenti (ed utilizzare strategie di scheduling come quelle viste).



RTP **Real Time Protocol**

Protocollo basato sullo UDP. Si deve collocare, quindi, al livello applicativo — ciascuna applicazione deve gestire i pacchetti RTP (mediante incapsulamento), con relativi problemi di incompatibilità tra applicazioni differenti.

Al protocollo RTP si associa, specie nelle trasmissioni multimediali in multicast, il protocollo **RTCP** (Real Time Control Protocol).

header RTP:

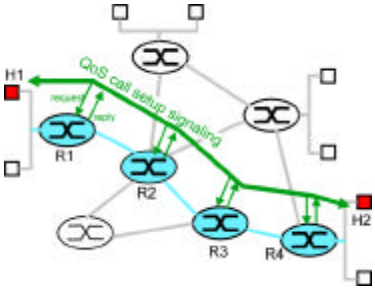
- Payload type — Indica il tipo di dati trasportati (contenuti nel campo Data): voce, video, immagini,…
- Sequence number — Numero di sequenza del pacchetto
- Time stamp — Indica l'istante (in ms) in cui è creato il dato → permette di scartare i pacchetti che giungono con ritardi superiori all'eventuale massimo consentito (concordato tra sender e receiver)
- Synchronization source ID — Identifica il flusso di dati (è possibile anche il multiplexing dei dati tra source e receiver)
- Miscellaneus field

RSVP **Resource reSerVation Protocol** (RFC 2205)

Il protocollo RSVP permette alle applicazioni di riservare, sulla rete, ampiezza di banda per i propri flussi di dati.
L'RSVP è usato dagli host per richiedere una certa larghezza di banda, e dai routers per inoltrare tale richiesta e riservare effettivamente la banda (per implementare l'RSVP, il software deve essere attivo nei senders, receivers e routers).

caratteristiche:

E' protocollo di **signaling** — ricerca la disponibilità delle risorse (ampiezza di banda) da allocare per alberi di multicast (unicast può vedersi come caso particolare)



E' **receiver-oriented** — viene guidato dal ricevente (si procede sul percorso inverso fino al source)

E' **soft-state** — i messaggi di prenotazione delle risorse hanno una validità temporale molto limitata, da ciò:

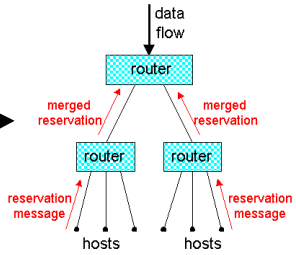
a. è necessario rinnovare la richiesta frequentemente
b. non è consentito prerilascio della risorsa

NB: Un esempio di protocollo hard-state è il TCP (data la complessa procedura di fine connessione)

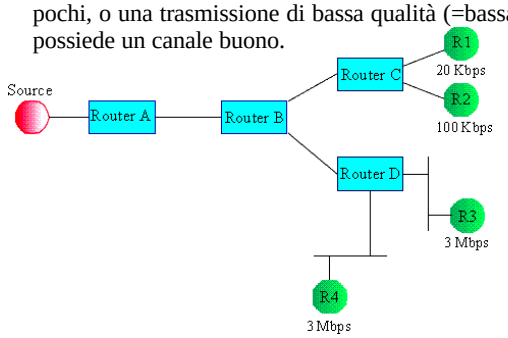
Implica **call admission** — richiesta di prenotazione delle risorse (dal potenziale ricevente alla sorgente)

osservazioni:

1. I pacchetti RSVP seguono le normali tabelle di routing IP. Il "circuito" sorgente-destinatario che si viene a creare, può considerarsi **dedicato** (nel quanto di tempo di validità della prenotazione delle risorse).
2. I routers possono effettuare un **merge** tra richieste per uno stesso flusso →



3. Problema dell'**eterogeneità dei riceventi**:
I receiver di una sessione di multicast potrebbero avere caratteristiche di link (ampiezza di banda della propria connessione con il primo router) molto diverse.
Il sender si troverebbe nella situazione di poter effettuare una trasmissione di alta qualità, che però è accessibile a pochi, o una trasmissione di bassa qualità (=bassa occupazione di banda), accessibile a tutti ma non accettabile da chi possiede un canale buono.



Per risolvere tale problema, si può pensare di codificare video ed audio (flussi continui interattivi) a diversi livelli sovrapposti (similmente ad un certo tipo di codifica di immagini a livelli di grigio); quando l'ampiezza di banda non consente la trasmissione del flusso a qualità massima, si procede con l'eliminazione dei livelli qualitativi superiori (con un deterioramento inizialmente quasi trascurabile) fino a raggiungere l'ampiezza consentita.

Modello di Rete Sincrona

Introduzione

Rappresentazione mediante un grafo orientato: G=(V,E), V: nodi; n=|V|; E: archi
M= insieme dei possibili messaggi

Definizioni:
- in_nbrs $_i$: insieme dei nodi adiacenti, con link in input (verso i);
- out_nbrs $_i$: insieme dei nodi adiacenti, con link in out (da i);
- distance (*i*,*j*): distanza (in hop) sul cammino minimo i→j
- diametro: valore massimo tra le distanze tra nodi di un grafo

a. arco=**canale** (può contenere solo un messaggio per round)
b. nodo=**processo** (un solo processo per nodo)

- processo identificato da:
  - insieme di stati (anche infinito): states $_i$
  - insieme di stati di partenza (non vuoto): start $_i$
  - funzione di generazione dei messaggi: msg $_i$
    (msg: states X out_nbrs → M U {null})
  - funzione di transizione di stato: trans $_i$
    (trans: states X $[M]_{1..in\_nbrs}$→states) [ ]:vettore colonna
- i processi applicano ciclicamente i passi (ad ogni *round*):
  1. applicazione msg $_i$: pongono i messaggi sui canali
  2. applicazione trans $_i$: prelevano (cancellandoli) messaggi dai canali
- Terminazione: non ci sono stati finali nei sistemi distribuiti
  I sistemi terminano indipendentemente l'uno dagli altri, finendo in uno stato di *halt* (situazione stabile)
- Fallimenti possibili: divisi in due categorie: *process failure*, *link failure*
- Esecuzione: Definita come la sequenza $C_0\ M_1N_1C_1\ M_2N_2C_2$ …
  - C_r ← assegnazione dello stato ad ogni processo, al round r
  - M_r ← assegnazione dei messaggi in uscita, al round r
  - N_r ← assegnazione dei messaggi in ingresso, al round r

  Due esecuzioni si dicono indistinguibili al processo i, se hanno le stesse sequenze di messaggi (in ingresso ed in uscita).

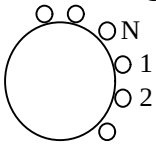
  Per comportamenti *non deterministici*, è necessario introdurre una componente di casualità: funzione rand $_i$ (S): stato → stato
  L'esecuzione diverrebbe: $C_0\ D_1M_1N_1C_1\ D_2M_2N_2C_2$ …
  - D_r ← nuova assegnazione degli stati, ottenuta dall'assegnazione precedente (C_{r-1}), mediante l'applicazione della rand $_i$.

Metodi di prova:
a. metodo delle *invarianti*: dimostrazione per induzione
b. metodo della *simulazione*: si sfruttano sistemi analoghi a quello da studiare

Misure di complessità:
a. *temporale*: numero di round necessari per giungere dallo stato iniziale alla terminazione
b. *di comunicazione*: numero di messaggi non nulli che è necessario inviare

La complessità temporale è quella più importante; quella di comunicazione si considera solo se interviene il problema della congestione

Rete sincrona ad anello: *ring*



- Sistema con N processi (nodi)
- Il singolo nodo non ha conoscenza della rete se non nei suoi vicini
- La numerazione dei nodi è considerata mod N (nodo n+1= nodo 1)

Problema **ELEZIONE DI UN LEADER**

Algoritmi:

1. *Ipotesi*: I processi sono identici (secondo la definizione data) → nessuna soluzione possibile!
2. Algoritmi basati sui confronti

Ipotesi: I processi si differenziano in base ad uno UID∈N+

LCR a propagazione del token circolare

Ogni processo manda il suo UID lungo l'anello. Quando un processo viene raggiunto da un UID, lo confronta con il proprio: se è maggiore, viene inoltrato questo; se è minore, non viene effettuata alcuna operazione; se è uguale, il processo si dichiara leader!
(se mi giunge il mio UID, significa che questo ha fatto tutto il giro dell'anello senza subire modifica)

Formalizzazione:

- insieme dei messaggi: M={UID}U{null}
- Variabili di stato:
  - U ∈ {UID}, init: UID del processo
  - *send*={UID}|{null} init: UID del processo
  - *status*={unknown, leader} init: "unknown"
- funzione generazione dei messaggi:
  - send the current value of *send* → processo *i*+1
- funzione di transizione di stato:

```
send←null
if incoming(message)=v then                v is UID
        case
                v>U : send←v
                v=U : status←leader!
                V<U : --
        end case
```

Dimostrazione formale della correttezza:

Con il processo i_{max} intendiamo quello con UID più alto.

lemma1: Il processo i_{max} produce la risposta "leader" dopo N rounds.
Quindi, dopo N passi: stato i_{max} = leader!
Dimostriamo quindi (per induzione) che:
$\forall r: 0 \le r \le N-1$, dopo r rounds: send $i_{max+r}=u_{max}$

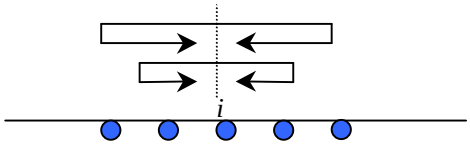
lemma 2: Nessun processo i, $i \neq i_{max}$ produce status={leader}
Si tratta quindi di dimostrare che:
$\forall r, \forall i,j$, dopo r rounds, if $i \neq i_{max}$, $j \in [i_{max}, i] \rightarrow send_j \neq u_i$
(l'UID di i viaggia in senso orario, quando giunge ad i_{max} viene tolto dall'anello (è i_{max} il leader!))

Terminazione: Il leader eletto manda un messaggio del tipo "leader=UID";
il singolo processo inoltra il messaggio (in avanti) e si blocca.

Complessità temporale: N passi (rounds) (2N nel caso di conferma per la terminazione)
Complessità di comunicazione: N^2 (ad ogni round vi sono N messaggi sull'anello)

HS a diffusione di token rientranti

Ogni processo opera in diverse fasi. Nella fase l, il processo i invia il proprio UID in entrambe le direzioni (fino ad una distanza massima pari a 2^l, poi tornano indietro). Se entrambi i token tornano indietro invariati (il rewind avviene senza controlli), il processo passa alla fase successiva (l+1).



Ogni processo confronta gli UID dei token che li attraversano, con il proprio UID. Il token vienelasciato proseguire solo se superiore al proprio UID (altrimenti viene scartato).
Se gli UID coincidono, il processo si elegge leader!

Formalizzazione:

- insieme dei messaggi: M={UID}U{hop counter}U{in|out}
- Variabili di stato:

U ∈ {UID},		init: UID del processo
send+={UID}|{null}	verso destra,	init: UID, in, 1
send- ={UID}|{null}	verso sinistra,	init: UID, out, 1
status={unknown, leader}		init: "unknown"
phase∈N+		init: 0

- funzione generazione dei messaggi:
  send the current value of *send* + → processo *i*+1
  send the current value of *send* - → processo *i* -1
- funzione di transizione di stato:

```
send+ ←null
send - ←null
il (msg da i-1)=(v,out,h) then
        case
                v>u & h>1: send+ ←(v,out,h-1)      --inoltra
                v>u & h=1: send - ←(v,in,1)
                v=u: status←leader                 --rispedisce indietro un token che è giunto a "fine onda" (in rewind hop non decrementato)
        end case
il (msg da i+1)=(v,out,h) then
        case
                v>u & h>1: send - ←(v,out,h-1)     --inoltra
                v>u & h=1: send+ ←(v,in,1)         --fine onda
                v=u: status←leader
        end case
if (msg from i-1)=(v,in,1) & (v≠u) then send+ ←(v,in,1)        --rewind
if (msg from i+1)=(v,in,1) & (v≠u) then send- ←(v,in,1)        --rewind
if (msg from i-1)=(msg from i+1)=(u,in,1) then                 --nuova fase!
        { phase++;      send+ ←(u,out,2^phase);   send - ←(u,out,2^phase)
```

Complessità temporale: Si dimostra essere 3N (per N potenza di 2) o 5N (else); quindi, comunque: O(N)

Complessità di comunicazione:

phase=0 — 4 messaggi per ogni processo (2 in uscita, 2 in ingresso), quindi 4N messaggi

phase=l:
- Inviano messaggi quelli che nella fase precedente (l-1) non hanno ricevuto (indietro) entrambi i propri token: $n / (2^{l-1}+1)$ -- ogni $2^{l-1}+1$ (=raggio onda+1(mittente)) processi c'è solo un vincitore (→ che passa alla fase successiva: l-1→l)
- numero messaggi per processo: $4\ 2^l$

quindi i messaggi nella fase l sono: $4\ (2^l\ (N/\ 2^{l-1}+1)\ ) \leq 8N$

Concludendo: in totale si hanno [1+log N] fasi,
--la fase massima ha dimesione N, le fasi crescono esponenzialmente:
$N=2^l$ → $l \approx \log_2 N$

quindi i messaggi sono al più: $8N[1+\log N] \propto o(N \log N)$

3. Algoritmi non basati sui confronti:

Time Slice

Le fasi si ripetono continuamente, numerate in modo consecutivo: 0,1,2,…
I processi stanno "in silenzio" finchè il numero di fase coincide col proprio UID (=leader!).
NB: Tutti i processi devono conoscere il numero N (ring statico!), per attendere la fine di ogni fase.

Complessità temporale: $N * UID_{min}$ (non prevedibile a priori)
Complessità di comunicazione: N --solo quello inviato dal leader per notificare l'elezione

Algoritmo a velocità variabile

Ogni processo invia un token con il proprio UID attraverso l'anello. Diversi token viaggiano a velocità differenti → un token contenente l'UID v attraversa l'anello alla velocità di un messaggio ogni 2^v rounds (cioè il processo intermedio lo inoltra 2^v rounds dopo averlo ricevuto).

Complessità temporale: $N * 2^{UIDmin}$ (non prevedibile a priori)
Complessità di comunicazione: $(<2N) \propto o(N)$

Rete sincrona qualsiasi

Problema: **ELEZIONE DI UN LEADER**

Algoritmo:

Flood Max (massima inondazione)

Ogni processo mantiene, in ogni istante, il massimo UID che lo ha attraversato. Ad ogni round il processo trasmette su tutti i link in uscita il massimo UID al momento riscontrato (inizialmente il proprio).

NB: L'algoritmo LCR (relativo all'anello) è un caso particolare di questo

Ipotesi: Nell'algoritmo ogni processo deve conoscere il diametro della rete.

Formalizzazione:

- insieme dei messaggi: M={UID}
- Variabili di stato:
  - U ∈ {UID}, init: UID del processo
  - maxUID ∈ {UID} init: UID del processo
  - *status*={unknown, not-leader, leader} init: "unknown"
  - *round* ∈N+ init: 0
- funzione generazione dei messaggi:
  - if round<diam then send(maxUID) → out_nbrs
- funzione di transizione di stato:
  - round++
  - U←{UID from in_nbrs}
  - maxUID←max (maxUID, U)
  - if round=diam then
    - if maxUID=U then status="leader"
      else status="not-leader"

Dimostrazione formale della correttezza:

Il processo i_{max} sarà leader e tutti gli altri processi saranno not-leader, in un numero di rounds pari al diametro della rete.

Asserzione: ∀r: 0≤r≤diam, ∀j, dopo r rounds,
if dist (i_{max} , j)≤r then $maxUID_j=U_{max}$

dimostrabile per induzione.

Terminazione: L'algoritmo ha termine dopo un numero di round pari al diametro della rete.

Complessità temporale: diam passi (rounds) (2N nel caso di conferma per la terminazione)

Complessità di comunicazione: diam x n°archi (in ogni round, c'è un messaggio su ogni link)

variante ottimizzata

Non trasmette un UIDmax se questo rimane (nel round in esame) uguale a quello inviato al round precedente (basta inserire una variabile boolean che tiene conto di questo apetto).

Dimostrazione formale della correttezza:

Dimostrazione effettuabile per simulazione, partendo dall'algoritmo nella sua versione base (le due versioni hanno gli stessi stati). Una dimostrazione rigorosa risulterebbe troppo complessa.

problema: **RICERCA**: raggiungimento di tutti i processi (nodi del grafo)
Ipotesi: Ogni processo non ha alcuna informazione sulla rete.

Algoritmo:

BFS: Bradth-First Search (ricerca in ampiezza in una rete sincrona)

Necessario marcare i singoli nodi (processi), quando questi vengono "raggiunti" dal messaggio di *search*. Si definisce "padre" di un processo quello che lo raggiunge per primo con un messaggio di *search*. Se il processo riceve più messaggi di search in un solo round, sceglie (arbitrariamente) il padre.
Una volta marcato, il processo invia a sua volta i messaggi di *search* (flooding).

Correttezza: Asserzione: Dopo r rounds, ogni processo p_i: dist $(i_0,p_i)=d$ $1 \le d \le r$ ha il "parent" definito (è stato raggiunto!).
dimostrabile per induzione.

Complessità temporale: O(diam)

Complessità di comunicazione: O(|E|) --nell'ordine del numero di archi della rete

Per un processo i, un "figlio" è tale solo se da esso giunge una conferma di elezione (necessario quindi un messaggio che confermi l'elezione di un processo a "padre").
Questo è realizzabile molto semplicemente in una *rete simmetrica*.

Qualora la rete fosse *asimmetrica*, sarebbe necessaria la costruzione di un BFS da ogni nodo "figlio" per la notifica (al padre) dell'elezione a padre.
In questo caso, si avrebbero:

Complessità temporale: resta O(diam)

Complessità di comunicazione: O (diam x |E|)

miglioramenti: - un processo evita di inviare il messaggio di search al padre (ed agli event. padri potenziali).
- broadcasting: Quando si vuole costruire l'albero BFS per usarlo più di una volta è necessario mantenere informazioni (inerenti ai figli: *chindren ponters*) residenti sui vari processi.

Terminazione: Necessaria una fase di **converge-cast** (approccio bottom-up: dalle foglie alla radice) per la notifica del raggiungimento di tutti i nodi.

Applicazioni possibili:
- broadcast;
- elezione del leader;
-calcoli globali sulla rete (solo operazioni associative e commutative)
- esplorazione rete, misura diametro, conteggio nodi,…

problema: **SHORTEST PATH**
Ipotesi: Ogni processo non ha alcuna informazione sulla rete.

Algoritmo:

Bellmann-Ford ← in realtà l'algoritmo migliore è quello di *Dijakstra* (che però è fortemente centralizzato).

Agli archi del grafo è associato un peso: arco(i,j,ω)
Fa parte degli algoritmi dinamici.

Assumiamo che ciascun processo conosca i pesi di tutti gli archi adiacenti ed il numero di nodi del grafo. Ogni processo deve conoscere il proprio *padre* (nodo nel cammino minimo verso la radice) ed la distanza di questo dalla radice.
Ogni nodo mantiene un proprio **vettore delle distanze** (una entry per ogni nodo della rete):

Il generico elemento di questo vettore è il record:

distanza da *i*	link per il cammino min. verso *i*

Tale vettore viene inizializzato con le distanze dai nodi adiacenti, per le altre entry si considera distanza ∞. Ad ogni passo il processo invia il suo vettore delle distanze ai propri vicini, la tabella viene aggiornata in base alle nuove informazioni.

Il numero dei rounds perché l'algoritmo converga è n-1: tutti i nodi devono trasmettere tutte le informazioni.

Asserzione: Dopo r rounds, ciascun processo P_i ha le variabili distanza e parent che corrispondono al cammino minimo tra i cammini minimi da i a j che abbiano un numero di salti al più r (se ∃ un cammino minimo tra i e j, che necessita di un numero di passi superiore ad r, non può essere individuato in r round).

Applicazioni possibili: - usato all'interno del RIP quando il numero dei nodi su Internet era più limitato

problema: **MINIMUM SPANNING TREE**

Minimizza il costo complessivo (=somma di tutti i costi degli archi) dell'albero di copertura.
Proprietà: Il MST risulta indipendente dalla radice scelta (→ si ottiene comunque lo stesso albero).

Ipotesi: { - grafo simmetrico
- ogni processo conosce il numero (N) dei nodi del grafo

Correttezza:

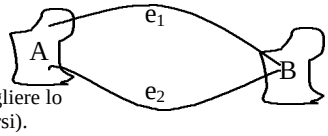
lemma: Sia G=(V,E) un grafo orientato pesato e sia $\{(V_i,E_i): 1 \le i \le k\}$ una foresta per G, con k>1;
Fissato $i \in [1..k]$, consideriamo !un arco con il più piccolo peso tra gli {e': e' ha !un estremo in V_i} ← Ovvero*: per ogni nodo del generico albero della foresta, scegliamo l'arco di peso minimo tra quelli che si dipartono (dall'albero stesso) verso nodi non appartenenti all'albero.*

allora:
∃ uno spanning tree per G che include V_i,E_i ed e.
Tale albero è quello di peso minimo tra gli spanning tree per G che includono la foresta V_i,E_i.

Strategia di risoluzione:
inizialmente gli alberi della foresta sono formati da un solo nodo (quindi all'inizio vi sono N alberelli di un nodo ciascuno).
Per ciascun nodo di ogni albero si ricerca l'arco di peso più piccolo uscente dall'albero stesso (=**MWOE**: minimum weight outgoing edge). I due alberi si collegano.
Questo passo avviene "in parallelo" in tutti gli alberi → dopo ogni passo, gli alberi della foresta si riducono (in numero) *almeno* alla metà (fondendosi!).

Problemi della versione distribuita:
- all'unificazione di due alberi è necessario scegliere il "nome" identificativo del nuovo albero → necessaria elezione leader
- possibile creazione di cicli per l'aleatorietà data nella scelta dell'arco di "unione" tra archi di peso uguale:

A potrebbe scegliere e_1,
B scegliere e_2
→ per evitare un ciclo, A e B dovrebbero scegliere lo stesso arco (possibile se i pesi sono tutti diversi).



Algoritmo:

GHS: Si svolge in livelli (o passate) composti da diversi rounds;
All'inizio vi sono N alberelli di un solo nodo. Al generico livello k vi saranno diversi alberi (grumi), all'interno di questi c'è un nodo leader. Ogni nodo del grumo conosce l'UID del suo nodo leader (quindi conosce il suo grumo di appartenenza e conosce quali dei suoi link puntano a nodi dello stesso grumo e quali puntano all'esterno dello stesso).
Ciascun grumo cerca il suo arco (verso l'esterno del grumo stesso) di peso minimo (**MWOE**):
Il leader manda in broadcast (sul grumo) una richiesta di ricerca del MWOE. Ogni processo, alla ricezione del messaggio, individua il *suo* arco di peso minimo in uscita dal grumo e lo invia (converge cast) verso il leader.
Il leader (al termine del converge cast) conosce l'MWOE.
Avvenuta la "fusione" dei diversi grumi, si procede alla rielezione del leader all'interno del nuovo grumo.

Complessità temporale: Numero grumi in ciascun livello: 2^k → numero di livelli al piu $\log_2 n$; Ogni livello ha complessità n (per il broadcasting), quindi: $O(n \log_2 n)$

Complessità di comunicazione: Per ogni livello: $n + |E|$ messaggi → $(n+|E|)\log_2 n$
|E|: un messaggio per ogni arco
n: dovuto alla ricerca del MWOE

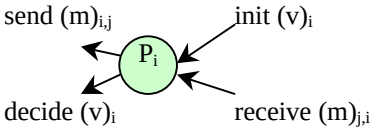
Modello di Rete Asincrona

Introduzione

Rappresentazione mediante un automa.
Le transizioni di stato sono chiamate **azioni** e si distinguono in:
- *di ingresso* (non gestibili dall'automa)
- *di uscita*
- *interne* (similmente al concetto di stato)

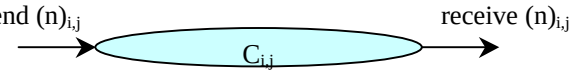
Le azioni di input sono sempre permesse in ogni stato ← l'automa è detto **input enable**
Si suddivide in due entità:

1. **Processo**



v: valore
m: messaggio
i,j: source/destination

2. **Canale**



Formalismo:

signature S dell'automa: descrizione di tutte le azioni che possono essere compiute dall'automa
S=(in(S), out(S), int(S)), indicando rispettivamente le azioni di ingresso, di uscita ed interne di S.

definizioni:
- azioni esterne: ext(S)=in(S) U out(S)
- azioni localmente controllate: local(S)=int(S) U out(S)
- insieme delle azioni in S: acts(S)= in(S) U out(S) U int(S)
- external signature: S= (in(S), out(S), ϕ) ←int(S)=ϕ

definizione dell'automa: l'automa viene definito da:
- signature S=sig(a)
- insieme degli stati states(A)
- stati iniziali start(A)
- insieme delle transizioni trans(A)
  → tripla formata da: (stato sorgente, azione, stato destinazione), quindi trans(A) ⊆ states(A) x acts(sig(A)) x states(A)
- sequenze delle azioni nei processi task(A)

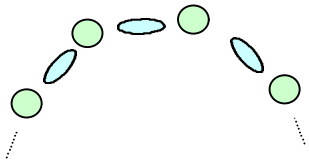
← uno stato si dice **quiescente** se ha solo azioni di input (es. server in ascolto)

Utilizzando il formalismo precedentemente descritto, diamo una definizione di:

Canale di I/O:
- sig (C_{ij}): input: $send(m)_{i,j}$ m∈M; output: $receive(m)_{i,j}$ m∈M; internal: ϕ → quindi il canale è un *external signature*
- states (C_{ij}): consideriamo le variabili: -coda messaggi∈M (inizialmente ϕ)
- trans (C_{ij}): $send(m)_{ij}$ *precondizione*: -- ; *effetto*: add m in queue; $receive(m)_{ij}$ *precondizione*: m is first in queue; *effetto*: remove m from queue
- task (C_{ij}): L'azione che compie C_{ij} è consegnare messaggi: $receive(m)_{ij}$, m∈M

Processo *i*-mo:
- sig (P_i): input: $init(v)_i$ v∈V; $receive(v)_{j,i}$ " 1≤j≤N, j≠i; output: $decide(v)_i$ "; $send(v)_{i,j}$ " 1≤j≤N, j≠i; internal: ϕ → quindi il processo è un *external signature*
- states (P_i): consideriamo il vettore: -val [1..N] val [*i*]∈V U {null}, inizialmente tutti null
- trans (P_i): $init(v)_i$ *precondizione*: -- ; *effetto*: val[i]=v; $send(v)_{ij}$ *precondizione*: val[i]=v; *effetto*: -- ; $receive(v)_{j,i}$ *precondizione*: -- ; *effetto*: val[j]=v; $decide(v)_i$ *precondizione*: val[j]≠null, ∀j=1→N; *effetto*: v=*f* (val[1..N])
- task (P_i): for j≠i do { $send(v)_{i,j}$ v∈V } $decide(v)_i$ v∈V

Rete asincrona ad anello: *ring*



- Sistema con N processi (nodi)
- Il singolo nodo non ha conoscenza della rete se non nei suoi vicini
- La numerazione dei nodi è considerata mod N (nodo n+1= nodo 1)
- Si assume il canale come coda FIFO

Problema **ELEZIONE DI UN LEADER**

Algoritmi:

LCR a propagazione del token circolare

Identico alla versione sincrona, basta considerare un buffer su ogni nodo (processo).

Formalizzazione:

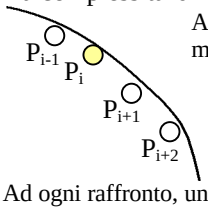
- signature:
  - input: $receive(v)_{i-1,i}$ v is UID
  - output: $send(v)_{i,i+1}$ v is UID
  - $leader_i$ ← flag di identificazione leader
- variabili di stato:
  - $U \in \{UID\}$, init: UID del processo
  - *send*= coda FIFO di UID init: send contiene solo UID_i
  - *status*={unknown, chosen, reported} init: "unknown"
- transizioni:
  - $send(v)_{i,i+1}$ *precondizione*: v first in queue *send*
    effetto: remove v from *send*
  - $receive(v)_{i-1,i}$ *precondizione*: ---
    effetto: case
      v>*u* add v to queue *send*
      v=*u* *status* ← chosen
    end case
  - $leader_i$ *precondizione*: *status*=chosen
    effetto: *status* ← reported
- task: $send(v)_{i,i+1}$ v is UID
  $leader_i$

HS a diffusione di token rientranti

Praticamente identico al modello sincrono.
Complessità di comunicazione o(nlogn), come nella versione sincrona, con link bidirezionali

Algoritmo con espansioni unidirezionali di Peterson

Ha complessità di comunicazione $o(n\log n)$, come l'algoritmo HS, ma utilizza link unidirezionali.



Ad ogni passo, ogni P_i invia il proprio UID ai due processi che lo seguono (P_{i+1}, P_{i+2}). Allo stesso modo riceve due UID (rispettivamente dei processi P_{i-1} e P_{i-2}), che confronta con il proprio:

- se l'UID più alto è il proprio oppure quello del processo P_{i-2} diviene un nodo "trasparente" (si occupa solo dell'inoltro degli UID, senza confronti)
- se l'UID più alto è quello del processo P_{i-1}, P_i copia, come proprio UID, quello di P_{i-1} e rimane "attivo".

Ad ogni raffronto, un processo diviene "trasparente" (relay) → il numero di processi si dimezza continuamente.

Non necessariamente il leader è quello con l'UID massimo. Viene eletto il processo che riceve in ingresso un UID coincidente con quello che ha memorizzato.

Formalizzazione:

- signature:
  - input: $receive(v)_{i-1,i}$ — v is UID
  - output: $send(v)_{i,i+1}$ — v is UID
  - output: $leader_i$ ← flag di identificazione leader
  - internal: $get\text{-}secondUID_i$ ← legge UID_{i-1}
  - internal: $get\text{-}thirdUID$ ← legge UID_{i-2}
  - internal: $advance\text{-}phase_i$
  - internal: become relay
  - internal: $relay_i$
- variabili di stato:
  - *mode* ∈ {active,relay} — init: "active"
  - *status* ∈ {unknown, chosen, reported} — init: "unknown"
  - *uid*[1..3], uid[*i*] ∈ {UID}∪{null} — init: [null,null,null]
  - *send* coda FIFO d'uscita — init: proprio UID
  - *receive* coda FIFO in ingresso — init: ϕ
- transizioni:
  - getsecondUID$_i$
    - *precondizioni*: *mode*=active; *receive* is not empty; *uid*[2]=null
    - *effetto*: *uid*[2] ← first from *receive*; remove first from *receive*; add *uid*[2] to *send*; if *uid*[2]=*uid*[1] then *status* ← chosen
  - getthirdUID$_i$
    - *precondizioni*: *mode*=active; *receive* is not empty; *uid*[2]≠null, *uid*[3]=null
    - *effetto*: *uid*[3] ← first from *receive*; remove first from *receive*
  - advance phase$_i$
    - *precondizioni*: *mode*=active; *uid*[3]≠null ← ricevuti entrambi gli UID; *uid*[2]>*uid*[1],*uid*[3]
    - *effetto*: *uid*[1] ← *uid*[2]; *uid*[2],*uid*[3] ← null; add *uid*[1] to *send*
  - become relay$_i$
    - *precondizioni*: *mode*=active; *uid*[3]≠null ← ricevuti entrambi gli UID; *uid*[2]≤*uid*[1],*uid*[3]
    - *effetto*: *mode* ← relay
  - relay$_i$
    - *precondizioni*: *mode*=relay; *receive* is not empty
    - *effetto*: move first from *receive* to *send*
  - leader$_i$
    - *precondizione*: *status*=chosen
    - *effetto*: *status* ← reported
  - send(v)$_{i,\,i+1}$
    - *precondizione*: v first in queue *send*
    - *effetto*: remove v from *send*
  - receive(v)$_{i-1,i}$
    - *precondizione*: ---
    - *effetto*: add v to *receive*
- task:
  - $send(v)_{i,i+1}$ v is UID
  - $get\_secondUID_i$, $get\_thirdUID_i$
  - $advance\_phase_i$ ⇔ $became\_relay_i$, $relay_i$
  - $leader_i$

Rete asincrona qualsiasi

Problema: **ELEZIONE DI UN LEADER**

Algoritmi:

Flood Max (massima inondazione)

L'algoritmo di *flooding* è fondamentalmente sincrono (si basa sul numero di rounds). Si può simulare un asincronismo effettuando un conteggio del numero di nodi attraversati (hop), sostituendo tale misura ai rounds.
La condizione di terminazione sarà quindi: numero di hop = diametro della rete.

Ipotesi: Nell'algoritmo ogni processo deve conoscere il diametro della rete (come nella verione sincrona)

variante ottimizzata

La variante ottimizzata di questo algoritmo, basata, nella versione sincrona, sulla riduzione del numero dei messaggi (messaggi inoltrati solo se contenenti informazioni nuove), non può essere riportato nella versione asincrona (per la quale la terminazione, non potendosi basare su un *timer* assoluto come il numero di round, deve basarsi proprio sul numero di hop dei messaggi stessi).

Per l'elezione di un leader in una rete generica possono essere utilizzati algoritmi ideati per problemi diversi.

Problema: **SPANNING TREE**

Spesso molte applicazioni eseguite su reti asincrone hanno lo scopo di costruirne un **albero di copertura**, partendo da un nodo iniziale i_0, per effettuare operazioni di broadcast o di convergecast.

Algoritmi:

AsySpanningTree

Ipotesi: Il grafo G=(V,E) è non orientato e connesso; i processi non hanno necessità di alcun dato sulla rete (a meno dei link verso i neighbors), perfino gli UID non sono necessari.

L'algoritmo produce un albero di copertura non necessariamente minimo.

Formalizzazione:

- signature:

input:	receive("search")$_{j,i}$	$j \in nbrs_i$
output:	send("search")$_{i,j}$	$j \in nbrs_i$
	parent(j)$_i$	$j \in nbrs_i$

- variabili di stato:

parent $\in$ {nbrs} U {null}	init: null
reported is boolean	init: false
send= array[1..#nbrs] di {search,null}	init: l'array della radice i_0 ha tutti i valori a "search" gli array di tutti gli altri processi sono inizializzati con tutti gli elementi a "null".

- transizioni:

send(*search*)$_{i,j}$ $i \neq i_0$	*precondizione*:	*send*[j]="search"
	effetto:	*send*[j] ← "null"
receive(*search*)$_{j,i}$	*precondizione*:	---
	effetto:	if $i \neq i_0$ & parent$_i \neq$null then parent$_i$ ← j for k=1..#nbrs –{j} do send[k]←"search"
parent(j)$_i$	*precondizione*:	*parent*=j *reported*=false
	effetto:	*reported*←true

- task: parent(j)$_i$ $j \in nbrs_i$ ← troviamo il padre del processo i
  for $\forall j \in nbrs_i$ do send("search")

Broadcast asincrono con ACK

L'insieme dei messaggi è dato da: $M \equiv \{ (\text{“bcast”}, w), w \in W \} \cup \{\text{“ACK”}\}$

Formalizzazione:

- signature:
  - input: $receive(m)_{j,i}$ — $m \in M$, $j \in nbrs_i$
  - output: $send(\text{“search”})_{i,j}$ — $m \in M$, $j \in nbrs_i$
  - internal: $report_i$
- variabili di stato:
  - *val* $\in W \cup \{null\}$ — init: null, $\forall i \neq i_0$; “bcast” per $i = i_0$
  - *parent* $\in \{nbrs\} \cup \{null\}$ — init: null
  - *reported* is boolean — init: false
  - *acked* $\subseteq$ nbrs — init: ϕ
  - *send*= array[1..#nbrs] di code FIFO — init: l'array della radice i_0 ha tutti i valori a *val* (=“bcast”) gli array di tutti gli altri processi sono inizializzati con tutti gli elementi a “null”.
- transizioni:

$send(m)_{i,j}$	*precondizione*:	*m* first of send[j]
	effetto:	remove m from *send*[j]
$receive(bcast, w)_{j,i}$	*precondizione*:	---

	effetto:	if *val*=null then
		val←w; *parent*←j
		for $\forall k \in nbrs_i - \{j\}$ do
		add (“bcast”,w) to *send*[j]
		else add(“ACK”) to *send*[j]

$receive(ACK)_{j,i}$	*precondizione*:	---
	effetto:	acked ← acket U {j}
$report_i$ $\forall i \neq i_0$	*precondizione*:	*parent*≠null *acked*=nbrs-{*parent*} *reported*=false
	effetto:	*reported*←true add (“ACK”) to *send*[*parent*]
$report_i$ $i = i_0$	*precondizione*:	*acked*=nbrs *reported*=false
	effetto:	*reported*←true

- task: $report_i$ ← test sulla condizione di uscita
  for $\forall j \in nbrs_i$ do $send(m)_{i,j}$ $m \in M$

problema: **RICERCA**: raggiungimento di tutti i processi (nodi del grafo)
Ipotesi: Ogni processo non ha alcuna informazione sulla rete.

Algoritmi:

BFS: Bradth-First Search (ricerca in ampiezza)

Tiene conto della distanza dal nodo radice i_0.

Formalizzazione:

- signature:
  - input: $receive(n)_{j,i}$ — $n \in N$, $j \in nbrs_i$
  - output: $send(n)_{i,j}$ — $n \in N$, $j \in nbrs_i$
- variabili di stato:
  - *dist* $\in N \cup \{\infty\}$ — init: $\infty, \forall i \neq i_0$; $0, i = i_0$
  - *parent* $\in \{nbrs\} \cup \{null\}$ — init: null
  - *send*= array[1..#nbrs] di code FIFO di numeri naturali ($\in N$)
- transizioni:

$send(n)_{i,j}$	*precondizione*:	n first of send[j]
	effetto:	remove n from *send*[j]
$receive(n)_{j,i}$	*precondizione*:	---
	effetto:	if n+1<*dist* then *dist*←n+1 *parent*←j for $\forall k \in nbrs_i - \{j\}$ do add *dist* to *send*[k]

- task: for $\forall j \in nbrs$ do $send(n)_{i,j}$ $n \in N$

SISTEMI DI ELABORAZIONE II

I sistemi distribuiti
Architetture di elaboratori: Sistemi Centralizzati, Sistemi Paralleli, Sistemi Distribuiti.
Motivazioni Tecnologiche ed Economiche dei sistemi distribuiti.
Le tecnologie di comunicazione.
Le applicazioni delle reti di elaboratori.

Lo scambio di informazioni.
Messaggi e Pacchetti.
Il concetto di sessione.
Commutazione di Pacchetto- Commutazione di Circuito.
Architetture modulari e stratificate. I livelli funzionali del modello OSI.

Introduzione alla Rete Internet
Definizione di protocollo. I nodi esterni della rete. I nodi interni della rete

Gli algoritmi distribuiti.
La complessità di un algoritmo.
Complessità spaziale e temporale. Efficienza asintotica di un algoritmo.
Crescita polinomiale ed esponenziale, Complessità di comunicazione.

Il modello di rete sincrona
Sistemi di rete sincroni.
Il problema dell'elezione di un leader su un anello:
- Algoritmi basati sui confronti.:
  L'algoritmo LCR a propagazione del token circolare. L'algoritmo HS a diffusione di token rientranti
- Algoritmi senza confronti: Algoritmo TimeSlice, Algoritmo a velocità variabile.

Limite inferiore di complessità per algoritmi basati sui confronti e non.
Elezione del Leader su una rete sincrona generica
Algoritmi di rete per ricerca in ampiezza, cammini minimi, albero di copertura minimo.

Le applicazioni di Internet e i Protocolli del livello Applicazione:
- Il World Wide Web e HTTP (Hyper Text Transfer Protocol),
- il File Transfer e lo FTP (File Transfer Protocol),
- la posta elettronica e lo SMTP (Simple Mail Transfer Protocol),
- il servizio di risoluzione dei nomi e il DNS (Domain Name System).

Programmazione di applicazioni in ambiente Unix con l'utilizzo dell'astrazione delle Socket.

I protocolli di livello Trasporto.
Principi generali.
Il multiplexing ed il demultipexing dei flussi generati dalle applicazioni.
- UDP (User Datagram Protocol), un protocollo non orientato alla connessione.
- Principi generali per effettuare un trasferimento di dati affidabile.
  TCP (Transmission Control Protocol) un protocollo orientato alla connessione.
  Il controllo di flusso del TCP.
  Principi generali di controllo della congestione. Il controllo della congestione del TCP.

Algoritmi e protocolli utilizzati a livello Network.
Introduzione ai servizi del livello Network.
Il problema del routing. Il routing gerarchico.
Il protocollo IP (Internet Protocol). L'indirizzamento del protocollo IP.
Gli algoritmi di routing adottati dai protocolli di routing di Internet. I protocolli RIP, e OSPF.
Problematiche relative a IP e il protocollo Ipv6.
Il routing per il multicast.

Il modello di rete asincrona e relativi algoritmi.
Elezione di un leader su una rete ad anello asincrona.

Applicazioni multimediali su rete,
il concetto di Qualità del Servizio, il protocollo RTP (Real Time Protocol).
Politiche di Scheduling e di Policing per garantire le Qualità del Servizio.
Il Protocollo RSVP (Resource Reservation Protocol).

Il problema della sicurezza.
Cenni di crittografia. Chiavi simmetriche e asimmetriche.
L'algoritmo RSA.
Il problema dell'autenticazione.
La firma digitale per i documenti elettronici.