Reti di Calcolatori T - 9 CFU

Prof Antonio Corradi

Appunti di
Federico Fanalista
federico.fanalista@studio.unibo.it

Indice

1 Generalità, obiettivi e modelli di base

1.1 Sistemi Distribuiti

Sistema Distribuito:
La locuzione sistema distribuito, in informatica, indica genericamente un insieme di calcolatori interconnessi tra loro da una rete informatica per l'espletamento di una certa funzionalità e in cui le comunicazioni interne avvengono tramite lo scambio di opportuni messaggi

Insieme di sistemi distinti in località diversa che usano la comunicazione per ottenere risultati coordinati

PRO
- Accesso alle stesse risorse remote da qualunque parte
- Condivisione di risorse remote

CONTRO
- Concorrenza: stesse risorse accessibili da molti
- Nessuna sincronizzazione globale
- Possibili fallimenti indipendenti di singoli nodi

REQUISITI
- Dinamicità del sistema (scalabilità)
- Qualità del servizio
- Tolleranza ai guasti (ridondanza)
- Apertura a evoluzioni dei sistemi

1.2 Architettura software

Per applicativi che comunicano nella rete servono dei modelli di conformità legati a degli standard.
Per un'architettura sono importanti le fasi di *mapping* e di *binding*.
Nella prima fase si parte con una configurazione architetturale e su questa si decide come fare il mapping delle parti di programma sull'architettura. Se cambia l'architettura, è necessario cambiare tale fase.
La seconda fase invece è la decisione che consente di legare una risorsa *logica* (quella da noi definita) ad una risorsa *fisica corrispondente* (quella utilizzata durante l'esecuzione), per esempio un processo che opera su un certo processore o una certa posizione di memoria.
Il legame di binding può essere gestito sia in modo:
- **statico:** prevalentemente nei sistemi concentrati, in questo case il legame è effettuato prima dell'esecuzione ottenendo maggiore efficienza.
- **dinamico:** usato nei sistemi distribuiti, è un legame che viene effettuato al momento dell'esecuzione, ovvero si decide al momento del bisogno dove trovare la risorsa.

In caso di guasto il binding statico non permetterebbe di continuare l'esecuzione, mentre quello dinamico si limita a cercare un'altro server.

UNIX è stato adottato nei sistemi distribuiti per alcune sue caratteristiche positive che realizzano a livello di sistema uno standard di conformità per le funzioni di accesso (***API*** in *UNIX):*
- *Accesso ai file* (open/read e write/close)
- Possibilità di creare programmi di tipo *filtro*, ovvero programmi in grado di consumare tutto l'input producendo un output. I comandi unix sono dei filtri.
- Modello a *concorrenza* basato su processi pesanti.
- Comunicazioni con primitive di sistema intoccabili da ambienti diversi per mezzo di standard chiamati socket.

Essendo UNIX lo standard, il modello standard è a *processi pesanti* e si è passati, per abbattere l'overhead, a microkernel modulari invece che usare kernel monolitici.

OVERHEAD: (letteralmente *in alto*, *che sta di sopra*) serve per definire le risorse accessorie, richieste un sovrappiù rispetto a quelle strettamente necessarie per ottenere un determinato scopo in seguito all'introduzione di un metodo o di un processo più evoluto o più generale.

1.3 Processi

I processi pesanti richiedono molte risorse e portano molto overhead in operazioni quali il cambio di contesto. I processi leggeri, i thread, sono invece contenuti dentro un unico processo pesante e condividono

risorse e sono visibili fra loro, per questi particolari processi il cambio di contesto risulta molto meno pesante e si riduce l'overhead.
Per ovviare al problema dell'eterogeneità dei modi della rete, oltre all'uso di UNIX come architettura standard possono esistere approcci ed ambienti aperti unificati che superano le differenze delle piatteforme in runtime (ad esempio la JVM).
Usando Java come standard, si ha la JVM come processo pesante contenitore dei vari thread. La JVM ha librerie per legarsi al sistema nativo della piattaforma.

1.4 Modello Client/Server

Modello distribuito in quanto le due entità in gioco, il *Client* colui che chiede e il *Server* colui che risponde risiedono su macchine differenti.
Si basa su due aspetti fondamentali: *SINCRONO* (la richiesta del cliente prevede una risposta) e *BLOCCANTE* (il client di **blocca** fin quando non arriva tale risposta).
Tale modello è di tipo molti a uno, ovvero un server deve poter gestire le richieste di *N* nodi client, inoltre risulta un modello molto accoppiato in quanto non è possibile che uno dei due processi termini prima che l'interazione sia completata.
Il *binding* fra il server e il client è *dinamico*, in quanto non si può sapere a priori chi richiede il servizio, il client possiede un informazione che consente di trovare il server al momento del bisogno e in quell'istante viene anche a conoscenza del server.
Il server deve essere sempre attivo, quindi è un *demone*, e pronto a rispondere, se così non fosse il client che effettua una richiesta ad un server non raggiungibile riceve un'eccezione o un segnale di errore.
Il suo progetto è complesso e deve rispondere alle seguenti esigenze:

- Richieste multiple
- Richieste concorrenti
- Accessi a diverse risorse
- Integrità dei dati
- Problemi di accesso e privacy.

Infine tale modello risulta anche *asimmetrico* ovvero non c'è conoscenza da parte del server dei vari client, ciò significa che deve essere pronto ad accettare richieste da qualunque client.
Nel modello sincrono bloccante, il client si blocca fino all'ottenimento di una risposta: se questa non arriva non ci si sblocca più.
La risposta può non arrivare in quanto il server è in crash oppure molto congestionato (simile al crash per il client) di conseguenza non ha senso attendere una risposta all'infinito e quindi vengono settati dei *time-out*.
Quando scatta un time-out il client può decidere se riprovare a richiedere il servizio, cambiare server o altro.
Un server con iterazione sequenziale lavora nel seguente modo: accoda le richieste, le serve, fornisce una risposta e passa alla successiva richiesta seguendo una logica FIFO. Il server per non inviare la stessa risposta alla stessa richiesta che un client ha inviato due volte, deve essere in grado di distinguere le richieste e quindi dare una sola risposta alle richieste multiple per far ciò deve possedere uno stato per ogni client. Il client deve quindi porre un identificativo alle richieste che manda. Il server poi dovrà mantenere il risultato prodotto fino all'avvenuta consegna. Quindi si trattano azioni *idempotenti* ovvero operazioni che fatte più volte hanno lo stesso effetto, quindi le richieste che il client marca con un identificatore risultano idempotenti.

1.5 Modelli asincroni

POLLING O PULL: modello in cui il client esegue le richieste impostando un timeout molto piccolo, per riprendere l'esecuzione e rifare successivamente la richiesta. Ci si aspetta che il server non riesca a rispondere subito, ma conservi il risultato dell'operazione, per cui quando il client ripete la richiesta, riceve il risultato in tempi brevissimi, poiché il server ha già la risposta. Tale modello rende più complicato il ciclo del client, mentre semplifica quello del server che, anche se ha dei tempi molto lunghi, riesce a gestire tutti i client.

PUSH: modello in cui si suppone che il client chiede un'operazione non bloccante, o al limite con una risposta di conferma alla ricezione della richiesta da parte del server. Quando il server ha ottenuto il risultato, questo è etichettato con l'identificativo del client. Il risultato viene spinto (push) dal server verso il client, su iniziativa del server e il client diventa così il servitore dell'operazione di consegna del risultato. Un esempio di interazione push sono i Feed RSS.

1.6 Modello a delegazione (Proxy)

Un client che è interessato ad attendere una risposta non aspetta in prima persona ma comunica al server che ha delegato un'altra entità, il proxy, ad attendere tale risposta e lui continua a fare altro. Il client, per

conoscere il risultato, può ottenerlo lavorando in modo sia pull (chiede se ha il risultato) che push (lo fornisce quando lo ha) sul proxy . Così risulta efficiente da parte del client ma allo stesso tempo il server deve conoscere a priori l'entità proxy incaricata ad attendere la risposta.

1.7 Modello C/S

Adatto alle interazioni di ambienti distribuiti, sincrono bloccante, il client conosce il server solo alla richiesta. Le entità in gioco sono molto legate e devono essere presenti entrambe per interagire. Se una delle due manca, non c'è interazione. *Accoppiamento forte o strong coupling.*

1.8 Modello Pub/Sub (eventi e scambio di messaggi)

Tre entità: **sub**, **gestore** e **pub**.
Il sub si registra ad un gestore, il pub genera eventi e li consegna al gestore. Il gestore riceve gli eventi e ne fa *push* ai sub registrati. E' basato su un modello *molti a molti, asincrono disaccoppiato.*
A differenza del modello C/S:

- Accoppiamento molto debole (non impone compresenza delle parti)
- Flessibile
- Permette comunicazioni **broadcast** e **multicast**.

1.9 Iterazioni C/S

Un aspetto tecnologico importante è quello identificato dalla possibilità di avere un'interazione fatta:

- *con connessione:*
  si stabilisce un canale di comunicazione, *stream,* prima dell'invio dei dati e lo scambio dei messaggi avviene rispettando un certo ordine. La connessione garantisce che la comunicazione avviene con qualità, senza perdite di informazioni, ordinata e senza copie (a livello di trasporto usiamo il protocollo **TCP** in quanto più affidabile e ordinato).
- *senza connessione:*
  interazioni occasionali fatte da scambio di messaggi isolati (protocollo utilizzato: **UDP**, non affidabile e non ordinato). Ha un costo nettamente minore rispetto a quella precedente.

A livello di trasferimento di messaggi, con la connessione si stabilisce una strada fra i nodi, statica, che sarà usata da tutti i messaggi (route) impiegando risorse intermedie.
Senza connessione la scelta della route è libera per ogni messaggio e viene scelta dinamicamente (non si impegnano risorse), ciò provoca un arrivo non ordinato dei messaggi.

1.10 Visibilità della comunicazione

I sistemi di comunicazione formati da molte entità si possono classificare sulla base di due modelli di visibilità:

- *Globale*: permette ad ogni partecipante dello scenario di esecuzione, di comunicare direttamente con tutti gli altri, non mette vincoli sulla visibilità ma si ha un problema di scalabilità.
- *Locale*: vengono messi dei vincoli alla visibilità, un nodo non comunica con tutti, ma solo con un suo vicinato e su questo si realizza un buon protocollo finché si lavora in locale.

1.11 Lo stato

Si parla di stato quando l'interazione tra il client e il server non è singola ma si ha una sequenza di operazioni. Possiamo avere quindi due tipi di stato:

- **Statefull:**
  lo stato è gestito dal server quindi i messaggi sono più ridotti e si ha più efficienza. Di contro il server si carica del compito di dover riconoscere un cliente e di mantenere memoria delle operazioni passate. Si hanno garanzia di ripristino in caso di problemi ai clienti.
- **Stateless:**
  non c'è traccia dello stato che deve essere autocontenuto nei messaggi. I messaggi sono quindi più pesanti in quanto rappresentano più informazioni. Si semplifica il server ma si complica il client che deve gestire richieste più complesse (deve lui mantenere lo stato). Affidabile a malfunzionamenti della rete.

Per poter fare operazioni stateless è importante avere istruzioni *idempotenti,* ovvero che alla stessa richiesta si risponde sempre con la stessa risposta (a meno di modifiche alla risorsa).
Lo stato lato server ha un costo in base alla durata di mantenimento che può essere permanente o a tempo.

1.12 Tipologie di server

Il progetto del server risulta più complicato in quanto, per quanto detto prima, deve essere in grado di servire più client. Esso può quindi presentarsi sotto due modalità di servizio:

- **Sequenziale/iterativo:**
  processa una richiesta alla volta e accoda le altre e non riuscendo a vederle fino a quando non le serve. Può portare a lunghe attese a seconda della lunghezza della coda.
- **Concorrente:**
  può servire più richieste insieme, concordantemente (processi figli o thread più scheduling) o in parallelo (multiprocesso). Ad ogni richiesta è associato un processo diverso, ovvero si genera un processo figlio che esegue la richiesta.
  La complessità e il costo della concorrenza dipende dal sistema di supporto:
  - In UNIX: **fork,** processi pesanti, costo rilevante.
  - In JAVA: **thread,** processi leggeri, minor overhead.

Nel caso iterativo per accorciare i tempi di risposta si può accorciare la coda e rifiutare richieste a coda piena.
Nel caso concorrente si avanzano diverse richieste ma si introducono ritardi relativi alla generazione dei processi e al loro cambio di contesto.

1.13 Progetto C/S

Client: più semplice e di solito sequenziale.
Server: sempre attivo (demone che lo attiva all'inizio di una sessione di sistema) e infinito (per la durata del sistema). Sequenziale o parallelo.
Di default abbiamo una tipologia di progetto SINCRONO BLOCCANTE ASIMMETRICO.
Possiamo avere anche delle variazioni: ASINCRONO quando non ci interessa il risultato, NON BLOCCANTE quanto non si attende il risultato, ASIMMETRICO il cliente conosce il servitore ma il servitore non conosce a priori i clienti possibili.

1.14 Modello ad agenti multipli

Può capitare che la risposta a un servizio non sia reperibile da un solo servitore, quindi il client si interfaccia a un sistema di agenti coordinati che si preoccupano di dare la risposta. Tale servizio degli intermediari (agenti) può anche essere sfruttato per scollegare il client dai server, per ottenere protezione e per scalabilità. Ponendosi dietro un agente, i server possono quindi essere replicati per miglior affidabilità, senza che il client se ne accorga. Tale modello nasce in internet quando vi era la necessità di gestire un insieme di servizi molto esteso con obiettivi di scalabilità.

1.15 Sistema di nomi

Il client deve riferire al servitore tramite il suo *nome.* Questo può essere:

- *trasparente:* nomi indipendenti dalla locazione fisica del servizio, quando un client utilizza tale nome si aspetta di poter raggiungere tutti i servitori che realizzano quel servizio. Se i server modificano la loro posizione fisica il nome rimane invariante facilitandone l'uso per il client ma complicando il lavoro del DNS.
- *non trasparente:* nomi che dipendono dalla locazione fisica del servizio, permettono di vedere l'indirizzo della risorsa che realizza il servizio. L'uso di tali nomi rappresenta un binding statico.

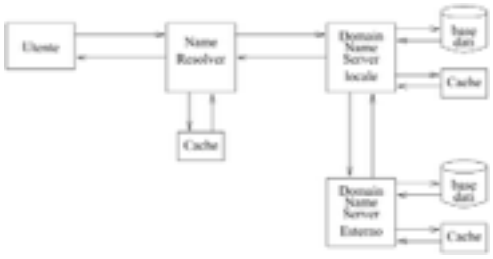
I nomi servono a identificare le entità della rete, ma vanno risolti in modo:

- **statico:** il binding viene fatto prima dell'esecuzione
- **dinamico**: il binding è risolto a runtime al momento del bisogno ed è usato dai nomi trasparenti.

In sistemi concentrati si preferisce un binding statico mentre nel distribuito uno dinamico e quindi serve un servizio di nomi per risolvere le relazioni ovvero ottenere una traduzione dal nome della variabile, contenente il nome del server, posseduta dal client al server con cui si vuole effettuare il binding. Tipicamente i *name server* (gestori di nomi) usano delle tabelle di allocazione.
Si devono poter aggiungere nomi al sistema; esiste una partizione locale dei nomi e degli agenti che coordinano.
Il servizio *Domain Name System* (DNS) gestisce una tabella di corrispondenze, capace di passare da un sistema di nomi (logici) ad un altro (fisico). In particolare è in grado di avere una trasformazione dal nome di dominio al nome di IP corrispondente. Inoltre introduce una gerarchia ordinata di nomi ad albero, dividendo i compiti e il coordinamento e risulta fortemente scalabile. Tale servizio ha qualità grazie anche alla replicazione dei gestori (ovvero la replicazione della tabella) e il partizionamento dei gestori (partizionamento della tabella).
I servitori devono registrarsi presso un DNS per poter avere il nome visibile globalmente. Ogni dominio ha un server dei nomi responsabile della registrazione dei servitori di quella località. L'agente che gestisce i nomi è detto *name RESOLVER.*
Il sistema DNS deve essere affidabile, ovvero server con qualità che non si guastino, efficiente, favorisce buoni servizi, e locale ovvero mantiene una cache con le richieste già effettuate. Per l'efficienza uso a vari

livelli di cache (ovvero si memorizzano le richieste già effettuate) e protocolli UDP, per l'affidabilità necessita la replicazione delle tabelle per mezzo dei *primary DNS* e diversi *secondary DNS* per ogni zona. Un client che vuole risolto un nome non può rivolgersi direttamente al server DNS perché non autorizzato, la richiesta è realizzata così dal *name resolver* nel seguente modo:

1. Un client chiede di risolvere un nome.
2. Il risolver risponde o dalla cache o fa una richiesta C/S a un *nameServer locale.*
3. I DNS agiscono come agenti per ottenere reciprocamente informazioni



1.16 Risoluzioni delle richieste

Un resolver conosce il suo server DNS di domini, per risolvere una query non in cache e quindi per uscire dal dominio locale ci sono due tipi di richieste:

- **Ricorsiva:** si chiede e si ottiene o una risposta o un errore, la risposta può essere ottenuta come richiesta di diversi DNS di gerarchie superiori che rischiano di essere congestionati di richieste. Il risultato ottenuto viene così inserito in tutte le cache dei vari server DNS coinvolti.
- **Iterativa:** si ottiene o la risposta o il suggerimento del miglior DNS server che può rispondere: le richieste non sono passate ma sempre fatte dal richiedente di partenza che sarà l'unico a salvare in cache il risultato.

Se non si hanno risposte in tempi ragionevoli si usano dei timeout per cambiare il server da consultare. A un DNS si può accedere in modo diretto cioè con un nome per avere un IP, o in modo inverso con un IP e avere un nome logico.

1.17 Nomi di internet

Se ne occupa OSI con l'obiettivo di creare interoperabilità fra reti e tecnologie di diversi proprietari. Il trasporto definisce la parte dei servizi, il livello IP definisce i nomi IP dei nodi.
NIC autorità che assegna il numeri di una rete.

Classi	A, B, C	comuni e in diversi numeri di host
	D	multicast
	E	usi futuri

Possibilità di avere sottoreti con l'uso di maschere.

1.18 Directory X.500

In internet è nata la necessità di altri sistemi di nomi globali che non avessero il vincolo del DNS di essere legati a sistemi di nomi fissi e a query semplici, per tali motivi OSI ha definito uno standard chiamato X.500 che consente di inserire qualsiasi informazione nel sistema di nomi con caratteristiche di qualità. Tale standard è una directory ovvero un sistema di nomi in grado di catalogare qualunque informazione e renderla sempre disponibile con un minimo di qualità. La struttura che possiede una directory è ad albero il che permette di fare query complesse di ricerca su tutta la struttura attraverso determinati valori degli attributi.

1.19 Organizzazione interna della directory X.500

Per accedere a tale directory serve un *directory user agent* (DUA) utilizzato dagli utenti per effettuare le query. La directory inoltre necessita di una serie di agenti, ovvero delle macchine su cui vengono memorizzate parti della directory stessa e che bisogna consultare per ottenere la query. Tali agenti della directory sono:

- DSA (Directory System Agent): agenti che contengono le informazioni.
- DSP (Directory System Protocol): agenti che scambiano le informazioni, attraverso un protocollo standardizzato.
- DAP (Directory Access Protocol): agenti che si occupano dell'accesso alla directory attraverso un protocollo non compatibile con internet, per tale motivo è stato creato un protocollo compatibile che è LDAP (Lightweight Directory Access Protocol).

Domande tipiche:

1. **Descrivere il modello C/S sincrono non bloccante e individuare i ruoli e le entità che entrano in gioco.**
   Il modello ha due entità, il CLIENT, sequenziale, che chiede il servizio e il SERVER, sequenziale o parallelo, che risponde. Il modello di DEFAULT è quello SINCRONO BLOCCANTE ASIMMETRICO, ma si prevedono altre forme di modello e iterazione fra le due entità. La caratteristica di sincronicità prevede che ci sia una risposta, l'essere non bloccante prevede che il client non rimanga in attesa della risposta. Questo permette al client stesso di inviare una richiesta ad un server e inviare per esempio un'altra richiesta ad un secondo server. La risposta è prevista ma prima che giunga al client, quest'ultimo può compiere altre azioni. L'utilizzo di un proxy può essere indicativo, in quanto gli vengono delegate le risposte che poi verrano ricevute dai client, quindi utilizzati come cache per i nodi server.

2. **Definisci precisamente le caratteristiche di un rapporto CLIENTE/SERVITORE in un sistema di distribuito nelle sue caratteristiche di sincronicità, blocco, locale o meno, e dinamicità del rapporto**
   Un rapporto C/S nei sistemi distribuiti può presentare diverse caratteristiche. Si dice sincrono quando abbiamo una risposta da parte del server al client, al contrario si dice asincrono. Bloccante, invece quando il client si blocca in attesa della risposta da parte del server fino ad un determinato tempo (timeout), viceversa sarà non bloccante quando il client non attende la risposta ma per esempio delega un suo agente come per esempio un proxy. Infine questo rapporto è sempre dinamico, ovvero il binding viene fatto solo al momento della richiesta da parte del client.

3. **Definire il modello Cliente/Server nelle sue caratteristiche a default.**
   Modello a due entità (1 a molti): Cliente, sequenziale, che effettua una richiesta e un Servitore, sequenziale o parallelo, che offre il servizio e risponde. Il modello è Sincrono (si prevede risposta dal Servitore al Cliente), Bloccante (il Cliente aspetta la risposta da parte del servitore), Asimmetrico (il cliente conosce il servitore ma il servitore non conosce a priori i clienti possibili) e Dinamico (binding tra le entità fatto solo quando un cliente fa una richiesta). Strong coupling: accoppiamento molto stretto tra le entità interagenti che devono essere presenti entrambi per interagire. Il Client fa la richiesta e aspetta, se arriva tutto ok, altrimenti solleva un'eccezione o un'azione compensativa. Se non arriva la risposta non si aspetta per sempre ma esiste un TIMEOUT, cioè è prevista una risposta dopo un determinato arco di tempo oltre il quale si potrebbe fare un'altra richiesta ad un altro server o sollevare un'eccezione locale. Il reale utilizzo del timeout è quello ti ottemperare alle perdite di dati sulle linee o a congestioni sui server.

4. **Modi che ha un client per comunicare col server nel modello C/S (Modelli di interazione C/S)**
   Se il cliente ha iniziativa ed è il primo a fare la richiesta si parla di interazione PULL, ciò semplifica il progetto del server. In caso contrario, se il cliente esegue una sola richiesta asincrona non bloccante ed il server si occupa di consegnare i dati quando sono disponibili si parla di interazione PUSH, tale tipo di interazione sgrava il client da cicli attivi di richieste ma complica il progetto del server. Descrizione e confronto modello C/S e PUB/SUB

5. **Modello C/S ASINCRONO**
   Un ulteriore modello C/S è quello ASINCRONO ovvero senza risposta in contrapposizione ai modelli a Default Sincroni. Si parla di modello di iterazione PULL, quando il client ha sempre l'iniziativa. Viene semplificato il progetto del server ed è il client a decidere. Nel modello di iterazione PUSH, il client fa la richiesta una volta e si sblocca e può fare altro. Il server può fare il servizio ed ha la responsabilità di spedire il risultato al cliente. Il modello PUSH fa diventare il server cliente di ogni cliente. Tipico esempio di modello PUSH è quello per ampliare la fascia di utenza. Per esempio un utente si registra ad un feed RSS che si incarica ad inviare i messaggi ai registrati. Poi riceve ogni nuova notizia su iniziativa del Server. Altro esempio è il modello PUB/SUB.

6. **Modello a Delegazione**
   Un modello a Delegazione viene realizzato quando i tempi di risposta sono troppo lunghi. Infatti per ovviare a questo problema che potrebbe presentarsi nell'iterazione C/S a Default (Sincrono/Bloccante), vengono realizzati sistemi e modelli SINCRONO/NON BLOCCANTI. Il loro funzionamento consiste nella possibilità di delegare una funzionalità ad una entità che opera al posto del responsabile e lo libera di un compito. Le entità che possono svolgere tale mansione sono i PROXY, DELEGATE, AGENTI, ATTORI. Un cliente lascia un'altra entità ad aspettare una risposta ad un operazione fatta ad un server lento. Il proxy lavora in modo push per fornire la risposta al cliente stesso. Quindi un Cliente invia la

richiesta con delega ad un proxy, che viene ricevuta dal Sistema che l'elabora e poi inviata ad un proxy che fornisce la risposta al richiedente.

7. Modello PUBLISH/SUBSCRIBE (ad eventi)

Il modello PUB/SUB è un modello molti a molti, asincrono e disaccoppiato. Le entità in gioco sono: da una parte i CONSUMATORI che si sottoscrivono (SUBSCRIBE) ad un GESTORE cioè un Servitore DI NOTIFICA che riceve gli eventi da un PRODUTTORE che genera le informazioni pubblicandole (PUBLISH). Infine il gestore, notifica gli eventi ai consumatori registrati.

8. Descrivere modello a eventi (pub/sub)

Questo modello è un modello C/S MOLTI a MOLTI asincrono e disaccoppiato, ciò significa che prevede di avere molti PRODUTTORI e molti CONSUMATORI ed un sistema di supporto che gestisce l'invio di messaggi disaccoppiando gli interessati, detto SISTEMA gestore dell'offerta degli eventi. Il ruolo dei PRODUTTORI è quello di segnalare un evento, quello dei CONSUMATORI e ricevere dopo la sottoscrizione (modello pub/sub), la risposta i relativi messaggi dal GESTORE DEGLI EVENTI.

9. Modello a Framework

Modello diverso rispetto a C/S di richieste sincrone a kernel. Il Framework tende a rovesciare il controllo (per eventi di sistema). Il processo utente non aspetta, ma registra con una propria funzione Esempio: Windows che prevede per i processi un loop di attesa di eventi da smistare ai richiedenti. All'arrivo di un risultato questo viene portato al processo significativo. Le risposte dal framework al processo utente sono dette backcall o upcall (push del framework) Sono assimilabili a eventi asincroni generati dal supporto che le applicazioni devono gestire.

10. Modello ad AGENTI MULTIPLI

Schema in cui i servizi sono forniti dal coordinamento di più servitori detti AGENTI MULTIPLI che forniscono un servizio globale unico. Tali agenti possono:

- partizionare le capacità di un servizio
- replicare le funzionalità di un servizio in modo trasparente al cliente.

Un esempio di modello ad agenti multipli è il servizio di posta. Gli agenti di posta sono UA = user agent e MTA = mail transfert agent, quest'ultimo è il servizio che porta la mail da una parte all'altra. La mail tipicamente è un modello ASINCRONO (non prevede risposta) NON BLOCCANTE (non attende risposta).

11. Descrivere le funzioni del servizio DNS.

Al fine di conoscenze reciproche tra entità e servizi nelle relazioni tra C/S è necessario reperire il nome dei servitori nel cliente. I sistemi di nomi sono il primo supporto architetturale, i nomi sono i riferimenti ad altre entità e sono distribuiti nel codice dei clienti. I nomi si qualificano e risolvono riferimenti attraverso: BINDING STATICO ovvero i riferimenti risolti prima dell'esecuzione a differenza del BINDING DINAMICO dove i riferimenti sono risolti al momento del bisogno. Nei sistemi CONCENTRATI il binding è statico in quanto non necessita un sistema di nomi vista l'invarianza dei nomi. Nei sistemi DISTRIBUITI invece è necessario un sistema di nomi che mantiene e risolve i nomi e fornisce il servizio durante l'esecuzione (NAME SERVER). Tipicamente si utilizzano delle tabelle di allocazione controllate da opportuni GESTORI di NOMI. Tra le caratteristiche base per i sistemi di nomi abbiamo:

- Partizionamento dei gestori: ciascuno responsabile di una sola parte (partizione) dei riferimenti località (in generale i riferimenti più richiesti).
- Replicazione dei gestori: ciascuno responsabile con altri di una parte (partizione) dei riferimenti coordinamento.

STRUTTURA DNS

DNS = Domain name system è considerato come un'insieme di gestori di tabella di nomi logici e indirizzi IP, con l'obiettivo di attuare corrispondenze tra nomi logici (HOST) e fisici (IP). DNS introduce nomi logici in un gerarchia ad albero con a capo una radice innominata, poi vengono individuati i domini di primo, secondo livello ed eventuali sotto-livelli. Ogni nome di dominio corrisponde ad un server di responsabilità. Ogni dominio è organizzato su un server di responsabilità primaria, detto di ZONA. La suddivisone in zone è stata fatta per ragioni geografiche, ogni zona riconosce un'autorità cioè un server che fornisce le giuste corrispondenze. I diversi servitori possono delegare ad altri server che se ne assumono responsabilità e autorità. Ogni richiesta di utente viene fatta al servizio dei nomi in modo indiretto tramite un agente specifico chiamato NAME RESOLVER per la gestione dei nomi nella località.

ARCHITETTURA DNS
I diversi server DNS sono organizzati per ottenere requisiti quali: affidabilità, efficienza e località. Ogni dominio ha: NAME RESOLVER: un'entità che fornisce la risposta o perché la conosce tramite la cache o la trova attraverso una richiesta C/S ad un NAME SERVER CACHE.
NAME SERVER DOMAIN
Sono dei server che ottengono informazioni reciprocamente dalla più corretta autorità. Un'ulteriore specifica dei DNS è la REPLICAZIONE, degli stessi domini al fine di fornire il servizio anche in caso di guasti. In generale ogni zona ha un primary master e diversi secondary master copie del primario.

12. Quali tipi di richieste (query) DNS possono essere fatte? Cos'è il DNS resolver? Che entità utilizza il DNS per risolvere le query e in che modo?
Un DNS Resolver è un'entità che fornisce la risposta ad una richiesta DNS. Fornisce la risposta o perché ha la corrispondenza cercata in cache o la trova attraverso una richiesta C/S a un Name Server (che mantiene effettivamente le tabelle di corrispondenza). Ci sono due tipi di query DNS. Quella ricorsiva prevede che il resolver fornisca la risposta (chiedendo ricorsivamente ad altri) o segnali un errore. Quella iterativa prevede che il resolver fornisca la risposta o il miglior suggerimento come riferimento al miglior DNS Server e in questo caso il name resolver non passa la richiesta.

2 Progetto C/S con Socket in Java

2.1 Introduzione

Esiste la necessità di standard in quanto la comunicazione può avvenire fra macchine molto eterogenee. Come standard di comunicazione si fa riferimento alle **socket**. Le socket sono il terminale locale (**end-point**) di un canale di comunicazione bidirezionale in grado di far comunicare in internet.
Ne esistono di due tipi che rispecchiano gli standard **TCP** e **UDC**:

- con connessione —> socket **stream,** usano TCP, prevedono l'apertura di un canale di comunicazione con qualità e quindi un relativo costo elevato.
- senza connessione —> socket **datagram,** usano UDP e hanno un costo basso

La comunicazione nel distribuito avviene fra due processi distinti, un problema deve quindi essere in grado di identificare univocamente i processi in gioco, ovvero un processo client che vuole comunicare con un processo server situato su un altro nodo deve conoscere tale server. Quindi abbiamo bisogno di un *sistema di nomi di internet* in quanto risulta più adatto rispetto a quello dei processi che non ha visibilità globale. Questo quindi risulta costituito da un *Nome Globale:* indirizzo IP + numero di porta (TCP/UDP).
Il processo con *Nome Locale* di una macchina dovrà agganciarsi ad una porta. Le porte sono al massimo 64k dove le prime 1024 sono riservate a servizi standard (well know).

2.2 Server Java

Possiamo avere server *sequenziali* senza o con connessione e server *concorrenti* dove un master server riceve le richieste e genera dei thread per servirle.

2.3 Socket Datagram

Permettono a due thread di scambiarsi messaggi senza creare una connessione, quindi lo scambio avviene in modo non affidabile e non ordinato. Viene istanziata mediante la classe `DatagramSocket` la quale mediante il suo costruttore la crea e fa un binding locale.
La comunicazione avviene con:

- `void send (DatagramPacket p)` —> consegna il datagramma a livello di kernel sottostante, non si coordina con il processo ricevente, è asincrona rispetto al ricevente ma bloccante fino a che il messaggio non è consegnato al driver.
- `void receive (DatagramPacket p)` —> genere un'attesa del pacchetto fino la ricezione, quindi basta ricevere un pacchetto per sbloccarsi (sincrona bloccante).

Servono classi di appoggio come la `DatagramPacket` la quale è divisa in:

- *Parte dati*: un array di byte con indirizzo di inizio e lunghezza.
- *Parte di controllo*: indirizzo IP e porta del ricevente.

2.4 Socket Multicast

Per comunicazioni multicast associate a classi IP di tipo D esiste la classe `MulticastSocket` la quale è in grado di creare una socket che può appoggiarsi o togliersi da un gruppo con la *JoinGroup* e la *leaveGroup*. Su uno stesso indirizzo IP possono risiedere diversi gruppi distinti da una porta.
Una volta che un nodo del gruppo fa una send, la manda a tutti gli che possono leggere con una receive.
Opzioni: esistono diverse opzioni per modificare il comportamento di una socket che di default è sincrono bloccante:

- `setSoTimeout()`
- `setSendBufferSize()`
- `setReceiveBufferSize()`

Dopo un certo tempo l'operazione termina con un'eccezione oppure si modificano i relativi buffer della driver. La comunicazione multicast può essere fatta solo con protocolli non connessi in quanto per definizione la comunicazione connessa è definita punto a punto e non fra più parti. A livello di rete locale, la scheda di rete legge tutti i messaggi che passano sulla rete, seguendo un filtraggio applicativo, portando ai livelli sovrastanti tutte le informazioni di proprio interesse, quindi tutti i messaggi che hanno come ricevente l'indirizzo della scheda di rete scartando tutto il resto.

2.5 Socket Stream

Le socket *Stream* sono gli estremi di un canale di comunicazione creati prima della comunicazione stessa. La comunicazione è *bidirezionale, affidabile, in sequenza e senza duplicati* (semantica *at-most-once*). Se due processi devono scambiarsi un solo messaggio creare una connessione risulta dispendioso mentre non lo è se si vuole scambiare un file.

La connessione è definita univocamente da una quadrupla: `<IP client, PORTA client; IP server, PORTA server>` per tale motivo una connessione risulta unica.
Esistono due tipi di socket stream: una per il client e una per il server.

- **Client:**

  `Socket` —> crea e apre in modo implicito una comunicazione con l'altro terminale specificando nel costruttore IP e porta del server facendo inserire automaticamente dal sistema il suo IP e una porta libera oppure si specifica l'intera quadrupla. In caso di errori il costruttore lancia un'eccezione.
  E' importante la sua chiusura con una `close()` per evitare di impegnare sia le sue risorse che quelle dell'interagente. Close è synchronized in quanto non ha parametri in ingresso a differenza del costruttore che ha parametri unici. Essendo uno stream bidirezionale esistono i metodi per ottenere i due stream come oggetti utilizzati:

  `-getInputStream()` `-getOutputStream()`

  Questi a loro volta saranno inseriti in degli adapter, `DataInputStream()` e `DataOutputStream()`, per evitare la lettura e la scrittura byte a byte e aggiungere funzioni più evolute.

- **Server:**

  `ServerSocket` —> si collega ad una porta locale e ha il ruolo di predisporre una coda e di accettare richieste servendole secondo una modalità FIFO. Il suo costruttore prende in ingresso la porta locale e/o un intero che rappresenta la lunghezza della coda, che a default è 5.
  Una richiesta accodata *non* è servita prima di non essere esplicitamente accettata con una `accept()`. Questa primitiva crea una *socket* connessa con il client che l'ha richiesta.

La semantica dell'accept del server è sincrona bloccante fino a quando non consuma una richiesta. Con la socket ottenuta dall'accept si hanno i due capi della connessione fra C/S e si prosegue. La socket una volta terminato il suo lavoro deve essere **chiusa**.
Si può creare un server parallelo per andare in contro ai costi elevati della connessione. Un processo padre esegue una accept stabilendo la connessione con i client, la gestione della richiesta invece viene gestita da un thread figlio che completa la comunicazione (un figlio per ogni client).
Quando una socket viene chiusa per iniziativa di uno dei due estremi, questa ha effetti anche sull'altro estremo. Ogni socket è responsabile del proprio canale di output mentre per l'input dipende dal pari, ciò significa che i dati che stanno nel canale di output non saranno eliminati del tutto ma si cercherà di consegnarli all'altro end-point e questo li riceverà fino ad un segnale indicato con un EOF che gli dice che la socket è stata chiusa.
Quindi si può pensare a chiusure dolci di un canale tramite una `shutdownOutput()`. In questo modo si chiude l'output ma non l'input che resta ancora aperto per ricevere dal pari.
Dopo una close possono esserci ancora dei dati nel proprio buffer di output che devono ancora essere inviati con l'opzione sul tempo di *linger* (`setSoLinger()`) e quindi si definisce quanto aspettare prima buttare tali dati.
Altre opzioni sono relative a:

`-setKeepAlive` `-setTCPNoDelay`

L'ultima determina una spedizione immediata non bufferizzata.

Domande tipiche:

1. **Nominare le classi coinvolte in una connessione socket TCP in Java.**
   Classe Socket:
   Crea una socket stream, il costruttore prende in ingresso porta e IP del server. La creazione produce in modo atomico anche la connessione con l'entità remota (bind implicito). Lato client è l'unica ad essere usata, lato server è usata per richiamare l'accept() della socket del server
   Classe ServerSocket:
   Crea una socket di ascolto sulla porta specificata, ovvero quella del server. Questa permette di accodare e accettare successivamente (con primitiva accept()) le richieste di connessione provenienti da diversi client, lato Server. Infine altre classi usate in questo tipo di connessione sono le *DataInputStream()* e la *DataOutputStream()* che permettono di inviare e ricevere i pacchetti tra i due pari.

2. **Nominare e descrivere classi e primitive in Java per una comunicazione attraverso protocollo UDP distinguendo tra cliente e servitore**
   Le fasi di interazione tra cliente e servitore e le classi utilizzate sono le stesse. La classe *DatagramSocket* permette di creare un end-point di comunicazione fra due thread diversi senza stabilire una connessione (modello non affidabile). Una volta creata la socket lo scambio di messaggi può avvenire mediante meccanismi primitivi di comunicazione. Su una istanza di DatagramSocket si invocano i metodi s*end(DatagramPacket p)* e *receive(DatagramPacket p)* che sono reali operazioni di comunicazione. La send invia il messaggio consegnandolo solamente al livello sottostante che si occupa dell'invio (primitiva passante), mentre la receive ha una semantica bloccante localmente: alla prima ricezione si sblocca e si prosegue. La classe DatagramPacket serve a preparare e usare i datagrammi. I client userà il costruttore contenente sia la parte di controllo inserendo IP e porta del Server che quella di dati, settando opportunamente un array di byte, il server invece usa il costruttore che prende in ingresso la parte dei dati.

3. **Nominare e descrivere classi e primitive in Java per una comunicazione attraverso protocollo TCP distinguendo tra cliente e servitore**
   Al fine di realizzare una socket Stream in Java, è necessaria una connessione, esistono delle classi di supporto per la realizzazione. Nel package java.net di networking, sono presenti le classi: *ServerSocket* per il Server e *Socket* per il Client. I costruttori relativi alle due classi possono essere diversi:
   Per il CLIENT:
   public Socket(InetAddress remoteHost, int remoteport); *public Socket(String remoteHost, int remoteport)*; La classe Socket consente di creare una socket attiva e connessa stream(TCP) per il collegamento C/S.
   Per il SERVER:
   possiamo avere due costruttori: *public ServerSocket(int localPort)* che crea una socket in ascolto sulla porta specificata oppure *public ServerSocket(int localPort,int count)* dove count è la lunghezza della coda.

4. **Multicast socket in java**
   Nelle socket in java è presente una classe del relativo package utile alle comunicazioni multicast: *MulticastSocket*. Ovvero consente di inviare/ricevere messaggi di gruppo, tra un server e diversi client, il cui costruttore è *MulticastSocket(int multicastport)*. Un gruppo multicast è specificato da un indirizzo di multicast di *classe D* [224.0.0.0 - 239.255.255.255] e da un numero di porta. La comunicazione avviene tramite delle funzioni primitive: *send()* e *recv()* utilizzando delle classi accessorie per la preparazione del gruppo.
   Il modello di comunicazione è:
   - preparazione dell'IP di classe D con la *InetAddress*
   - creazione socket e set porta con la *MulticastSocket(int Port)*
   - azioni di gruppo con i metodi *leave()* e *join()*
   - creazione packet
   - invio e ricezione con le primitive *send()* e *recv()*.

   Problema delle socket multicast si presenta nel caso volessero essere utilizzate in reti locali o dove non ci sia una connessione di rete.
   Infine esistono diverse opzioni per modificare il comportamento di una socket che di default è sincrono bloccante:
   - setSoTimeout - setSendBufferSize - setReceiveBufferSize

   Dopo un certo tempo l'operazione termina con un'eccezione oppure si modificano i relativi buffer della

driver.

5. **Descrivere le primitive della fase di chiusura delle socket in java**
   Nelle socket stream che sono socket connesse, è necessario chiudere le connessioni al fine di risparmiare risorse mediante il metodo *public synchronazid void close()throws SocketException,* il quale chiude l'oggetto Socket precedentemente creato e disconnette il Client dal Server. In caso di una Socket chiusa le risorse vengono mantenute per un certo periodo di tempo, per inviare i bytes che devono essere ancora spediti. Esiste un modo più 'dolce' per chiudere la connessione attraverso l'uso delle primitive: *shutdownInput()* e *shutdownOutput()* che consentono di chiudere la connessione in un solo verso ovvero l'output lasciando il verso entrante aperto per il rispettivo pari.

6. **Descrivere lo stream di comunicazione socket TCP java**
   Dopo aver qualificato le risorse di una connessione socket stream, l'invio e la ricezione dello stream avviene tramite: *public InputStream getInputStream()* e *public OutputStream getOutputStream()*. Attraverso gli stream di java si possono inviare/ricevere solo bytes, questi arrivano ordinati e non duplicati e al più una volta secondo la semantica AT-MOST-ONCE, quindi con una certa affidabilità, ma senza nessun controllo in caso di malfunzionamento. Un'ulteriore vantaggio degli oggetti in Java è la possibilità di incapsulare tali oggetti e utilizzare funzionalità di più alto livello, come: *DataInputStream* e *DataOutputStream*. Le funzionalità più usate sono la *writeUTF*, *writeInt, readUTF* ecc.ecc

3 Progetto C/S con Socket in C

3.1 Introduzione

Usiamo UNIX come modello per lo scambio di messaggi e strumenti come i *segnali* per la **sincronizzazione** e di *file* che condividono lo stesso file system sullo stesso nodo per la **comunicazione.** Fra **processi** che risiedono sullo stesso nodo invece usiamo le *pipe* o le *pipe con nome.*
Per la sincronizzazione remota invece si usano le *socket*.
In UNIX i file aperti sono identificati dal *file descriptor* e mappati nella tabella dei file aperti. Le socket sono conformi ai file e usano le stesse primitive come: *open*, *write*, *read* e *close*. Le socket che con la quadrupla IP-Porta Locale, IP-Porta Remota rappresentano l'end-point di comunicazione, sono mappate con i socket descriptor come dei file.
Le socket in UNIX sono state standardizzate perché presentano caratteristiche di:

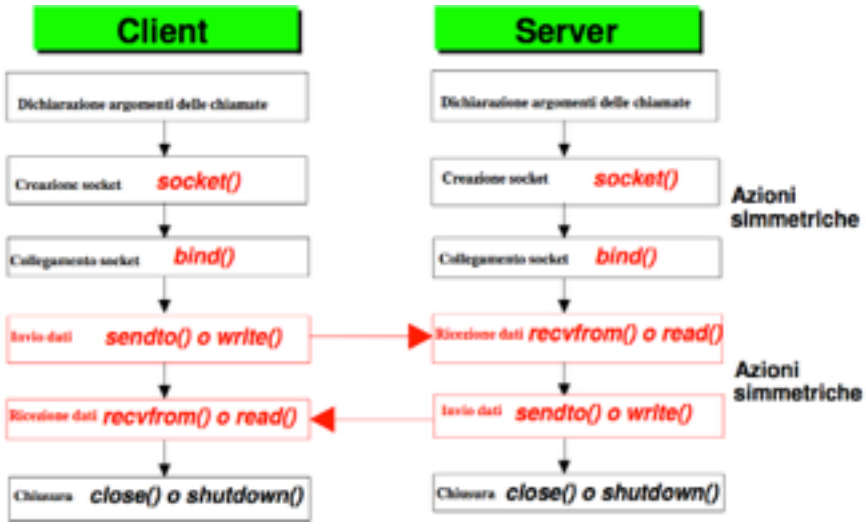
- *eterogeneità:* possibilità di comunicare tra ambienti diversi su architetture diverse.
- *trasparenza*: comunicano in modo unico, indipendente dal processo destinatario.
- *efficienza*: comunicano nel miglior modo possibile, esempio buffer.
- *compatibilità*: le possono usare anche file o programmi scritti in precedenza senza aver la necessità di ricompilarle.
- *completezza*: garantiscono che tutti i protocolli di comunicazione possono essere accettati nello scenario delle socket.

Le socket lavorano con *protocol family PF* (PF_UNIX, PF_INET) e i domini di *Address family AF* (AF_INET, AF_UNIX). Le tipologie di servizio legate ai protocolli sono le *datagram* e le *stream*.
Il nome logico (locale) di una socket è il suo indirizzo nel dominio, il nome fisico (globale) è una parte sul nodo, quindi necessita di un binding fra le due entità.
Per rappresentare un nome di socket si ha una generale `sockaddr(…)` ovvero una struttura (di 16 byte, 2 riservati alla socket address family e il resto alla socket address data) contenente l'indirizzo generico di una socket che può ospitare molti nomi di dominio, non solo di internet.
Per rappresentare invece gli indirizzi di dominio di internet si usa una struttura chiamata `sockaddr_in(…)` (di 8 byte) della famiglia AF_INET dove le due strutture C sono perfettamente compatibili.
Per il supporto del servizio di nomi esiste la funzione `struct hostent* gethostbyname(char* name)`, la struttura *hostent* contiene le informazioni sul nodo passato come parametro.
Analogamente esiste la funzione `struct servent* getservbyname(char* name, char* proto)` per ottenere il numero di porta di un servizio contenuto nella struttura, passandogli in ingresso il nome logico del servizio e il protocollo.
Analizziamo ora le primitive preliminari delle socket:

- `int socket (int dominio, int tipo, int protocollo)` —> crea la socket a livello applicativo rappresentata dal file descriptor nel kernel ad essa associata.
- `int bind (int s, sockaddr* indirizzo, int lunghezza)` —> crea il legame fra la socket applicativa con l'indirizzo di dominio passato per argomento

Questa socket creata è un end-point di comunicazione e rappresenta una *half-association* (mezza associazione del canale di comunicazione) locale pronta per comunicare.

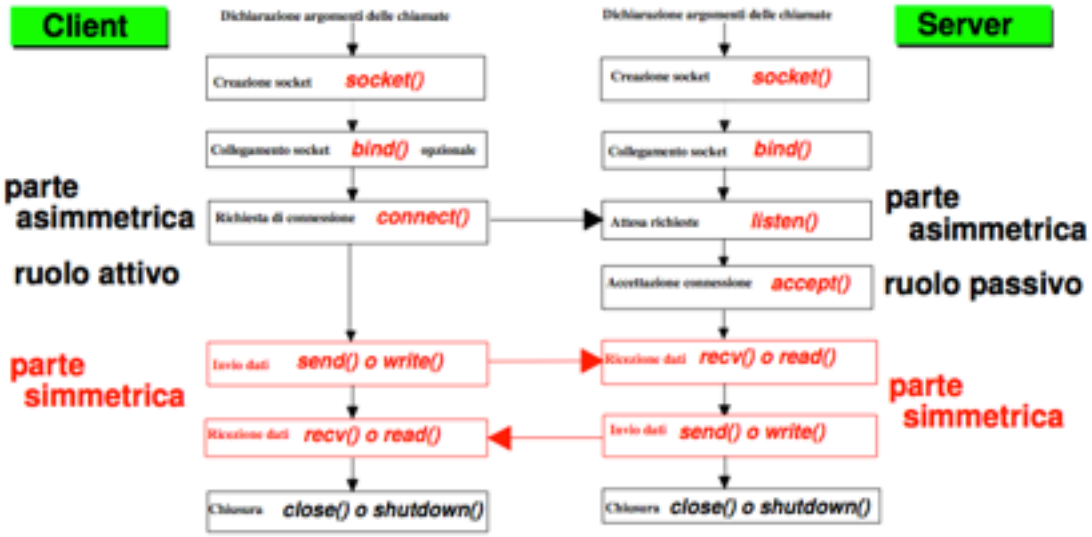
3.2 Socket Datagram



Le operazioni con questo end-point sono la `sendto()` per inviare e la `recvfrom()` per ricevere; entrambe come parametro necessitano della struttura `sockaddr_in` che rappresenta l'altro nodo della comunicazione e restituiscono il numero di byte inviati/ricevuti in caso di successo un intero negativo in caso di fallimento.
I dati (datagrammi) così inviati sono di protocollo UDP quindi non affidabili: come tecniche di prevenzione si possono limitare il numero di datagrammi da inviare al minimo e richiedere datagrammi di avvenuta consegna.
Problemi legati al protocollo UDP:
- *affidabilità*: se si perde il messaggio o la risposta il client rimane ad aspettare quindi si usano i *timeout*.
- *controllo del flusso:* se il server riceve troppi messaggi e non riesce a elaborarli li scarta senza avvisi.

Esistono opzioni per modificare le dimensioni dei buffer di IN e OUT di una socket.

3.3 Socket Stream



Abbiamo una entità attiva che richiede la connessione e una passiva che invece accoda una richiesta e cerca di servirla.
Protocollo di comunicazione:
Parte *Asimmetrica:*
- entrambi i lati creano le socket localmente con la `socket()` la quale restituisce un socket descriptor in caso di successo o -1 se fallisce
- Viene invocata la `bind()` che collega la socket appena creata localmente alla porta e al nodo globale (sistema di nomi globali);
- il client fa una `connect()` che indica la volontà di volersi connettere al server.
- il server attende richieste con una `listen()` e le mette in coda .
- il server accetta e serve le richieste con una `accept()` che genera una socket per la vera comunicazione, il suo successo produce quindi la connessione.

Parte *Simmetrica:*
- invio fra client e server sullo stream con `write()/read()` o `send()/recv()`. Il risultato di tali operazioni sarà il numero di byte realmente inviato o ricevuto.
- chiusura delle relative connessioni con la `close()` o la `shoutdown()`.

Usando le stesse primitive di accesso ai file i filtri *naive* possono lavorare nel distribuito sulle socket senza rendersene conto.
Da notare che per il client la bind è opzionale in quanto non hanno bisogno di essere visti dall'esterno e può essere implicita. Infatti è eseguita dalla connect() che assegna la prima porta libera di sistema, quindi ogni server che vuole essere raggiunto dai client deve eseguire la bind().
La primitiva *connect()* di un client è sincrona e termina solo quando la richiesta è accodata. Restituisce il file descriptor se il risultato è positivo, oppure se il risultato è negativo restituisce un errore memorizzando il motivo di tale fallimento nella variabile *errmo (EISCONN, ETIMEDOUT o ECONNREFUSED).* In uscita dalla *connect()* si ha una connessione solo dal lato client.
La *listen()* (primitiva senza attesa) crea una coda al server per possibili richieste di servizio, il numero massimo, che di default è 5, è dato da un parametro. Il risultato di tale primitiva è positivo in caso si successo

restituendo il file descriptor e negativo in caso di errore. Se arriva una *connect()* quando si è già al massimo della coda, questa è scartata senza che il server se ne accorga. Tale primitiva deve essere eseguita dal server prima che il client esegua la connect.
La connessione è completata dal server quando questo fa una *accept()* (sincrona bloccante verso la coda) su una richiesta in coda. In questo modo si produce una socket connessa al client che si appoggia sulla stessa porta della listen. Il suo risultato sarà negativo in caso di errore se positivo invece restituisce una nuova socket connessa al client mediante il suo socket descriptor.
Si possono avere socket diverse con bind alla stessa porta. Le diverse connessioni si distinguono dall'IP del client.

3.4 Comunicazione

Quando a livello applicativo si utilizza una primitiva di scrittura sullo stream, questo dato non è subito inviato ma viene bufferizzato dal driver TCP e inviato a blocchi.
Soluzione: messaggi di lunghezza del buffer o *flush* espliciti.
Per la ricezione c'è il problema analogo in quanto la read leggerà i dati bufferizzati dal driver.
Soluzione: messaggi di lunghezza fissa o variabili ma preceduti da un messaggio di lunghezza fissa con indicazione sulla lunghezza variabile del successivo.
Per ottimizzare lo stream si deve fare in modo che ogni flusso di dati sia consumato per intero, per questo si fanno circolare dei segnali di fine flusso. Quindi ogni end-point legge da IN fino a fine flusso e farà terminare il suo OUT con un fine flusso.

3.5 Chiusura

Alla chiusura con una normale *close()* vengono liberate le risorse ma non istantaneamente, dopo un tempo di *linger* settabile. Con la socket abbiamo dei buffer di IN e OUT associati: il contenuto del buffer IN viene buttato, mentre l'OUT ha il tempo di linger per spedire al pari il contenuto. A questo punto si può deallocare la memoria dei buffer, il pari quindi se si trova in stato di lettura ottiene un *fine file,* se invece è in stato di scrittura ottiene un segnale di *connessione non esistente.*
Con l'uso delle `int shoutdown(int s, int how)` si può decidere di chiudere solo uno dei due canali di comunicazione per ogni socket:
- Chiusura di *_RD:* non si riceve ma si trasmette, una send del pari ritorna -1 e il processo riceve *SIG_PIPE.*
- Chiusura di *_WR:* si riceve e non si trasmette, se il pari legge trova un *EOF.*

Tipicamente una socket decide di chiudere il proprio canale di OUT, il pari così consuma il buffer e poi trovando *EOF* capisce di non dover più leggere ma potrà comunque inviare finché vuole se dall'altro lato però non si è chiuso l'IN.

3.6 Presentazione dei dati

Avendo a che fare nella rete con piattaforme e sistemi eterogenei, bisogna garantire che i dati che vengono scambiati siano compatibili. Le diverse macchine possono essere *big* o *little-endian* mentre per convenzione la rete è *big-endian*. Esistono funzioni di libreria che servono a uniformare i formati di rete e interno degli interi *htons()* e *htonl(),* queste funzioni prendono uno short o un long dal formato locale e lo convertono in uno a formato internet. Nei formati interni big-endin la htons non fa nulla. Esistono altre funzioni *accessorie* per conversioni da indirizzi IP a stringhe in notazione puntata e viceversa. Altre funzioni servono per lavorare con blocchi di memoria a partire da indirizzi dati senza badare al contenuto.

3.7 Situazione di server reale multi-processo con demone

- Il processo padre crea la socket, fa il bind e crea la coda con la listen.
- Con la *setsid()* si stacca dalla console e crea una nuova gerarchia.
- Crea il figlio demone con una *fork().*
- Il demone chiude *stdin e stderr* e ridireziona *stdout* su un file di log, ignora SIGLD per non avere zombie.
- Il demone si mette in ciclo infinito di *accept* e crea un figlio con una *fork* per ogni richiesta.
- Chiude la socket creata e continua il ciclo.
- Il processo chiude la socket d'ascolto e svolge il servizio e infine termina.

3.8 Select

Per ovviare al costo di creazione di figli con *fork()* si utilizza la primitiva *select()* che già esisteva prima delle socket per gestire possibili attese multiple di un processo che si bloccava su diversi I/O con le primitive di accesso ai dati (lettura e scrittura), le quali essendo primitive sincrone bloccanti possono causare tempi di attesa lunghi.
La primitiva select() consente di creare un server concorrente mono processo in C che possa accettare servizi molteplici, bloccando un processo in attesa di un evento che può essere di lettura, scrittura o eccezione.

Tale primitiva quindi è in grado di attuare una scelta riguardo a quale socket gestire, se datagram o stream, poiché tali socket forniscono un risultato senza causare momenti di blocco.
Generare processi figli risulta molto dispendioso, quindi la select fornisce uno strumento per ottimizzare i costi e gestire più entità in un unico processo.
La select è sincrona e limitata da un timeout:

```
int select(int nfds, fd_set* readfds, fd_set* writefds, fd_set* exceptfds,struct timeval* timeout)
```

- *nfds:* numero massimo di eventi attesi e lunghezza delle maschere
- *readfds, writefds, exceptfds:* sono maschere (lettura, scrittura ed eccezione) che rappresentano dei file descriptor (o socket) sui quali si attendono eventi.
- *timeout:* tempo di attesa massimo, se settato a 0 la select sarà sincrona bloccante e quindi pooling.

La select ritorna un numero negativo in caso di errore oppure il numero di eventi accorsi e li segnala con un bit settato a 1 nelle diverse maschere:

- Eventi di *lettura: recv, read, accept, EOF.*
- Eventi di *scrittura: send, write, connect, SIGPIPE.*
- Eventi di *eccezione: close e shoutdown.*

Se in polling o al timeout, si esaminano le richieste attraverso le maschere: in ogni maschera c'è un 1 in corrispondenza di un *FD* (File Descriptor), allora tale FD è in attesa dell'evento relativo alla maschera. Se sono più di uno si può decidere se servirli tutti o meno e in che ordine. Le maschere sono modificate ad ogni select (se ci sono eventi). Esistono funzioni e macro per ottimizzare il lavoro bit a bit con le maschere: settare i FD, verificarne lo stato e infine azzerarli.

3.9 Multiserver

Può aver senso avere un unico servitore per più servizi, questo può portare a termine completamente dei servizi semplici e delegare ad altri processi i servizi più complessi.
Questo servitore è un demone, *inetd* in UNIX, e schedula i servizi con una select.

3.10 Tipi di Client

Anche da lato client si può gestire concorrenza e parallelismo:

- *Concorrenza:* un client che deve instaurare diverse connessioni con un server, gestione con la select.
- *Parallelismo*: un client master genera diversi slave che interagiscono in maniera diversa con diversi server (es multicast).

3.11 Opzioni su socket

Con le *getsockopt() e setsockopt()* abbinate a diversi parametri si possono configurare i comportamenti di una socket in modo diverso da quello a default:

- *SO_SNDTIMEO, SO_RCVTIMEO:* settano il tempo massimo di attesa per send/receive.
- *SO_SNDBUF, SO_RCVBUF*: settano le dimensioni dei buffer.
- *SO_KEEPALIVE*: controlla periodicamente lo stato della connessione.
- *SO_REOSEADDR*: permette la convalida dell'indirizzo di una socket senza un controllo di unicità di associazione.
- *SO_LINGER*: permette di impostare il tempo con il quale viene deallocato il buffer in uscito dopo una close.

3.12 Socket non standard

Tramite l'uso delle primitive *ioctl()* e *fcntl()* e relative opzioni si possono ottenere socket *asincrone* e/o non bloccanti. Le socket asincrone sono senza attesa di risposta, ma tipicamente questa è segnalata con il segnale SIGID (di default ignorato). Di natura i segnali sono inviati a un gruppo di processi, bisogna fare in modo che arrivi solo a quello giusto (flag con pid negativo). In caso non bloccante le tipiche primitive restituiscono particolari condizioni di errore e il risultato è consegnato tramite segnali.

Domande tipiche:

1. **Elenca le funzioni per creare una datagram socket in C di client e server e dire cosa esprime il loro valore di ritorno.**
   Le socket Datagram in C prevedono il supporto di alcune primitive per la loro creazione, scambio di messaggi e chiusura. Le primitive utilizzate sono:
   - int Socket(int AF_INET,int SOCK_DATAG,0). Consente la creazione della socket restituisce un intero, se ha un valore maggiore di zero la socket è stata creata correttamente altrimenti minore
   - int bind(struct socketaddr_in addr, int addrlen). Consente il legame tra nomi fisici e nomi Globali, ovvero tra indirizzi e numero porta se il valore di ritorno (intero) è maggiore di zero la bind è andata a buon fine, altrimenti no.
   - int sendTo() e recvfrom() o ancora read() e write() sono funzioni per l'invio e la ricezione dei messaggi, restituiscono un int che rappresenta il numero di bytes trasmessi.
   - void close(int fd) e void shutdown(int fd,int how) sono invece funzione che chiudono la comunicazione tra C/S.

2. **- Socket TCP in C, primitive C/S e a cosa servono.**
   - Nominare le primitive C per l'apertura di una connessione con socket TCP.
   - Listare e descrivere le primitive in C per una interazione con connessione, sia per un Cliente sia per un Servitore: si dia di ogni primitiva la specifica precisa del caso di successo.
   Le primitive per una socket di tipo stream in C sono:
   1. **socket()** crea l'entità locale logica di connessione è usata sia lato Client che Server e restituisce un socket descriptor in caso di successo o -1 se fallisce
   2. **bind()** collega la socket locale a un indirizzo e al sistema di nomi globali, opzionale per il Client, necessaria per il Server in quanto gli serve per essere raggiunto dai Client che richiedono un suo servizio.
   3. **connect()** primitiva di comunicazione sincrona utilizzata dal Client che termina quando la richiesta è stata accodata o in caso di errore. Restituisce il file descriptor se il risultato è positivo, oppure se il risultato è negativo restituisce un errore memorizzando il motivo di tale fallimento nella variabile errmo.
   4. **listen()** primitiva senza attesa utilizzata dal Server la quale crea una coda per possibili richieste di servizio. Ancora una volta restituisce il file descriptor in caso di successo o un errore in caso di fallimento.
   5. **accept()** primitiva con attesa, anche questa utilizzata dal Server, che stabilisce la reale connessione accettando e servendo le varie richieste in coda. Il suo risultato sarà negativo in caso di errore se positivo invece restituisce una nuova socket connessa al client
   6. **read/write o send/recv** lettura e scrittura di byte dalla/nella socket connessa. Il risultato di tali operazioni sarà il numero di byte realmente inviato o ricevuto
   7. **close()** e **shoutdown()** chiudono la connessione in un tempo stabilito (linger) con la differenza che quest'ultima la chiude in modo unidirezionale, ovvero o il canale di IN o quello di OUT.

   Dal punto 1 al punto 5 abbiamo la parte asimmetrica, gli ultimi due punti sono quelli simmetrici.

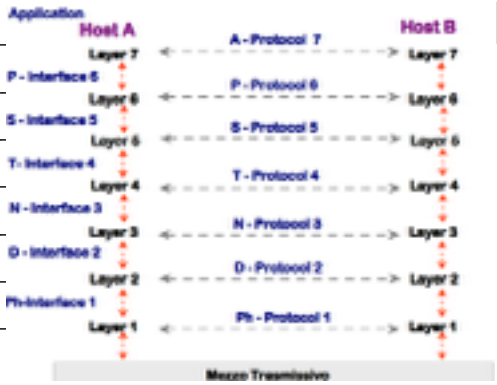
3. **- Primitiva select() in C, indicare quando si usa, la sintassi, il valore di ritorno e gli argomenti.**
   - Dimensione delle tre maschere e utilità del primo parametro
   - Descrivere come funziona la primitiva select a partire dalla sua firma
   - Primitiva Select() in C
   La realizzazione delle socket in C, fa si che si utilizzano delle primitive che a volte hanno un comportamento sincrono bloccante, ciò penalizza il funzionamento del processo in quanto può rimanere bloccato in attesa di richieste che non arrivano. La gestione di queste attese viene risolta con l'introduzione della primitiva select() infatti tale primitiva consente di creare un server concorrente mono processo che possa accettare servizi molteplici, bloccando il processo in attesa di almeno un evento che può essere di tipo: lettura, scrittura o eventi anomali (eccezioni).
   La funzione select() si presenta con i seguenti parametri: *int select(size_t nfds, int* readfds, int* writefds, int* exceptfds, struct timeval* timeout),* dove il valore di ritorno è un intero e indica il numero di eventi occorsi e indica quali per mezzo delle maschere. I parametri in ingresso invece sono nfds il numero massimo di eventi attesi, le tre maschere che rappresentano dei file descriptor, o socket, sui quali si attendono eventi, e un valore di timeout che se settano a 0 diventa una funzione sincrona bloccante.
   La chiamata prevede l'esame degli eventi per il file descriptor specificate nelle tre maschere ovvero quelli con bit a 1.

4 OSI - Open System Interconnection

4.1 Introduzione

OSI è uno standard di comunicazione tra sistemi aperti con l'obiettivo della *interoperabilità* tra i sistemi eterogenei. OSI è uno schema di progetto e non un'implementazione quindi non ha legami con realizzazioni proprietarie.
E' organizzato in sette livelli:

7) Applicazione
6) Presentazione
5) Sessione
4) Trasporto
3) Network
2) Data Link
1) Fisico



Ogni livello comunica con il suo pari attraverso un protocollo e sfrutta i servizi del livello sottostante. Servono quindi interfacce fra i vari livelli di una stessa catasta. Ogni livello OSI definisce un servizio o interfaccia da offrire al livello superiore e un protocollo per definire lo scambio di dati e informazioni con il suo pari. Tale interfaccia prende il nome di **SAP**, *service access point*, che rappresenta un punto di accesso ad un servizio che un livello offre al suo sottostante

4.2 Sistema di nomi

Ogni elemento attivo di un livello è detto entità. Si differenziano i vari livelli anteponendo l'iniziale maiuscola del nome (S-Layer per Sessione). L'interfaccia logica fra due entity di tipo N-1 e N è detta SAP o N-SAP ed è definita da API. Ogni SAP deve avere un nome unico per essere identificata, per identificare un entità si devono nominare tutti i SAP di ogni livello fino al livello 1.

4.3 Comunicazione

La comunicazione avviene fra un mittente che la inizia e un ricevente che la accetta e la sostiene, passando per eventuali intermediari che servono a sostenerla. Ogni azione comporta un passaggio per tutti i livelli, dall'applicativo al fisico da parte di mittente e ricevente e almeno fino al Network per gli intermediari.
Per la comunicazione tra i diversi livelli si riconoscono il **SDU** (*service data unit)* e il **PDU** (*protocol data unit*) che descrivono i messaggi che permettono di chiedere il servizio e di specificare il protocollo. Il PDU è formato aggiungendo informazioni al dato passato, fornito dal protocollo diventa SDU del livello sottostante. Le modifiche avvengono con il **PCI** *(protocol control information)*. In pratica ogni livello aggiunge proprie informazioni specifiche al dato ricevuto e le passa al SAP sottostante (mittente), viceversa nel ricevitore, non le aggiunge bensì le toglie.

4.4 Protocollo

OSI definisce solo specifiche e né suggerisce né indica tecniche implementative di soluzione. Esistono implementazioni di diverso tipo come a procedure, a oggetti, a processi dal momento che l'organizzazione a SAP è astratta.

4.5 Modalità e Qualità

Esistono due modalità di comunicazione:

1. *CONNECTIONLESS:*
   Dati spediti in modo indipendente e automaticamente senza ordine, con costi limitati, scarse garanzie, non c'è qualità di servizio, non c'è storia e negoziazione.
2. *CONNECTION-ORIENTED:*
   Si stabilisce tramite negoziazione una connessione fra due entità pari che conta di tre parti apertura - trasferimento - chiusura. Si richiede qualità di servizio.

La prima è adatta a l'invio di dati occasionali, costa poco ma è senza garanzie.
Con la connessione si ha un costo più elevato ma una maggiore affidabilità sull'arrivo e l'ordine dei messaggi. In fase di negoziazione si possono stabilire gli intermediari che non necessariamente

impiegheranno risorse per sostenere la connessine. Ogni servizio deve rispettare le qualità si servizio (QoS) determinata che può essere scelta a seconda delle esigenze.

4.6 Primitive

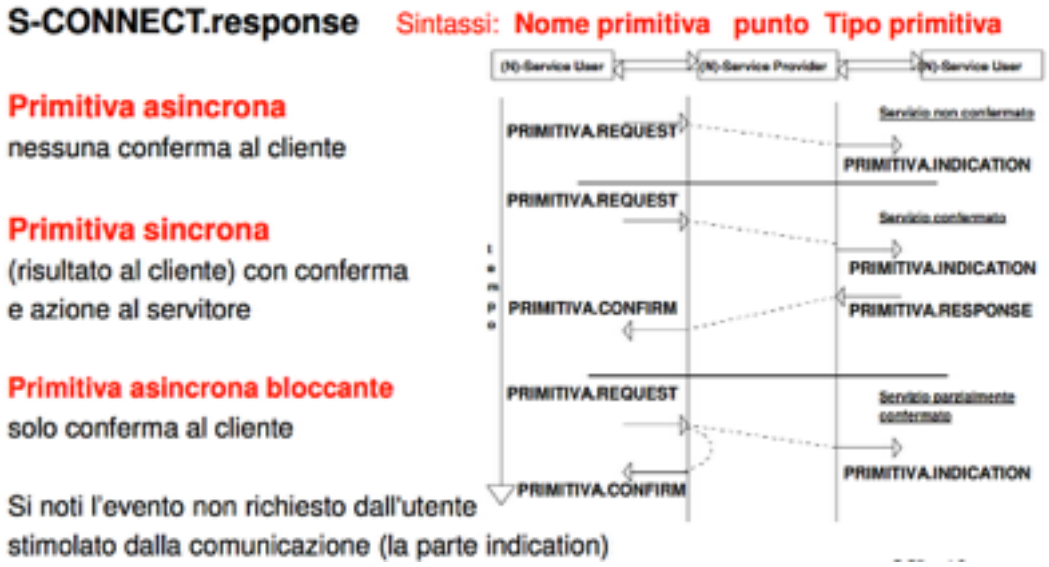
Due entità pari cooperano tramite **primitive** per svolgere le funzionalità del livello a cui appartengono:

- *connect:* apre la connessione
- *data:* trasferisce i dati
- *disconnect:* chiude la connessione

Ogni primitiva ha quattro **forme:**

1. *request*: si richiede un'azione.
2. *Indication*: viene segnalata la richiesta di un evento.
3. *Response*: si risponde alla richiesta.
4. *Confirm*: si segnale la risposta alla richiesta.

Nel caso *sincrono* sono presenti tutte e quattro le fasi.



Una sequenza di primitive può essere:

connect (con le quattro forme), molte *data* (con *request e indication)* ed infine *disconnect.*

4.7 Livelli OSI

Connessioni OSI: tipicamente con QoS e impegno degli intermediari
Connessione TCP/IP: *best effort* e impegno solo di end-point.
Best effort: fa il massimo possibile ma senza garantire controlli di errori, flusso e congestione.
3 livelli OSI inferiori: *fisico, data link e il network;* 1 intermedio: *trasporto.* Questi quattro livelli forniscono un meccanismo trasparente per il trasporto end-to-end e le funzioni base includono:

- controllo degli errori dovuti al rumore o altra causa
- controllo del flusso dei dati
- modelli di indirizzamento per identificare end system (naming)
- per la rete, strategie di routing per trasferire i dati (internetworking)

4.8 Network

Questo livello si occupa di realizzare il *routing* fra le diverse reti tenendo conto dei nodi intermedi. Lo strumento che permette tale scopo è il *router* che è appunto un dispositivo di infrastruttura. L'obiettivo di questo livello è quello di cercare di far passare le informazioni fra gli intermediari senza toccare i livelli applicativi.
I compiti a livello di rete sono:

- indirizzamento (*routing)*
- controllo del flusso tra i due pari
- controllo della *congestione* dell'intero sistema

4.9 Trasporto

Rappresenta un livello intermedio di separazione tra quelli sottostanti di comunicazione e quelli sovrastanti di applicazione.

Il trasporto potrebbe avere molte SAP, quindi un pari che vuole comunicare deve indicare senza ambiguità tutte le proprie SAP e quelle che permetterebbero di arrivare all'altro. Lo stesso vale per ogni livello superiore. Una funzione possibile del trasporto è da un lato spezzare un dato e consegnarlo al Network e dall'altro lato ricevere i pezzi del Network ricomponendoli. L'obiettivo di tale livello è spedire nel canale dati corretti, affidabili e con certi tempi di risposta su richieste di livello S.
La modalità *connessa* è composta dalle primitive *connect, disconnect, data e data expedited,* dove i dati expedited sono soggetti ad un controllo di flusso separato che permette l'invio di messaggi di controllo anche se il servizio per i dati normali è bloccato.
Per quanto riguarda la modalità *non connessa* invece si ha solo la *connect* confermata e i vari parametri del servizio sono semplici.
Tale livello lavora in modo end-to-end.

4.10 Sessione

La sessione genera il supporto al dialogo, questo può essere bidirezionale, molteplice e strutturato e può avere garanzie di correttezza e affidabilità. Il servizio offerto dalla sessione si basa su un numero di unità funzionali divisi in un insieme di primitive. Il numero delle unità funzionali cresce per i livelli verso l'applicazione, il servizio di Sessione offre 58 primitive raggruppate in diverse unità funzionali (14). Si fa uso di connessione e QoS negoziata: ogni pari può chiedere il livello di servizio adatto alle sue esigenze.
Il dialogo si struttura in:

- azioni di controllo
- attività indipendenti gestibili
- eccezioni notificabili

Il livello Sessione offre servizi simili a quelli di altri livelli, ovvero:

- apre e termina connessioni
- trasferisce dati
- varie modalità di dialogo (half-duplex, full-duplex o simplex)
- sincronizzazione (uso di checkpoint, gestione delle eccezioni)

I punti di sincronizzazione sono dei nuovi dati. La sincronizzazione permette di intervenire sul dialogo e di creare dei punti di ripristino che possono essere di due tipi:

1. punto di sincronizzazione *maggiore*: ci si blocca in attesa di risposta (sincrono bloccante)
2. punto di sincronizzazione *minore:* non necessariamente deve essere confermato e il mittente può inviarne più di uno, la conferma di un punto conferma in automatico anche i precedenti.

Il numero di minori che si possono attendere prima di sospendersi è negoziabile e agisce come una *sliding windows* che scorre sui punti non confermati. A finestra piena di attende la conferma per continuare.
I punti di sincro di un dialogo possono servire in recovery per ritrovare uno stato significativo.
Le strategie di recovery previste sono:

- *abbandono:* reset della comunicazione
- *ripristino:* ci si riporta all'ultimo stato confermato da un maggiore
- *ripristino diretto dall'utente:* ci si riporta ad uno stato arbitrario senza controllo della conferma mancante.

La sessione struttura il dialogo mediante oggetti astratti detti *token,* da intendere come autorizzazioni: un solo utente possiede il token in ogni momento e ha diritto di uso sull'incisione di servizi di Sessione.
La S.CONNECT permette di negoziare i token che possono essere:

- *data token:* invio dati in half-duplex
- *release token:* richiedere terminazione
- *synchronize minor token:* creare punto di sincronizzazione minore
- *synchronize major token*: creare un punto di sincronizzazione maggiore

4.11 Presentazione

Il livello di Presentazione si preoccupa di uniformare la codifica dei formati dei dati fra i due pari, inoltre migliora la comunicazione con efficienza mediante la compressione dei dati e la sicurezza mediante la crittografia. Nell'esempio di Java, l'uso della JVM che uniforma gli ambienti, porta a non attuare trasformazioni a livello di presentazione.
Se non c'è accordo bisogna determinare un linguaggio comune. I dati possono viaggiare o come solo valori (efficiente) o come valore a sua descrizione (affidabile).
In caso di dati non omogenei si può:

1. dotare ogni ogni nodo con tutte le funzioni di conversione possibili per ogni tipo di dato
2. concordare un formato comune e ogni nodo possiede solo le funzioni di conversione da quel formato al proprio.

La prima soluzione porta ad elevare le performance ma è inattuabile mentre la seconda porta a un minor numero di funzioni da implementare.
Se c'è accordo i dati possono essere inviati anche con diversi gradi di ridondanza a seconda del costo della comunicazione, **se non c'è accordo** è necessario una notazione standard sulla quale accordarsi. L'accordo può essere ottenuto con protocolli detti di *negoziazione,* a molte fasi non predicibili, come il **bidding** (*contract net)* tra il sender e il receiver e prevede almeno cinque fasi, anche ripetute:

1. il sender manda la richiesta in *broadcast*
2. i receiver fanno l'offerta (bid)
3. il sender sceglie fra i bid
4. il receiver accoglie la risposta definitiva (*contract*)
5. si instaura l'accordo

Ci si può bloccare e ricominciare in diversi punti, flessibile ma costosa.
Il livello di presentazione fornisce due linguaggi: il primo: linguaggio astratto ASN.1 (*Abstract Syntax Notation*) è un linguaggio astratto di specifica usato per descrivere le funzioni di accordo fra entità che non si sono accordate in modo statico sulla rappresentazione. Non descrive solo dati ma è possibile descrivere anche in modo non ambiguo informazioni che servono a stabilire il contesto, il soggetto e la semantica di comunicazione. Si usa raramente.
L'altro linguaggio è il BER (*Basic Encoding Rules*) è un linguaggio concreto di descrizione dei dati. Stabilisce delle regole di codifica della rappresentazione standard dei dati che viene scambiata tra un nodo ed un altro. Viene usato sempre. Ogni dato deve viaggiare con una tripla Tag-Length-Value (tipo, lunghezza, valore).

4.12 Applicazione

Il livello di applicazione è quello che si interfaccia con l'utente e ha come obiettivo *l'astrazione*, nascondere le complessità dei livelli inferiori e coordinare il dialogo fra applicazioni distribuite. Si definiscono un insieme di servizi indipendenti dal sistema come il servizio di *directory (X.500),* il *file transfer (FTAM), terminale virtuale (VT)* etc…
Il servizio di nomi a direttorio X.500 consente di collocare e classificare ogni entità di interesse in un sistema gerarchico molto accessibile (7 giorni su 7, 24 ore su 24).

4.13 OSI Vs Internet

Confronto tra OSI e TCP/IP oltre al numero di livelli.

OSI	TCP/IP
Uso di Object-Orientation	protocolli e implementazioni insieme
Uso di interfacce	
Uso di implementazioni	
Progetto completo	implementazione solo dei prodotti
Interesse in standard	Progetto in crescita
Qualità di servizio	

Internet cerca si seguire le specifiche OSI.

Domande tipiche:

1. **Descrivere la struttura della comunicazione fra due pari.**
   La comunicazione avviene tra un mittente che la inizia e un destinatario che la accetta e la sostiene dello stesso livello, ovvero due pari. Ogni azione comporta un passaggio per tutti i livelli, da quello applicativo a quello fisico. Si riconoscono il SDU (*service data unit)* e il PDU (*protocol data unit*) che descrivono i messaggi che permettono di chiedere il servizio e di specificare il protocollo. Il PDU è formato aggiungendo informazioni al dato passato, fornito dal protocollo diventa SDU del livello sottostante. Le modifiche avvengono con il PCI *(protocol control information)*. In pratica ogni livello aggiunge proprie informazioni specifiche al dato ricevuto e le passa al SAP sottostante (mittente), viceversa nel ricevitore, non le aggiunge bensì le toglie.

2. **- Parla dei punti di sincronizzazione: a quale livello si trovano e qual'è il loro compito**
   - Cosa sono i punti di sincronizzazione maggiori e minori nel livello S di OSI? come si usano?
   I punti si sincronizzazione si trovano nel livello 5 della pila OSI, ovvero il livello di Sessione. Il suo principale compito è quello di generare il supporto al dialogo. Tra i servizi offerti dal livello di Sessione (oltre ad aprire e chiudere le connessioni e trasferire dati) vi è la *sincronizzazione* mediante l'uso di check point. Il suo compito è intervenire sul dialogo creando dei punti di ripristino che sono il maggiore che si blocca in attesa di risposta e quindi è di tipo sincrono bloccante, e il minore che invece non necessita di conferma e il mittente può inviarne più di uno, la conferma di uno conferma anche i precedenti. Quest'ultimo agisce su una sliding windows che scorre sui punti non confermati. Lo scopo di questi punti è quello di ritrovare uno stato significativo in una recovery.

3. **Pila ISO/OSI livello Presentazione.**
   Nella pila ISO/OSI il livello di presentazione si trova al layer 6 e fa parte dei layers applicativi. Si occupa della presentazione e trasformazione dei dati fra i due pari, migliora la comunicazione con efficienza mediante la compressione dei dati e la sicurezza mediante la crittografia. In caso di dati non omogenei si adottano due approcci, il primo dota ogni nodo di tutte le funzioni di conversione possibili per ogni tipo di dato aumentando le performance, il secondo invece concorda un formato comune così ogni nodo ha una sola funzione di conversione, dal proprio linguaggio a quello concordato ottenendo così minor numero di funzioni da implementare. L'accordo è ottenuto mediante protocolli di negoziazione e ha molte fasi tra cui il bidding tra sender e receiver.

4. **- Nel livello di presentazione OSI, quali sono le caratteristiche dei linguaggi BER e ASN.1 e per cosa vengono usati?**
   - Nel Livello OSI di presentazioni, cosa sono i due linguaggi previsti e opportuni definire? Sono usciti sempre in ogni caso?
   ASN.1 (Abstract Syntax Notation) è un linguaggio astratto di specifica usato per descrivere le funzioni di accordo fra entità che non si sono accordate in modo statico sulla rappresentazione. Non descrive solo dati ma è possibile anche descrivere in modo non ambiguo informazioni che servono a stabilire il contesto, il soggetto e la semantica di comunicazione. Usato raramente.
   BER (Basic Encoding Rules) è un linguaggio concreto di descrizione dei dati. Stabilisce delle regole di codifica della rappresentazione standard dei dati che viene scambiata tra un nodo ed un altro. Usato sempre. Ogni dato deve viaggiare con una tripla Tag-Length-Value (tipo, lunghezza, valore).

5. **Pila ISO/OSI livello Applicazione.**
   E' il livello che si interfaccia con l'utente e ha come obiettivo l'astrazione, nasconde le complessità dei livelli inferiori e coordina il dialogo fra le applicazioni distribuite. Si definiscono un insieme di servizi indipendenti dal sistema come il servizio di directory X.500 che è un servizio di nomi a direttorio che colloca e classifica ogni entità di interesse in un sistema gerarchico molto accessibile, il file transfer e il terminale virtuale.

5 Reti, proprietà e Routing

5.1 Introduzione

La rete è il mezzo di interconnessione punto a punto che consente la comunicazione fra diverse entità, viene misurata dal costo, della velocità e dall'affidabilità dell'invio dei messaggi.
I parametri per stabilire se una rete è buona o meno sono:

1. *tempo di latenza*: ritardi sulla comunicazione.
2. *banda*: dati trasmessi per unità di tempo.
3. *connettività*: tipo di interconnessione e topologia.
4. *costo degli apparati*.
5. *affidabilità*.
6. *funzionalità*: frammentazione, ricomposizione.

5.2 Topologia

Distinguiamo le reti in:

- **Dirette** (statiche): non variabili, regolari e di interconnessione
- **Indirette** (dinamiche): variabili e collegate ad un *bus* dinamico.

Si può pensare di connettere tutti con tutti (*interconnessione completa):* staticamente cresce in modo esponenziale il numero di connessioni grazie al bus dinamico. Con il bus unico invece, se viene occupato da qualcuno gli altri devono aspettare.
Lo switching (interconnessione dinamica) permette di dedicare le risorse a più richieste in tempi diversi e non di mantenerle allocate ad una entità fissa, quindi crea possibilità di condivisione e si contrappone all'uso esclusivo della risorsa di comunicazione.
Lo switching dei pacchetti può avvenire attraverso canali virtuali creati dinamicamente utilizzando risorse nei nodi intermedi, oppure essere libero senza connessioni di alcun tipo usando la frammentazione in datagrammi.

5.3 Circuit Switching

Fare circuit switching significa avere canali dedicati, statici e mantenuti anche se non in uso. Lo switching dei circuiti funziona mantenendo un canale end-to-end per un flusso atteso. Questa è una tecnica pessimista e statica per due motivi:

1. aumenta l'impiego di risorse anche senza un flusso con conseguente aumento dei costi
2. si usa il multiplex inverso: per un canale logico si possono impegnare anche N canali fisici.

Un esempio di CS è l'ISDN.

5.4 Circuiti Virtuali

Tecnica dinamica per creare dei canali virtuali fra i nodi tramite l'uso di *virtual circuit identifier* (**VCI**), che sono identificatori locali ai nodi intermedi e unici per la diversa connessione, in modo da migliorare l'uso del sistema di comunicazione, condividendo la connessione (tecnica ottimista).

5.5 Switching a datagrammi

Nessun canale ma solo datagrammi mandati fra nodi vicini. Ogni datagramma contiene il nome del ricevente e viene smistato in modo indipendente da altri. Si possono quindi creare percorsi diversi. Con i datagrammi non si garantisce nessun controllo di flusso e la QoS e possono essere introdotti molti ritardi.

5.6 Switching a pacchetti

Si inviano nella rete dei pacchetti tutti di dimensione fissa ottenuti dalla frammentazione di messaggi applicativi di dimensione variabile.

5.7 Signaling o Controlli

Se si deve gestire una connessione, si ha la necessità di crearla, mantenerla, garantirla e al termine chiuderla.
Per i controlli si possono inviare segnalazioni che viaggiano:

- *in-band*: con i dati sugli stessi cammini.
- *out-of-band*: su cammini separati per controllo e signaling.

Il signaling avviene su su tre piani differenti:

1. *user*: per i protocolli utente.
2. *controllo*: per il controllo della connessione.
3. *management*: per la stabilizzazione e la gestione del canale.

5.8 Tipi di Rete

Le reti possono essere molto differenti tra loro e le possiamo distinguere in:

- **Wide Area Network** (*WAN*): reti geografiche e coperture molto grandi
- **Metropolitan Area Network** (*MAN*): copertura di una città
- **Local Area Network** (*LAN*): reti di dimensione limitata e con forte limitazione sui partecipanti
- **Personal Area Network** (*PAN*): reti con dimensioni limitate, ad uso personale, riservate e ad-hoc (reti wireless).

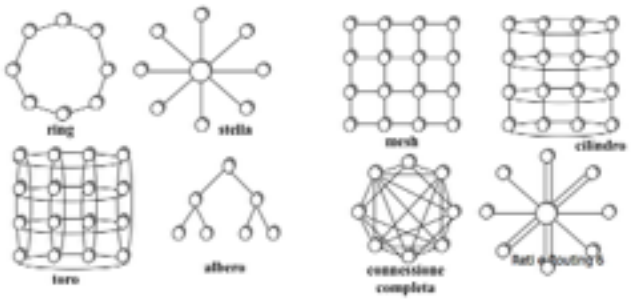
5.9 LAN

Sono reti locali caratterizzate da alta velocità e ampia banda con bassa probabilità di errori e broadcast facilitati. Possiamo distinguerle per *topologia, mezzo trasmissivo e controllo di accesso.*

Topologia:

- stella
- bus o insieme di bus
- anello
- hub, un bus inglobato in un'unica unità centrale di connessione simile al nodo centrale di una stella

Le LAN tendono ad evolvere verso la ridondanza per ottenere alte prestazioni.



Mezzo trasmissivo:

Queste reti hanno cablaggi semplici, alcuni esempi sono:

- *doppino*, schermato e non schermato
- *cavo coassiale*, sottile o grosso a seconda della banda (base o estesa)
- *fibra ottica,* con diversa propagazione

Un ulteriore mezzo trasmissivo usato per questo tipo di reti è il WIFI (IEEE 802.11b) che è un mezzo non cablato.

Controlli di accesso:

Tipici controlli di accesso sono:

- *CSDMA/CD* (Carrier Sense Multiple Access with Collision Detection): (Ethernet, standard 802.3) controllo standard, reattivo e non proattivo, in caso di bus impegnato si avverte la collisione e si ritrasmette dopo un tempo random.
- *token:* (standard 802.5) solo un possessore del token ha diritto di trasmettere, si forza il passaggio del token da un vicino all'altro dopo un intervallo.
- *slotted ring:* (standard 802.4) il messaggio gira nell'anello occupando slot vuoti, i nodi visti come dei contenitori.

5.10 Interconnessioni

Esistono diversi apparati che agiscono nei vari livelli OSI:

- *ripetitore*: rigenera un segnale fisico indebolito (L1-Fisico)
- *bridge*: collega reti diversi con capacità di separazione evitando congestioni (L2-Data Link)
- *router*: sistema per il passaggio da una rete all'altra con l'obiettivo di supportare il passaggio di messaggi e il routing (L3-Network)
- *protocol convert:* si collegano reti con protocolli diversi (dal Trasporto in su)

5.11 Bridge

Un bridge collega e separa due o più reti a livello di Data Link, tratta situazioni di errore, bufferizza i frame fra le reti, controlla le performance e l'affidabilità. Il buffer però rallenta e non è illimitato e il frame va trasformato con controllo da una rete ad un'altra. I bridge multiporta servono a collegare segmenti di rete, mentre quelli trasparenti lavorano come dei filtri per far passare solo i pacchetti che possono transitare, bloccando quelli che devono rimanere in locale. In questo caso si parla di *router isolato* e si deve prevedere una tabella di *forwarding* che può anche autocompilarsi in fase di learning: il bridge inizialmente lascia passare tutto e controlla i frame, poi si adegua e inizia a filtrare.
In situazioni in cui abbiamo più bridge questi si devono coordinare mediante il cosiddetto algoritmo di *spanning tree.* Questo algoritmo funziona scegliendo un algoritmo radice e scambiando messaggi con gli altri bridge si cerca di costruire un albero unico per le interconnessioni, ogni altro bridge così troverà il cammino più veloce per collegarsi alla radice.

5.12 Routing

I router risolvono il problema dell'interconnessione fra mittente e ricevente con il routing. Il routing deve essere tollerante ai guasti e alle variazioni, semplice, ottimale e giusto ovvero senza sovraccarichi. Il routing può essere *locale,* decisioni a bassa visibilità, o *globale,* con visuale su tabelle con tutte le possibili interconnessioni. Poi può anche essere *statico,* con cammini di propagazione fissi, o *dinamico* quando i cammini variano. Infine può essere *adattativo* se sfrutta dinamicamente risorse che si liberano o *non adattativo* diversamente.

5.13 Strategie di routing

Le strategie si classificano in base a:

- **Chi prende decisioni di routing:**
  1. La *sorgente:* il mittente decide l'intero o parte del cammino
  2. *hop-by-hop*: in modo indipendente ogni intermedio decide il prossimo passo del cammino
  3. *broadcast*: si invia ad ogni possibile ricevente e così via, strategia molto costosa e non ci sono decisioni per il mittente ma si consegna a tutti (eliminando duplicati)

- **Chi attua le decisioni di routing**
  Le decisioni sono attuate da agenti intermedi che possono essere:
  - *centralizzati*: si ha il controllo su tutta la rete quindi si fanno decisioni attuali, unico gestore.
  - *distribuiti*: diversi luoghi di controllo distribuiti che comunicano in modo più o meno coordinato
  - *coordinati*: con scambio continuo di informazioni
  - *isolati*.

- **Il momento delle decisioni di routing**
  Il routing può essere *statico* (fisso o deterministico) dove la strategia è fissa e non cambia o *dinamico* (adattativo), dove l'algoritmo evolve e si adatta al sistema mediante le informazioni che riceve: si superano così guasti non previsti.

5.14 Algoritmi di routing

GLOBALI

Questi algoritmi in caso di modifica delle tabelle propagano globalmente le informazioni.
Vediamone alcuni:

- **Shortest Path First di Dijkstra** e consiste nel far costruire ad ogni nodo un grafo completo delle interconnessioni e stabilisce le distanze con dei pesi; vengono poi determinate le distanze minime per ogni nodo e il traffico di routing segue la via più breve. Ha il vantaggio di ottimizzare le risorse ma lo svantaggio di avere un costo elevato di propagazione dei valori in caso di variazione (o creazione iniziale).
- **Spanning tree:** algoritmo statico, si identifica un albero fra tutti i nodi consentendo l'eliminazione di cicli e determinando cammini in modo globale.
- **Distance Vector:** algoritmo dinamico, ogni gateway mantiene una tabella con la sola distanza in passi e il primo passo d'uscita per il routing, le tabelle sono minime e consentono un instradamento statico facile.
- **Link state**: dinamico e multipath, ogni nodo mantiene tutto il grafo e limita la propagazione dell'informazione, le variazioni sono propagate in broadcast.

MULTIPATH

Ogni nodo mantiene una tabella con diversi possibili percorsi per uno stesso nodo. La scelta della route è random o con bilanciamento di carico, si ottiene così anche un'affidabilità in caso di guasti.

ADATTATIVI

- **Backward Learning:** ogni messaggio indica il mittente così permette agli intermediari di stimare le distanze e la topologia. In questo modo si permette di evitare situazioni di ciclo o livelock dove il messaggio si perde in passaggi inutili. Algoritmo molto costoso.

DINAMICI - ISOLATI

Indipendente dalla topologia si basano su informazioni locali o solo del vicinato, sono efficaci in quanto non ci sono costi di coordinamento e si limita l'overhead in caso di variazioni. Si rischia di introdurre però cicli e livelock in quanto manca visibilità.

- **Algoritmo Random:** ogni messaggio è smistato su una via d'uscita a caso non di input. Risulta ottimale in caso di sistema dinamico con numero infinito di nodi.
- **Algoritmo Patata Bollente**: il messaggio è smistato sulla via d'uscita più scarica. Il numero di passi per arrivare a destinazione non è predicibile e dipende dal traffico.
- **Algoritmo di Flooding:** il messaggio è smistato su tutte le code d'uscita del nodo. Si usano contatori per limitare i passi del messaggio e indicatori per evitare generazioni senza fine. Costo elevato per il mantenimento dello stato sui nodi.

5.15 Problema di routing

Analizziamo alcune situazioni critiche per il supporto al routing:

- **Congestione:** controllo dei buffer, si scartano messaggi successivi ad uno atteso, si prevede un numero massimo di messaggi circolanti.
- **Deadlock**: blocco critico dovuto all'impiego totale del buffer, può essere:
  - *avoindance*: si acquisiscono i buffer in ordine.
  - *prevention*: si mantengono buffer per far scambi in caso di saturazione.
  - *recovery*: si tratta il problema quando rilevato.
- **Livelock**: messaggi che permangono nel sistema senza giungere a destinazione (per esempio persi in cicli). Si può evitare ciò in due modi:
  - *a priori*: si mantiene il percorso e si evitano loop.
  - *a posteriori*: dopo un tot di passi si getta il messaggio (time to live).

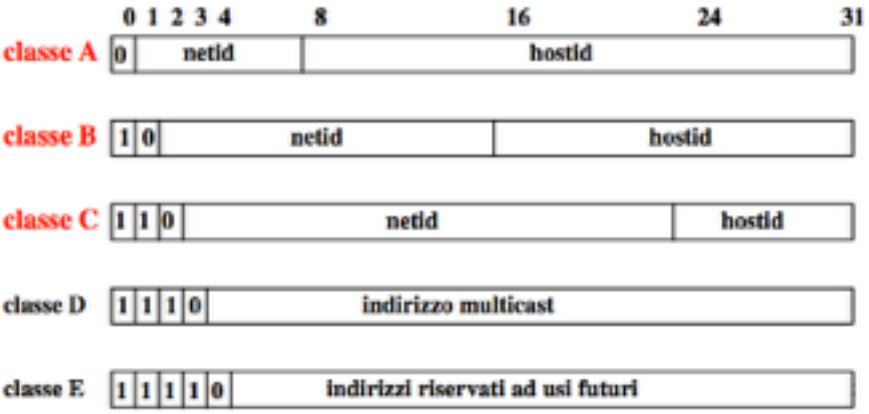
5.16 Livello Network - Indirizzi IP

Ogni host connesso ad una rete ha un indirizzo IP diviso in:

- *net id:* identificatore della rete.
- *host id*: identificatore dell'host connesso alla rete.

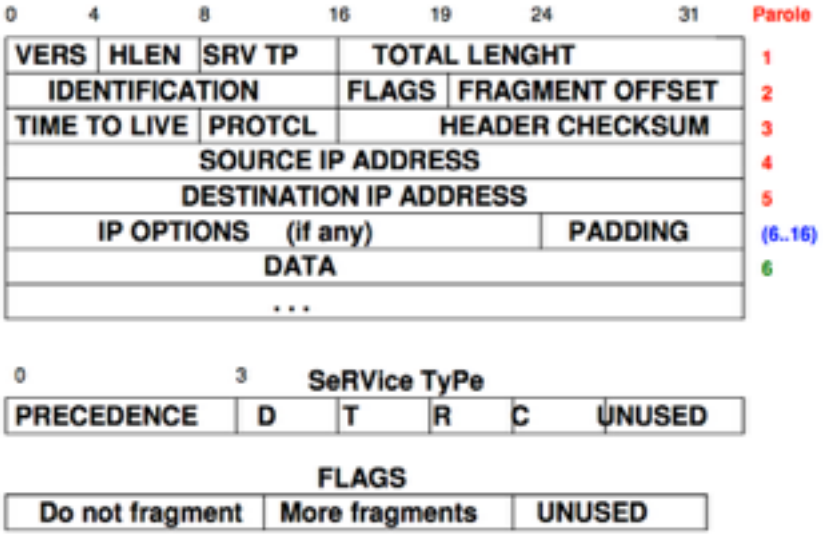
Host con diverse connessioni hanno più indirizzi.

La porta di rete è assegnata dal **Network Information Center** (*NIC*) in base a 3 classi primarie che determinano il numero possibile di nodi collegabili. Le WAN hanno classe **A** con molti nodi collegabili, le LAN invece hanno IP di classe **B** e **C**. Essendo sempre gli IP a 32 bit, se cala il numero di nodi aumenta il numero di reti.



5.17 Datagramma IP

Il datagramma IP è l'unità base che viaggia nella rete, è suddiviso in due parti principali: l'*header* e i *dati.* IP non specifica il formato dei dati ma l'header.



L'header ha minimo 20 byte e massimo 64 ed è composto da 5 parole da 32 bit:

1. *vers, hlen, srv tp, total lenght*: versione del protocollo, lunghezza dell'header, tipo di servizio e lunghezza totale.
   Il service type ha 3 bit di precedenza e il tipo di trasporto desiderato (**D** minimo ritardo, **T** massimo throughput, **R** massima affidabilità e **C** costo minimo).
2. *identification, flags, fragment offset*: identificatore per ricomporre il frame se scomposto, bit utilizzati per il controllo del protocollo e della frammentazione dei datagrammi, indica l'offset di un particolare frammento relativo all'inizio del pacchetto IP originale.
3. *time to live, prorcl, header checksum*: tempo di vita del datagramma, il tipo di protocollo superiore e checksum, ovvero un rilevatore di errori.
4. *source IP address*: indirizzo IP del mittente.
5. *destination IP address*: indirizzo IP del destinatario.
6. altre opzioni.

In generale IP è un protocollo senza QoS, best effort e a basso costo. I datagrammi viaggiano in modo autonomo anche frammentati.
IPv4 specifica il *servizio accoppiato a protocollo*. Dove il **servizio** indica cosa dobbiamo garantire ai livelli superiori invece il **protocollo** è la specifica di come lo si deve realizzare.

Il servizio risulta quindi:

- *connectionless*: pacchetti indipendenti, non c'è ordine e si possono seguire route diverse.
- *unrealible*: non ci sono controlli sulla ricezione del pacchetto.
- *best-effort*: l'inaffidabilità del trasferimento è dovuta all'esterno, non ci sono messaggi di errore.

Il protocollo invece:
Ogni nodo deve essere un router e quindi implementare l'instradamento, ha due funzioni principali:

- *elaborazione del messaggio*: incapsulamento e frammentazione dei dati dal livello superiore
- *routing*: instradamento dei pacchetti che consiste nel tradurre l'indirizzo logico IP a fisico di frame (**ARP**) oppure scelta della porta d'uscita in base al percorso.

5.18 Invio di datagrammi IP

Dal mittente al destinatario, ogni intermediario può agire sul messaggio operando frammentazioni in base al proprio **MTU** (M*aximum Transfer Unit*) di livello fisico. La decisione del MTU, ovvero la massima dimensione del datagramma, può essere presa:

- alla partenza: dell'MTU minimo della rete con tempi di trasmissioni lunghi

- passo passo: MTU fissato a 64 kbyte e pacchetti frammentati dai nodi se il mezzo fisico non è supportato. Il destinatario ricompone il datagramma usando ID e offset: se non si ricostruisce tutti il messaggio viene gettato via.

5.19 Opzioni

Alcune opzioni per il controllo e monitoraggio dei percorsi sono:

- *record route:* si genera una lista di massimo 9 IP dei gateway che il frame ha attraversato, ottenendo informazioni sui gateway intermedi attraversati.
- *timestamp*: si genera una lista dei tempi di attraversamento sugli intermedi, ottenendo un indicazione dei tempi di passaggio e della permanenza del datagramma nei gateway intermedi.
- *source route*: il sorgente fornisce indicazioni sul cammino da seguire nel routing del frame.
- *strict source*: il datagramma porta nella parte opzione una indicazione di tutti i gateway intermedi da attraversare
- l*oose source*: indicazione di un insieme di percorsi da attraversare non in modo contiguo ed unico

5.20 Famiglie di algoritmi di routing

Si conoscono due famiglie principali e sono globali e basate su tabelle:

1. distance vector: la tabella contiene solo la distanza in passi e il primo passo d'uscita per il routing.
2. link state: ogni nodo ha tutto il grafo e sono possibili molteplici cammini.

5.20.1 Distance Vector

La propagazione è a passi molto lenta: a regime si ha una tabella che contiene, per ogni gateway, la distanza in passi fino alla destinazione, ovvero il numero di gateway da attraversare, e il primo passo in uscita per il routing e quindi il primo gateway da attraversare. La propagazione risulta esponenziale al numero di nodi. Eventuali modifiche alla rete portano ad una scelta locale di ogni nodo di cambiare la tabella se conviene (numero di passi inferiori). La propagazione di variazioni può portare a situazioni non stabili ed è lenta. Periodicamente in modo asincrono i gateway vicini si confrontano.
Non tenendo traccia di chi fornisce una distanza da un nodo si può creare un loop che aumenta la distanza fino all'infinito, *counting-infinity,* limitandola a 16. A fronte di variazioni nella rete, i nodi si danno informazioni sbagliate l'uno sull'altro. Per evitare i problemi appena citati si usano delle strategie migliorative:

- split horizon: non si offrono cammini ai nodi da cui abbiamo attenuto informazioni (lenta convergenza del sistema)
- hold down: dopo la notifica di un problema si ignorano le informazioni di cammino per un certo intervallo per propagare errori (loop già creati non si risolvono e vengono mantenuti)
- split horizon con poisoned reverse e triggered broadcast: in caso di variazioni ogni nodo invia un broadcast con indicazione del problema e del cammino.

Da ciò si riscontrano ulteriori problemi: queste politiche generano fasi di broadcast in caso di variazioni e l'evoluzione della famiglia di algoritmi per privilegiare variazioni.

5.20.2 Link State

Ogni gateway ha una conoscenza completa del grafo: tiene sotto controllo le proprie connessioni e le verifica periodicamente. Quando si verifica un problema chi lo rileva invia il messaggio di variazione a tutti i partecipanti in broadcast o flooding. I problemi sono dovuti alle risorse per mantenere la topologia e le azioni costose per segnalare variazioni. I vantaggi sono la possibilità di scegliere cammini differenti e di conoscere i cammini completi. In caso di variazione ci sono problemi per protocolli dinamici: bisogna limitare i domini di conoscenza reciproca.
Flooding: protocollo di instradamento usato dai router che inoltrano un pacchetto in ingresso su tutte le linee ad eccezione di quelle da cui proviene. Ogni pacchetto in arrivo viene inoltrato su ogni linea di uscita eccetto quella da cui è arrivato.

5.20.3 Shortest Path First

Usato nei sistemi Link State per determinare i percorsi. Si ha il grafo con tutti i nodi e si vuole memorizzare il cammino minimo per giungere ad ogni nodo. L'algoritmo funziona in modo iterativo per passi: parte dal vicino e poi si allarga. Dati N nodi dopo N-1 cicli si raggiungono tutti i nodi e si trovano i percorsi minimi.

5.21 Routing Internet

Il routing di internet si basa sulla separazione delle reti e sulla loro interconnessione. L'indirizzamento può essere diretto solo su nodi della stessa rete altrimenti bisogna usare gateway che hanno almeno due IP, ovvero connessi su due diverse reti.

5.22 Sottoreti

All'interno di una rete si possono creare sottoreti non visibili fra loro se non con gateway interni, dall'esterno invece non sono visibili. Una connessione in una sottorete può comunicare direttamente solo con ogni nodo della sottorete e non con quelli di altre sottoreti. Per creare sottoreti si usa il meccanismo di subnet con le maschere. In questo modo si creano ulteriori suddivisioni di spazio di nomi IP (oltre le classi). All'interno di una rete i messaggi sono recapitati fra le sottoreti sempre con un routing. In internet il routing è gerarchico, esistono diverse aree locali di routing con algoritmi locali collegati fra loro da router di livello più alto.

5.23 Routing Globale

Si distinguono sistemi:

- *core:* gateways chiave con tabelle complete e replicate
- *non core*: informazioni di routing parziali e locali.

I nodi core si scambiano tutte le informazioni di routing con DV e LS. Per garantire scalabilità serve un sistema gerarchico con sottosistemi autonomi. Esistono gateway controllati da unità centrali che collegano reti le cui politiche di routing sono interne e non visibili, i sistemi autonomi scambiano informazioni di routing solo all'interno. La comunicazione interna è libera mentre quella con l'esterno deve essere predeterminata.

Si usano due protocolli:

- *External Gateway Protocol (EGP):* protocollo per gateway di controllo per raggiungere i core con struttura ad albero e il core situato in radice.
- *Interior Gateway Protocol (IGP):* protocollo per trovare percorsi interni di un sistema autonomo (multipath).

5.24 Routing Locale

Routing Information Protocol (RIP): i nodi *attivi* partecipano a determinare il percorso, mentre quelli *passivi* ascoltano. Si manda al vicinato la propria tabella locale e si aggiornano eventualmente le tabelle, dopo un timeout ogni cammino scade. Se un nodo non manda messaggi per un certo tempo si considera guasto. Questo tipo di routing è adatto alle piccole reti.

5.25 Routing IP - Non Globale

Il routing IP è l'instradamento, si capisce il destinatario dal datagramma in ingresso e con le tabelle di routing si decide dove mandarlo. E' basato su informazioni di rete e non di nodi.

Le sue proprietà sono:

- *routing statico*: stesso percorso, tutto il traffico passa per una rete
- *autonomia*: ogni gateway è autonomo.
- *visibilità*: solo il gateway finale comunica con il destinatario e verifica la sua operatività. Gli intermediari smistano e basta.
- *percorso di default*: se non si trova il nodo nella tabella lo si invia al gateway di default.

L'algoritmo usato è il seguente:

*dato un IP destinazione **se** la rete è direttamente raggiungibile **allora** invio il datagramma alla rete; **se** l'indirizzo ha un cammino proprio **allora** invio in base alla tabella; **se** l'indirizzo è nella tabella, tenendo conto anche le subnet, **allora** invio il datagramma al prossimo gateway **altrimenti** invio al gateway di default.*

Domande tipiche:

1. **Protocollo IP**
   Internet protocol è il protocollo utilizzato a livello 3 ovvero quello di Network, è un protocollo a pacchetto connectionless, quindi si basa su una semantica best-effort a scarsa affidabilità. I pacchetti che non raggiungono la destinazione vengono scartati, non avremo controllo di flusso e per questo motivo dovranno pensarci i protocolli a livello superiore (trasporto) magari con affidabilità come TCP. Il compito fondamentale è l'indirizzamento e l'instradamento tra sottoreti eterogenee. L'indirizzamento può essere diretto o indiretto, a livello network vengono usati IP address al fine di assegnare un piano di indirizzamento e quindi un percorso di raggiungibilità, ma principalmente identificare un dispositivo di rete. E' compito delle sotto-reti se la rete che viene attraversata è di transito, raggiungere l'ip di destinazione. Infine attraverso ARP si commutano gli indirizzi logici in fisici MAC.

2. **Parlare del Datagramma IP**
   Il Datagramma IP è l'unità di base che viaggia in internet, è suddiviso in due parti parti principali: i dati e l'header che a sua volta è costituito da 5 parole da 32 bit:
   1. *vers, hlen, srv tp, total lenght*: versione del protocollo, lunghezza dell'header, tipo di servizio e lunghezza totale.
   2. *identification*: identificatore per ricomporre il frame, se scomposto.
   3. *time to live, prorcl, header checksum*: tempo di vita del datagramma, il tipo di protocollo superiore e checksum, ovvero un rilevatore di errori.
   4. *source IP address*: indirizzo IP del mittente.
   5. *destination IP address*: indirizzo IP del destinatario.
   6. altre opzioni

   Non specifica il formato dei dati ma contiene solo le informazioni dei livelli superiori che verranno passati per incapsulamento al livello data-link attraverso la frammentazione. Essa può avvenire in maniera statica passando l'intero frame, oppure frammentando il datagramma riassemblandoli a destinazione.

3. **Parlare in generale del routing**
   Il router è il mezzo che si occupa di risolvere il problema dell'interconnessione fra mittente e ricevente mediante il routing. Il routing deve essere tollerante ai guasti e alle variazioni, semplice, ottimale e giusto ovvero senza sovraccarichi. Il routing può essere *locale,* decisioni a bassa visibilità, o *globale,* con visuale su tabelle con tutte le possibili interconnessioni. Poi può anche essere *statico,* con cammini di propagazione fissi, o *dinamico* quando i cammini variano. Infine può essere *adattativo* se sfrutta dinamicamente risorse che si liberano o *non adattativo* diversamente.

4. **- Descrivere le famiglie di routing Distance Vector e Link State.**
   - Quali sono le 2 principali famiglie di routing e cose le caratterizza rispetto alle altre.
   - Delineare le famiglie di routing per il livello di rete IP.
   Le 2 principali famiglie di routing sono Distance Vector e Link State, entrambe applicano algoritmi di routing dinamici.
   DV mantiene una tabella che contiene, per ogni gateway, la distanza in passi fino alla destinazione, ovvero il numero di gateway da attraversare, e il primo passo in uscita per il routing e quindi il primo gateway da attraversare. Ha il vantaggio di avere tabelle di dimensioni ridotte ed un instradamento facile e veloce, ma ha problemi in caso di cambiamenti e propagazione degli stessi e può creare dei count-to-infinity che possono essere contenute attraverso degli algoritmi quali: SPLIT_HORAZION o SLIPT_HORAZION CON POISION REVERSE.
   LS consiste nel far mantenere ad ogni nodo tutto il grafo della rete, in modo da limitare la quantità di dati da propagare tenendo sotto controllo le proprie connessioni. Il traffico segue il cammino più corto. I vantaggi sono quelli di rendere possibili cammini multipli (multi-path), sfruttando meglio le risorse e source routing. Le variazioni vengono propagate in broadcast (problemi di costo).

6 TCP/IP: protocolli e scenari di uso

6.1 Introduzione

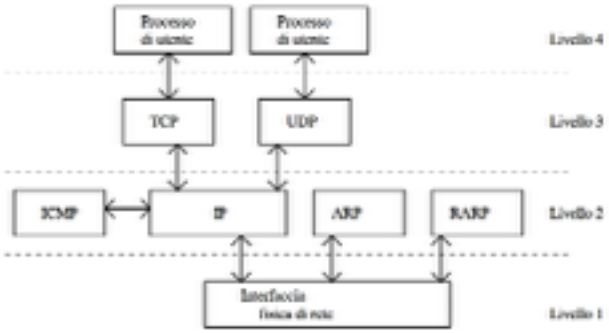
Internet nasce dall'idea di poter interconnettere tutte le reti in un'unica globalità con protocolli liberi e senza costi.

A livello di **Trasporto** troviamo i seguenti protocolli:

1. **Transmission Control Protocol - TCP**: canale virtuale best-effort, senza duplicati e con controllo del flusso.
2. **User Datagram Protocol - UDP**: scambio di messaggi end-to-end

A livello di **Network** invece troviamo:

1. **Internet Protocol - IP**: scambio di datagramma senza garanzia di consegna ai vicini
2. **Internet Control Message Protocol - ICMP**: scambio di messaggi di controllo
3. **ARP e RARP Protocol:** interazione con nomi di livello fisico.



6.2 Strutture di comunicazione

MAY-BE

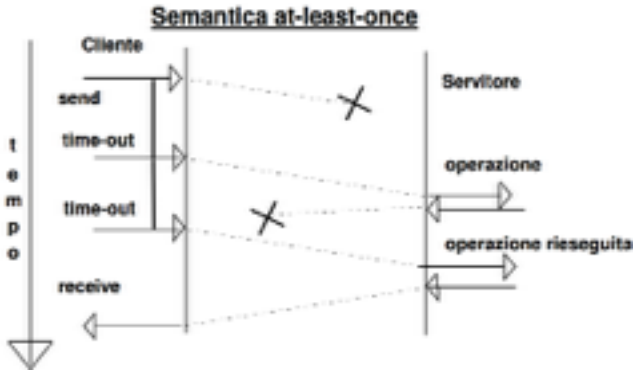
Semantica conosciuta anche come **Best-Effort,** un solo invio, può arrivare o meno. Internet è tutto best-effort per puntare all'efficienza: si veda IP e UDP che non guardano all'affidabilità. Questo è un tipo di standard che sacrifica la QoS.

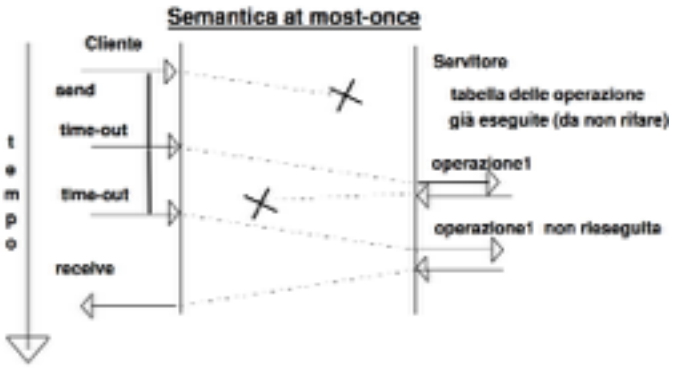
AT-LEAST-ONCE

Il messaggio deve arrivare e se ne tiene conto lato mittente. Sono previste ritrasmissioni allo scadere di timeout fino all'arrivo della risposta, al server possono anche arrivare diverse richieste identiche ma le serve senza accorgersene dei duplicati. Il cliente si preoccupa dell'affidabilità. Questa è una semantica adatta per azioni idempotenti.

AT-MOST-ONCE

Si ha ritrasmissione a lato client e uno stato a lato server che ricorda le richieste ricevute, se un'operazione è già stata eseguita, alla successiva stessa richiesta non viene rieseguito nulla ma si rinvia il primo risultato. Client e Server sono coordinati per avere garanzie di affidabilità. Lo stato fra i pari è mantenuto a tempo. Progetto reliable come per il caso del TCP.





EXACTLY-ONCE

Insieme di At-most-once e At-least-once. I pari lavorano entrambi per ottenere il massimo dell'accordo e della reliability. Tutto o niente, non c'è durata massima. Se va bene arriva uno e un solo messaggio e si riconoscono i duplicati, altrimenti sia client che server non sanno se il messaggio è arrivato o meno. Completo coordinamento, durata non predicibile: se uno dei due fallisce, l'altro deve aspettare il suo recovery.

6.3 Semantiche dei protocolli

IP e UDP usano la semantica *may-be,* TCP invece usa la semantica *at-most-once.* Non c'è garanzia di accordo sullo stato in caso di successo, si è decisa questa semantica per mantenere accettabile la durata delle operazioni e il carico dei protocolli.

6.4 Semantiche di trasporto

TCP è un protocollo connesso con connessione bidirezionale, con dati differenziati ovvero quelli normali e quelli prioritari, con controllo del flusso byte cioè si ha un ordine corretto dei byte con eventuale ritrasmissione di quelli persi, con controllo del flusso e quindi del buffer e con una semantica at-most-once. L'obiettivo di tale protocollo è quello di consentire una durata limitata e di avere eccezioni trasparenti.

6.5 Azioni di gruppo TCP/IP

A livello globale non sono consentiti *broadcast* vista la dimensione del sistema e per evitare elevati costi, ma sono permessi solo a livello locale.
Un broadcast può essere:

- *limitato:* su tutta una rette locale di un nodo
- *diretto*: su tutti i nodi di una rete specificata, broadcast trasmesso sulla rete internet fino alla rete bersaglio.

Per quanto riguarda il *multicast* esistono indirizzi di classe **D** sul quale tutti i nodi registrati possono ricevere e spedire messaggi di gruppo, i protocolli sono ancora in via sperimentale.

6.6 Address Resolution Protocol - ARP

Abbiamo bisogno di strumenti in grado di risolvere i nomi fisici degli host a partire dal numero di IP e viceversa. **ARP** è un protocollo locale basato su broadcast, funziona inviando, in broadcast appunto, un IP, il nodo con quell'IP, che sarà unico, risponde al server ARP con il suo indirizzo fisico. Si deve tener conto dei costi quindi si crea una memoria cache delle associazioni risolte e prima di inviare il broadcast, si cerca nella tabella. Per creare un servizio efficiente si ottimizza il broadcast:

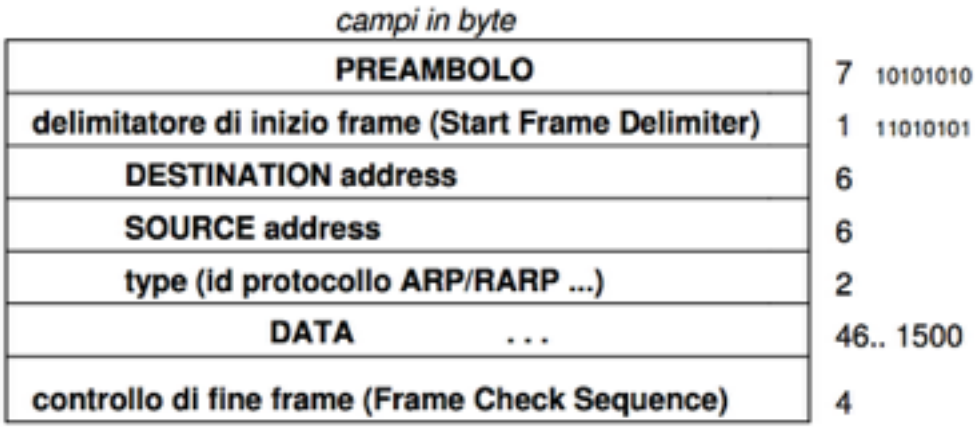
- l'associazione relativa alla macchina richiedente viene memorizzata anche dalla macchina che risponde ad ARP
- ogni richiesta broadcast viene memorizzata da tutti
- ogni nuova macchina al collegamento invia sulla rete locale un broadcast con la propria coppia indirizzo fisico - indirizzo IP

Ogni nodo realizza due ruoli distinti nel protocollo:

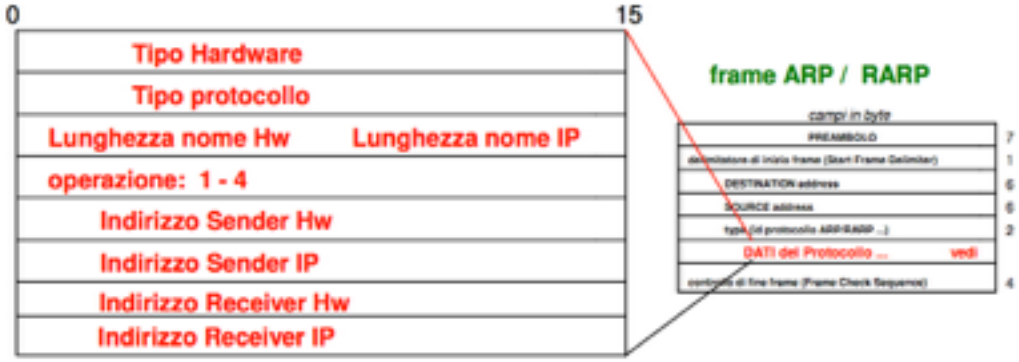
- uno *attivo*: cerca di risolvere l'indirizzo localmente se no manda il messaggio in broadcast
- uno *passivo*: risponde alle richieste di altri.

Il messaggio di ARP viene incapsulato in frame di livello fisico e rinviato richiedente.

6.7 Frame Data Link - Ethernet/MAC



Ethernet è lo standard a livello MAC (Medium Access Controll), anche gli altri frame di tale livello, MAC, hanno la stessa forma, in generale sono composti da preamboli di sincronia, delimitatori iniziali e finali, controlli come il CRC e gli indirizzi sono a 6 byte. Nei frame per l'invio di pochi dati c'è molto overhead (per 1 byte, overhead di 46 byte). L'header dei protocolli **ARP** e **RARP,** considerati identici, è contenuto nella zona **DATA** del frame:



6.8 Reverse Address Resolution Protocol - RARP

Visto il protocollo ARP, vediamo il suo duale, il protocollo **RARP**. Il protocollo funziona ricercando l'indirizzo IP per nodi che conoscono solo il proprio **MAC** (indirizzo fisico) questo perché gli indirizzi IP solitamente sono memorizzati nel disco fisso di una macchina e recuperati dal SO, ma in caso di macchine diskless necessita il supporto del RARP. Un client invia un broadcast di data link per raggiungere il server RARP che gli invierà la risposta. Su una rete spesso ci sono diversi server RARP che però non devono interferirsi nelle risposte:

- modello *dinamico*: alla prima richiesta risponde il primario, alle successive gli altri anche in gerarchia.
- modello *statico*: risposte con ritardi, prima risponde immediatamente il server primario, dopo tempi random invece rispondono gli altri abbassando la probabilità di risposte simultanee.

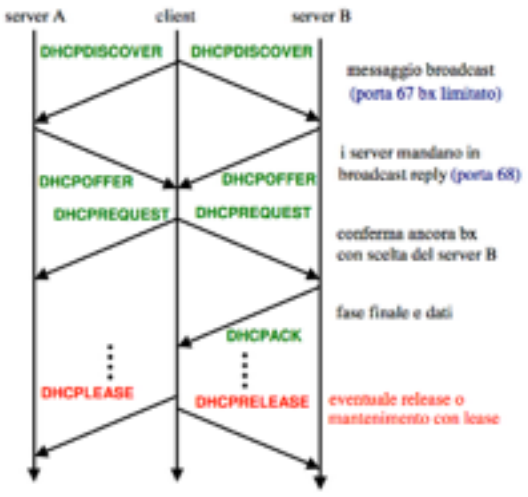
6.9 Dynamic Host Configuration Protocol - DHCP

E' un protocollo per l'attribuzione dinamica di numeri IP in una rete. Si cerca di risparmiare numeri IP, assegnandoli solo al momento del bisogno. Si basa su un sistema a C/S con un protocollo d'offerta tipo asta (bidding) con iniziativa del client:

- il client manda un pacchetto chiamato DHCPDISCOVER in broadcast per richiedere un IP sulla rete dove è collegato
- i vari server DHCP ricevono la richiesta e inviano, se lo vogliono, un'offerta DHCPOFFER
- il client aspetta di ricevere le varie offerte per un tempo stabilito e dopo ne sceglie una inviando, sempre in broadcast, un pacchetto DHCPREQUEST con il server selezionato
- il server che vede il suo indirizzo nel pacchetto conferma l'offerta con un DHCPACK
- gli altri server vengono automaticamente informati che la loro offerta non è stata scelta dal client

L'attribuzione del contratto è a tempo e se non viene usato può anche essere revocato e riassegnato dal server. Prima del rilascio si può riconfermare l'uso con un messaggio di base, per non rieseguire il protocollo di bidding. DHCP usa il protocollo UDP con porte 67 per il server e 68 per il client.
Successivamente avremo delle fasi di LEASE o RELEASE, cioè verranno aggiornati gli indirizzi che non saranno utilizzati, questo per un risparmio di risorse.
Il protocollo DHCP viene usato anche per assegnare all'host che richiede l'IP diversi parametri necessari per il suo corretto funzionamento sulla rete a cui è collegato. Tra i più comuni, oltre all'assegnazione dinamica dell'indirizzo IP, si possono citare:

- Maschera di sottorete
- Default gateway
- Indirizzi dei server DNS
- Nome di dominio DNS di default
- Indirizzi dei server NTP
- Indirizzo di un server tftp e nome di un file da caricare per calcolatori che caricano dalla rete l'immagine del sistema operativo (si veda Preboot Execution Environment).



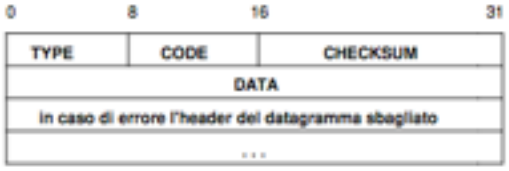
6.10 Network Address Translation - NAT

Usato per traslare indirizzi privati intranet in indirizzi IP pubblici globali. Si usano appositi router NAT che utilizzano tabelle di traslazione. Per rendere il servizio più flessibile esiste il NAT a porte che permette di mappare molti indirizzi interni privati su un unico indirizzo esterno pubblico, i diversi indirizzi interni sono determinati da diverse porte di connessione (TCP) che fungono da tag del nodo. Così si limita l'uso di indirizzi esterni.

6.11 Internet Control Message Protocol - ICMP

E' un protocollo che si occupa della gestione e del controllo degli IP per migliorare la qualità best effort. Serve per coordinare le entità di livello IP e a inviare messaggi di controllo o errore ai mittenti. Mentre i nodi intermedi non sono informati dei problemi i nodi mittenti (o sorgente) possono provare a correggerli.
I messaggi ICMP sono incapsulati in datagrammi IP, seguono il normale routing, non hanno priorità e non hanno canali dedicati. Possono quindi creare *congestione* che però è evitabile non mandando messaggi ICMP d'errore in risposta ad errori nei ICMP.

Il messaggio ICMP è incapsulato nell'IP, contiene sempre l'header e i primi 64 bit dell'area dati del datagramma che ha causato il problema. I suo header prevede un checksum e i campi type e code che indicano al mittente la causa del messaggio, in particolare, il campo code identifica il codice di errore.



Possibili valori del campo type

0	*Echo Reply*
3	**Destinazione irraggiungibile**
4	**Problemi di congestione (*source quench*)**
5	Cambio percorso (*redirect*)
8	*Echo Request*
11	**Superati i limiti di tempo del datagramma**
12	**Problemi sui parametri del datagramma**
13	Richiesta di timestamp
14	Risposta di timestamp
15	Richiesta di Address mask
16	Risposta di Address mask

Possibili valori del campo code

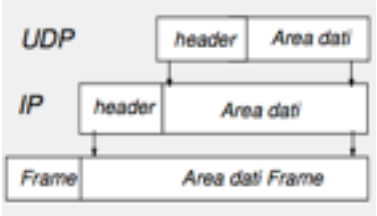
0	Rete irraggiungibile
1	Host irraggiungibile
2	Protocollo irraggiungibile
3	Porta irraggiungibile
4	Frammentazione necessaria
5	Errore nel percorso sorgente (source route fail)
6	Rete di destinazione sconosciuta

ICMP permette l'invio di informazioni di routing tra gateway:
- controllo di raggiungibilità di un sistema (echo request / reply)
- richiesta di maschere
- sincronizzazione degli orologi
- cambi di percorso

Il protocollo viene gestito mediante alcuni strumenti:
- comando *ping* per stimare il RoundTripTime mediante l'invio di echo request
- comando *traceroute* per visualizzare il percorso fra due nodi: si mandano messaggi con time-to-live crescente, il nodo su cui scade il time-to-live lo scarta mandando un ICMP al mittente. Il mittente registra chi manda gli ICMP e può ricostruire il cammino.

6.12 User Datagram Protocol - UDP

Protocollo di trasporto besteffort a basso costo. Per distinguere i diversi processi attivi su un nodo, li identifica con un numero di porta. I messaggi UDP sono molto ridotti e con pochissimo overhead non fornendo un servizio a connessione e affidabile, i datagrammi possono essere disordinati, persi o duplicati.



Gli User Datagram non sono frammentati a livello IP nonostante sono interamente contenuti all'interno dell'area dati di quest'ultimo. La driver UDP si preoccupa di smistare in entrata e in uscita i diversi messaggi provenienti dai diversi processi sulle varie porte. Dietro alle prime 1024 porte ci sono servizi standard statici, oltre invece fino a 64 kbyte l'assegnamento è libero e avviene con binding dinamico.

Quindi ogni programma ha quindi una porta (o più porte) per inviare/ricevere datagrammi. Si distinguono due diverse decisioni sulle porte:

1. *multiplexing:* messaggi da più processi applicativi paralleli con un solo servizio IP
2. *demultiplexing*: lo stesso messaggio recapitato alla porta corretta.

6.13 TCP - Entità e connessioni

Il TCP permette la connessione fra processi di nodi distinti, tale connessione è data dalla quadrupla {*hostIP1, port1; hostIP2, port2}* ed è univoca. Ogni nodo e ogni porta possono ospitare diverse connessioni insieme, ma nel complesso le quadruple devono essere diverse. Come per l'UDP, il TCP ha determinate porte dinamiche, *wellknow,* e fa il multiplexing/demultiplexing fra i processi. Le porte TCP non sono le stesse dell'UDP.

6.14 ARQ

Per ottenere QoS dobbiamo trovare il compromesso fra *affidabilità,* attendere l'avvenuta ricezione e *asincronismo* ovvero non aspettare per troppo tempo.
Automatic Repeat-reQuest è una strategia di controllo di errore, che svolge il compito di rivelare un errore (ma non di correggerlo). I pacchetti corrotti vengono scartati e viene richiesta la loro ritrasmissione.
Affinché il sistema sia capace di riconoscere i messaggi corrotti è necessario che questi vengano preliminarmente codificati da un codificatore. Dopo la trasmissione il decodificatore decodifica il messaggio e, a seconda che questo sia integro o meno, si comporta in base a uno dei 3 diversi protocolli più comuni:

- *Stop-and-wait:* il mittente invia un messaggio e attende dal destinatario una conferma positiva (ACK, acknowledge), negativa (NACK, contrazione di negative acknowledge) o un comando; se scade il tempo di attesa (time-out) per uno di questi tre, il mittente provvederà a rispedire il pacchetto e il destinatario si incaricherà di scartare eventuali repliche. Nel caso in cui si verificasse un errore nella trasmissione del segnale di conferma (ACK), il mittente provvederà a rinviare il pacchetto; il destinatario riceverà in questo modo una copia del pacchetto già ricevuto, credendo che gli sia pervenuto un nuovo pacchetto. Per ovviare a questo problema si può procedere numerando i pacchetti trasmessi, ovvero inserendo un bit di conteggio.
- *Go-Back-N:* il mittente dispone di un buffer dove immagazzina N pacchetti da spedire, man mano che riceve la conferma ACK svuota il buffer e lo riempie con nuovi pacchetti; nell'eventualità di pacchetti persi o danneggiati e scartati avviene il reinvio del blocco di pacchetti interessati. I pacchetti ricevuti dal destinatario dopo quello scartato vengono eliminati.
- *Selective Repeat:* in questo caso anche il destinatario dispone di un buffer dove memorizzare i pacchetti ricevuti dopo quello/quelli scartati; quando i pacchetti interessati vengono correttamente ricevuti, entrambi i buffer vengono svuotati (mittente) o i pacchetti contenuti salvati (destinatario).

6.15 Continuous Request

Si supera la sincronia e si mandano un certo numero di messaggi pari alla dimensione della finestra diversi in modo ripetuto. Il mittente mantiene una finestra di messaggi di una certa dimensione e si blocca solo quando è piena. Quando arriva un ACK di conferma il messaggio è tolto dalla finestra ed essa scorre. Il ricevente passa i messaggi al livello superiore in ordine. E' un protocollo più complesso rispetto a ARQ e le conferme sono con overhead per full-duplex, gli ACK invece sono mandati in piggybacking sul traffico opposto. In caso di errore o di messaggi saltati si attuano due strategie:

1. *selective retransmission*: attesa dell'esito dei messaggi tenendo conto degli ACK ricevuti e anche ACK negativi (dovuti al time-out del ricevente) e ritrasmissione di quelli persi.
2. *go-back-N:* ritrasmissione di un gruppo di messaggi se solo uno ha problemi, il ricevente svuota il buffer.

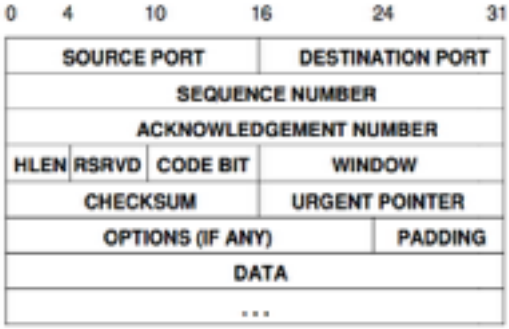
6.16 Transmission Control Protocol - TCP

Il protocollo TCP fornisce un servizio di trasmissione affidabile basato su:

- reliable stream full-duplex
- canale bidirezionale

La connessione è fra i due end user e non impiega i nodi intermedi, lo stream è ordinato anche se esistono due tipi di dati:

- dati normali
- dati prioritari, a banda limitata, con segnalazione della loro presenza nel flusso



L'header del TCP è costituito da 20 byte ed è composto da 5 parole:

1. contiene le porte del mittente e del destinatario
2. numero di sequenza
3. numero di ACK
4. lunghezza dell'header, della finestra e il code bit
5. checksum e puntatore urgente
6. varie opzioni

Nella quarta parola sono presenti i code bit, i quali occupano 6 bit, e se ne conoscono 6:

1. **URG**: c'è un dato urgente segnalato da ogni header del flusso e puntato dall'urgent pointer
2. **ACK**: nel campo relativo c'è un ACK significativo
3. **PUSH**: richiesta di non bufferizzare e quindi di invio immediato
4. **RST**: problemi nella connessione e quindi necessità di riavviare la connessione
5. **SYN**: si stabilisce la connessione
6. **FIN**: si inizia la fase di chiusura lato mittente

Il TCP può spezzare i messaggi in segmenti di dimensione variabile, può anche raggrupparli per ottimizzare:

- se troppo corti possibili overhead
- se troppo lunghi rischio di altra frammentazione a IP e possibili perdite.

TCP per comunicare usa il protocollo Continuos Request. Essendo un traffico full-duplex (nei due sensi), gli ACK sono inviati in piggybacking, ovvero inglobati nel flusso opposto. Si usa la *sliding window* espressa in byte, determinata dal ricevente e comunicata per ogni invio, il mittente invia fino a saturare la sua finestra poi si ferma attendendo gli ACK di avvenuta consegna dal destinatario. Se si confermano i segmenti si scorre e si continua il flusso, altrimenti si rinviano; gli ACK possono giungere in modo non ordinato.
Quando invece non c'è stata ricezione del segmento, TCP usa il protocollo Go-Back-N. Il ricevente può scartare segmenti successivi e attendere il mancante, il mittente invece deve rimandare quello che manca. In realtà il ricevente cerca di non buttare i successivi ma di mantenerli e poi integrarli.
TCP usa gli ACK cumulativi, ovvero la conferma di un segmento conferma anche tutti i precedenti, quindi dice poco sullo stato di chi li sta ricevendo. Non cambia molto a livello mittente che con un ACK capisce solo che alcuni sono confermati e altri ancora no (magari ne manca uno e sono arrivati i successivi, ma non c'è ACK cumulativo per via del buco); si procede ancora con rinviare tutti, anche quelli ricevuti dal ricevente (Go-Back-N). Altrimenti si può provare a mandare solo il primo e vedere se viene riempito il buco, in questo caso arriverà l'ACK cumulativo anche di quelli successivi già ricevuti.
TCP conta di tre fasi di operatività:

1. fase *iniziale*: three-way-handshaking, si stabiliscono dei parametri operativi per la connessione e si prepara l'avvio.
2. fase di *comunicazione*: si parte da un transitorio iniziale e si raggiunge quello di regime.
3. fase *finale*: chiusura in modalità mono o bidirezionale.

Analizziamo nel dettaglio le tre fasi:

INIZIALE:
Il mittente attua un protocollo di tipo *three-way-handshaking* per realizzare la connessione. Per coordinare mittente, A con ruolo attivo, e ricevente, B con ruolo passivo, si attuano tre fasi di comunicazione:

1. A invia a B un segmento con SYN nell'header e un numero casuale X che indica il valore iniziale del flusso scelto da A, richiedendo la connessione.
2. B riceve e invia a sua volta un SYN con un nuovo numero di sequenza casuale Y e un ACK con X+1 per dare conferma
3. A riceve SYN+ACK e conferma Y con un ACK Y+1

I valori di X e Y di sincronia sono casuali e non sono pari a 0 così non si rischia di avere errore con vecchi segmenti. In questa fase di studio, le due entità si accordano anche sulla dimensione del blocco di dati massimo (MSS), sulle dimensioni della finestra e il coordinamento degli orologi (tempi di trasmissione e risposta, timeout…). La corretta impostazione del timeout è difficile ma molto importante, in quanto un timer troppo breve comporta ritrasmissioni inutili (il timer scatta mentre il riscontro o il pacchetto sono ancora in viaggio), mentre un timer troppo lungo comporta attese in caso di effettiva perdita di pacchetti. Tale intervallo dovrà essere almeno pari al RTT (Round Trip Time) cioè al tempo di percorrenza a due vie di un

pacchetto per tornare al mittente sotto forma di ACK, tale tempo varia in modo casuale così il protocollo tende ad aggiustare il timeout basandosi su una stima di tale RTT.

FINALE:
Come per le socket la chiusura della connessione in questo protocollo può avvenire in due modi:

1. *three-way-handshaking*: istantanea per entrambi i canali, corrisponde alla close() delle socket.
2. *four-way-handshaking*: chiusura di un solo canale per volta, corrisponde alla shoutdown() delle socket.

La prima funziona come per la connessione, dove il segmento SYN viene sostituito con FIN.
La seconda chiusura, detta *graceful,* avviene invece in quattro fasi: viene chiuso il canale di output, anche se non del tutto in quanto è ancora permesso il flusso di ACK di controllo, quindi rimane aperto l'altro rimanendo in ricezione di altri dati del pari.
Nel dettaglio abbiamo:

1. A invia un segmento FIN e numero di sequenza X
2. B riceve FIN e conferma con ACK di X+1
3. il canale che va da B ad A non è chiuso quindi B può mandare dati che sono confermati da A con appositi ACK. Quando B vuole chiudere manda un FIN e il numero di sequenza Y
4. A riceve e conferma la chiusura con ACK di Y+1

La gestione di eventi anomali come fallimenti ripetuti, mancanza dal server, avviene con l'invio di un segmento di reset (RST) per tentare il ripristino o chiudere tutto.

COMUNICAZIONE:
Nella fase del transitorio **iniziale** si deve evitare di congestionare la rete, quindi una volta stabilita la connessione, il flusso dei dati parte lentamente e si porta a regime dopo un certo tempo, per verificare di andare alla velocità che non congestiona la rete. Quindi anche grandi quantità di dati subito disponibili vengono trasmesse gradualmente.
Nella fase di **regime** dopo un primo calcolo del timeout principale o *Round Trip Time* (RTT), nella fase di connessione, questo viene ricalcolato (tenendo presente la storia dei RTT precedenti) per ogni segmento arrivato con successo e senza ritrasmissione. Il timeout principale è multiplo di 100 ms.
Tra mittente e ricevente esistono altri intervalli di tempo derivati da quelli principali:

- il ricevente cerca di sfruttare il piggybacking per gli ACK ma usa un timeout per limitare i tempi di ritardo, dopo tale tempo viene inviato un ACK apposito.
- il mittente e il ricevente cercano di accumulare messaggi per inviare segmenti di MSS (dimensione media negoziata), ma dopo un timeout si invia ugualmente il dato corto per limitare i ritardi.
- entrambi gli end-point se non c'è traffico si mandano messaggi per verificare lo stato della connessione, ad intervalli dettati da un timeout.

La fase di comunicazione deve garantire:

- la consegna ordinata ed eliminazione di duplicati
- il riscontro dei pacchetti e ritrasmissione.
- il controllo del flusso
- il controllo della congestione

Il Sequence number si occupa di consegnare ordinatamente i pacchetti ed eliminare i duplicati. Serve a identificare e posizionare in maniera ordinata il carico utile del segmento TCP all'interno del flusso di dati. Con la trasmissione tipica a commutazione di pacchetto della rete dati infatti ciascun pacchetto può seguire percorsi diversi in rete e giungere fuori sequenza in ricezione e quindi in ricezione TCP si aspetta di ricevere il segmento successivo all'ultimo segmento ricevuto.
In relazione al numero di sequenza TCP ricevente attua le seguenti procedure:

- se il numero di sequenza ricevuto è quello *atteso* invia direttamente il carico utile del segmento al processo di livello applicativo e libera i propri buffer.
- se il numero di sequenza ricevuto è *maggiore* di quello atteso deduce che uno o più segmenti ad esso precedenti sono andati persi, ritardati dal livello di rete sottostante o ancora in transito sulla rete. Pertanto, memorizza temporaneamente in un buffer il carico utile del segmento ricevuto fuori sequenza per poterlo consegnare al processo applicativo solo dopo aver ricevuto e consegnato anche tutti i segmenti precedenti non ancora pervenuti, aspettandone l'arrivo fino ad un tempo limite prefissato (time-out). All'istante di consegna del blocco ordinato di segmenti tutto il contenuto del buffer viene liberato.
- se il numero di sequenza ricevuto è *inferiore* a quello atteso, il segmento viene considerato un duplicato di uno già ricevuto e già inviato allo strato applicativo e dunque scartato permettendo l'eliminazione dei duplicati di rete.

Per quanto riguarda il riscontro dei pacchetti e la ritrasmissione ad occuparsene è il Acknowledgment Number o numero di riscontro che per ogni segmento ricevuto in sequenza TCP ne invia uno. Il numero di riscontro presente in un segmento riguarda il flusso di dati nella direzione opposta. In particolare, il numero di riscontro inviato da A a B è pari al numero di sequenza atteso da A e, quindi, riguarda il flusso di dati da B ad A.
In particolare il protocollo TCP adotta la politica di conferma cumulativa, ovvero l'arrivo di numero di riscontro indica al trasmittente che il ricevente ha ricevuto e correttamente inoltrato al proprio processo applicativo il segmento avente numero di sequenza pari al numero di riscontro indicato ed anche tutti i segmenti ad esso precedenti. Per tale motivo TCP lato trasmittente mantiene temporaneamente in un buffer una copia di tutti i dati inviati, ma non ancora riscontrati: quando questi riceve un numero di riscontro per un certo segmento, deduce che tutti i segmenti precedenti a quel numero sono stati ricevuti correttamente liberando il proprio buffer dai dati. La dimensione massima dei pacchetti riscontrabili in maniera cumulativa è specificata dalle dimensioni della cosiddetta finestra scorrevole.
Per evitare tempi di attesa troppo lunghi o troppo corti per ciascun segmento inviato TCP lato trasmittente avvia un timer, detto timer di ritrasmissione RTO (Retransmission Time Out): se questi non riceve un ACK di riscontro per il segmento inviato prima che il timer scada, TCP assume che tutti i segmenti trasmessi a partire da questo siano andati persi e quindi procede alla ritrasmissione.
Si noti che, in TCP, il meccanismo dei numeri di riscontro non permette al ricevitore di comunicare al trasmettitore che un segmento è stato perso, ma alcuni dei successivi sono stati ricevuti (meccanismo ad Acknowledgment Number negativi), quindi è possibile che per un solo pacchetto perso ne debbano essere ritrasmessi molti. Questo comportamento non ottimale è compensato dalla semplicità del protocollo. Questa tecnica è detta Go-Back-N.

In questa fase vi è un importante controllo che è quello del **flusso**, fondamentale data la presenza di macchine molto diverse tra di loro.
L'obiettivo di tale controllo è evitare che il mittente invii una quantità eccessiva di dati che potrebbero, in alcune situazioni, mandare in overflow il buffer di ricezione del destinatario generando una perdita di pacchetti e la necessità di ritrasmissione, risultando dunque particolarmente utile per il mantenimento delle prestazioni della connessione.
I meccanismi fondamentali di coordinamento sono:

- la *dimensione preferenziale:* si deve cercare di inviare segmenti MSS (Maximum Segment Size) per ottimizzare mediante l'uso di buffer.
- *finestre*: la dimensione della finestra viene inviata con ogni segmento e indica al pari il suo stato di memoria in byte. Se la finestra è pari a 0 si chiede di bloccare il flusso.

Per avere un buon flusso si deve evitare di avere finestre troppo piccole e non inviare segmenti troppo corti.
L'algoritmo di **Nagle** serve per evitare di spedire segmenti corti: se c'è un segmento non confermato, prima di spedirne altri si cerca di arrivare al MSS.
Le applicazioni possono anche cercare di superare la trasparenza di TCP, usando il segmento con code bit:

- **PUSH:** il segmento è inviato immediatamente e portato all'applicazione il prima possibile.
- **URG**: se ne segnala la posizione nel flusso, e il ricevente deve consumare i dati per arrivare quanto prima al byte urgente.

A regime ogni segmento produce coordinamento con il pari su:

- posizione nel flusso
- dimensione della finestra
- timeout

La connessione diventa critica in caso di **congestioni** in quanto non si riescono più a consegnare dati in tempo utile rispetto lo stato attuale. Il problema può essere non solo dovuto agli end-point ma può anche riguardare l'intero sistema a causa dei buffer dei router pieni. La congestione è identificata allo scattare ripetuto di un timeout, anche se questo scatta solo una volta. Per limitare i danni si attuano subito azioni di recovery, il mittente dimezza la finestra d'invio e raddoppia i timeout, finita la congestione si va gradualmente (*slow start)* a regime. Slow start è lo stesso principio in avvio per evitare congestioni iniziali.
Il controllo della congestione avviene mediante l'uso di alcuni strumenti:

- **rwnd:** receiving window, finestra segnalata dal ricevente corrispondente al controllo del flusso.
- **cwnd**: congestion window del mittente che parte da MSS e cerca di allargarsi fino alla rwnd
- **ssthresh**: la soglia si slow start

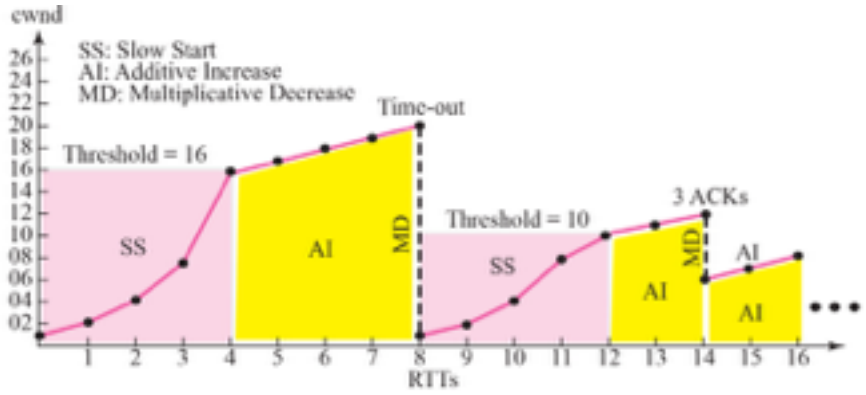
Finché cwnd è minore di ssthresh, cwnd raddoppia crescendo in modo esponenziale per ogni ACK ricevuto fino al valore di ssthresh, poi entra in una fase lineare in cui aumenta di uno fino al regime rwnd.

Ad ogni congestione riscontrata, ovvero ogni qualvolta che si presenta un timeout, si verificano le seguenti azioni:

- la cwnd riparte da uno
- si ricomincia con la fase di SS
- e la ssthresh assume un nuovo valore pari alla metà della cwnd precedente

Le tipiche strategie del TCP sono:

- ricalcolo del timeout in modo dinamico come media con la stima del precedente
- in caso di ritrasmissioni il timeout raddoppia (*exponential backoff*)
- per non lavorare con segmenti corti, la rwnd rimane 0 fino a MSS/2 (*silly window*)
- fingere di avere buffer più grandi per pipe veloci in modo da mantenerle sempre piene (*long fat pipes*)



Domande tipiche:

1. **Semantiche di comunicazione may-be, at-least-once e at-most-once**
   Discutere delle semantiche di realizzazione dei protocolli
   MAY_BE/BEST_EFFORT: semantica utilizzata per protocolli UDP e IP con bassa qualità dove non è prevista nessuna connessione, i messaggi vengono mandati uno alla volta, possono essere duplicati, e non vi è alcun controllo di errore ne di flusso. Tale semantica però presenta il vantaggio di essere a basso costo, qualità importante in internet.
   AT_LEAST_ONCE: semantica dove viene migliorata la precedente con ritrasmissioni ad intervallo introducendo un timeout. Il client invia la richiesta se arriva al server e viene restituita la risposta se tutto ok, in caso di errore invece esiste un timeout entro il quale si può fare la ritrasmissione. In caso scade il timeout la richiesta viene scartata. Anche questa non molto affidabile e non c'è controllo sull'esito dell'iterazione ne in caso di successo ne insuccesso. La richiesta può arrivare almeno una volta, se arriva più volto il server la serve senza accorgersi dei duplicati.
   AT_MOST_ONCE: tale semantica è utilizza dal protocolli TCP in quanto protocolli con affidabilità, connessi. Similmente alla precedente per via del timeout e quindi delle ritrasmissioni, qui vi è mantenimento dello stato sul server delle richieste ricevute. Tale semantica prevede l'invio di dati in ordine e non duplicati e la richiesta arriva al server al più una volta. Anche in questo caso non abbiamo un coordinamento in caso di insuccesso, ma lo avremo in caso di successo.
   EXACTLY_ONCE: semantica migliore dal punto di vista dell'affidabilità e del coordinamento. Prevede che l'iterazione C/S vada a buon fine, semantica del TUTTO o niente. Molto costosa in termini di risorse.

2. **Descrivere ARP e RARP**
   Address Resolution Protocol. Protocollo che permette di ottenere l'indirizzo MAC di un nodo a partire dal suo indirizzo IP. Questo serve in caso di routing diretto (all'interno della stessa rete locale). È un protocollo costoso perché fa broadcast sulla rete locale. Per questo i nodi mantengono una cache con le corrispondenze trovate. Le tabelle vengono qualificate ad ogni richiesta perché tutti i nodi della rete vedono la risposta. Due ruoli: Attivo richiede l'indirizzo fisico (esamina la cache locale altrimenti broadcast della richiesta), passivo risponde alle richieste di altri. Il messaggio ARP è incapsulato in un frame di livello sottostante, data link .
   RARP: Reverse Address Resolution Protocol è utilizzato per compiere la risoluzione di indirizzi fisici in logici, spesso sono presenti tanti Server ma la richiesta in broadcast potrebbe creare conflitti per via di risposte simultanee, per questo esiste un server primario che sarà lui a rispondere per primo e successivamente i secondari (modello dinamico). Invece nel modello statico il primario risponde immediatamente e eventualmente i secondari con un certo timeout random assicurando la non sovrapposizione delle risposte.

3. **DHCP**
   Dynamic Host Configuration Protocol è un protocollo utile per superare i problemi di indirizzamento statico in reti di dimensione ampie. Il suo scopo principale è quello di fornire dinamicamente indirizzi IP logici ai dispositivi collegati in una rete assegnandoli al momento del bisogno. Il suo principio di funzionamento si basa sul modello C/S con protocollo d'offerta tipo asta. Il client invia un messaggio in broadcast, detto DHCPDISCOVERY, a tutti i server presenti, i quali ricevono il messaggio e a loro volta invieranno un offerta, DHCPOFFERT, il client sceglie un'offerta e invia una DHCPREQUEST al server selezionato, quest'ultimo infine assegnerà l'IP. Successivamente avremo delle fasi di LEASE o RELEASE, cioè verranno aggiornati gli indirizzi che non saranno utilizzati, questo per un risparmio di risorse.

4. **NAT**
   Network Address Traslation è un protocollo che consente la trasformazione di indirizzi intranet (privati) in indirizzi internet (pubblici). Si utilizzano a tal proposito router NAT che mantengono delle tabelle apposite. Utilizza le porte per poter migliorare il suo funzionamento, grazie all'associazione ad una porta più indirizzi, quindi mappando tanti indirizzi interni in un unico esterno. Distinguendo i vari indirizzi interni tramite le porte diverse di connessione. Tale ottimizzazione è chiamata nat a porte è ha un notevole risparmio di indirizzi pubblici.

5. **ICMP**
   Serve per migliorare le prestazioni di IP viste le sue caratteristiche besteffort. Il protocol Internet Controll Message, è un protocollo di rete che lavora insieme all'IP riconosciuto come un protocollo non affidabile. Si basa sul controllo dei messaggi inviati da parte del client, i quali possono non raggiungere

il Server. In tal caso vengono inviate al mittente dei messaggi di notifica segnalando l'errore, ma non per corregerlo. Sarà compito del Client inviare di nuovo la richiesta. In caso di un secondo fallimento la comunicazione verrà abbandonata, in quanto gli stessi messaggi ICMP potrebbero creare congestione, questi ultimi sono incapsulati in datagrammi IP e non hanno priorità. ICMP presenta un HEADER costituito da solo un campo CODE, TYPE e CHECKSUM. Attraverso i campi CODE e TYPE si riesce a individuare il tipo di errore.

6. **UDP protocol**
   User Data Protocol è un protocollo di trasporto che si basato sul Best-effort ed è a basso costo, quindi non affidabile. Lo scambio di messaggi da un processo ad un altro avviene da nodo a nodo senza connessione, i dati vengono inviati non in ordine, anche duplicati e possono essere anche persi. Il suo vantaggio è quello di avere un costo molto basso. Sfrutta il protocollo IP e la porta per inviare a destinazione i datagrammi. L'HEADER UDP è costituito da un IP source e IP destination, un UDP messaggeLen, e un UDP checksum più la parte Dati ed entrambi Header e DATA sono contenuti nella parte data del datagram IP.

7. **Continuos requests**
   Al fine di migliore la comunicazione tra C/S si adottano tecniche di comunicazione più efficienti come CONTINUOS REQUEST, che implica l'invio di richiesta dal Client al server sino a saturare il buffer del server che le contiene. Anche il Cliente avrà un buffer dove contenere tutti gli eventuali messaggi di acknowledgment che vengono inviati in piggybacking sul traffico opposto. Nel caso ci fossero problemi nella ricezione degli ack si può risolvere il problema grazie a dei protocolli come: SELECTIVE_RETRASMISSION: che permette l'invio selettivo dei soli ACK non arrivati correttamente. GO_BACK_N: prevede l'attesa degli ACK e l'invio di quelli del ricevente. Il mittente scarta i msg successivi non in sequenza e li rimanda al ricevente. TCP utilizza il go_back_N con ACK cumulativi.

8. **Descrivere fase iniziale e finale protocollo TCP in termini di messaggi scambiati e mettere quest'ultima in relazione con le funzioni socket.**
   La fase di apertura di una connessione TCP avviene tramite Three-way handshake. A vuole connettersi con B. (1) A invia un segmento a B con SYN e numero di sequenza X. (2) B riceve il segmento SYN da A e risponde con un nuovo segmento identico un nuovo numero di sequenza Y con ACK con X+1. (3) A riceve il segmento SYN e ACK da B e risponde a B con un ACK con Y+1. Le fasi sono tre perché servono a dimensionare time-out e finestre di ricezione. Se si dovesse perdere un messaggio si ritrasmette a time-out crescenti e dopo si chiude.
   La fase di chiusura graceful invece è strutturata in quattro fasi: (1) A invia un segmento FIN con numero di sequenza X. (2) B riceve FIN e conferma con un ACK con X+1. (3) B può mandare ancora dati ad A in quanto questo canale non è stato ancora chiuso e a sua volta A può mandare ACK di conferma, unico invio permesso. Quando B decide di chiudere il canale manda un FIN con numero di sequenza Y. (4) A riceve e conferma la chiusura con un ACK Y+1. La chiusura dei canali, come per le socket, può avvenire in modo unidirezionale: close() e chiusura three-way-handshaking o bidirezionale: shoutdown() e chiusura four-way-handshaking.

9. **Nel datagramma TCP, come vengono gestiti i dati urgenti?**
   L'header del datagramma TCP è formato da 5 parole, dove una di queste contiene i code bit, ovvero 6 bit che opportunamente settati permettono di richiamare varie funzioni per il TCP, una di questa è URG che se settato a 1 indica che il flusso contiene un dato urgente che punta all'urgent point. Il campo URGENT POINTER indica poi la distanza dalla posizione corrente nel flusso. Se viene segnalato un dato urgente sul flusso da livello applicativo tutti gli header successivi porteranno l'indicazione della sua presenza.

10. **In tutte le fasi del protocollo TCP, cosa sono e come vengono usati il sequence number e l'acknowledgment number.**
    Il segmento TCP prevede due campi nel suo HEADER quali: Sequence number [32 bit] - Numero di sequenza, indica lo scostamento (espresso in byte) dell'inizio del segmento TCP all'interno del flusso completo, a partire dal Initial Sequence Number (ISN), negoziato all'apertura della connessione.
    Acknowledgment number [32 bit] - Numero di riscontro, ha significato solo se il flag ACK è impostato a 1, e conferma la ricezione di una parte del flusso di dati nella direzione opposta, indicando il valore del prossimo Sequence number che l'host mittente del segmento TCP si aspetta di ricevere.

11. Come si settano i time-out principali in una connessione

In una connessione i timeout vengono settati nella fase iniziale, ovvero nella negoziazione a tre fasi, dove ogni pari manda e riceve una risposta per il calcolo del proprio timeout. La corretta impostazione di questo timer è difficile ma molto importante, in quanto un timer troppo breve comporta ritrasmissioni inutili (il timer scatta mentre il riscontro o il pacchetto sono ancora in viaggio), mentre un timer troppo lungo comporta attese in caso di effettiva perdita di pacchetti. Tale intervallo dovrà essere almeno pari al RTT (Round Trip Time) cioè al tempo di percorrenza a due vie di un pacchetto per tornare al mittente sotto forma di ACK, tale tempo varia in modo casuale così il protocollo tende ad aggiustare il timeout basandosi su una stima di tale RRT.

12. TCP-IP, come si definisce la dimensione della finestra di trasferimento? come si risponde a una presunta congestione?

Come rilevare congestione e come trattarla.

Il caso di congestione è critico per connessioni TCP. Rileviamo la congestione come scenario in cui non si riescono più a consegnare dati in tempi utili (rispetto alla operatività corrente). Può dipendere dai soli end-point della connessione stessa, sia da una più ampia situazione dell'intera rete. Tutti i router sono con buffer pieni e nessuno scambio può più avvenire, fino alla de-congestione. Identificazione della congestione: time out che scatta in modo ripetuto: si assume che il pari non sia raggiungibile e che la congestione sia in atto (anche solo 1 timeout). In caso di congestione, in modo unilaterale, il mittente dimezza la finestra di invio e raddoppia il time-out al termine della congestione, per ritornare ad una situazione di regime si riparte con un transitorio con finestra piccola (slow start). Slow start è anche la politica iniziale per evitare una potenziale congestione iniziale (avoidance).

7 Java RMI - Remote Method Invocation

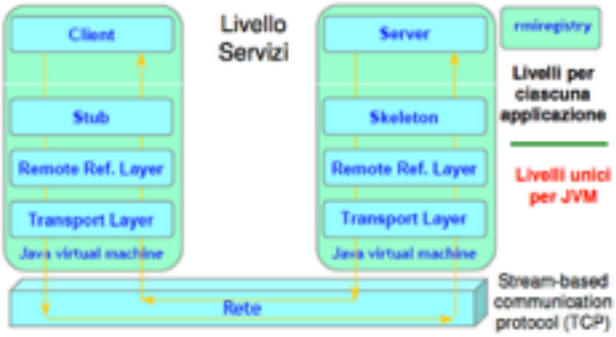
7.1 Introduzione

Questo tipo di operazioni costituiscono delle *Remote Procedure Call* (**RPC)** in Java, ovvero la possibilità di richiedere esecuzione di metodi remoti.
Costituiscono un insieme di strumenti, politiche e meccanismi che permettono ad un'applicazione Java su un nodo di invocare metodi di oggetti attivi su un altro nodo dando la percezione di invocazioni locali lavorando invece con metodi remoti. Java utilizza una semantica per *rifermento*, dove ogni variabile che riferisce un'istanza, è un riferimento nella memoria a quell'istanza. Per avere la possibilità di invocare un metodo remoto, con una semantica per rifermento, è necessario l'utilizzo di un riferimento remoto ovvero un rifermento che contiene le informazioni necessarie per richiedere l'invocazione del metodo.
In locale viene creato solo il riferimento all'oggetto e mantenuto in una *variabile interfaccia,* cioè variabili che possono contenere un riferimento ad un qualsiasi oggetto che implementa l'interfaccia.
I riferimenti remoti vengono costruiti con RMI mediante l'uso di due *proxy* (interfacce che nascondono a livello applicativo la natura distribuita dell'applicazione):

1. lo *stub* per il client
2. lo *skeleton* per il server

L'uso di questi proxy è necessario in quanto non è possibile riferire direttamente l'oggetto remoto e quindi abbiamo bisogno di questa infrastruttura attiva e distribuita.
L'istanza, realizzata con la variabile interfaccia che contiene il riferimento remoto, non riferisce realmente un'entità remota, ma possiede un riferimento allo stub, invocato ad ogni utilizzo del riferimento remoto. Questo poi si coordina con lo skeleton, che una volta ricevuta la richiesta la esegue e restituisce il risultato allo stub che a sua volta la consegna al client. Quindi il client interroga lo stub come se fosse un server e lo skeleton invece assume un ruolo da client interrogando il server, ciò appunto conferisce un adeguato livello di trasparenza all'utente finale.
RMI lavora con il protocollo TCP, non modificabile, per garantire qualità del servizio, a differenza di altri protocolli che potrebbero essere modificati rispetto al default. La semantica di invocazione dei metodi rimane *sincrona bloccante.*
Analizziamo nel dettaglio l'architettura di RMI, ovvero due pile a livelli che rappresentano due JVM separate. A livello più alto, quello applicativo, troviamo client e server (scritte dal programmatore) che rispettivamente chiedono e forniscono i metodi, gli altri strati invece, generati dalle rispettive JVM, sono:

1. *stub:* proxy locale sul quale l'utente fa invocazioni destinate all'oggetto remoto.
2. *skeleton*: entità remota che riceve le invocazioni dello stub e le realizza con chiamate sul server.
3. *Remote Reference Layer (RRL):* si occupa del passaggio degli argomenti, gestisce i riferimenti agli oggetti remoti, i parametri e le astrazioni dello stream.
4. *Transport Layer:* gestione delle connessioni mediante protocollo TCP e delle attivazioni integrate nella JVM.
5. *Registry*: risiede su una JMV a parte e si occupa della gestione dei nomi, consente al server di pubblicare un servizio e al client di recuperarne il proxy.



7.2 Oggetti distribuiti

Un oggetto remoto consente di invocare i suoi metodi, accessibili da remoto, quindi da diversi host, definiti da un'interfaccia, da parte di una JVM in esecuzione su un altro nodo.
Per **interfaccia** intendiamo un contratto astratto con la dichiarazione dei soli metodi. Chi si occupa di implementarla, implementa anche i suoi metodi.
Chiamate locali e remote (su interfaccia) hanno la stessa sintassi ma semantica diversa, infatti la chiamata remota è sincrona bloccante e può fallire, mentre quella locale è affidabile. Un server remoto esegue sempre in modo indipendente e **parallelo.**

7.3 Interfacce e Implementazioni

Interfacce (definizioni del comportamento) estendono java.rmi.Remote, ogni metodo invece deve propagare java.rmi.RemoteException e restituire o un tipo primitivo o un oggetto che estende la classe serializable (i tipi primitivi sono sempre serializable).
Le classi invece che si occupano dell'implementazione devono, appunto, implementare le relative interfacce ed estendere java.rmi.UnicastRemoteObject.

7.4 Passi di utilizzo

Quindi è necessario:

1. Definire interfacce e implementazioni remote (*server*)
   (**javac EchoInterface.java EchoRMIServer.java**)
2. Compilare le classi e generare stub e skeleton, con *rmic,* delle classi utilizzabili in remoto
   (**rmic -vcompat EchoRMIServer**)
3. Pubblicare il *registry* e registrarvi il servizio (il server fa bind al registry)
   (**rmiregistry** e **java EchoRMIServer**)
4. Ottenere lato client il riferimento all'oggetto remoto dal name service con una *lookup* sul registry e compilare il cliente
   (**javac EchoRMIClient.java** e **java EchoRMIClient**)

7.5 Server

Un processo in esecuzione sul nodo server (main) registra i servizi invocando tante bind quanti sono gli oggetti, passando come parametro una stringa contenente il *nome globale* di registrazione (IP:Porta/ NomeServizio). La bind o rebind è possibile solo sul registry locale. Tale classe quindi implementa i metodi che possono essere invocati da remoto. I server RMI sono sempre in parallelo.

7.6 Client

Si reperisce un riferimento remoto con una lookup e si memorizza in una variabile interfaccia, mediante un cast necessario poiché lookup non restituisce un specifico dato, ottenuta con una richiesta al registry. Si invoca il metodo remoto come specificato da interfaccia, invocazione che risulta sincrona bloccante.

7.7 RMI Registry

Il client deve sapere il nodo su cui risiede il server in anticipo, lo dice l'utente oppure mediante il servizio di naming in una locazione nota, che funge da punto di indirizzamento. In RMI il name service è svolto dal RMIRegistry che mantiene un insieme di coppie univoche in una tabella {*nome servizio, riferimento*} senza trasparenza di locazione, cioè significa che se cambia il riferimento il servizio non funziona più. Il registry va attivato sull'host del server con *rmiregistry* su una shell a parte specificando la porta (1099 di default); il registry così è attivato su una nuova istanza della JVM.

7.8 Comunicazione

E' resa possibile dai due proxy:

1. il client ottiene il riferimento remoto
2. il client invoca metodi sullo stub e attende
3. lo stub serializza la richiesta e invia via RRL le informazioni allo skeleton
4. lo skeleton deserializza i dati, invoca la chiamata sull'oggetto che implementa il server (dispatching), serializza il valore di ritorno e lo manda allo stub
5. lo stub deserializza il valore e lo restituisce al client

7.9 Passaggio dei parametri

Tipo	Metodo Locale	Metodo Remoto
Tipi primitivi	Per valore	Per valore
Oggetti	Per riferimento	Per valore (interfaccia `Serializable` deep copy)
Oggetti Remoti		Per riferimento remoto (interfaccia `Remote`)

Per gli oggetti remoti vengono presi un riferimento ad uno stub remoto e quest'ultimo viene serializzato.
Deep copy: copia dell'intero grafo degli oggetti.
Serializzazione: se un oggetto è sia *remotizable* che *serializable* viene trattato come remotizable poiché risulta più "potente".
Solitamente in RPC i parametri subiscono un *marshalling* (processo di codifica degli argomenti e dei risultati per la trasmissione) e un *unmarshalling* (processo inverso) per risolvere problemi di rappresentazione eterogenee. In Java non serve grazie al *bytecode* e i dati sono semplicemente serializzabili e deserializzabili utilizzando le funzionalità offerte dal linguaggio come per esempio *writeObject e readObject.* Stub e skeleton usano queste funzionalità per lo scambio dei parametri. La serializzazione si può fare solo ad oggetti serializable e che contengono riferimenti ad altri serializable. Vengono trasferiti solo i dati dell'istanza e alla deserializzazione viene creata una copia riempita dei dati, deve essere accessibile il .class per la struttura, ovvero viene verificato che le due classi sono uguali mediante l'uso dell'HASH.

7.10 Stub e riferimenti remoti

Il server è implementato da una classe in remoto. Il client implementa uno stub passato dal registry per mezzo della lookup (ottenuto per riferimento remoto) al cui interno c'è un riferimento remoto al server che viene usato dal RRL per raggiungerlo. Due riferimenti dello stesso client possono puntare allo stub, se però lo si pone in remoto si crea un altro stub.
Il registry è un serverRMI che può essere creato dall'interno del codice di un server. In questo caso il registry è creato nella stessa istanza della JVM mediante il metodo *createRegistry;* così facendo si evita la creazione di una seconda JVM. Il metodo rebind è usato per evitare più registrazioni dello stesso metodo remoto.
Lo stub invece si appoggia su RRL, effettua invocazioni, serializza e deserializza, spedisce e riceve argomenti e il risultato (uguale al RRL se la chiamata va a buon fine).
Lo skeleton gestisce la deserializzazione e serializzazione, spedisce/riceve i dati appoggiandosi sul RRL, ed invoca il metodo richiesto ovvero fa dispatching. Il metodo dispatch invocato dal RRL, prende un processo che non serve più e lo mette da parte inserendo al suo posto un processo attivo che serve.

7.11 Livello di trasporto: concorrenza e comunicazione

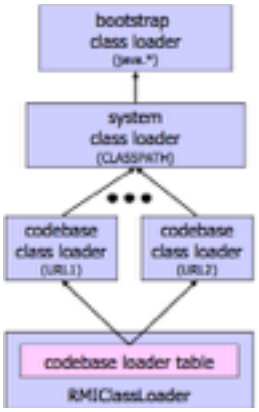
Uso di processi, thread, per ogni invocazione sull'oggetto remoto in esecuzione sella JVM, quindi si ha un server parallelo che a livello applicativo deve essere thread-safe e quindi gestito con *synchronized*. La concorrenza è nascosta all'utente e viene gestita dalla JVM che possiede un processo principale che rimane in attesa delle richieste e genera i thread che si occupano dell'esecuzione del metodo. Se esiste già una comunicazione a livello di trasporto fra due JVM si cerca di riutilizzarla.

7.12 Deployment

In un a applicazione RMI è necessario che siano presenti gli opportuni .class nelle località che lo richiedono. Il server deve vedere interfacce di servizio, implementazioni del servizio, stub e skeleton, oltre classi di utilizzo del server. Il client invece deve essere in grado di vedere interfacce del servizio, stub, classi del server (eventualmente valori di ritorno), oltre a classi di utilizzo del client.

7.13 Class Loader

In Java è l'entità capace di caricare le classi dinamicamente o da locale o dalla rete (RMI classloader) con diversi gradi di protezione, più elevata a livello remoto. Sono in gerarchia o sono diversi e definibili dall'utente, ognuno crea una località e non interferisce con gli altri. RMI usa un classloader particolare che esegue due operazioni particolari: estrae il campo *codebase* dal riferimento remoto e usa il codebase classloader per caricare da remoto le classi necessarie. Il codebase è associabile ad un qualsiasi URL valido. La sicurezza dell'RMI è garantita con l'attivazione sulla JVM di un Security Manager sia sul client che sul server che gestiste gli accessi tramite un file di policy specificato.



7.14 Problema di boostrap

Trovare lo stub del registry senza consultare il registry. Si costruisce localmente un'istanza dello stub a partire da:

1. indirizzo e porta del server ottenuti dall'oggetto remoto
2. identificatore dell'oggetto registry sul server, costante fissa

7.15 Sicurezza

Solo servitori sullo stesso nodo di un registry possono farvi invocazioni e richieste. Il codice scaricato dalla rete è eseguito in un contesto di classloader separato per non creare rischi al sistema. Le informazioni su dove reperire il codice sono sul server e passate al client a richiesta: nel server RMI con l'opzione java.rmi.server.codebase di specifica dove prelevare le classi necessarie. L'URL può essere: *http, ftp, file locale.* Il codebase è annotato nel RemoteRef pubblicato sul registry (quindi nello stub). Le classi sono cercate prima nel classpath e poi eventualmente nel codebase. Anche il server può scaricare classi dal client usando il suo codebase.

Domande tipiche:

1. **RMI parlare dello Stub e Skeleton**
   Lo stub è un proxy locale lato client che riceve le invocazioni destinate all'oggetto remoto, le quali vengono serializzate e inviate allo skeleton, entità remota che riceve il riferimento all'oggetto remoto deserializza e invoca il registry su cui sono registrati i servizi remoti (dispatching), ottiene la risposta la serializza e invia allo stub che a sua volta la deserializza e restituisce il risultato al client. Il metodo dispatch è invocato dal RRL.

2. **Definire i livelli di un'architettura RMI e le entità' coinvolte.**
   RMI utilizza il pattern proxy, l'oggetto remoto non viene riferito direttamente ma attraverso una infrastruttura a più livelli. I proxy vengono chiamati Stub (lato cliente), su cui vengono effettuate le invocazioni destinate all'oggetto remoto, e Skeleton (lato server), che riceve le invocazioni fatte sullo stub e le realizza effettuando chiamate sul server.
   Il Remote Reference Layer (RRL), si occupa di gestire i riferimenti agli oggetti remoti e astrae dalle connessioni stream-oriented. Livello di trasporto, sempre connesso (TCP), si occupa della gestione delle connessioni a basso livello.
   RMIRegistry, sistema di nomi che consente al server di pubblicare un servizio. I Clienti per ottenere e raggiungere il riferimento remoto fanno una richiesta al RMIRegistry mediante lo stub.

3. **Java RMI, cos'è RMI registry, a cosa serve, come si interagisce? E' obbligatorio un riferimento al registry in RMI?**
   RMI registry: cos'è? serve sempre? perché?
   Un client in esecuzione su una macchina ha bisogno di localizzare il server a cui connettersi (su un'altra macchina). Serve un servizio di nomi, in una locazione ben nota, che funga da punto di indirizzamento. È un server RMI che agisce come sistema di nomi in un'architettura RMI. Ha l'obiettivo di gestire una tabella costituita da coppie nome del servizio e riferimento dell'oggetto remoto che fornisce il servizio. Lavora con una logica di unicità di nomi. Permette ai server RMI che realizzano servizi di registrarsi (bind/rebind) per essere poi trovati dinamicamente dai clienti interessati (lookup).

4. **Remote Reference Layer**
   Il RRL insieme al Transport Layer risiedono nelle JVM di client e server in una architettura RMI. Il RRL è responsabile della gestione dei riferimenti agli oggetti remoti, dei parametri e delle astrazioni della connessione a stream. Quando trasferisce variabili per riferimento (es Classi) trasferisce l'intero grafo delle relazioni di dipendenza.

5. **Come avviene il passaggio dei parametri nei metodi remoti in RMI**
   Il passaggio dei parametri nei metodi remoti avviene per passaggio per riferimento remoto. Gli oggetti la cui funzione è strettamente legata alla località in cui eseguono (server) sono passati per riferimento remoto: ne viene serializzato lo stub, creato automaticamente a partire dalla classe dello stub. Ogni istanza di stub identifica l'oggetto remoto al quale si riferisce attraverso un identificativo (ObjID) che è univoco rispetto alla JVM dove l'oggetto remoto si trova.

8 Sistemi RPC e sistemi di Nomi

8.1 Remote Procedure Call

Le RPC mediante i server mettono a disposizione delle procedure che possono essere invocate in remoto dai vari client, rendendo questa invocazione come se fosse locale, quindi abbiamo l'estensione della chiamata a proceduta su locale mediante un C/S nel distribuito. In una chiamata a procedura remota sia i parametri che i risultati viaggiano attraverso la *rete.*
Quindi si ha un approccio applicativo di alto livello, livello 7 OSI. I processi sono distinti su due nodi e non condividono spazio d'indirizzamento, si possono avere possibili problemi sui nodi o nell'infrastruttura. RPC prevede una struttura di supporto che ha il compito di *identificare* le procedure richieste e le risposte corrispondenti, gestire l'eterogeneità dei dati facendo il marshalling/unmarshalling dei dati fra i diversi ambienti infine gestire gli errori.
Proprietà delle RPC sono:

1. trasparenza
2. controllo dei tipi
3. controllo concorrenza ed eccezioni
4. binding distribuito
5. trattamento degli orfani (processo server che non riesce a fornire risultato o il cliente senza risposta)

Il default è un client sincrono bloccante e non ci sono regole standard e ogni sistema usa primitive diverse, ma il client invia richieste call servizio (con eventuali parametri) il server accetta e risponde. Il server ha due possibilità di concorrenza, la prima è sequenziale (esplicita) la seconda invece vede processi indipendenti per ogni richiesta (implicita).

8.2 Tolleranza ai guasti

Un obiettivo, a livello applicativo, è quello di mascherare i malfunzionamenti, come perdita dei messaggi, e i crash di uno dei due nodi. In base a questo si possono tentare diverse semantiche lato cliente:

- may-be: attesa
- at-least-once: attesa e ritrasmissione
- at-most-once: tabella delle azioni effettuate
- exactly-once: azione fatta fino alla fine

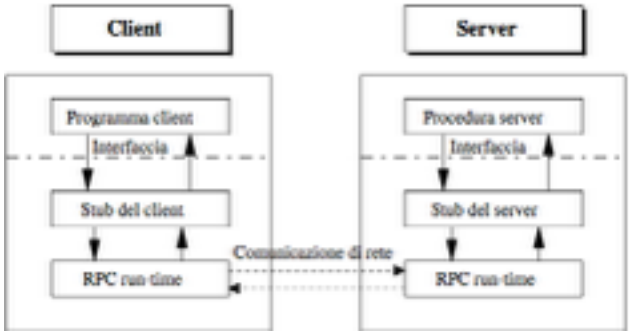
In caso di crash del cliente, si devono gestire i processi orfani a lato server, ovvero in attesa di consegna. Le tipiche politiche usate sono:

- sterminio: orfani distrutti
- terminazione a tempo: ogni azione ha una scadenza oltre la quale viene distrutta
- reincarnazione: tempo diviso in epoche, ciò che appartiene all'epoca precedente è distrutto.

8.3 Open Network Computing - ONC

Sun propone una chiamata RPC, disomogenea dalla locale, primitiva **callrpc()** con parametri: *nome del nodo remoto, Id della procedura, specifiche di trasformazione degli argomenti.* Queste specifiche di trasformazione degli argomenti sono dettate dal formato standard **XDR**: eXternal Data Representation. Questo tipo di modello risulta essere non trasparente.

8.4 Network Computing Architecture - NCA



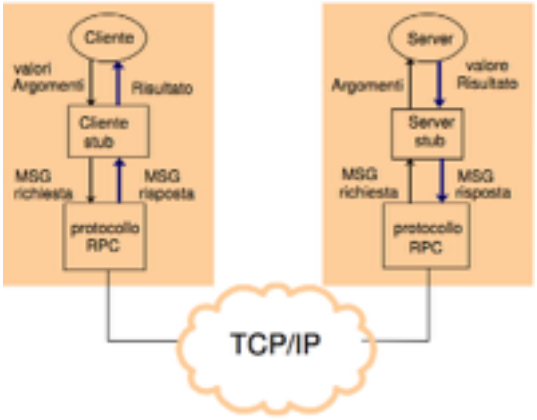
Si introducono in entrambi i lati due stub per ottenere trasparenza come se fossero delle chiamate locali. Gli stub vengono generati automaticamente e nascondono le operazioni ma in compenso si aggiunge il costo di

avere due chiamate per ogni RPC fatta. Il modello con stub quindi risulta asimmetrico a molti client e a un solo client.

Il client invoca lo stub e il servitore riceve la richiesta dallo stub:

- **stub client:** ricerca il server, marshalling argomenti, invio di richieste, ricezione risposta, unmarshalling riposta, consegna il risultato.
- **stub server:** attende richieste, unmarshalling degli argomenti, invoca operazioni locali sul server, riceve risposta, marshalling della risposta, invia la riposta.

Negli stub si concentra il supporto nascosto all'utente. Mancano ritrasmissioni possibili e gestione della concorrenza del server.



8.5 Passaggio dei parametri

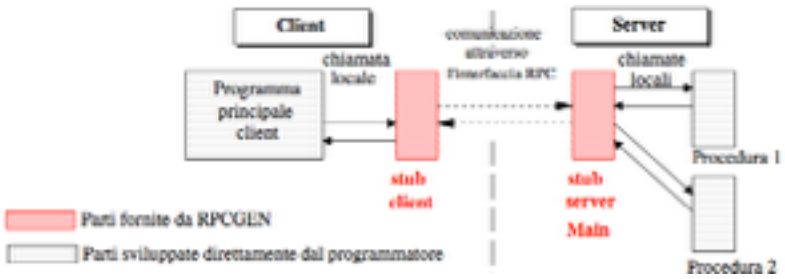
Passaggio dei parametri può avvenire o per valore o per riferimento, a default è per valore. Se è per valore abbiamo un trasferimento, se invece è per riferimento non c'è tale trasferimento rendendo l'oggetto remoto.
Le RPC prevedono il trattamento delle eccezioni tramite un gestore, per eventi anomali dipendenti dalla distribuzione o dai guasti. Una RPC può avere successo o insuccesso e in tal caso produrre una eccezione locale con semantica tipica (at-least-once se SUN)

8.6 Interface Definition Language IDL

Sono linguaggi per la descrizione di operazioni remote, specifiche del servizio a generazione degli stub. Si devono consentire:

- identificazione unica del servizio con un nome astratto e un numero di versione
- definizione astratta dei dati da trasmettere in IN e OUT

Non esistono standard di IDL, un esempio per la SUN è **XDR**: *eXternal Data Representation.* Definisce le operazioni remote e le informazioni per la generazione degli stub. Si prevedono versioni diverse e tipi primitivi anche definibili dall'utente, l'utente descrive la logica in file appositi. Per passare da file IDL a stub si usa un compilatore di protocollo chiamato RPCGEN che produce gli stub per il server e il client da un insieme di costrutti descrittivi per tipi di dati e per le procedure remote in linguaggio RPC.



8.7 Fasi di supporto RPC

Prima si specifica un contratto in IDL, poi si implementa. Le tipiche fasi dell'implementazione sono:

- *compilazione di sorgenti e stub:* la compilazione produce gli stub che semplificano il progetto e porta il raggiungimento dell'accordo sul servizio e server.
- *trasporto dati*: dipende dallo strumento e può essere connesso o meno (TCP o UDP)

- *controllo concorrenza*: consente di usare gli stessi strumenti per funzioni diverse (esempio condivisione connessione)
- *supporto alla rappresentazione dei dati*: si trasformano i dati per superare l'eterogeneità
- *binding*: come ottenere il giusto aggancio fra client e server.

8.8 RPC Binding

Il client si aggancia al server seguendo due strade:

- secondo una strada *pessimistica e statica*: definito in compilazione prima dell'esecuzione, costo limitato.
- secondo una strada *ottimistica e dinamica*: fatto alla necessità, dirige le richieste sul gestore presente e più scarico, costo maggiore.

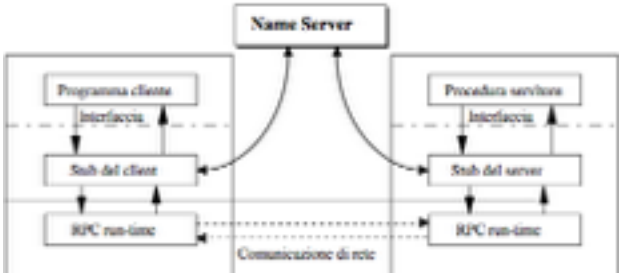
Il binding dinamico viene ottenuto distinguendo due fasi nella relazione C/S:

- servizio (fase statica): prima dell'esecuzione il client specifica a chi vuole connettersi con un nome unico identificativo del servizio (**naming),** si associano i nomi unici di sistema alle interfacce astratte e si fa il binding con l'interfaccia specifica di servizio.
- indirizzamento (fase dinamica): all'uso il client deve essere realmente collegato al server (**addressing**), si cercano, usando sistemi di nomi, gli eventuali server pronti per il servizio.

La parte di naming è risolta con un numero associato staticamente all'interfaccia del servizio.
La parte di addressing deve avere costi limitati durante il servizio:

- esplicita: si fa richiesta del server in broadcast e si accetta la prima risposta.
- implicita: uso di un name server che registra i server e gestisce tabelle di binding.

In caso di binding dinamico, serve un binding ad ogni chiamata: soluzione costosa, quindi si usa dopo un primo legame sempre lo stesso per diverse chiamate allo stesso server, come se fosse statico.



8.9 Sistemi di nomi

Un binder deve consentire agganci flessibili mediante operazioni come:

- lookup
- register
- unregister

Se il nome del server è dipendente dal nodo di residenza, ogni variazione deve essere comunicata al binder. Il binding è attuato come servizio coordinato di più server

8.10 RPC Asincrone

Servono RPC che non si bloccano, asincrone, dove il cliente non si blocca ad aspettare il server, per far ciò abbiamo due opzioni:

- RPC a *bassa latenza*: inviano richieste e trascurano il risultato
- RPC ad *alto throughput*: raggruppano in un unico messaggio diverse richieste.

A seconda del supporto usato, UDP o TCP, si ottengono diverse semantiche:

- *Realmente asincrone* (senza risultato): Athena usa UDP e bassa latenza con semantica may-be; SUN usa TCP ad elevato throughput - semantica at-most-once
- *asincrone* (con risultato): Chorus - bassa latenza; Mercury - alto throughput

8.11 Proprietà delle RPC

Proprietà **visibili** all'utilizzatore:

- *entità che si possono richiedere:* operazioni o metodi di oggetti
- *semantica di comunicazione:* maybe, at most once, at least once
- *modi di comunicazione:* a/sincroni, sincroni non bloccanti
- *durata massima e eccezioni*: ritrasmissioni e casi di errore

Proprietà **trasparenti** all'utilizzatore:
- *ricerca del server*: sistemi di nomi unici centralizzati o multipli
- *presentazione dati:* linguaggio IDL e generazione stub
- *passaggio parametri:* valore, riferimento

8.12 RPC Sun vs RMI

	SUN RPC	RMI
Entità da richiedere:	solo operazioni e funzioni	metodi di oggetti via interfacce
Semantica:	at-most-once, at-least-once	at-most-once (TCP)
Comunicazione:	sincrona e asincrona	solo sincrone
Durata - Errori:	timeout	trattamento di casi di errori
Ricerca Server:	portmapper sul server	registry nel sistema unico centrale
Presentazione dei dati:	XDR e RPCGEN (XDR)	generazione stub e dello skeleton
Passaggio parametri:	per valore, strutture complesse sono linearizzate e ricostruite al server	per valore di default, per riferimento gli oggetti con interfacce remotizzabili

8.13 Nomi
Necessità nel distribuito di trovare le altre entità del sistema. Il problema sono la presenza di più livelli di nomi e diverse funzioni di trasformazione da nome a entità. Una stessa entità può avere diversi nomi:
- **esterni (utente)**: stringhe, significative per l'utilizzatore
- **interni (sistema)**: numeri, più efficiente.

8.14 Livelli di nomi
Possiamo avere tre livelli per i nomi nel distribuito:
1. *nome*: logico esterno (come un oggetto)
2. *indirizzo*: fisico (dove risiede l'oggetto)
3. *route*: organizzazione per raggiungibilità

Esistono poi funzioni di *mapping* per passare da una forma ad un'altra (es mapping da nomi a indirizzi)

8.15 Sistemi di nomi
Gli spazi dei nomi più usati sono:
1. *piatto* (flat): senza struttura, adatto per poche entità
2. *partizionato*: con gerarchia (DNS)
3. *descrittivo*: riferimento ad una struttura di attributi per identificare le entità (X.500)

I client di un name server sono i clienti che devono risolvere un nome per riferire la risorsa oppure le risorse che devono registrarsi per essere riferite. Il sistema è ottimizzato all'uso: le operazioni dei client sono più frequenti di quelle di registrazione delle risorse. I name server devono fornire operazioni per agire sulle proprie tabelle di corrispondenza. La realizzazione può essere:
- centralizzata in un unico server
- distribuita e replicata con diversi agenti multipli (DNS)

Nel caso di agenti multipli il servizio prevede una comunicazione tra gli agenti pur continuando a fornire il servizio, il coordinamento deve essere minimizzato in tempo e risorse in quanto si possono creare problemi di affidabilità, complicati dalle replicazioni.

8.16 Risoluzione dei nomi
Nei sistemi globali, DNS, si distribuisce e risolve il nome in contesto locale e si ricorre ad altri contesti solo se necessario, in tal caso si utilizzano risoluzioni **ricorsiva** o **iterative.**

8.17 Directory X.500
E' un servizio di nomi standard partizionato, decentralizzato e sempre disponibile. La directory è organizzata come un albero logico (*Directory Information Tree - DIT*) costruito in base al valore di attributi liberi e a

scelta dell'utente. Ogni entry ha diverse notazioni: ha un *Distinguished Name (DN)* univoco su tutto il DIT e la *Relative Distinguished Name (RDN),* univoca solo all'interno di un contesto. I nodi sono selezionabili come chiave univoca, DN o con tutte le associazione attributo-valore. La comunicazione tra directory client e server usa il *Directory Access Protocol (DAP),* che essendo un protocollo del livello applicazione richiede l'intera pila OSI.
Le informazioni o attributi possono essere molto eterogenee. Ci sono diverse operazioni: la prima è un *bind* con il directory, poi *ricerche* (frequenti) e *cambiamenti* (rari). La ricerca avviene tramite agenti che con determinati protocolli chiedono ai componenti del directory le informazioni, questi a loro volta si coordinano con protocolli stabiliti.
Il **servizio di directory** quindi non è un database ma è un programma o insieme di programmi che provvedono ad organizzare e memorizzare informazioni riguardanti reti di computer e risorse condivise tramite rete. Può essere considerato un *database organizzato* ma ha molte differenze con i tradizionali database relazionali. Le directory per loro natura ricevono accessi in lettura, la fase di scrittura è lasciata all'amministratore di sistema. Per tale motivo sono ottimizzate per la lettura e quindi risultano non idonee ad immagazzinare informazioni usate con frequenza.

8.18 Uso del directory

Esistono due tipologie di evoluzione protocolli di **Directory** e protocolli di **Discovery**:

- *Directory*: nomi globali, servizi completi e complessi a costo elevato. Garantire QoS (replicazione, sicurezza, rispetto di tempistiche) e tutto il supporto per memorizzare informazioni prevedendo molte letture e poche scritture.
- *Discovery*: nomi locali, servizi essenziali a costo limitato adatto a variazioni rapide. Si definisce un servizio per ritrovare informazioni correttamente visibili (località delle risorse).

Domande tipiche:

1. **Definire il modello architetturale alla base delle RPC.**
   Il modello della base RPC è costituito da: XDR:external DATA Rappresentation rappresentazione e conversione dei dati simile all'interfaccia rmi.server per dichiarare dati costanti e strutture utili per la dichiarazione delle procedure remote. RPCGEN: remote procedure call generator, è uno strumento che funziona similmente ad un compilatore per trasformare i dati del file XDR e genera i file stub C/S.
   PORTMAPPER: servizio di nomi simile a rmiregestry di rmi, utile per reperire i servizi quando dal lato client vengono invocati. Il port mapper è un unico processo che gestisce le tabelle chiamate port map dove vengono memorizzate le informazioni, riguardante i servizi registrati attraverso la registrerpc() e la svc_run(). NETWORK FILE SYSTEM: file system di SUN

   Tre entità fondamentali cliente, servitore e portmapper (sistema di nomi per le procedure remote). Il client crea un gestore di trasporto, una struttura dati utilizzata per tenere traccia e comunicare con i server. Anche il server crea un gestore di trasporto per mantenere il collegamento con i potenziali clienti e con il servizio corrente. Il port mapper si occupa di fornire ai clienti il numero di porta a cui risponde la procedura da chiamare (inviando numero di programma e versione). Su ogni porta possono risiedere più procedure, il dispatching a quella corretta è possibile grazie al fatto che il messaggio RPC inviato dal client contiene anche il numero della procedura.

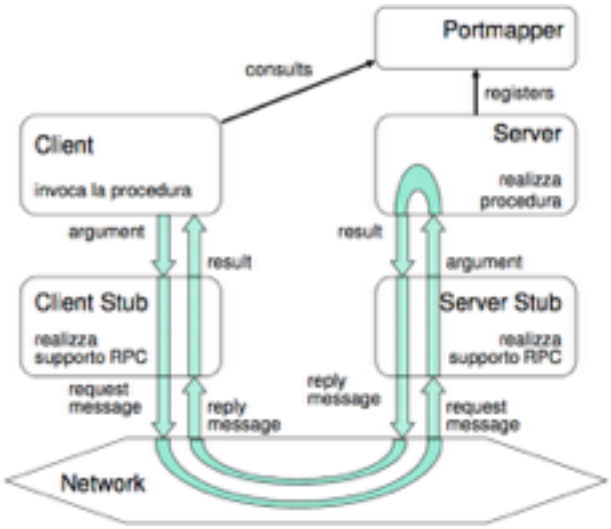
2. **Descrivere RPC in termini di entità coinvolte e operazioni effettuate distinguendo tra cliente e servitore**
   Le entità coinvolte in una interazione RPC sono: stub lato cliente, stub lato servitore e portmapper. Lato Cliente: consulta il portmapper per ottenere la porta su cui invocare la procedura (l'indirizzo lo conosce), insieme ad argomenti invoca la procedura e passa i dati al gestore di trasporto (stub) che si occupa di realizzare il supporto RPC e inviare il messaggio. Lato Server: il gestore di trasporto (stub) riceve il messaggio, legge gli argomenti e chiama la procedura passando i risultati nuovamente allo stub che realizzerà il supporto e invierà la risposta al cliente.

9 Implementazione RPC

9.1 Introduzione

Possibilità di invocare una procedura non locale con un modello tipo C/S, con parametri tipati e valore di ritorno. Uso di un supporto inglobato nei processi interagenti che:

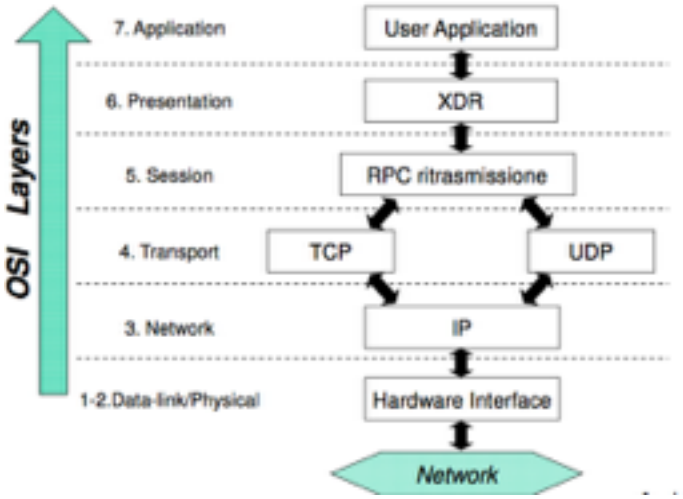
- scambia messaggi: identifica i messaggi e le procedure remote.
- gestisce i dati: un/marshalling dei dati, serializzazione
- gestisce gli errori: della distribuzione dell'utente



La RPC di Sun usa una semantica di tipo at-least-once (protocollo UDP, con ritrasmissioni se non sopraggiunge risposta) con server sequenziale (serve una richiesta per volta) e client sincrono (attende risposta), tali caratteristiche sono quelle di default ma possono essere modificate. L'implementazione di Sun è basata sul modello **Open Network Computing (ONC)** che include alcuni componenti essenziali, che sono:

- XDR: linguaggio di interfaccia che rappresenta e converte i dati nella comunicazione.
- RPCGEN: strumento per la generazione automatica degli stub
- Portmapper: risolutore dell'indirizzo del server, gestore di nomi. Entità a cui i server si devono registrare per offrire le procedure e a cui i client devono fare riferimento per ottenere il servizio scelto.
- NetworkFileSystem (NFS): file system nel distribuito

RPC è un servizio di livello 7 quindi esistono comportamenti che riguardano l'intera catasta OSI.



9.2 Definizione del programma RPC

Abbiamo due parti descrittive in linguaggio RPC:

1. definizione dei programmi: identificazione dei servizi e dei tipi di parametri
2. definizione XDR: definizione del tipo di dato dei parametri

In un file XDR (con estensione .x) si deve descrivere il contratto di interazione, ovvero descrivere l'insieme delle procedure implementate dal server e che quindi il client può richiedere, rispettando i seguenti vincoli:

- ogni procedura ha un solo parametro d'ingresso e un solo parametro d'uscita
- gli identificatori sono progressivi
- ogni procedura è identificabile da un numero di versione unico all'interno del programma

Oltre a definire il contratto il file XDR permette di trasformare i dati dal formato del client allo standard XDR così che il server possa ritrasformarlo nel suo linguaggio e viceversa, quindi XDR risulta un linguaggio di presentazione dei dati.
Il programmatore deve sviluppare:

1. Nel client il programma con main a interazione con l'utente per le richieste e la logica per reperimento a bind dei servizi remoti
2. nel server l'implementazione di tutte le procedure del servizio.

Gli stub sono generati automaticamente da RPCGEN che è comunque basato su XDR.

9.3 Sviluppi implementativi Remoto vs Locale

Server

Ha il compito di realizzare le procedure, il main e le funzioni di comunicazioni sono contenute nello stub.
Le procedure remote differiscono da quelle locali:

- argomenti d'ingresso e uscita passati per riferimento
- il risultato punta ad una variabile statica (globale) per sopravvivere oltre la chiamata a procedura
- il nome della procedura presenta il numero di versione ed è minuscolo

I parametri della procedura cambiano, viene aggiunto un secondo parametro (struct svc_req *rqstp), struttura che contiene tutte le informazioni riguardanti la richiesta di procedura remota, invece il primo parametro viene modificato per diventare un puntatore al tipo o alla struttura dati che gli viene fornita.
Tale procedura viene invocata dallo stub attraverso un passaggio di parametri per riferimento, in quanto la copia dei dati è fatta tra gli stub.
Il risultato viene fornito allo stub per indirizzo e deve essere fornito attraverso una **variabile statica**. Le variabili di procedura vengono allocate nello stack e deallocate nel momento in cui la procedura termina, affinché il risultato venga passato per indirizzo e non venga cancellato al termine della procedura, questo deve essere statico per acquistare il tempo di vita del *programma* e non della procedura che termina prima che lo stub server consegni una copia del risultato allo stub client.

Client

Il client è invocato con il nome dell'host remoto e il parametro utile al servizio. Viene creato un gestore di trasporto client (si utilizza UDP). Il client deve conoscere: l'host remoto, informazioni sulla procedura ovvero programma, versione e nome. Prima di invocare la funzione viene qualificata un'entità, il **gestore di trasporto,** ovvero una variabile di tipo `CLIENT` che viene richiesta al portmapper attraverso l'invocazione della funzione:

```
CLIENT *clnt_create(char *host, u_long n_prog, u_long n_vers, char *proto);
```

che restituisce un argomento nullo in caso di errore, altrimenti un'area di memoria di tipo client. Tale funzione necessita solo del nome IP del nodo poiché essa ottiene il gestore di trasporto direttamente dal **portmapper,** un servizio RPC che sta sulla porta 111 di UDP e TCP.
L'invocazione remota è fatta specificando come argomenti il parametro (unico) e il gestore di trasporto che si ottiene dal portmapper. Il risultato della procedura è un puntatore, che se è nullo indica un errore, che viene riempito con un'area di memoria allocata direttamente dallo stub e contenente i valori forniti dal server.
Il client gestisce gli errori che possono incorrere durante l'invocazione remota.

Il passaggio degli argomenti e del risultato per *indirizzo* tra stub client e client tra stub server e server, è dovuto affinché si mantenga una certa efficenza. Se tale passaggio fosse stato per *valore* si avrebbe avuto una copia di tutti i dati fra entità che siedono sulla stessa macchina e condividono la stessa memoria. Con il passaggio per riferimento lo scambio dei dati fra stub e le corrispettive entità applicative non comporta una copia dei dati ma si fornisce solamente l'indirizzo di memoria. Quindi avremo un passaggio per riferimento a livello locale ed uno per copia tra i due stub attraverso le operazioni di marshalling e unmarshalling.

Passi di sviluppo:

1. definire servizi e tipi di dato (file .x)

2. generare gli stub con RPCGEN
3. realizzare i codici per client e server e compilare i file sorgente e fare linking dei file oggetto
4. pubblicare lato server i servizi: attivare il portmapper e registrarvi il servizio
5. reperire lato client l'end-point del server tramite il portmapper e creare il gestore di trasporto
6. da questo punto in poi inizia l'interazione C/S

9.4 Caratteristiche RPC Sun

Le RPC sfruttano i servizi delle socket sia per stream TCP che per datagrammi UDP che sono le default in Sun. Un programma contiene più procedure con diverse versioni dove ognuna ha solo un argomento di ingresso e di uscita. Il server è **sequenziale** e garantisce la mutua esclusione, esegue una sola invocazione per volta. Il client attende in modo **sincrono bloccante.** Esiste lato server la possibilità di deadlock qualora il server richiedesse un servizio al client stesso, che risulta però bloccato, inoltre ad un nodo servitore possono arrivare diverse richieste RPC che vengono gestite in sequenza. Sun prevede delle modifiche all'implementazione di default per evitare tali situazioni.
La semantica del servizio è UDP di tipo at-least-once: a default si fanno diverse ritrasmissioni dopo un intervallo di timeout.

9.5 Identificazione

Ogni messaggio RPC deve contenere i numeri di programma, di versione e di processo per l'identificazione globale. Il primo è a 32 bit e quelli disponibili dall'utente sono da 20000000h - 3fffffffh. Ogni programma non ha assegnato statisticamente un numero di porta, ma l'assegnamento avviene dinamicamente ed è gestito dal portmapper che funge da gestore di nomi.
Per sicurezza, sia il client verso il server che viceversa, quando fanno delle chiamate si devono identificare o autenticare.

9.6 Utilizzo di servizi RPC

RPC Sun è stato creato come un'implementazione di libreria, quindi sopra il kernel, e offre tre livelli d'uso:

1. Alto: livello utente, offre funzioni di libreria remote con implementazioni nascoste all'utente.
2. Intermedio: livello a default di RPC, permette la definizione e l'invocazione di nuove funzioni, utilizzando strumenti standard.
3. Basso: gestione avanzata, modificazione del comportamento di default.

Le funzioni di libreria fornite all'utente dal livello **alto** sono:

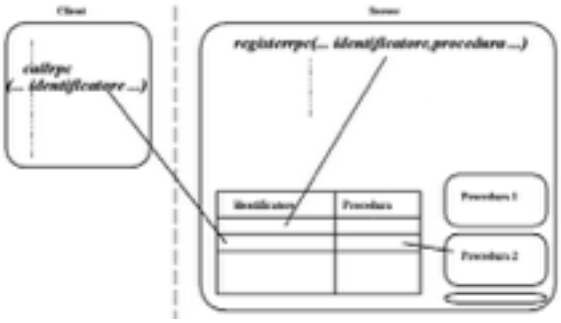
Routine RPC	Descrizione
musers()	Fornisce il numero di utenti di un nodo
rusers()	Fornisce informazioni sugli utenti di un nodo
rstat()	Ottiene dati sulle prestazioni verso un nodo
rwall()	Invia al nodo un messaggio
getmaster()	Ottiene il nome del nodo master per NIS
getrpcport()	Ottiene informazioni riguardo agli indirizzi TCP legati ai servizi RPC

Tutte le funzioni richiedono come parametro il nodo a cui si vuole fare richiesta e tale nodo deve avere attivo il servizio RPC in quanto sono procedure definite e implementate da programmi noti di RPC. Queste funzioni funzionano in modo molto semplice e automatizzato e la gestione è fatta dalle librerie stesse.

A livello **intermedio** invece la libreria offre due primitive:

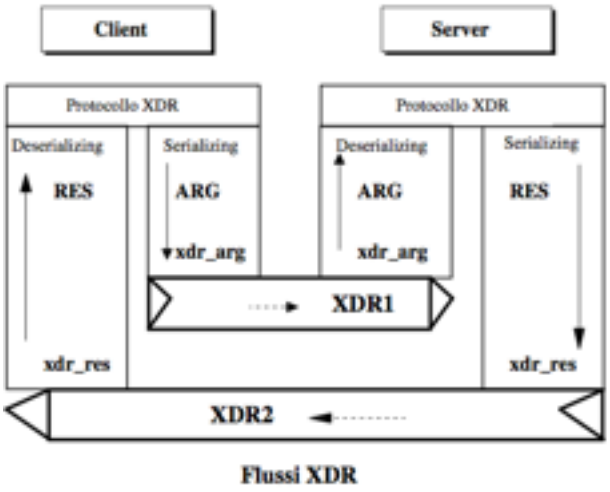
- *callrpc():* il client provoca l'esecuzione della procedura remota. Restituisce 0 in caso di successo oppure la causa dell'insuccesso.
- *registerrpc():* il server associa un ID unico alla procedura remota

Come prima operazione il server dovrà registrarsi al portmapper fornendo il proprio id completo, ovvero numero di programma, di versione e di procedura; successivamente il client, fornendo l'id, richiede al portmapper la porta su cui



sta il servizio, invocando la procedura e ottenendo il risultato.

9.7 Omogeneità dei dati



Flussi XDR

Per comunicare fra nodi eterogenei si possono avere due vie: o si fornisce ogni nodo di tutte le funzioni di conversione di dati possibile oppure si definisce un formato per i dati e ogni nodo possiede solo le funzioni di conversione dal suo formato a quello standard che in questo caso è XDR.
XDR appartiene al livello di presentazione e definisce funzioni di marshalling, definisce tipi atomici e standard e permette di definire nuove strutture poiché ne abbiamo necessità in quanto la *callrpc()* accetta solo un solo parametro in ingresso e uno in uscita. Ogni informazione trasmessa porta a due conversioni, una trasformazione dal linguaggio del mittente al formato comune, XDR, e una da quest'ultimo a quello del destinatario. Tali conversioni sono permesse in quanto XDR fornisce delle funzioni di conversione (che lavorano in modo invertibile ovvero usano la stessa funzione con la stessa signature per fare conversioni in entrambi i sensi) per i tipi primitivi, invece per le trasformazioni di strutture XDR genera una funzione ad hoc che si basa sulle trasformazioni delle primitive.
Le strutture complesse devono essere aggregati di tipi standard o di altre strutture definite.
Lo strumento in grado di compilare tale linguaggio interfaccia è *rpcgen.*

9.8 Registrazione

Il server registra le procedure nella tabella dinamica dei servizi del nodo. La *registerrpc()* è eseguita dal server e registra solo una entry alla volta. Se la funzione ha successo, ovvero restituisce 0, il portmapper inserirà le informazioni nella propria tabella. Poi con *svc_run()* il server si mette in attesa indefinita di una richiesta di servizio, come un demone. Le procedure registrate sono compatibili con chiamate basate su UDP e non su stream TCP.

9.9 Portmapper

E' il gestore di nomi con una tabella che si basa sulla registrazione fisica di:
- una tripla con numero di programma, numero di versione e protocollo
- un numero di porta

La tabella è strutturata come lista dinamica. Non si indica il numero di programma in quanto condividendo lo stesso gestore di trasporto è il medesimo per tutte le procedure. La gestione delle tabelle è gestita su ogni nodo da un processo unico di tipo demone detto **portmapper.**
Il portmapper è un server RPC sulla porta 111 con numero programma 100000 e versione 2, identifica il numero di porta associato ad un qualsiasi programma: allocazione dinamica dei servizi sul nodo. Il portmapper registra servizi e offre procedure per:
- inserire servizi
- eliminare servizi
- lista delle corrispondenze
- corrispondenza tra associazione astratta e porta

Un limite è che ogni chiamata del client interroga il portmapper.

9.10 Architettura e funzioni

Client: creazione mediante la porta ottenuta dal portmapper e utilizzo di un gestore di trasporto, supporto usato per tener traccia o poter comunicare con potenziali servitori.

Server: creazione di un gestore di trasporto registrata nel portmapper per mantenere collegamenti con potenziali client. Tale gestore di trasporto riceve le richieste e seleziona attraverso il dispatching la procedura richiesta dal client. Quindi quando il client chiede al portmapper la porta su cui risiedono tutte le procedure di un programma, versione e protocollo, si ha la creazione di due gestori di trasporto.

9.11 Gestione avanzata

Il livello **basso** è il livello che permette a un utente esperto di modificare i protocolli a default ottenendo nuovi comportamenti, per far ciò RPC permette di modificare i file sorgente dello stub client e server, conferendo loro nuove funzioni che permettono la gestione avanzata del protocollo.

Server: cinque funzioni di libreria eseguite automaticamente dallo stub.

svcudp_create() o svctcp_create() sono funzioni con cui viene creato il gestore di trasporto eseguite a default.

pmap_unset() funzione che elimina la registrazione dei servizi che lo stub vuole registrare.

svc_register() funzione di registrazione al portmapper.

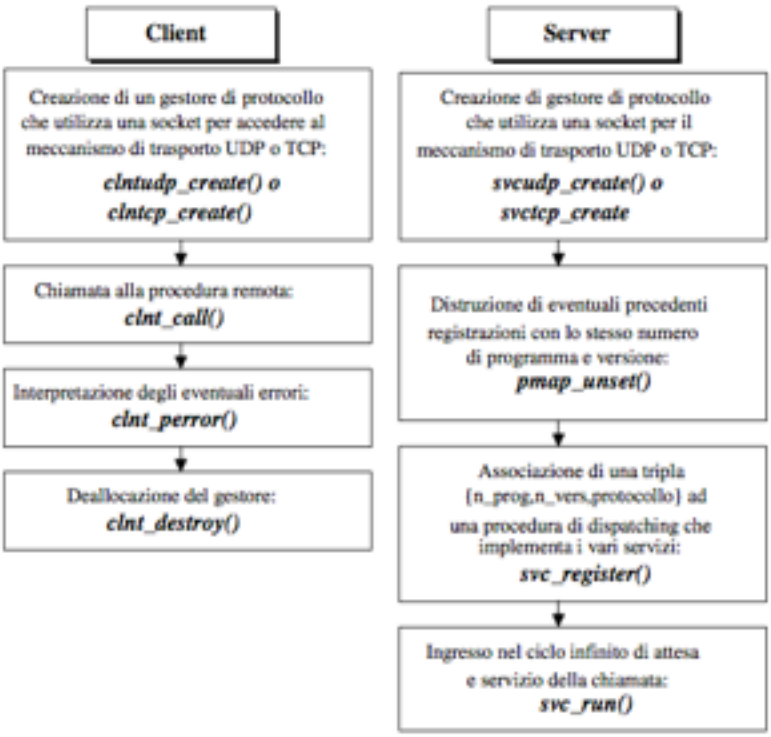
svc_run() rende il server un demone rimando in attesa di una richiesta di chiamata remota.

Client: ha a disposizione cinque nuove funzioni. Per la creazione del gestore di trasporto per RPC si usa la *clntudp_create()* (*clnttcp_create()* per TCP, senza timeout e definisce i buffer di input e output), come parametri si definisce un timeout per le ritrasmissioni.

La chiamata della procedura remota avviene dopo aver creato il gestore mediante la *clnt_call()* (specifica solo gestore di trasporto e numero della procedura) specificando il numero della procedura si raggiunge l'entità, essa è invocata dallo stub.

La primitiva *clnt_perror()* analizza il risultato della call stampando sullo standard error una stringa con l'errore riscontrato.

Infine *clnt_destroy()* distrugge il gestore deallocando il suo spazio in memoria senza chiudere la socket, operazione necessaria in quanto più gestori possono condividere la stessa socket.



9.12 Sun RPC: analisi completa

XDR specifica i dati e le interazioni RPC, si possono avere due sottoinsiemi di definizione:

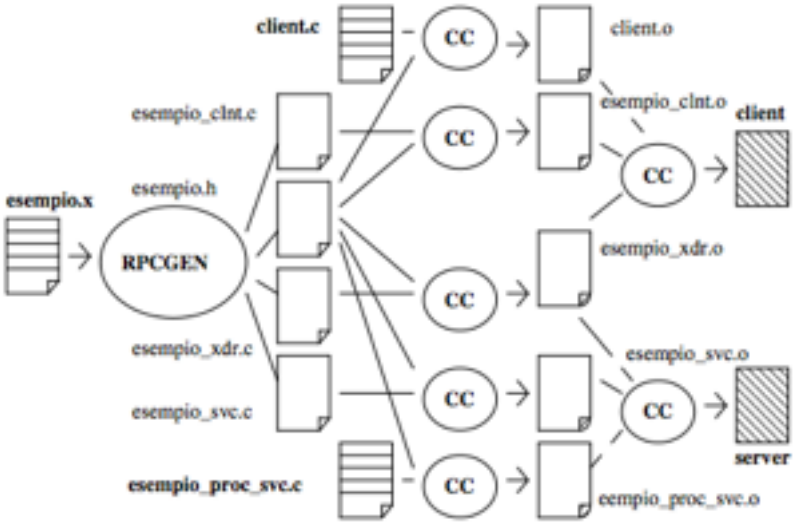
1. definizione di dati dei parametri d ingresso e uscita di tutte le procedure descritte per permettere la creazione di funzioni di conversione. Definizioni di costanti.
2. definizioni delle specifiche di protocolli: di definiscono le procedure con numero di procedura, di versione e di programma.
   NB: 0 è riservato alla NULLPROC, funzione RPC per testare se il server è attivo. Ogni procedura ha solo un ingresso e un'uscita.

Con RPCGEN si generano gli stub automaticamente a partire da un *esempio.x*, in particolare:

- un *esempio.h* da includere nei due stub, si definiscono i nuovi tipi e le nuove strutture per le quali sono generate le funzioni di trasformazione
- lo stub del client *esempio_clnt.c* con la reale chiamata remota (clnt_call())
- lo stub del server esempio.svc.c con il main per registrare, il run per ottenere e la per attendere e la funzione di dispatch()

- la routine XDR *esempio_xdr.c* contenente le funzioni di conversione.

E' compito dello sviluppatore realizzare i programmi client e server (*client.c* e *esempio_svc_proc.c*).



9.13 Osservazioni

Il valore di ritorno della procedura sul server deve essere **static,** deve infatti sopravvivere all'invocazione della procedura per poter essere elaborata dalle funzioni di trasformazione ed essere inviata. Se non è static viene rimossa dopo l'invocazione.

9.14 Modalità asincrona Batch

A default l'RPC è asincrono, ma si può intervenire sul cliente per avere un servizio asincrono. Il client usa TCP, specifica timeout nullo nella clnt_call() e il server non prevede risposta. Si usa TCP per non perdere messaggi. Senza timeout il client continua senza fermarsi ad attendere. Il server non invia risposta. Le richieste del client sono poste nel buffer TCP e gestire dalla driver senza bloccare il processo che le genera.
L'esecuzione è divisa in due parti:

- asincrona: le richieste sono registrate nel buffer TCP ma non servite
- sincrona: finita la fase di ottenimento delle richieste, il client sblocca la situazione con una chiamata sincrona a NULLPROC e si svuota il buffer con le risposte.

Batch: non interattività, esecuzione accorpata e non immediata.

Domande tipiche:

1. **Confronto RMIRegistry e Portmapper**
   Sono entrambi sistemi di nomi fondamentali per il funzionamento di architetture RMI e RPC. Consentono entrambi di far registrare dei servitori che forniscono servizi e consentono ai clienti di trovarli. La differenza sostanziale sta nel fatto che con il registry RMI il client ottiene effettivamente un riferimento remoto all'oggetto, residente sul server, su cui chiamare la procedura e agire direttamente. Mentre con il portmapper i clienti ricevono la porta su cui fare la chiamata a procedura remota e l'azione di per se viene fatta dal server e non direttamente dal cliente che richiede il servizio.

2. **Le attività e quello che privilegia il portmapper**
   Il portmapper è un processo demone che gestisce la tabella port_map presente in ogni nodo RPC. Servizio di nomi (server RPC) che mantiene una tabella di corrispondenze tra numero di programma, versione e porta su cui insiste la procedura. Il port mapper registra i servizi sul nodo e permette di inserire o eliminare un servizio, recuperare la porta da una corrispondenza astratta per la procedura remota e fornisce supporto alla esecuzione remota. Per l'interrogazione abilita due gestori di trasporto propri, uno TCP e uno UDP, che insistono sulla stessa porta (111). A default utilizza solo UDP.

3. **Come funziona il portmapper**
   Il portmapper è un processo unico di tipo demone che gestisce le tabelle su ogni nodo allocando dinamicamente dei servizi e associando due porte alla tripla <programma, versione, protocollo>, una per socket tcp e una per socket udp.
   Il portmapper registra servizi e offre procedure per:
   - inserire servizi
   - eliminare servizi
   - lista delle corrispondenze
   - corrispondenza tra associazione astratta e porta.

4. **Semantica RPC Sun**
   La RPC di Sun ha una semantica del servizio UDP di tipo at-least-once con server sequenziale, che garantisce la mutua esclusione (esegue una sola invocazione per volta), e client sincrono (bloccante). L'implementazione di Sun è ONC che include XDR, RPCGEN, PORTMAPPER e NFS. A default si fanno diverse ritrasmissioni dopo un intervallo di timeout.

5. **RPC si descrivi le responsabilità dei vari componenti (chi genera chi e che funzioni hanno).**
   I componenti di un RPC sono: FILE XDR:Il file XDR con estensione .x, è il file dove vengono rappresentati e convertiti i dati. E' costituito da due parti, la prima dichiarazione dei dati utilizzati nella parte di dichiarazione delle procedura che saranno individuate da un numero di programma, versione e procedura.
   CLIENT: il programmatore sviluppa un programma client implementando il main() e la logica necessaria per richiedere il servizio remoto.Viene chiamata la procedura remota passando il parametro in ingresso vero e proprio più il gestore di trasporto.
   SERVER: La parte server non implementa nessun main perché la procedura remota viene invocata nello stub. vengono dichiarate le eventuali strutture dati e inizializzate, poi vengono implementati i servizi a cui vengono fatto le invocazioni. Gli argomenti d'ingresso che uscita vengono passati per riferimento. Il parametro di uscita viene memorizzato come static, in quanto si prevede che sia presente anche dopo la chiamata alla procedura.
   STUB:Gli stub sia lato client che server sono auto generati dando in pasto rpcgen il file xdr. Nello stub lato server vengono invocate le procedure remote definite nel file server.

10 Servizi Applicativi UNIX

Domande tipiche:

1. **Descrivere FTP**
   Tra i protocolli di accesso e trasferimento dei file abbiamo FTP (File Transfer Protocol), protocollo utilizzato per gestire il trasferimento dei file e ci permette di lavorare su un file system remoto. Dopo aver stabilito una connessione TCP/IP possiamo identificarci con user e pwd,lavorare sul file system quindi file e directory. Le operazioni possibili sono get, put, ls cd... L'implementazione del protocollo si basa su server Paralleli, quindi prevede un approccio concorrenziale da parte di più client e un server. In pratica si hanno almeno due collegamenti per ogni client: La connessione di controllo: prevede che il server sia sulla porta di default 21 e client una random, il server fa la listen e il client la connect, quindi svolge un ruolo attivo. La connessione data:La porta su cui ci si aggancia sul server è la 20, il client arbitraria. qui il ruolo attivo e del server che fa la connect. Il client attraverso i comandi get e put chiede il trasferimento del file.

2. **SMTP**
   Simple mail transfer protocol è il protocollo standard per il trasferimento di mail tra mail MTA. I ruoli tra sender e receiver possono essere inverti per consentire l'invio e ricezione della posta in verso opposto. Esistono altri protocolli di supporto al servizio mail quali: POP: poste Office Protocol IMAP: Internet Mail Access Protocol.

3. **Protocollo NNTP**
   Network News Transfer Protocol è un protocollo per l'invio di news. Un client locale di news mantiene le news e molti lettori SI basa su connessione TCP sulla porta 119.