Sistemi Distribuiti

2006/2007

prof. Cugola

Riva 'ZaX' Samuele

SOMMARIO

*P*agina

1) Introduzione

Un sistema distribuito è un insieme di computer indipendenti che appare all'utente come un unico sistema coerente.

Internet e il web sono sistemi distribuiti, ma non tutti i sistemi distribuiti operano via web, esistono altri settori come le Wireless Sensor Network (WSN), GPS, cellulari.

Perché è utile un sistema distribuito?
- ✔ prezzi e performance migliori;
- ✔ semplicità di evoluzione e modularità;
- ✔ resistenza ai guasti.

Lo sviluppo di questi sistemi è però molto più complesso rispetto ai classici sistemi informatici, bisogna infatti tener conto di 8 fattori fondamentali (Peter Deutsch):

1. la rete non è affidabile
2. la latenza può essere elevata
3. la banda di trasmissione è finita
4. la rete è insicura
5. la topologia del sistema è dinamica
6. esistono più amministratori
7. i costi di trasporto sono variabili
8. la rete non è omogenea

➔ *Funzionalità fondamentali*

Concorrenza
I nodi di un sistema distribuito (co)-operano in parallelo.

Assenza di clock globale
I clock dei client in rete sono diversi e solitamente non sincronizzati.

Guasti indipendenti
I guasti bloccano solo parzialmente il sistema, spesso a causa di errori di comunicazione.
E' complesso rilevare e correggere gli errori.

Gli obiettivi da raggiungere e i relativi problemi contro cui ci si scontra riguardano:

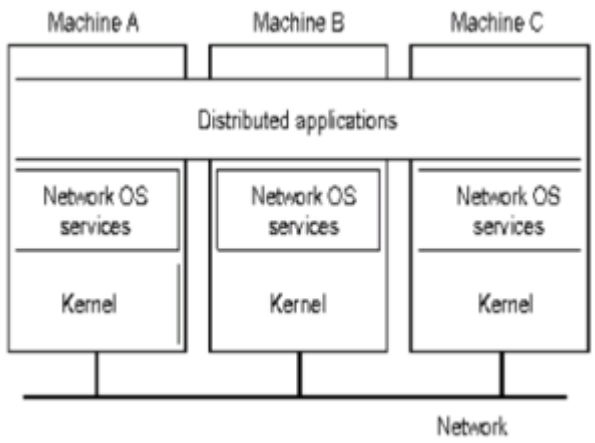
- **eterogeneità**: reti, hardware, OS e linguaggi di programmazione possono essere molto differenti.
- **openness**: determina se un sistema può essere esteso e/o reimplementato.
- **sicurezza**: è necessario mantenere confidenzialità, integrità, disponibilità.
- **scalabilità**: implemento la possibilità di incrementare/diminuire le performance a seconda dei nodi attivi; è complesso conoscere lo stato del sistema, non essendoci punti centralizzati.
- **gestione guasti**: complessità di individuazione e correzione dei guasti distribuiti.
- **concorrenza**: controllo sulle regioni condivise, a cui accedo in mutua esclusione.
- **trasparenza**: è necessario nascondere le differenze hardware, le rappresentazioni dei dati, la

replicazione...

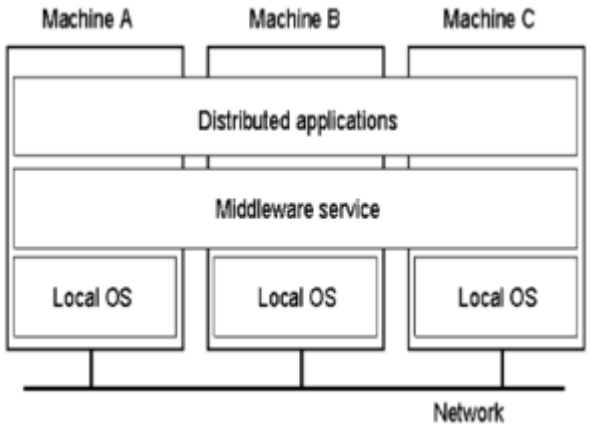
2) Architetture distribuite

Esistono 2 tipi di sistemi distribuiti: multiprocessore (più CPU nello stesso computer condividono memoria), multicomputer (N computer scambiano messaggi e cooperano).

Un'architettura software invece può essere:
- **Network OS**: le applicazioni usano direttamente i servizi di rete di un Sistema Operativo.
- **Middleware**: i programmi sono scritti sopra un nuovo livello che rende trasparente il tipo di SO, fornendo servizi dedicati ai sistemi distribuiti.



Network OS based



Middleware based

Il middleware è un software che permette interoperabilità tra diverse applicazioni, nascondendo le eterogeneità. Tramite middleware si alza notevolmente il livello di astrazione e viene semplificato lo sviluppo del software applicativo, fornendo servizi di comunicazione, applicazione e gestione anche molto avanzati.

➔ Modelli architetturali

I sistemi distribuiti sono costruiti seguendo diversi modelli:

- ✔ **Client-server**: server passivi forniscono servizi tramite API, che i client sfruttano usando comunicazione tramite messaggi (o RPC).
  I server sono implementati a livelli (tier), solitamente 2 o 3, rappresentati da più macchine che agiscono in modo distribuito.
  (es: 3-tier: interfaccia—*rete*—applicazione—*rete*—dati)
  Sfruttando al massimo la distribuzione è possibile avere architetture multi-tier.

- ✔ **Peer-to-peer**: tutti i nodi del sistema hanno lo stesso ruolo, fruiscono e forniscono servizi.
  Grande decentralizzazione permette di evitare *bottleneck* sul server e ha una buona scalabilità.

- ✔ **Object-oriented**: i componenti del sistema sono distribuiti come oggetti che incapsulano dati e forniscono API che ne permettono l'accesso. Sfrutta l'information hiding per creare un

"guscio" attorno ai dati che consente una migliore gestione delle risorse e il riutilizzo delle stesse. La comunicazione avviene tramite RPC.

- ✔ **Data-centered**: modello semplice in cui i diversi componenti comunicano con un nodo repository centrale da cui si può accedere ai dati in modo sincronizzato.

- ✔ **Event-based**: i vari nodi collaborano scambiandosi informazioni sugli eventi che avvengono nel sistema, tramite *publish* di notifiche e *subscribe* agli eventi di loro interesse. La comunicazione si basa su scambio di messaggi asincroni, che si appoggia ad un dispatcher unico che esegue routing in base alle sottoscrizioni verso i diversi eventi ricevuti dai nodi.

- ✔ **Mobile code**: permette di rilocare run-time codice e stato di un componente su un altro nodo, garantendo grande flessibilità a discapito di alcuni possibili problemi di sicurezza.

Il middleware viene usato come supporto in ogni modello, ma solitamente ad un tipo di middleware equivale un solo modello (RMI/CORBA sono object-oriented, Jini è data-centered, JMS event-based).

➔ ***Algoritmi e guasti distribuiti***

I sistemi distribuiti sono composti da molti processi in esecuzione allo stesso istante, che interagiscono in modo sincrono (esistono tempi massimi di "confine" che permettono di semplificare la comunicazione) o asincrono (non ci sono vincoli, alcune situazioni sono irrisolvibili).

Two-Army problem (A):
2 armate devono accordarsi su chi comanda e quando attaccare il nemico che è tra loro (l'attacco deve avvenire contemporaneamente per vincere), la comunicazione è SICURA, avviene tramite scambio di messaggi.

chi comanda? è possibile deciderlo sia nel caso sincrono che asincrono;
attacchiamo ora? non è possibile nel caso asincrono, i *clock* delle armate non sono sincronizzati (non posso decidere un istante in cui attaccare) e non conosco il tempo di latenza dei messaggi (arrivano in 2 minuti o impiegano ore?).

Nel caso sincrono è possibile definire la massima differenza tra tempi di attacco, selezionando un limite dopo il quale chi invia il messaggio attacca.
Poiché il messaggio prima o poi arriverà di certo (per ora la comunicazione è sicura), l'algoritmo è possibile anche se non perfetto.

(la contemporaneità non esiste, ma nel caso sincrono posso almeno avvicinarmi)

Nei casi reali sia i processi che la comunicazione possono avere dei guasti o errori.
Dividiamo le tipologie di *failure* in:

- **omission failures**: se avviene un fail stop il guasto può essere rilevato da altri processi, in caso

contrario avviene un crash.

- **byzantine failures**: i messaggi possono arrivare modificati o in più copie. Questo rende più complessa la gestione dell'errore perchè i processi guasti non si bloccano.
- **timing failures**: avvengono solo su sistemi sincroni quando sono violati i vincoli temporali.

Two-Army problem (B):
Come scoprire se una delle due divisioni è stata attaccata e sconfitta ?

Nel caso *sincrono* è sufficiente inviare un ACK ogni tot tempo per avvertire della propria presenza. In caso di mancanza di ACK per un periodo limite prestabilito, considero la divisione sconfitta.
Nel caso *asincrono* invece non è possibile capire se la divisione è stata sconfitta o se il messaggio è ancora in viaggio, non ho dei vincoli temporali a cui affidarmi.

Nel caso in cui i messaggeri possano essere catturati (canale con possibili errori), le due armate possono stabilire se attaccare o meno?
Il problema è irrisolvibile, anche nel caso sincrono, infatti anche usando gli ack se viene perso l'ultimo messaggio di conferma il destinatario non attaccherà, mentre il mittente sì.

Quindi nei sistemi distribuiti non è possibile raggiungere consenso deterministico tra i nodi. Nei casi reali ci si accontenta di avere una comunicazione sufficientemente buona per raggiungere consenso su base probabilistica piuttosto che deterministica.

3) Programmazione concorrente

La concorrenza permette di eseguire diversi flussi di processo in "contemporanea" su un unico processore, tramite switch tra un flusso ed un altro.

I processi non condividono memoria, hanno un proprio spazio di indirizzamento.
I thread condividono tra loro questo spazio e sono quindi più adatti al parallelismo (minor tempo di switch e facilità di manipolare dati condivisi).

I sistemi moderni usano un modello preemptive, che mette in stato di ready un processo in esecuzione, permettendo ad altri processi di entrare in esecuzione.

➔ Concorrenza in C

Usando il linguaggio C è possibile creare più processi tramite il metodo *fork()*, mentre i thread sono creati usando particolari funzioni della libreria *pthread*.

I thread condividono dati, file aperti, gestori di segnali, identificatori... mentre hanno un personale set di registri, puntatori allo stack, priorità.
La sincronizzazione tra thread avviene tramite *joining* usando la funzione *pthread_join(id, stato)*, che blocca l'esecuzione del flusso di processo chiamante fino al termine del thread specificato.

Una tecnica usata per creare mutua esclusione tra thread sono i *mutex*, indispensabili per proteggere dei dati condivisi. Tramite *condition variables* è possibile avere una sincronizzazione basata su un particolare valore "condizione", che evita di continuare a controllare la zona protetta (polling), in modo che i thread attendano semplicemente l'arrivo di un segnale che li "sblocchi".

PROCESSI	
`pid_t  fork()`	crea una copia identica del processo
THREADS	***pthread_t***
`int pthread_create(*id, *attr, funz, args)`	crea un thread associato alla funzione *funz*
`pthread_exit()`	termina un thread
`pthread_join(id, *exit_status)`	blocca il flusso fino al termine del thread id
MUTEX (lock sulle strutture condivise)	***pthread_mutex_t***
`pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;`	inizializzo il mutex *m* con i valori di default
`pthread_mutex_lock(*m)`	acquisisco un lock su *m*
`pthread_mutex_unlock(*m)`	sblocco la variabile mutex *m*
CONDITION VARIABLES (permette sincronizzazione basata sul valore di una variabile)	***pthread_cond_t***
`pthread_cond_t cv = PTHREAD_COND_INITIALIZER;`	inizializzo la condition variable *cv*
`pthread_cond_wait(cv) // sono in un mutex`	blocca il thread fino ad un *signal* su *cv*.

`pthread_cond_signal(cv) // sono in un mutex`	sveglia un thread messo in wait su *cv*.
`pthread_cond_broadcast(cv)  // sono in un mutex`	sveglia tutti i thread in attesa su *cv*.

➔ ***Concorrenza in JAVA***

Java supporta la concorrenza a livello di linguaggio (C deve importare la libreria pthread), fornisce quindi delle classi e metodi per istanziare i thread, sincronizzarli e implementare condition variables.

Il modello di esecuzione è non-deterministico (senza un ordine predefinito), Java inoltre usa di default la preemption (implementa RoundRobin tra i thread, se possibile), utilizzabile attivamente dal programmatore invocando il metodo *yield()* [metodo che cede l'esecuzione ad un altro thread].

I thread vengono realizzati ereditando dalla superclasse java.lang.Thread, oppure implementando l'interfaccia Runnable.
Entrambi i metodi comportano la specifica di una metodo *run()* in cui si specifica il comportamento del thread una volta messo in esecuzione:

THREADS	***java.lang.Thread***

`extends Thread --> implemento funzione run()`
`thr = new nomeClasse();`
`thr.start()`
	classe è un Thread; *run()* viene eseguita quando invoco *thr.start();*

`implements Runnable --> implemento run()`
`run = new nomeClasse();`
`new Thread(run).start();`
	classe è Runnable, la istanzio e poi eseguo *run()* con *new Thread(run).start();*

`thr.join();`	attendo il termine del thread *thr*

Safety vs Liveness

Un sistema concorrente è corretto solo se rispetta le proprietà di **safety** (senza errori) e **liveness** (si giunge sicuramente ad una soluzione). Spesso queste due proprietà contrastano (mutex migliorano safety ma possono causare deadlock), quindi è necessario sviluppare attentamente gli algoritmi distribuiti.

In un sistema *safe* ogni oggetto si protegge da possibili violazioni di intergrità tramite *tecniche di exclusion* che evitano effetti inconsistenti. Esistono 3 strategie nel caso multithread:

- Immutabilità: creo oggetti senza stato o con metodi che non lo modificano (utilizzabile solo in certe applicazioni, è un approccio totalmente non flessibile);
- Esclusione dinamica: tramite lock posso serializzare l'accesso a metodi sincronizzati, evitando conflitti in scrittura/lettura da parte di thread concorrenti.
- Esclusione strutturale: l'accesso seriale avviene come scelta esplicita nell'applicazione. (?)

MUTUA ESCLUSIONE (serializzo accesso ai dati)	
`synchronized funzione () {... }`	Creo funzioni ad accesso serializzato
`synchronized (oggetto) {... }`	Sincronizzo un oggetto
`synchronized (this) {... }`	Creo blocchi di codice sincronizzato

-- i metodi synchro non si ereditano	

Usando la clausola *synchronized* in Java posso quindi creare blocchi di codice ad accesso controllato:

```
public void funz() {
   ...                                  // accesso parallelo consentito
   synchronized (oggetto) { ... }       // zona ad accesso serializzato!
   ...                                  // accesso parallelo consentito
 }
```

E' possibile avere intere classi sincronizzate per avere una safety perfetta, ma non esisterebbe più alcuna concorrenza tra i threads.

Deadlock, variabili volatili...

- Un deadlock può avvenire se diversi thread accedono a 2 o più oggetti in mutua esclusione ed ognuno di loro mantiene un lock richiedendone altri, già bloccati. In questo caso il flusso dei processi si interromperebbe e sarebbe impossibile proseguire.
  [es: p1 chiede A bloccando B, p2 chiede B bloccando A... deadlock!!]

- Per evitare che certe variabili siano lette in modo *safe* ma poi vengano tenute in memoria a lungo senza verificare che siano ancora valide, si identifica la variabile come *volatile*, costringendo i processi a rileggere il suo valore ogni volta che la si vuole usare, evitando inconsistenze.

- Tramite adapters è possibile creare delle collezioni sincronizzate (generalmente non lo sono).
  Le classi adapter permettono a diverse classi di lavorare insieme anche se le proprie interfacce non sono compatibili.

Bisogna usare la sincronizzazione con cautela, perchè invocare un metodo synchronized è 4 volte più lento del normale, inoltre un uso massicio limita la concorrenza e può causare problemi di deadlock.

Condition Synchronization

In Java si usano classi Monitor che forniscono mutua esclusione attraverso condition variables, usando 2 metodi sincronizzati che modificano una variabile di stato.
Questi metodi controllano se possono agire nel monitor basandosi sul valore della condition variable, che poi modificheranno adeguatamente prima di uscire dal Monitor.

Tramite la funzione *wait()* un processo attende che altri lo risveglino (nel frattempo pur essendo in un metodo sincronizzato cede il controllo ad altri), mentre con *notify()* o *notifyAll()* un processo segnala e risveglia uno o tutti i processi in attesa sulla condition variable condivisa.

MONITOR (equivalente alle condition variables)	

`class Monitor {`
`  protected int condvar;`
`  // contiene metodi synch che modificano condvar`
`}`
	La classe Monitor inizializza una variabile protetta che viene aumentata/ridotta da metodi sincronizzati

`while (!condvar) { wait(); }`
`// modifico la variabile...`
`notifyAll();`
	Attendo se condvar è rispettata...
	Prima di uscire dal metodo synch notifico

E' possibile evitare eccessivo overhead di comunicazione usando *notify()*, ma solo se sono certo che ci sia al massimo un solo thread da svegliare dalla *wait()* (*es:* un solo produttore e molti consumatori, evito di svegliare tutti i consumatori in attesa)

Esempio di funzione interna al Monitor che agisce sulla variabile:

```
public synchronized void funz() throws InterruptedException {
   while (!cond)  {    // se cond è zero il processo attende
      wait();          // solo dopo un signal() verrà ricontrollata la condizione
   }
   ...                          // eseguo codice in mutua esclusione
   cond--;                      // modifico lo stato
   notifyAll();                 // esco dalla zona e lo segnalo a tutti i thread in attesa
 }
```

Per evitare deadlock si può sfruttare una wait temporizzata *wait(time)* che sblocca dall'attesa i thread dopo un tempo *time*, oppure usare la busy wait tramite *Thread.yield()* al posto della wait(), in modo da avere un loop continuo di controllo verso la variabile condition che non necessita di notifiche.

Nel caso in cui un thread client invochi una funzionalità lato server di cui voglio conoscere il risultato futuro?
Il server può rispondere quando ha terminato l'esecuzione con una completion callback, un messaggio verso il client che indica il termine dell'esecuzione passando parametri e risultati. Nel frattempo il client ha potuto proseguire le proprie operazioni asincronamente dal server.

Se il client dovesse attendere di conoscere quando e se il server ha eseguito il task è più comodo un approccio che consiste nell'invocare *Thread.join()*, in modo che il client attenda la fine del thread eseguito su server senza fare nulla.

4) Facilities di comunicazione

➔ Comunicazione tramite socket

Uno dei più comuni mezzi di comunicazione fa uso dei socket, un'astrazione software contenente le informazioni necessarie per la ricezione e trasmissione dei dati tra processi remoti e locali.
Ogni socket è identificato da una doppia coppia porta-indirizzo_IP che identifica i due processi che effettueranno la comunicazione.
Tramite socket è possibile implementare sia una comunicazione tramite datagram connectionless (UDP), sia una comunicazione orientata alla connessione (TCP).

Implementazione in C

L'API di implementazione della comunicazione TCP tramite socket in C è piuttosto complessa, comporta un diverso approccio lato client e lato server:

- **Server**: creo un socket [socket()] e lo inizializzo [bind() + listen()]. Attendo connessioni [accept()], a cui di solito si risponde creando un secondo processo che gestisce le richieste mentre il server ritorna in attesa di altre connessioni.
- **Client**: creo un socket [socket()] e mi connetto specificando ip e porta del server [connect()].

Client e server comunicheranno in seguito usando le primitive *send* e *recv*, in cui viene specificato il socket, i dati da trasferire/ricevere ed eventuali istruzioni avanzate.
Il socket viene chiuso tramite funzione *close*, che libera lo spazio in memoria.

Per quanto riguarda la comunicazione UDP senza connessione, l'inizializzazione è più semplice, infatti per entrambi i punti di comunicazione è sufficiente inizializzare il socket [socket], effettuare il binding [bind] e poi usare i metodi *sendto* e *recvfrom* per comunicare specificando ovviamente l'indirizzo a cui inviare i datagram.

Implementazione in JAVA

In Java esistono 5 classi principali che aiutano a sviluppare applicazioni che utilizzando socket per comunicare tra loro.
Per quanto riguarda TCP l'implementazione lato server consiste nell'usare la classe ServerSocket specificando la porta, poi si accettano richieste dei client tramite il metodo accept, che restituisce un oggetto Socket. La comunicazione vera e propria avviene usando gli stream.

Esempio:

```
ServerSocket sock = new ServerSocket(porta)
Socket s = sock.accept();

in = new BufferedReader( new InputStreamReader( s.getInputStream() ));
stringa = in.readLine();
```

Per UDP si usa la classe DatagramSocket e DatagramPacket, in cui diventa necessario specificare l'indirizzo del destinatario, sfruttando le funzionalità della classe InetAddress.
La comunicazione sfrutterà le funzioni *send* e *receive* di DatagramSocket.

Tramite il protocollo di rete IP multicast è possibile inviare datagrammi UDP a più destinatari in modo efficiente, specificando una porta e il gruppo di IP a cui inviare (192.168.2.255 invia a tutto il

sottogruppo di 253 destinatari identificati da 192.168.2.*).
Java supporta il multicast con la sottoclasse dei datagram MulticastSocket, che fornisce i metodi *joinGroup* e *leaveGroup*.

➔ ***Remote Procedure Comunication (RPC)***

RPC è una tecnologia che permette ad un'applicazione di eseguire delle funzioni in remoto (su altri computer) in modo trasparente e del tutto simile alla normale esecuzione di primitive di sistema locali.
Per rendere trasparente la comunicazione RPC è necessario creare un middleware che effettui la serializzazione dei dati (trasformo strutture dati in un flusso trasportabile in pacchetti) e il marshalling dei parametri (conversione in modo che i dati siano comprensibili ad entrambi i sistemi comunicanti).
Il middleware RPC consiste nella creazione di due *stub* client/server tra il livello applicativo e quello di rete.

Viene usato un Interface Definition Language (IDL) per alzare il livello di astrazione e abilitare la definizione di servizi in un linguaggio indipendente dalla piattaforma.

Lo standard *de facto* su internet è la **versione SUN di RPC**: specifica il formato dei dati con XDR; permette il passaggio di parametri solo per copia (al massimo posso effettuare una chiamata per value/result e sovrascrivere quindi il parametro locale) ; trasporto possibile su TCP e UDP.
Il ciclo di sviluppo di RPC consiste nella creazione di un file di specifica XDR (nome.x) che servirà per ottenere i due stub di middleware e la libreria principale, tramite esecuzione di *rpcgen nome.x*.
Il server e il client dovranno invocare le funzioni remote usando nomi e terminologie che corrispondano al file di specifica, importando la libreria precedentemente creata con *rpcgen*.

Per scoprire quale server fornisce una determinata funzionalità e come stabilire una connessione verso di esso, SUN introduce un processo demone detto *portmap* in cui i server inseriscono la porta e l'identificatore del servizio, permettendo ai client di connettersi dopo una richiesta al demone.
Se il client non conosce il server che contiene una determinata funzionalità può inviare richieste multicast.

La **versione DCE**, usata come base da DCOM e .NET, consiste in un insieme di specifiche e un'implementazione di riferimento, fornisce diversi servizi "sopra" RPC.

Anche DCE usa processi demone, uno per ogni server, dove vengono registrati gli *endpoint* che il client userà per usare direttamente le funzioni remote. I client per conoscere verso quale indirizzo effettuare delle richieste si connettono ad un directory server conosciuto (può essere anche distribuito) in cui esiste un secondo demone, detto *binder*, in cui i server registrano i propri servizi e l'indirizzo.

Esiston **versioni alternative** di RPC, come la versione leggera (si usa una memoria locale condivisa per memorizzare i dati) e quella asincrona (il client non attende il risultato dell'operazione ma prosegue appena riceve un ACK dal server che segnala l'inizio dell'esecuzione della funzione).

➔ ***Remote Method Invocation***

RMI sfrutta la stessa idea di RPC (a volte viene addirittura implementato "sopra" un layer RPC), ma

con diversi costrutti adattati all'object orientation del linguaggio di programmazione Java.
RMI fornisce una API che rende trasparente al programmatore i dettagli della comunicazione di rete, in modo del tutto simile a RPC, ma con funzioni più semplici ed intuitive, grazie all'utilizzo di oggetti e classi pre-fornite da Java.

L'IDL di RMI non necessita di compilazione (da Java 5 in poi), supporta l'ereditarietà, la gestione degli errori, si basa sull'utilizzo delle interfacce degli oggetti che si vogliono usare da remoto.
Prima di Java 5 il *proxy* lato client e lo *skeleton* lato server erano creati dal compilatore *rmic* e usati per il marshalling e la gestione delle connessioni. Ora sono generati automaticamente a run-time.
Tra gli aspetti innovativi abbiamo la possibilità di passare parametri per riferimento, il download delle classi e l'esistenza di proxy dinamici.

Le classi utilizzabili da remoto devono implementare un'interfaccia che a sua volta estende *java.rmi.Remote* ed i cui metodi gestiscono l'eccezione RemoteException, utile per identificare eventuali i problemi di collegamento server/client.
Per accettare connessioni una classe dovrà inoltre essere esportata (export) estendendo una sottoclasse RemoteObject (tipicamente UnicastRemoteObject).

Per ottenere un riferimento remoto uso il servizio di lookup *rmiregistry*, in cui le classi "server" si registrano mentre i "client" ottengono i riferimenti necessari (lo stub dell'interfaccia remota: il proxy) e può iniziare la comunicazione agendo sullo stub come se fosse una classe locale.
(una volta ottenuto lo stub è possibile passarlo ad altri componenti senza problemi (è serializzato))

rmiregistry mantiene un collegamento tra un nome simbolico e un oggetto, è possibile accedervi tramite la classe *java.rmi.Naming* e i suoi metodi *lookup*, *list*, *bind*, che permettono rispettivamente di ottenere il riferimento in base ad un nome simbolico, vedere una lista dei nomi disponibili, e registrarne un nuovo servizio. Tutti questi metodi sollevano l'eccezione MalformedURLException.

Il passaggio di parametri avviene automaticamente *per riferimento*, a meno che l'oggetto sia serializzabile, in quel caso viene passata una copia. La modalità di passaggio dei parametri viene comunque decisa staticamente.

RMI	***java.rmi.****

`classe implements interf {`
`// i metodi Remote lanciano RemoteException`
`}`
	La classe remota implementa un'interfaccia Remote e tutti i suoi metodi.

`interf extends Remote {`
`// medodi remoti...`
`}`
	L'interfaccia estende Remote e dichiara i metodi che saranno utilizzabili in remoto

`server extends UnicastRemoteObject implements interf { // implemento i metodi di interf }`	Una classe "server" deve essere esportata, cosa automatica se si estende URObject.
`Naming.rebind (//IP:porta/Nome, new server())`	Istanzio il "server" legandolo a rmiregistry. Specifico IP:porta e nome di identificazione
`interf serv = Naming.lookup (//IP:porta/Nome)`	Il "client" ottiene il proxy del "server" tramite lookup su rmiregistry.

esempio :

```
// creo interfaccia Remote
```

```
public interface NomeInt extends Remote {
      // funzioni varie() throws RemoteException;
}



// creo Classe ''server'' che estende UnicastRemote e implementa l'interfaccia Remote
public class NomeServ extends UnicastRemoteObject implements NomeInt {
    // implemento funzioni dell'interfaccia NomeInt
    // creo costruttore di default che lancia RemoteException
  // main {
      NomeServ c = new NomeServ();
      Naming.rebind(''//IP:porta/NomeID'', c);
  }
}
// creo classe ''client'', non è necessario estendere l'interfaccia Remote se
// voglio solo contattare il server, senza fornire servizi remoti a nessuno
public class NomeCli  {
  // main {
      NomeSer serv = null;      // creo interfaccia che otterrò dal registry
      serv = (NomeSer) Naming.lookup(''//IP:porta/NomeID'');
      // ora posso usare metodi sul server con serv.nomeMetodo();
  }
}
```

Su *rmiregistry* dovrà essere registrata l'interfaccia del server così che il client possa ottenerla tramite lookup.
Per quanto riguarda la concorrenza, il server RMI deve verificare che l'implementazione sia thread-safe, quindi è necessario aggiungere il costruttore *synchronized* ai metodi critici.

---

esempio 2 (senza codice):
Implemento una comunicazione tra due nodi alla pari, in modo che nessun oggetto svolga esclusivamente il ruolo di client o di server. Diventa quindi necessario creare anche un'interfaccia Remote per la seconda classe, oltre a registrarla su *rmiregistry* oppure passare alla prima classe il proprio riferimento in modo esplicito, in modo che la comunicazione sia bidirezionale.

// creo 2 interfacce *Int1* e *Int2*, che estendono Remote e definiscono i metodi utilizzabili da remoto.

// Il nodo N1 si crea come il precedente server, si registra su *rmiregistry* e attende.
// in questo caso poniamo che il costruttore inizializza una lista vuota di <*N2*>, che consente
// la comunicazione bidirezionale.

// Il nodo N2 estende UnicastRemoteObject ed implementa interfaccia *Int2*
    // creo costruttore e implemento i metodi definiti nell'interfaccia *Int2*
    // nel main istanzio la classe e ottengo il riferimento al nodo N1 tramite *lookup()* sul registro
    // eseguo metodi sul nodo N1, passando il riferimento alla mia classe tramite *this*

// Nel nodo N1 salvo i riferimenti passati dai N2 nella lista, a cui posso accedere
// tramite *for (Client c in ListaClient)*. Dopo ogni connessione è necessario effettuare *rebind()*

Attivazione dinamica degli oggetti

Gli oggetti remoti possono essere creati su richiesta in modo dinamico, usando i servizi delle classi contenute nel package *java.rmi.activation*, utilizzabili estendendo la sottoclasse di RemoteObject *Activatable* e lanciando il demone attivatore *rmid*.

Le classi "server" dovranno quindi estendere Activatable invece che UnicastRemoteObject, richiamando il costruttore tramite *super(activationId, porta)*. Nel main prima di eseguire rebind() si dovrà ottenere il descrittore del gruppo di attivazione (ActivationGroup), che poi registrerò presso il registry usando *Activatable.register()*.
Tutte queste modifiche sono ovviamente trasparenti alle classi "client", che non sono di tipo Remote.

Download del codice

Quando un client usa in remoto un sottotipo di un oggetto che è presente sul nodo destinatario, su quest'ultimo possono sorgere dei problemi perchè non ne conosce lo stub e quindi non può manipolare la sottoclasse adeguatamente, sollevando delle eccezioni.
Per questo motivo nasce la possibilità di effettuare il download del codice dell'interfaccia, mettendo in esecuzione il client specificando la posizione in cui si trova il bytecode del sottotipo, in questo modo:

```
java   nomeClasse   -Djava.rmi.codebase=''http://URL/del/file/'' .
```

Per quanto riguarda la sicurezza, ogni classe "server" deve specificarne un gestore tramite la funzione *System.setSecurityManager(new RMISecurityManager())*, specificano un file di policy *java.policy* contenente le path da cui è permesso scaricare il codice *grant codebase PATH {...}*.

RMI non maschera completamente la distribuzione, perchè la semantica delle classi varia (bisogna specificare eccezioni e interfacce remote), per una precisa scelta di design.
Rispetto ad altri sistemi di distribuzione di oggetti è meno potente, ma è la base di altri servizi più complessi, come Jini e J2EE.

➔ ***Comunicazione message-oriented***

RPC ed RMI supportano solo una comunicazione punto-punto e usano un modello sincrono, poco flessibile e costoso, è possibile usare un diverso approccio basandosi sullo scambio di messaggi-eventi per ottenere un modello di comunicazione asincrono, che supporti interazione multi-point.

Message Oriented Middleware (MOM) sono servizi forniti a livello applicazione da diversi server di comunicazione, che eseguono il routing dei messaggi.
La comunicazione nei MOM può essere sincrona o asincrona, non persistente (transient, si possono perdere messaggi se uno dei due nodi non è attivo) o persistente (i messaggi sono salvati e spediti in seguito se un nodo era inattivo, tipo e-mail).
Esistono 4 alternative di comunicazione transient (asincrona e 3 sincrone [receipt-based, delivery-based, response-based]) e 2 alternative per il modello persistente (asincrono e sincrono [attende ACK])

Esistono 2 modelli di MOM, entrambi basati su scambio asincrono di messaggi:

- **Message queuing**: comunicazione punto a punto, persistente e asincrona.
  Sono usate due code, una in cui sono inseriti i messaggi in attesa di essere ricevuti dal nodo e una per i messaggi in uscita (modello di JMS).

I nodi possono fare *get()* o *poll()* sulla propria coda e *put()* su code di altri nodi, dopo aver eseguito un *attach_queues()* sulla coda desiderata, identificata da un ID unico.
E' possibile sviluppare modello client/server, in modo che più client mandino richieste verso una singola coda in entrata condivisa tra i server, in modo asincrono.
Gestori delle code e servizi di lookup sono solitamente distribuiti in modo da rendere trasparente la comunicazione distribuita.

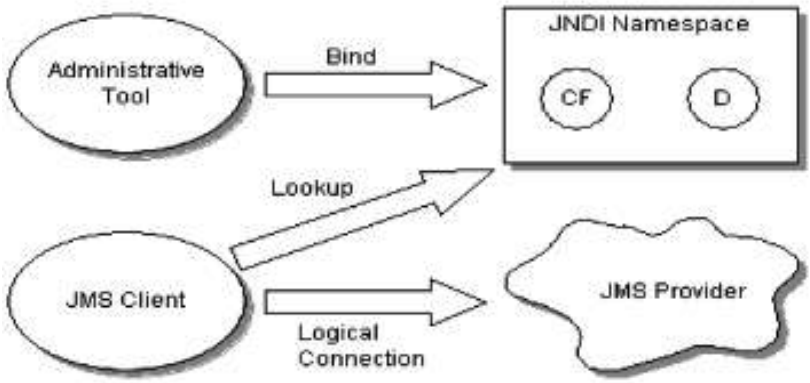
- **Publish-subscribe**: comunicazione multipoint, transiente e asincrona.
  I componenti possono pubblicare notifiche di eventi in modo asincrono e iscriversi a classi di eventi tramite sottoscrizione. La API è molto semplice, con due primitive: *publish()* e *subscribe()*.
  Le sottoscrizioni sono collezionate da un dispatcher di eventi centralizzato o distribuito.
  E' possibile iscriversi ad eventi specificando il tipo di evento (eventi su "Ricette", *topic-based*) o il contenuto desiderato ("Ricette"+ "primi piatti" + "..." permette di specificare maggiori filtri sugli eventi, è detto *content-based*).

Java Message Service

JMS definisce una API che permette di creare, inviare, ricevere e leggere messaggi.

Le parti fondamentali di JMS sono:
- provider: implementa le specifiche JMS;
- client: il software (JMS o meno) che usa i servizi del provider sfruttando la API fornita.
- dominio: si può scegliere o il modello queue-based o quello publish-subscribe (topic-based).

Per favorire la portabilità, i dettagli di un provider devono essere trasparenti. Per questo motivo vengono definiti degli oggetti amministrati da JMS, a cui i client possono accedere tramite un namespace JNDI: ConnectionFactory (usato per stabilire la connessione al provider) e Destination (usato per specificare la destinazione dei messaggi).



Gli amministratori creano gli oggetti ed effettuano binding sul namespace JNDI.

I client effettuano Lookup e possono usare gli oggetti per accedere al providere e spedire messaggi in modo trasparente

Le relazioni tra oggetti JMS vengono riprodotte da classi Java:
- Connection: collegamento tra il client e il provider;
- Destination: incapsula l'identità della destinazione di un messaggio;

- Session: contesto usato per inviare e ricevere messaggi, creato attraverso Connection

Un client JMS tipicamente usa JNDI per trovare un oggetto *ConnectionFactory* (con cui ottiene una Connection verso il provider) e uno o più oggetti *Destination*.
Tramite Connection ottengo una o più Session che, insieme alla Destination scelta, permette di creare MessageProducer e MessageConsumer con cui inviare e ricevere messaggi.

JMS	***javax.jms.* javax.Naming.****

`Context ic = new InitialContext()`
`Queue q = ic.lookup(''queue'');`
`QueueConnectionFactory cf = ic.lookup(''qcf'');`
`ic.close();`
	nel main inizializzo il contesto di init da cui ricavo la coda e la ConnectionFactory. Ho finito di usare JNDI.

`QueueConnection c = cf.createQueueConnection();` `QueueSession s = c.createQueueSession(false,0);`	Creo connessione e sessione.

`QueueSender qsend = s.createSender(q);`
`TextMessage mex = s.createTextMessage();`
`mex.setText(''stringa'');`
`qsend.send(mex);`
	Ottengo il Sender dalla sessione specificando la coda da usare. Creo messaggi e li invio usando send().

`QueueReceiver qrecv = s.createReceiver(q);`
`TextMessage mex;  // non serve fare new`
`c.start();`
`mex = qrec.receive();`
	Ottengo il Receiver spedificando un'altra coda. Creo messaggio vuoto, eseguo start e attendo l'arrivo dei dati in coda.

`Queue coda_temp = s.createTemporaryQueue();`
`mex.setJMSReplyTo(coda_temp);`
`mex.getJMSReplyTo();`
	Chi invia può creare coda temporanea su cui attendere risposte... chi risponde deve ottenere la coda tramite *getJMSQueue()*.

`qs.commit()` `qc.close()`	Commit della sessione, chiudo la connessione

Per ricevere risposta ai messaggi inviati posso creare una coda temporanea, l'approccio più efficiente è crearne una (tramite *setJMSReplyTo(coda)*) a cui tutti i server accederanno per rispondere. Tramite id di correlazione posso in seguito distiguere le diverse richieste.

JMS - Publish Subscribe

Il modello di comunicazione publish-subscribe consiste nello scambiarsi messaggi specificando un argomento (topic) che altri nodi interessati possono sottoscrivere.
I nodi pubblicano i messaggi ad un *broker* intermediario, su cui gli altri nodi possono sottoscrivere l'argomento, ricevendo così tutti i messaggi di quel topic.

Questo modello consente miglior scalabilità grazie alla possibilità di effettuare operazioni parallele, cache dei messaggi, routing tree-based...

In pub/sub non esiste una coda locale, quindi se i "sottoscrittori" non sono in linea al momento dell'arrivo dei messaggi, li perderanno, ma se più nodi si sottoscrivono allo stesso topic, riceveranno tutti i messaggi come in un modello multicast di comunicazione.

JMS	***javax.jms.* javax.naming.****

`Context ic = new InitialContext()`
`Topic topic = ic.lookup(''Topic'');`
`TopicConnectionFactory tf = ic.lookup(''tcf'');`
`ic.close();`
	Inizializzo il contesto di init da cui ricavo il topic e la connectionfactory.
	Ho finito di usare JNDI.

`TopicConnection c = tf.createTopicConnection();` `TopicSession s = c.createTopicSession(false,0);`	Creo connessione e sessione.
`TopicPublisher pub = s.createPublisher(topic);`	Il nodo che invia messaggi si inizializza

`TextMessage mex = s.createTextMessage();`
`mex.setText(''stringa'');`
`pub.publish(mex);`
	sul topic e crea il messaggio. Poi lo invia tramite *publish*.

`TopicSubscriber sub = s.createSubscriber(topic);`
`sub.setMessageListener(new MsgListener());`
`c.start();`
`System.in.read()`
`c.close();`
`-- creo MsgListener con metodo` *`onMessage(Message)`*
	Il nodo che sottoscrive si inizializza sul topic e crea un Listener dei messaggi di quel topic. Leggo usando *System.in.read()*.

`// importo classi joram`
`AdminModule.connect()`
`jndi = creo...`
`jndi.bind( topic_creata );`
	Permette di creare nuovi topic a cui è possibile sottoscriversi o pubblicare messaggi.

`s.commit()` `c.close()`	Chiudo la connessione e la sessione

E' possibile specificare se si vogliono usare o meno gli ACK per controllare l'arrivo dei messaggi e anche se si vuole una trasmissione persistente o meno, tramite flag nella creazione delle sessioni e dei messaggi:

`c.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);`
Creo una sessione spedificando la modalità di acknowledgement da utilizzare.

`producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);`
Il produttore specifica in che modo inviare i messaggi.

L'invio dei messaggi può essere persistente (la spedizione è garantita) o meno. Il caso NON_PERSISTENT specifica che il messaggio se arriva ne arriva al massimo uno, mentre con PERSISTENT arriva sicuramente un messaggio (a livello applicativo, mentre i duplicati esistono ma non vengono notificati).

Ovviamente in JMS ogni classe può usare comtemporaneamente sia Topic che Queue per gestire diversi tipologie di comunicazione in contemporanea, usando Common Interfaces :

- faccio lookup su JNDI di una Destination generica (sopra interfaccia di Topic e Queue)
- creo Session e Connection generiche che poi specifico a seconda dei bisogni.
- uso MessageProducer.send(Destination, msg) per inviare a topic o code.

In questo modo sintatticamente posso usare entrambe le tecniche senza differenze.

Per quanto riguarda la concorrenza possono nascere dei problemi con più thread.
L'ordinamento è FIFO tra mittente-destinatario, mentre ovviamente non ci sono garanzie per temporizzazione tra più nodi.

Il modello dei messaggi contiene un HEADER, [PROPRIETA'], CORPO del messaggio.
Nell'header inserisco i campi fissi più importanti: identificatore, priorità, eventuale ReplyTo.
In proprietà posso mettere campi addizionali informativi, come coppie <string,value>.
Il corpo del messaggio contiene il vero e porprio messaggio inviato.

➔ ***Data space condiviso (JavaSpaces)***

Il modello di condivisione dei dati è stato proposto negli anni 80 per programmazione parallela,

usando il linguaggio di programmazione LINDA, basato sulle tuple (lista ordinata di oggetti).
LINDA specifica una comunicazione persistente e content-based, i dati vengono memorizzati in tuple che vengono memorizzate in uno spazio condiviso (tuplespace) su cui sono possibili diverse operazioni standard (out, rd, in...) e altre opzionali.

LINDA quindi è un modello di comunicazione in cui i client condividono uno spazio virtuale organizzato in tuple.
La comunicazione è detta "generativa", usa le funzioni fondamentali *out* (inserisco una tupla nello spazio)*, in* (leggo ed elimino dallo spazio)*, rd* (leggo senza eliminare).
Sia per *in* che per *rd* vengono lette le tuple a seconda di un template passato per parametro. Se non trovo nessuna tupla che rispetta il pattern specificato si ferma la richiesta in attesa della tupla richiesta (esistono anche soluzioni più flessibili).

JavaSpaces

JavaSpaces è l'implementazione SUN del modello Linda, è fornito come parte dell'architettura Middleware Jini.
- L'API di JavaSpaces contiene le funzioni write() [*out*], read() [*rd* bloccante o meno], take() [*in*].
- Esistono le versioni di probe *xxxIfExists()* delle 3 funzioni.
- C'è una funzione di notifica *notify()*.

Le tuple sono definite dall'interfaccia Entry, usata sia per descrivere servizi che per scrivere negli spazi virtuali, può estendere l'oggetto *AbstractEntry* che fornisce le funzioni di base.
Non si possono inserire tipi primitivi in una Entry e deve esistere un costruttore vuoto (uso *super()*).

Una Entry *ent* si può usare come fosse un template per verificare la corrispondenza con un'altra Entry *ent2* (corrispondono se la classe di *ent2* è la stessa (o una superclasse) di *ent*, e se ogni campo non nullo del template corrisponde ad un campo di *ent*).

Architettura Jini

Jini fornisce vari servizi, tra cui JavaSpaces, registrati su un Lookup Service raggiungibile dai client del sistema; contiene classi "di aiuto" (Discovery Classes ...) che semplificano la fruizione dei servizi disponibili.

Vediamo esempi per quanto riguarda l'uso dei javaspaces:

> definisco una tupla:

```
class Pentry extends AbstractEntry {
 public String attr1, attr2;
 public int val1;

 public Pentry () {
   super();
 }

 public Pentry (String cc) {
   super();
   this.attr1 = cc; // ...
 }
```

```
}
```

> ottengo uno spazio condiviso e cerco di leggere dal javaspace in base ad un filtro:

```
JavaSpace js = s.lookup (new ServiceTemplate (null, new Class[]{JavaSpace.class}, null));
Pentry en = (Pentry) js.read(new Pentry (variabili,filtro), null, time);
```

Nell'implementazione Jini di publish-subscribe la sottoscrizione avviene invece tramite *addXXXListener()*, metodo che DEVE ritornare una classe EventRegistration.
La pubblicazione di messaggi invece avviene tramite *notify()* di una classe RemoteEvent.

(vedi capitolo 11 per maggiori informazioni su Jini)

5) Naming

I meccanismi di Naming sono utilizzati per fornire un nome logico ad un'entità: la struttura di questo nome influenza il modo con cui essa può essere utilizzata (e risolta) in un sistema distribuito.

Ad ogni entità si può accedere in diversi modi, che possono cambiare nel tempo... per questo motivo sono introdotti dei riferimenti univoci statici per distinguere i componenti, in modo che venga disaccoppiato il "naming service" (id delle entità) dal "location service" (indirizzi per raggiungerle).

Il processo che consente di ottenere l'indirizzo valido per accedere ad un'entità è detto name resolution, il cui funzionamento cambia in base allo schema di naming usato:

- **flat naming**: i nomi sono "piatti", stringhe senza struttura e contenuto (come RMI).
- **structured naming**: le entità sono organizzate in un namespace, un grafo etichettato composto di foglie (le entità) e directory (tipo FS).
- **attribute base naming**.

➔ *Flat Naming*

L'approccio più semplice per venire a conoscenza dei riferimenti di un'entità è fare **broadcast/multicast** (LAN) o **forward dei puntatori** (per nodi mobili, è una tecnica che crea una catena di proxy ognuno con i riferimenti al successivo).

Altri approcci sono: **home-based** (ha bassa mobilità, deve esistere un home-address statico per ogni entità che segnala eventuali spostamenti dell'indirizzo di riferimento), **DHT** (ha grande scalabilità, fornisce put()/get() verso una hash-table distribuita. I nodi sono raggruppati in modo da formare una rete "logica" ben definita [overlay network]).

E' possibile implementare un **approccio gerarchico**, in cui la rete viene rappresentata come un albero, dividendola in domini e sottodomini in cui le foglie sono le reti locali.
Il lookup avviene risalendo l'albero dalle foglie verso la radice. Quando un riferimento è conosciuto da qualche nodo, inoltra la richiesta verso la destinazione. Root conterrà tutte le informazioni su tutti i client. Ottimizzazioni sono possibili usando cache e distribuendo il nodo root in modo che non diventi bottleneck del sistema.

esempio DHT: Chord

Ogni nodo è associato un identificatore, così come ogni risorsa ha una chiave unica. Nodi e chiavi sono organizzati in un anello logico, ordinato in modo tale che ogni nodo con id x conosce il primo nodo con id $y > x$. La risorsa con chiave k viene gestita e memorizzata dal primo nodo con $x>k$ (in modo che quando cerco la risorsa k mi basta cercare il nodo appena superiore a k).

Per ottenere una determinata risorsa dovrei "girare" tutto l'anello, ma Chord usa una tecnica che permette di comunicare direttamente con nodi *finger* della rete (il 1,2,4,8,16, ... memorizzandone i riferimenti in una tabella), diminuendo di molto la comunicazione tra nodi, permettendo di "saltare" in zone dell'anello senza doverlo girare tutto.

I problemi nascono all'aggiunta/rimozione di nodi, che constringe alla riorganizzazione delle tabelle

di finger e anche allo spostamento delle risorse (entra nodo con id 40 dopo il nodo 35 e prima di 43, le risorse 36-39 devono essere passate dal nodo 43 al 40).

➔ ***Structured Naming***

Le risorse sono definite da un *path name* assoluto o relativo (es: */dir0/dir1/n2*) e come nei filesystem esistono hard e soft link.
I namespace su larga scala sono sistemi distribuiti su diversi *name server* organizzati gerarchicamente e partizionati su più livelli, come nel caso dei **DNS**.

Nel Domain Name System la risoluzione dei nomi può avvenire iterativamente (la gerarchia viene "navigata" dal client tramite richieste multiple sempre più precise) oppure ricorsivamente (la richiesta arriva al server di gerarchia più alta, che provvede a fare chiamate ricorsive verso altri server inferiori finchè non ottiene la risposta). Nel secondo caso si ha più carico sui name server, ma si risparmia sulla comunicazione e si può sfruttare più efficacemente i meccanismi di caching.

Un server DNS è organizzato come un albero ed è responsabile per una zona ben definita.
I server globali (al vertice della gerarchia) hanno diversi mirror e usano indirizzi IP Anycast per bilanciare il carico; vengono inoltre usati server secondari refreshati periodicamente con informazioni aggiornate e servizi di caching. Sono possibili temporanee inconsistenze (la cui durata temporale cresce all'aumentare della gerarchia del server).

Se sono presenti entità mobili che escono dal dominio generale del server, esistono due approcci:

- il vecchio server DNS fornisce comunque l'indirizzo della nuova locazione, anche se non sarebbe più sotto il suo controllo (si creano overhead amministrativi)
- il vecchio server DNS fornisce il nome della nuova locazione (lookup inefficiente)

In ambiti totalmente mobili queste soluzioni non sono accettabili e si passa allo sdoppiamento in naming service e location service (Jini).

➔ ***Attribute based Naming***

Con questo approccio non ci riferisce più alle entità tramite i loro nomi, ma attraverso un set di attributi che identificano le loro proprità. Diventa quindi possibile ricercare le entità in base ai loro attributi, cosa fondamentale al crescere del numero delle informazioni disponibili.
Questi sistemi di naming vengono chiamati directory service e sono implementati usando DBMS.

Un approccio comune per implementare directory service è combinare il namig strutturato con quello basato sugli attributi, come LDAP.

➔ ***Rimuovere entità senza riferimenti***

Come gestire la rimozione di entità non più raggiungibili dal rootset?
Nei sistemi convenzionali avviene un garbage collection automatizzata per eliminare le entità non più raggiungibili, ma nei sistemi distribuiti il tutto viene molto complicato a causa della mancanza di una conoscenza globale dello stato d'uso delle risorse e per i sempre possibili guasti di rete.

E' necessario quindi attuare diverse tecniche:

- Reference counting: ogni oggetto (nel proprio skeleton) mantiene traccia degli oggetti che si riferiscono a lui. Quando si rileva che un oggetto non è più "usato" da nessuno, viene eliminato. Bisogna garantire semantica exactly-once (ACK & DUP deletion) perchè messaggi multipli o mancanti causerebbero il fallimento dell'algoritmo.

Esiste il problema delle race condition tra più processi che cercano di eliminare/segnalare i riferimenti agli oggetti, ma è risolvibile introducendo il:

- Reference counting pesato: si usano 2 contatori, uno con *peso parziale* e uno con *peso totale* dei riferimenti. Il valore dei due pesi è inizialmente un numero alto prefissato (*es: 128*), ma ogni volta che il riferimento è copiato, metà del peso parziale va alla copia, metà resta all'oggetto originale (il peso totale invece rimane lo stesso). Quando si rimuove un riferimento, si sottrae il suo peso al contatore totale: quando i contatori parziale e totale coincidono, è possibile eliminare l'oggetto perchè non esistono altri riferimenti (eseguo solo decrementi: niente race condition!).
  Posso avere solo un numero limitato di riferimenti, a meno di usare il metodo indiretto di conteggio. In questo caso si usa uno skeleton interposto prima dell'oggetto che rimuove la limitazione ma introduce un hop addizionale sull'accesso.

- Reference listing: invece che memorizzare il numero di riferimenti si tiene traccia dell'identità dei proxies che accedono all'oggetto, con il vantaggio di ottenere idempotenza sull'inserimento-rimozione di un proxy (posso usare comunicazione non affidabile). Diventa inoltre più facile mantenere una lista consistente. Viene utilizzato in JAVA RMI.

Come identificare invece entità irraggiungibili (magari raggiungibili tra loro ma non dal tool-set)?
Si usano tecniche di *trace* che però richiedono la conoscenza di tutte le entità: mark and sweep.

- mark: tutti gli oggetti iniziano marcati di colore bianco. Vengono segnati in grigio gli oggetti raggiungibili dalla radice, e se tutti i suoi riferimenti sono grigi, segno in nero l'oggetto;
- sweep: tutti gli oggetti bianchi, cioè irraggiungibili, sono collezionati.

Mark & sweep funziona anche a livello distribuito.

6) Sincronizzazione

I processi distribuiti devono comunicare e cooperare, ma in modo sincronizzato e consistente.
In ambito distribuito, senza clock globale né memoria condivisa, la sincronizzazione diventa un problema molto complesso.

➔ Problemi di temporizzazione e clock

In un sistema distribuito prendere decisioni in base al tempo non è semplice, i processi di diversi nodi usano un orologio locale che può non coincidere con quello di altri nodi.
Come posso eseguire certi eventi in precisi momenti, o poterli confrontare per ricostruirne l'ordine?
In pratica, è possibile sincronizzare tutti i clock di un sistema distribuito?

Nei calcolatori ci sono dei timer che non funzionano in modo identico ma sono soggetti ad un errore detto *clock drift rate* (quanti secondi perde un orologio ogni tot tempo) di cui bisogna tener conto, rendendo necessaria una sincronizzazione ogni X secondi, dipendenti dal parametro ingegneristico *clock skew* definito inizialmente (è un limite predeterminato che indica ritardo massimo tollerabile).

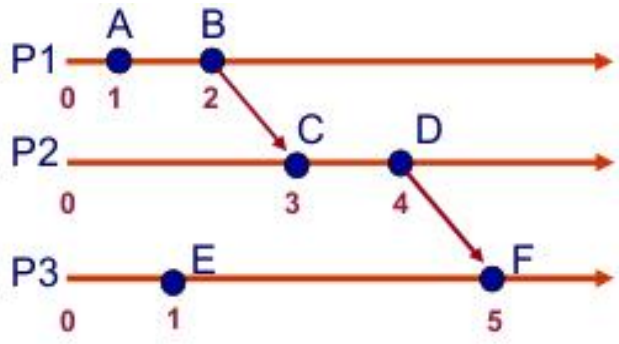
Nascono così diversi algoritmi che cercano di ri-sincronizzare i clock dei calcolatori distribuiti per evitare che i timer locali discostino troppo tra loro e possano quindi causare errori di temporizzazione.

Algoritmi di sincronizzazione su un clock globale:

- **Cristian's**: periodicamente ogni client manda messaggi di richiesta al *time-server che invia il tempo globale*. Lo scambio di messaggi deve essere rapido; eventualmente il tempo viene calcolato eliminando il RTT stimato.
- **Berkeley**: il time-server è *attivo*, prende il *tempo medio dei clock locali* dei diversi client, poi comunica broadcast il nuovo tempo a cui sincronizzarsi. Nessun nodo ha tempo "esatto".
- **NTP**: standard su internet, basato su una sottorete gerarchica che parte dalla radice, direttamente connessa alla sorgente UTC, fino alle foglie (i PC). (Calcola la stima dello scostamento mediandola tra diversi computer e collegamenti, trovando valori accettabili.)

Clock logici:
In molte applicazioni è importante solo controllare se un'evento avviene prima o dopo un'altro, oppure verificare la causalità delle operazioni... in questi casi non serve un timer globale ma è sufficiente un ordinamento logico delle operazioni e la definizione di relazioni happens-before che identificano un potenziale ordinamento causale (*a* segue *b* nel tempo... ma è davvero causato da *b*?).

- **Meccanismo di Lamport**: ogni processo ha un clock logico L*i* che parte da 0 e viene incrementato ad ogni evento. All'arrivo di un messaggio il processo setta il proprio clock a: MAX(TS_mittente, TS_proprio)+1.

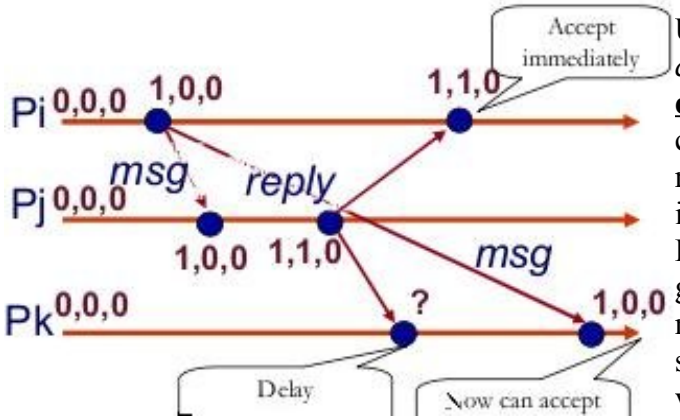


In questo modo si ottiene un ordinamento parziale, se volessi avere un ordinamento globale dovrei aggiungere ai clock l'ID dei processi.

Un esempio applicativo di Lamport, su database replicati (canali affidabili) è **multicast totalmente ordinato**, che ha come obiettivo mantenere un ordinamento coerente tra tutti i nodi nella ricezione dei messaggi (ogni nodo ha lo stesso ordinamento di messaggi). Vengono inviati messaggi multicast con un timestamp con clock logico scalare del nodo, ogni destinatario lo inserisce in una coda in base al valore, inviando poi un ACK. Solo alla ricezione di tutti gli ACK è possibile inviare allo strato applicativo il messaggio.
I clock logici non permettono di stabilire la causa effetto tra due eventi ordinati (B succede dopo A, ma è risposta ad A o è indipendente?)

Clock vettoriali:
per stabilire la causa effetto tra due eventi introduco i Vector Clock, che forzano un ordinamento tra due eventi causali ma non quello tra due eventi indipendenti (vincoli meno forti ma spesso sufficienti per la consistenza).
Ogni processo P*i* ha un vettore V*i*[n] che memorizza gli *n* clock logici dei processi distribuiti.

Ogni volta che un processo invia messaggi, allega anche il proprio vector clock, in modo che il ricevente possa aggiornare il proprio vettore. Gli eventi vengono parzialmente ordinati in modo molto migliore rispetto al metodo di Lamport, perchè ogni nodo che riceve un messaggio con il VC capisce quali *altri* eventi sono avvenuti al processo mittente prima di inviare il messaggio, è possibile quindi valutare rapporti causa-effetto.



VC nella versione totalmente distribuita

Una variazione dei VC può permettere invio *causale* dei messaggi in modo **totalmente distribuito**. In questo scenario incremento il clock solo all'invio dei messaggi (non alla ricezione, quando faccio merge dei VC senza incrementi).
L'algoritmo deve mettere in attesa le risposte giunte ad un nodo fino all'arrivo di tutti i messaggi con timestamp precedente (ordino solo se c'è causa-effetto tra eventi, allentando i vincoli troppo stretti del *multicast tot ordinato*).

➔ ***Mutua esclusione***

Sistemi con più processi che condividono e modificano gli stessi dati sono più facilmente programmabili tramite l'introduzione di zone critiche in cui l'accesso avviene per mutua esclusione, evitando interferenze e inconsistenze.
Requisiti: Safety (al massimo un processo in ogni zona critica) & Liveness (prima o poi tutti devono poter accedere alle zone critiche); opzionalmente posso ordinare l'accesso alla risorsa, per

evitare starvation.

Si assume che la trasmissione e i canali siano affidabili.

- La soluzione più semplice consiste nella creazione di un server centrale coordinatore che gestisce l'accesso alle risorse condivise. In questo modo però si crea un bottleneck e un unico punto di guasto che potrebbe bloccare il sistema e lo rende inaffidibile.

Alternative:
- uso *scalar clocks*: un processo interessato alla zona critica invia a tutti gli altri una richiesta.
  Se i processi Pj non sono interessati alla risorsa inviano un ACK, se un Pj ha il lock inserisce la richiesta in coda per quando rilascerà la risorsa, se un Pj è interessato confronta il timestamp del richiedente con il proprio, se è più basso del suo invia ACK, altrimenti mette in coda e segnalerà appena possibile.
  Quando un processo riceve tutti gli ACK, può ottenere la risorsa.
  Algoritmo Safe + Live, ordina in modo totale (con Lamport). Funziona solo con canali affidabili.
- uso *token ring*: collego logicamente tutti i processi in un anello. Faccio girare i token, il cui possesso determina la possibilità di usare la risorsa condivisa. Nascono problemi se viene perso il token, se si rompe il ciclo.

Alla fine risulta meglio seguire la soluzione centralizzata, più semplice ed efficiente.

➔ Algoritmi di election

Esistono algoritmi che permettono di eleggere dinamicamente un processo coordinatore e cambiarlo in caso di guasti. Questi algoritmi consentono di usare i molti algoritmi distribuiti che necessitano di un nodo centrale di controllo.

Gli algoritmi di elezione decidono il leader in base a diverse assunzioni (se tutti i nodi sono uguali posso scegliere a caso, se si preferiscono certi nodi posso usare tecniche per assegnare un peso...); in ogni caso i diversi nodi devono essere distinguibili.

- Bully election: assumo che i collegamenti siano affidabili ed è possibile decidere se un processo è andato in crash (sistema sincrono).
  Se un processo si accorge che il coordinatore non risponde più, inizia l'elezione inviando un messaggio ELECT con il proprio ID verso gli altri processi con ID più alto. Se nessuno risponde, invia messaggio COORD ai processi con ID minore, diventando il nuovo coordinatore. Se invece un processo riceve ELECT, risponde e prova a diventare leader a sua volta.
  Quando un processo rientra nel sistema dopo un crash prova a diventare coordinatore.
- Ring-based: assumo una topologia fisica o logica ad anello. Quando un processo si accorge del crash del leader, invia ELECT sull'anello, che si propaga fino a completare un ciclo (ogni nodo aggiunge il proprio ID). Quando il nodo riceve di nuovo ELECT, invia un COORD informando il nuovo leader (quello con ID più alto) che deve diventare il coordinatore.

Entrambe le tecniche richiedono una conoscenza totale della topologia di rete, per raggiungere correttamente i nodi.

➔ ***Catturare lo stato globale***

In molte occasioni è utile conoscere lo stato globale del sistema distribuito, cioè l'insieme degli stati locali di ogni processo più i messaggi correntemente in transito, già inviati ma non ancora ricevuti.
Analizzando lo stato globale è possibile eseguire debug, individuare eventuali deadlock o eseguire uno ***snapshot distribuito*** del sistema, un istante in cui si "fotografa" la situazione facendo in modo che sia consistente e riproducibile.

La conoscenza dello stato globale è complicata dall'assenza di un clock globale, che permetterebbe di sincronizzare i vari stati locali; per risolvere il problema vengono usati dei tool concettuali chiamati tagli (***cut***) che, rispettando certi vincoli (un messaggio arrivato DEVE essere stato spedito...), consentono di avere un'immagine accettabile e consistente del sistema.
Un taglio è l' "unione delle storie" di tutti i processi di un sistema fino ad un certo evento, è consistente se rispetta la happened-before e quindi per ogni evento include tutti gli eventi precedenti.

✔ Come registrare uno snapshot distribuito (**Chandy-Lamport**)?

Assumiamo comunicazione FIFO e link affidabili, deve esistere un percorso tra 2 processi.
Ogni processo può iniziare uno snapshot salvando il proprio stato interno e inviando un token in uscita, segnalando l'inizio dello snapshot e registrando tutti i messaggi in ingresso. A loro volta i riceventi salvano il proprio stato e i messaggi in ingresso A PARTE quelli provenienti da chi ha inviato il token (evito di ricevere messaggi che non risultano spediti).
In questo modo mantengo la consistenza ma allo stesso tempo i messaggi continuano a girare nel sistema (per questo lo snapshot è lo stato globale in cui il sistema POTREBBE essersi trovato).
Se lo snapshot iniziasse contemporaneamente in più nodi bastarebbe includere un ID che li distingua.

Si può ritenere che il sistema abbia terminato l'esecuzione o sia andato in deadlock se rilevo tutti i processi idle e senza messaggi nei canali. (La soluzione di Tanenbaum per usare snapshot distribuito per verificarlo non funziona perchè ci sono anelli che causano deadlock dei messaggi DONE).

Nel caso di *diffusion computation* (i processi sono inzialmente tutti fermi, si attivano e terminano solo ricevendo un particolare messaggio) si usa una tecnica alternativa allo snaphot distribuito, detta di **Dijkstra-Scholten**:

si crea un albero dei processi attivi in cui ogni nodo è padre dei nodi che attiva, in modo tale che si abbia controllo sulla terminazione dei singoli processi figli... quando anche la radice verifica la terminazione dei figli allora significa che tutto il sistema ha terminato l'esecuzione.

Quest'ultimo algoritmo è più efficiente di quelli di snapshot distribuito, ma specifico per certi casi.
L'overhead è elevato (in base ai messaggi) e non si tiene conto dei processi mai attivati.

➔ ***Transazioni distribuite***

Le transazioni sono un meccanismo tramite cui è possibile proteggere una risorsa condivisa dall'accesso simultaneo da parte di più processi concorrenti, garantendone la consistenza (e le altre ben note proprietà ACID).
Le transazioni sono una sequenza di operazioni definite da particolari primitive (begin/ end/ abort_transaction, read, write), permettono di accedere e modificare dati in modo atomico, in caso di errori vengono eseguite tutte le operazioni oppure nessuna (tramite abort e rollback delle operazioni allo stato iniziale posso annullare tutte le modifiche).

Le transazioni possono essere *flat* (normali...), *annidate* (hanno sotto-transazioni che devono supportare rollback anche dopo il proprio sotto-commit), oppure *distribuite* tra più basi di dati (necessitano di lock distribuito).

Nel caso distribuito com'è possibile avere:

- atomicità:
  - uso *workspace privati*, "copie" di ciò che viene modificato dalla transazione, in caso di commit poi viene sovrascritto l'originale (si usano puntatori per evitare eccessive scritture su disco).
  - un approccio più ottimista è il cosiddetto *WriteAhead Log*, che consiste nel modificare direttamente i dati reali, registrando però un file di log che permette di ritornare ai valori precedenti in caso di abort (meno spazio usato, rollback costoso).
- concorrenza: scompongo il problema in 3 parti, con un gestore per ognuna di esse: transaction manager + scheduler + data manager.

  Il *transaction manager* è il coordinatore unico per ogni transazione, comunica con lo scheduler distribuito sulle diverse macchine, controllando che tutto vada a buon fine.

  Lo *scheduler* distribuito controlla la *serialità* delle modifiche permettendo sovrapposizioni tra modifiche indipendenti ma evitando errori di race condition, garantendo isolamento e consistenza.

  Il *data manager* non conosce nulla delle transazioni concorrenti e si limita ad eseguire read/write.
- serializzabilità: avviene sullo *scheduler* secondo diversi metodi, uno dei quali è ***2PL*** (two phase locking): se un processo vuole accedere ai dati deve possederne un lock. Dopo che una transazione ha rilasciato un lock, non ne può ottenere altri... nel caso di *2PL strict* i lock sono rilasciati tutti nello stesso momento. Assicura serializzazione, ma c'è pericolo di deadlock.

  2PL può essere centralizzato, primario o distribuito, tutti e tre i metodi fanno uso di un lock manager che concede e gestisce i lock nelle varie repliche locali e fa da interfaccia con i data manager (nel caso distribuito).

  Lo scheduler usa clock logici come timestamp per stabilire a quali transazioni dare il controllo (abort delle scritture con TS precedente alla versione commited attuale, abort delle letture su dati scritti da transazioni più recenti). Le transazioni abortite ripartono con un nuovo timestamp. Non c'è deadlock ma c'è possibilità di starvation.
  E' possibile usare ordinamento di timestamp ottimistico che assume che i conflitti siano rari, controllandoli solo al momento del commit.

➔ Deadlock distribuiti

I deadlock in sistemi distribuiti sono molto difficili da determinare a causa della decomposizione dei dati su molte macchine; gli approcci sono i classici: ignorare il problema, detection, prevention, avoidance (mai usato perchè richiede conoscenza a priori dell'uso delle risorse).

In questo ambito i meccanismi transazionali sono utili, soprattutto nel caso della *detection*, perchè si evita di "killare" un processo ma piuttosto si usano meccanismi di rollback.

- Detection centralizzata: ogni macchina mantiene un grafo che memorizza le relazioni tra proprie risorse e processi, inviandole periodicamente ad un server centralizzato che analizza i diversi grafi, gestendo e controllando l'accesso distribuito alle risorse. Il coordinatore rileva

eventuali cicli e li risolve.
Può succedere che l'immagine del sistema presente nel coordinatore non sia consistente con la realtà a causa di messaggi di aggiornamento non ordinati temporalmente (falsi deadlock).

- Detection distribuita: non c'è un coordinatore, periodicamente ogni macchina manda messaggio *probe* verso il nodo da cui sta attendendo una risorsa. A sua volta il nodo inoltra il messaggio verso i nodi che occupano risorse a lui indispensabili per proseguire, e così via fino ad un eventuale ritorno del probe al mittente, cosa che indicherebbe un ciclo, cioè un deadlock. Nella pratica nascono problemi di gestione (visto che il nodo che scopre il deadlock si "suicida", potrebbe avvenire che molti processi si autointerrompono nello stesso momento -> si fa in modo che si esegua un kill del processo con ID maggiore).
- Prevention distribuita: si cerca di rendere impossibili i deadlock usando timestamp e algoritmi come *wait-die* (creo catena ordinata di processi in base a TS... si eliminano tutti i processi più "giovani" che chiedono una risorsa occupata da processi più "vecchi") o *wound-wait* (processi possono fare *preemption* sui processi più "giovani" che stanno usando le risorse richieste, mentre i processi con TS minore devono attendere – niente kill dei processi ma abort delle transazioni).

7) Consistenza e replicazione nei DB

I dati vengono generalmente replicati per migliorare l'affidabilità di un sistema e/o incrementare le performance suddividendo il carico di lavoro su diversi nodi.
Esempi classici sono le web cache (proxy), i file system distribuiti (replicazione dei dati) e i DNS.

Il problema principale della replicazione è il mantenimento della consistenza. L'obiettivo è assicurare l'accesso ai file replicati e originali pur mantenendo un basso overhead di comunicazione e quindi minimizzare il degrado delle performance

Esistono vari modelli: data-centrici e client-centrici, ognuno con diversa distribuzione degli update e un diverso modello di consistenza delle diverse repliche.
Studiamo le diverse alternative:

➔ *Modelli data-centrici*

Ci si concentra su un *data store* distribuito; ogni processo ha una versione locale dei dati e quindi il problema è dovuto al mantenimento delle diverse versioni.
Le scelte possibili sono molte a seconda che voglia preferire le performance o il livello di consistenza e le garanzie:

- **Consistenza stretta**: ogni lettura su X legge la sua scrittura più recente. In pratica ogni scrittura dovrebbe essere immediatamente visibile a tutti i processi.
  Dovrei avere un ordinamento globale (impossibile in contesto distribuito senza clock globale).

- **Consistenza sequenziale**: il risultato di ogni esecuzione è lo stesso che si avrebbe se le operazioni fossero eseguite in sequenza. Questa sequenza può essere diversa da quella reale, ma tutti i processi devono vedere lo stesso ordinamento di R/W (due processi scrivono A e in seguito B su X, ma poi tutti gli altri processi leggono prima il valore B poi A? Va bene, però TUTTI i processi devono leggere in questo modo!). Tutti i processi vedono lo stesso interleaving.
  Le operazioni di un processo non possono essere riordinate.

  Linearizzabilità: consistenza sequenziale a cui agguingo timestamp per ordinare parzialmente le operazioni. E' più forte della consistenza sequenziale, assume clock globali ma con precisione finita (posso usare algoritmi di sincronizzazione globali).

- **Consistenza causale**: le scritture potenzialmente legate causalmente devono essere viste da tutti i processi nello stesso ordine, mentre se non ci sono vincoli causali l'ordine può variare.
  E' un vincolo più leggero dell'happened-before di Lamport (non c'è scambio di messaggi).
  Necessita della costruzione di un grafo delle dipendenze che tenga conto delle relazioni tra operazioni. Un modo per fare questo è l'uso di *vector clock* VC.

- **Consistenza FIFO**: solo le scritture fatte da un singolo processo sono viste nello stesso ordine da tutti gli altri, ma non c'è ordinamento tra le W di processi differenti. In pratica tutte le scritture generate da processi diversi sono considerate concorrenti. Modello semplice ma diversi processi possono vedere le operazioni in modo differente.

Alcuni modelli di consistenza introducono la nozione di *variabili di sincronizzazione* **VS**, che permettono di rendere visibili delle scritture solo dopo esplicita richiesta da parte del processo (tramite operazione *synchronized()* ad esempio), in modo tale che spetti al programmatore decidere se e quando la consistenza è necessaria:

- **Consistenza debole**: semplifica ulteriormente l'approccio, nel caso in cui non sia necessarrio nemmeno ordinare le scritture di un processo (*es:* interessa solo il risultato finale!). Sono usate delle VS, tramite cui agisco sulla copia locale per poi propagare le modifiche al termine delle operazioni. L'accesso alle VS è consistente sequenzialmente, non sono possibili letture/scritture sui dati prima che siano completate le scritture synchro. Solo sincronizzando PRIMA di leggere si è però sicuri di avere i dati più aggiornati.
  Questo metodo permette di avere consistenza su gruppi di operazioni, non sull'intero sistema.

- **Consistenza release**: voglio distinguere la richiesta di propagazione di scritture dalla richiesta di letture consistenti (prima eseguita sempre con il comando di sincronizzazione). Introduco operazioni *Acquire()* e *Release()* globali che delimitano una zona critica e proteggono i dati. In base a come vengono propagati i dati dopo una *Release()* posso avere consistenza Eager (tutti gli aggiornamenti sono inviati alle repliche immediatamente) o Lazy (le modifiche avvengono solo su richiesta, tramite Acquire).

- **Consistenza entry**: associo esplicitamente dei dati con una variabile di sincronizzazione, in modo che sia possibile acquisire "lock" tramite Acquire e Release specifici per ogni tipo di risorsa, sia in modo esclusivo che non (se si tratta di letture). In questo modo aumento il parallelismo e la flessibilità, ma incremento la complessità (sono possibili anche incapsulamenti...).
  I dati sono aggiornati all'ingresso della zona critica. Dopo che un processo ha modificato i dati in modo esclusivo tutti i futuri accessi a quella variabile devono avvenire tramite Acquire

Riassumendo: i metodi data-centrici presentati differiscono per il grado di restrittività, per la complessità delle implementazioni e per le prestazioni.

Un buon modello è quello della *consistenza sequenziale*, flessibile e molto usato, ma ha il problema delle basse performance; se si punta alle prestazioni si possono usare *FIFO* o *consistenza causale*, che non necessitano di consenso globale, oppure usare variabili esplicite di sincronizzazione per specificare quando e su cosa richiedere la consistenza, aumentando le performance a discapito di un aumento della complessità e una consistenza del sistema solo parziale.

ESERCIZI

11 marzo 2005- 4 processi Pi.

P1	Wx1								Wx8			
P2		Wx2				Rx2	Rx1	Rx4				
P3			Rx2	Rx1	Wx4							
P4			Rx1	Rx2						Rx8	Rx4	

Il modello rispetta una consistenza di tipo: strict, sequential, causal, FIFO?

Strict?	NO	(ogni lettura non legge l'ultima scrittura, *vedi P4 con Rx1*)
Sequential?	NO	(le letture su P3 e P4 sono invertite)
Causal	SI	(le scritture causali sono W1-W4 e W2-W4 perchè Wx4 avviene dopo le letture Rx2 e Rx1. Le letture rispettano la consistenza 1-4 2-4.)
FIFO	SI	(è causale, poi viene rispettata la consistenza 1-8 di P1)

Secondo esercizio:

P1 Wx0 Rx1
P2 Wx1 Wy2
P3 Ry2 Rx0 Rx1

Consistente? NO (non si può leggere 2 su y prima che venga scritto!!)
Come posso modificare lo schema perchè diventi consistente?
Sposto Ry2 dopo la scrittura:

P1 Wx0 Rx1
P2 Wx1 Wy2
P3 Ry2 Rx0 Rx1

Non è strict (P3 legge 0 su x dopo che è stato scritto 1)
Non è sequenziale (se in P3 ho letto 2 su y allora dovrei aver aggiornato anche x, invece leggo 0 subito dopo!)
Rispetta gli altri due modelli di consistenza (non ci sono causalità, le scritture sono concorrenti visto che le uniche due che sono dello stesso processo agiscono su dati differenti)

Modifico ancora l'esercizio:

P1 Wx0 Rx1
P2 Wx1 Wy2
P3 Rx0 Ry2 Rx1

Ora leggo 2 da Ry, in seguito leggo 1 da Rx --> consistenza sequenziale
"posso leggere in modo disordinato ma la consistenza dei dati deve rispettare la sequenzialità"

➔ ***Modelli client-centrici***

Un'assunzione dei modelli centrati sui dati era la presenza di un data-store su cui i processi effettuavano update concorrenti. Il caso dei modelli client-centrici invece è adatto quando gli update non sono simultanei e se la gran parte delle transazioni riguardano le letture dei dati (come nelle Web Caches, i DNS...), per questo motivo sono introdotte tecniche alternative, più adatte.

Consistenza *eventual*: spesso nei sistemi client-centrici può essere sufficiente una consistenza meno restrittiva, che tollera un certo grado di inconsistenza. Per questo si parla di consistenza "eventual", cioè che prima o poi avverrà, senza badare troppo al quando, in modo da semplificare le operazioni e diminuire lo scambio di messaggi.

Abbiamo 4 modelli di consistenza client-centrici, indicati dal lavoro eseguito su Bayou (mobile DB):

- **monotonic read**: se un processo legge un dato, una seconda lettura su repliche non potrà restituire un dato meno recente. E' necessario verificare che una scrittura su una replica usi le modifiche delle scritture precedenti sugli altri nodi: *WS(x1,x2,x3)*, indica che viene scritto *x3*, tenendo conto di *x1*,*x2* scritte precedentemente.
  Quando leggo controllo che le scritture siano state propagate.
- **monotonic write**: se scrivo una modifica in un processo e poi cerco di scrivere una modifica

diversa in una replica, prima devo portare la replica in uno stato con scritture aggiornate (le operazioni write di uno stesso processo sono sempre sequenziali).
Quando scrivo controllo che le scritture siano state propagate.

- **read your writes**: una scrittura è sempre completata prima di una lettura successiva da parte dello stesso processo. Controllo che ogni scrittura tenga conto delle precedenti. (web cache)
- **write follow reads**: le scritture avvengono sui dati più recenti disponibili anche su altre copie. Controllo anche qui che ci sia WriteSet contenente le scritture precedenti. (newsgroup)

Questi schemi si possono implementare assegnando ad ogni operazione un id unico e facendo modo che ogni client definisca due set: **read-set** e **write-set**.
Tutti e 4 i modelli si basano su questi set che contengono gli id delle operazioni di lettura/scrittura e permettono di controllare la consistenza di volta in volta.

Implementazione della replicazione

posizionamento: le repliche possono essere permanenti, create dinamicamente dal server (si bilancia il carico muovendo i dati in prossimità del client) o dal client (cache).

propagazione update: la replicazione propaga le informazioni secondo diversi approcci: invio notifiche (avverto solo che c'è stata una modifica, sarà poi necessario richiedere esplicitamente i dati; si usano protocolli di invalidazione: adatto in presenza di tante scritture), trasferimento di tutti i dati (adatto in presenza di molte letture), active replication (propago le informazioni che indicano quali operazioni eseguire per effettuare un update remoto, basso overhead).

La propagazione degli aggiornamenti può seguire diverse modalità:
- push based: usato per maggior consistenza, propago gli update "a forza" sulle repliche.
- pull based: aggiornamenti solo su richiesta, usato se le letture sono poche (latenza di fetch).
(Si possono usare meccanismi di lease per passare tra i due metodi in modo dinamico.)

- algoritmi epidemici: eseguono la propagazione come se fosse una malattia: "infetto i vicini" in modo scalabile, resistente ai guasti e intrinsecamente distribuito. E' una tecnica probabilistica, quindi non esistono garanzie di consistenza.

Esistono diverse strategie di propagazione epidemiche:

Anti-entropy: casualmente un server ne sceglie un altro con cui scambiare gli aggiornamenti.

Gossiping: un update su un nodo da' il via ad un altro update verso una terza replica.
Queste tecniche permettono un grande calo di overhead, possono però nascere problemi con la cancellazione dei dati (che spesso viene considerata come un update "differente").

➔ ***Protocolli per la consistenza***

Un protocollo di consistenza consiste di una serie di regole che descrivono un'implementazione di un modello di consistenza scelto tra quelli possibili: i migliori sono *consistenza sequenziale*, la *consistenza debole con SV* e le *transazioni atomiche*.

Dividiamo i protocolli in base alla presenza o meno della copia primaria:

Protocolli primary-based: esiste un nodo centrale che coordina le scritture.
- Remote-write: inoltro letture/scritture ad un server remoto fisso, che contiene i dati consistenti e risponde alle richieste dei client. Eventualmente posso introdurre un nodo di backup.

- Local-write: il coordinatore inoltra i dati al nodo che ne fa richiesta, che diventa il nuovo primario (usato nei sistemi con memoria condivisa distribuita). Diventa quindi primario un nodo diverso a seconda delle operazioni che vengono richieste (si porta tutto al nodo locale al client), riducendo così l'overhead per le operazioni successive alla prima (si propagano tutte le informazioni in base a chi le vuole usare... primario totalmente dinamico).

Protocolli remote-based: le scritture possono essere inoltrate a più repliche.

- Active replication: ogni replica ha un processo che esegue le operazioni fatte sui diversi nodi, piuttosto che copiare i dati. Servono *timestamps* o *sequencer* centralizzato per eseguire le operazioni nell'ordine corretto.
- Quorum-based: i client devono richiedere il permesso ad certo numero (quorum) di server prima di accedere ai dati, in modo di avere la certezza che la versione sia la più recente ed evitando conflitti RW e WW. E' molto usata la scelta ROWA (read one, write all).

8) Fault tolerance

Una caratteristica che distingue i sistemi distribuiti dai sistemi convenzionali è il concetto di fallimento "*parziale*", che avviene quandi si guasta un componente, cosa che può non interrompere né interferire con le altre parti del sistema. Uno degli obiettivi cardine nel design dei sistemi distribuiti è riuscire a costruire il sistema in modo tale che non si blocchi e, se possibile, che riesca a recuperare da questi fallimenti parziali senza rallentare le performance generali.

La resistenza ai guasti è essenziale per raggiungere certi obiettivi:
- ✔ Availability: capacità del sistema di essere sempre attivo quando c'è bisogno.
- ✔ Reliability: capacità di funzionare senza soste per il tempo più lungo possibile.
- ✔ Safety: evitare che il sistema fallisca o esegua funzioni non corrette.
- ✔ Mantainability: semplificare la correzione di eventuali errori e l'aggiornamento del sistema.

Un sistema *fail* quando non riesce a fornire i servizi che dichiara di poter eseguire. Un *errore* invece è una parte dello stato del sistema che conduce al fallimento e viene causato da un *guasto/fault* (transiente, intermittente, permanente).
Se alcuni guasti sono evitabili, altri non lo sono, quindi costruire un sistema resistente ai guasti è complesso poiché deve continuare a fornire servizi anche in presenza di errori.

Esistono 3 tipi di fallimenti:
- omission failures: il processo smette di rispondere alle richieste --> 3 sotto categorie:
  fail-safe (output permette di riconoscere facilmente l'errore)
  fail-stop (il crash può essere rilevato dalla mancanza di risposte)
  fail-silent (non è possibile accorgersi del crash di un processo)
- timing failures: valido in sistemi sincroni, avviene se si violano i vincoli di tempo.
- byzantine failure: il processo può continuare a rispondere ma in modo arbitrario e impredicibile.

Come gestire la presenza di guasti? Con la **ridondanza**:
- ✗ Ridondanza informativa: aggiungo altra informazione in modo che sia possibile riconoscere e/o correggere gli errori.
- ✗ Ridondanza di tempo: ho slot temporali in cui eseguire azioni, in caso di errori re-invio.
- ✗ Ridondanza fisica: duplico i dispositivi fisici (dei dispositivi detti *voter* ricevono più ingressi e danno in output i dati ricevuti "in maggioranza", in modo da resistere ad un guasto - TMR).

➔ Comunicazione client/server affidabile (esempio)

Gran parte degli errori nella comunicazione tra client e server è conseguenza di guasti nella comunicazione, TCP maschera certi tipi di guasti tramite ACK e ritrasmissione, ma non ci sono protezioni per la perdita di connessione o crash del sistema.
Passando a costrutti di più alto livello vediamo che tramite RPC viene introdotto il meccanismo delle chiamate a funzione in un sistema distribuito... che causa il sorgere di una serie di problemi legati ad eventuali fallimenti (i primi due sono *benigni*):

- il client non riesce a localizzare il server. Gestisco la situazione con un'eccezione.
- viene perso il messaggio di richiesta verso il server. Lo reinvio dopo un tempo prestabilito.
- la mancanza di una risposta da parte di un server "crashato" non ci permette di sapere *perchè* non ci sia arrivata (è un guasto o si è perso il messaggio?).

Il client non può sapere se la funzione richiesta sia stata eseguita o meno. Questo non ci permette di essere certi che una funzionalità richiesta sia stata eseguita una ed una sola volta, possiamo scegliere tra la tipologia *at-most-one* oppure *at-least-once*.

- problemi simili avvengono se dal server viene inviata una risposta, ma questa è persa nella trasmissione; il server può tenere in memoria la richiesta del client evitando di ripetere più volte stesse funzionalità (ma per quanto tempo tengo in cache?).
- se il client va in crash dopo aver richiesto delle funzionalità al server, ho risorse bloccate sul server dette *orfane*, posso "*sterminarle*" al riavvio (devo tenere un log di tutte le richieste), "*reincarnarle*" (al riavvio di un client invia messaggi broadcast che permettono di uccidere le richieste passate), attendere la sua "*morte*" (dopo un certo tempo T la chiamata remota termina se non viene rinnovata).

➔ ***Protezione contro guasti ai processi***

Si può usare ridondanza in modo da mascherare processi guasti tramite creazione di gruppi di processi ridondanti che eseguono le stesse funzionalità in parallelo, secondo un'organizzazione piatta (nodi comunicano tra pari) o gerarchica (esiste un coordinatore).

La gestione di set di processi introduce **problemi di coordinamento e gestione di un gruppo**:

- come gestire modifiche alla topologia del gruppo?
- quanto deve essere grande un gruppo per avere k-fault-tolerance?
  - ✔ Se ipotizziamo sistemi fail-silent bastano k+1 processi.
  - ✔ Se i guasti sono byzantine diventano necessari 2k+1 processi (la maggioranza k+1 deve funzionare correttamente per avere un meccanismo di voting funzionante)
    Non posso quindi garantire tolleranza ai guasti infinita, ma si accetta un compromesso (es: k=5)
- se voglio avere **consenso distribuito** (agreement) (elezione leader, commit di transazioni...)?
  - ✔ La decisione è interna al gruppo, bisogna attendere che tutti i processi rispondano, la decisione non è presa dall'esterno tramite voting... nasce il problema del consenso.
    Creo un protocollo che permetta ai processi non guasti di prendere una decisione su qualche valore assicurando l'accordo, la sua validità e il suo raggiungimento in un tempo finito.
    Considerando solo la possibilità che alcuni processi crashino o abbiano guasti bizantini (con sistemi sincroni e senza problemi di comunicazione), il problema è risolto in f+1 cicli (con f=numero di fallimenti possibili):
  - ✔ Algoritmo Flood-set: i processi inviano/ricevono messaggi e cambiano stato in modo sincrono ad ogni round. Al primo round tutti i componenti inviano il proprio valore, inserendolo nel subset W, subset a cui di volta in volta aggiungono eventuali valori differenti ricevuti dagli altri processi. Dal secondo round in poi quindi si reinvia e aggiorna W per f+1 volte. Al termine vedo quanti valori ha W, in caso siano più di uno seleziono un valore di default (il max), altrimenti il risultato è quello contenuto nel set.
    [ho almeno un round senza crash che mi assicura che tutti gli W siano uguali nei diversi nodi]
    Posso migliorare l'algoritmo stabilendo che W viene inviato solo se ad esso viene aggiunto un nuovo valore. Non funziona con guasti bizantini.
  - ✔ Introduciamo guasti bizantini, cioè il processo può avere un comportamento arbitrario.
    Nel caso dei "generali sulle colline" di Lamport, possiamo avere generali traditori che indicano risposte scorrette. Perchè si raggiunga un accordo servono 3k+1 nodi (di cui k sono i

generali traditori). Esempio classico di 4 processi di cui 1 guasto... decisione a maggioranza (2 vs 1).

- ✔ Nel caso di sistema asincrono non posso sapere se un task è andato in crash, quindi non posso raggiungere consenso distribuito nemmeno nel caso fail stop.

La **comunicazione affidabile di gruppo** è di importanza critica, prendiamo due casi:

- gruppi di dimensione fissa in cui i processi non falliscono mai. Ho multicast non affidabili, voglio implementare una versione affidabile ... usando acknowledgement. La quantità di ACK necessari ad implementare l'affidabilità è elevata, causa problema di overload detto *ACK implosion*. Una soluzione è inviare Not-ACK e, nel caso in cui venga inviato, gli altri nodi che dovrebbero segnalarlo (dopo un timeout casuale) non lo fanno perchè non è più necessario.
  Posso usare un controllo gerarchico in cui esiste un coordinatore per ogni gruppo di nodi, il problema consiste nella creazione e mantenimento delle gerarchia.
- gruppi di dimensione variabili con processi che possono avere guasti. Ogni messaggio deve essere spedito a tutti o nessuno, con ordinamento uguale per ogni nodo --> *problema multicast atomico*. Serve un meccanismo per identificare i guasti, gli eventuali messaggi inviati dal task guasto devono essere processati da tutti o nessuno; mi accontento di ricevere i messaggi rilevanti in un ordine specifico, anche diverso tra i vari processi.
  La sincronia non è possibile, cerco di ottenere una *sincronia virtuale* aggiungendo un layer di comunicazione che riceve il messaggio, controlla se rispetta i vincoli e lo inoltra eventualmente all'applicazione (*delivery*). La trasmissione non è istantanea ma tutti i membri di un gruppo devono poter aver un ordinamento in base alla *membership* (nodi attualmente presenti nel gruppo), in caso di guasti il messaggio verrà bloccato nel livello di comunicazione.
  Dobbiamo garantire che i messaggi siano inviati e ricevuti prima o dopo una *view change* (un cambiamento della topologia della rete).

Per quanto riguarda l'ordinamento dei messaggi, abbiamo diverse possibilità:
- Non ordinati: non è rilevante l'ordine dei messaggi ricevuti dal singolo nodo.
- FIFO
- Causalmente ordinati

Qualsiasi ordinamento sia scelto deve valere per tutti i nodi.

Come implementare la sincronia virtuale?
ISIS è un sistema distribuito che usa canali affidabili e FIFO, assicura che i messaggi inviati ad un gruppo siano ricevuti prima di un view change. Questo è possibile tramite notify: ogni processo aspetta a fare delivery del messaggio fino a conferma. In caso di ricezione di un messaggio di cambio vista (qualche nodo ha rilevato un cambio della topologia o un guasto), un task invia in multicast tutti i messaggi instabili e segnala il *flush*. Solo quando tutti i processi hanno effettuato flush avviene il view change.
L'assunzione base vincola il funzionamento dell'algoritmo alla mancanza di cambiamenti di membership durante la fase di flush (in ISIS ci sono tecniche apposite).

➔ ***Commit distribuito***

- 2PC (two phase commit): abbiamo coordinatore (Transaction manager) e repliche (Resource

manager). Il coordinatore da' inizio al commit, le repliche rispondono con un commit o abort locale, il transaction manager comunicherà eventuale global commit (tutte le repliche hanno fatto commit) e global abort (qualche replica ha eseguito abort).
I guasti possono essere diversi, si creano problemi soprattuto in caso di crash del coordinatore, a seconda del momento in cui si trovano le risorse. Se queste sono in attesa di decisione globale READY devono attendere la risposta indefinitamente, oppure "chiedere" ad un nodo che conosce la risposta... e se non c'è? protocollo bloccante

- 3PC (3 phase commit): si introduce una fase di precommit, in caso di guasto del coordinatore durante la fase di attesa di decisione globale, si contattano i vari nodi ed è possibile decidere a maggioranza, perchè se esistono risorse in precommit c'è stata una decisione globale del coordinatore prima del guasto.

➔ ***Tecniche di recovery***

Le tecniche di recovery sono utili per far tornare ad uno stato corretto e consistente il sistema dopo che i suoi processi sono stati riavviati a causa di un guasto:

- backward recovery: il sistema viene portato in uno stato passato consistente; ritrasmissione dei messaggi.
- forward recovery: cerco di portare il sistema in un nuovo stato corretto diverso da quelli passati; usa la correzione tramite codici ed informazione ridondante.

Per recuperare uno stato precedente di un sistema distribuito devo usare *checkpoint* (ogni processo salva lo stato periodicamente) o *log* (salvo gli eventi avvenuti in un sistema in memoria stabile, poi rieseguo le operazioni per ritornare allo stato prima del guasto).

Per effettuare un checkpoint devo trovare un taglio consistente (una linea temporale tra più checkpoint), detto **recovery line**, che eviti di salvare uno stato in cui anche solo un messaggio è *arrivato* senza risultare *partito*.

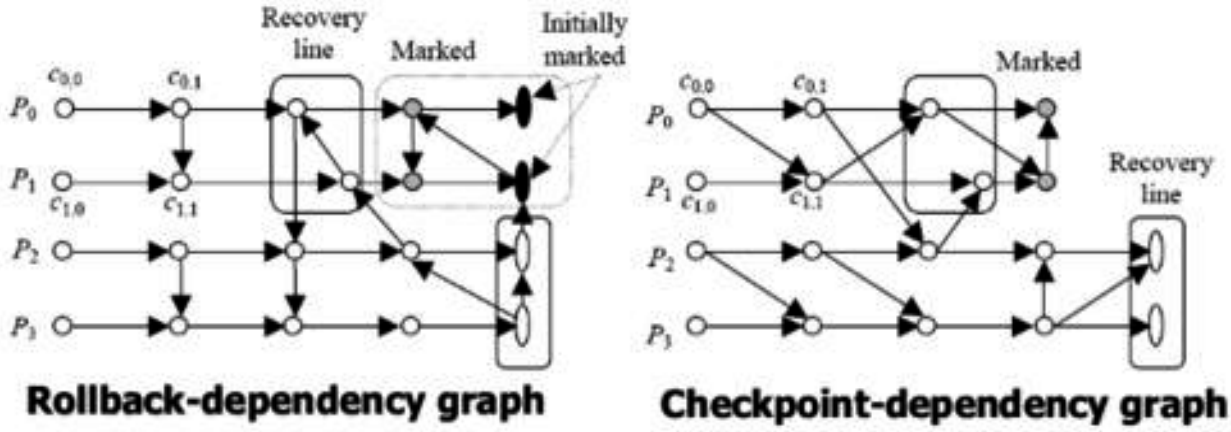
| La soluzione più semplice è detta *independent checkpoint*, in cui ogni processo salva il proprio stato senza sincronizzarsi con gli altri. L'algoritmo per salvare lo stato è semplice, ma può causare un *effetto domino* che riporta il sistema allo stato iniziale, senza contare che il controllo della consistenza tra i tagli è piuttosto complessa (dipendenze di messaggi, temporizzazioni, relazioni di causalità...).

| Alternativamente, si può gestire il processo di recovery con un coordinatore che in caso di guasto manda una richiesta di recovery in broadcast. Tutti i processi si fermano e inviano le proprie informazioni. Un processo calcola la *recovery line* e trasmette la richiesta di rollback a tutti (solo quelli guasti devono PER FORZA tornare indietro al checkpoint, gli altri processi, se lo stato del sistema rimane consistente, possono proseguire).

Le recovery line può essere calcolata tramite due approcci:
[indico checkpoint come $C_{id\_processo,\ numero\_checkpoint}$]

- grafo rollback-dependency: ho una relazione tra il checkpoint Ci,x e Cj,y se:
  i = j *(il processo è lo stesso)* AND y = x+1 *(ck successivi nel tempo)*
  OR
  i != j *(processi diversi)* AND esiste messaggio inviato da Ci,x verso Cj,y

  Si marcano i nodi corrispondenti agli stati di failure, e poi a ritroso tutti quelli da essi raggiunbili.
  Il rollback avviene sui primi nodi non marcati.



- grafo checkpoint-dependency: funziona in modo simile ma segnando le dipendenze tra i vari nodi.

Altri approcci*:*
Checkpoint coordinato: usando un coordinatore semplifico la soluzione, scambio messaggi e ACK.
Snapshot globale distribuito: non blocca i processi ma complica l'algoritmo.

9) Sicurezza

La sicurezza di un sistema riguarda due proprietà del sistema: la confidenzialità (informazioni il cui accesso è permesso solo ad entità autorizzate) e l'integrità (le alterazioni al sistema hw/sw sono possibili solo se effettuate da chi ne ha facoltà).

I problemi di sicurezza più rilevanti sono l'intercettazione (sniffing), l'interruzione di servizio (DOS), le modifiche irregolari (defacement), il forging (injection).

E' necessario introdurre delle ***security policy*** che definiscano cosa è permesso e cosa non lo è. Queste regole di sicurezza possono essere rafforzate tramite crittografia, autenticazione, autorizzazione, auditing (IDS).

Una security policy classica divide un dominio in sottogruppi con regole locali, usa mutua autenticazione e permette all'utente di delegare a processi delle funzionalità richieste.

Si possono inserire meccanismi di sicurezza su vari livelli della pila ISO/OSI (livello applicativo o di rete? I livelli superiori dipendono da quelli inferiori).
Il gruppo di meccanismi che usiamo per rafforzare una policy è detto Trusted Computing Base (TCB).
TCB rappresenta il set di meccanismi base affidabili su cui ci si basa per poter usare una certa politica di sicurezza (ad esempio, suppongo che il mio kernel sia "sicuro", applico regole ai livelli superiori).
Un modo per ridurre TCB è separare i servizi fidati da quelli insicuri usando una RISSC (Reduced Interfaces for Secure System Components)

Un altra tecnica che porta ad inserire meno errori in un sistema consiste nel mantenerlo il più semplice possibile, ma non sempre la semplificazione è sufficiente a costruire sistemi davvero sicuri.

➔ Crittografia e hash

Messaggio P -> funzione di crittazione -> Messaggio C (e viceversa)
Se la funzione viene scoperta si possono decrittare TUTTI i messaggi nascosti tramite essa quindi è preferibile usare funzioni parametriche, per cui $C = E_k(P)$, dove la chiave k è segreta.

Esistono sistemi crittografici simmetrici, per cui crittazione e decrittazione usano la stessa chiave.
Altri sistemi sono detti asimmetrici, per cui ogni nodo ha due chiavi differenti, pubblica e privata, per cui diventa possibile dare la propria chiave pubblica all'esterno per crittare i dati, mantenendo segreta la chiave privata, usata per decrittarli.

Algoritmi comuni sono:
DES: usa chiave simmetrica. In 16 *round* viene modificato il dato in base ad una chiave di 56bit.
RSA: usa chiavi pubblica e privata. Si basa sulla difficoltà di fattorizzare due grandi numeri, sfrutta i numeri primi per generare le coppie di chiavi.

Hash

Messaggio M -> funzione hash -> Hash H
Data una funzione di hash è "impossibile" passare da H ad M.

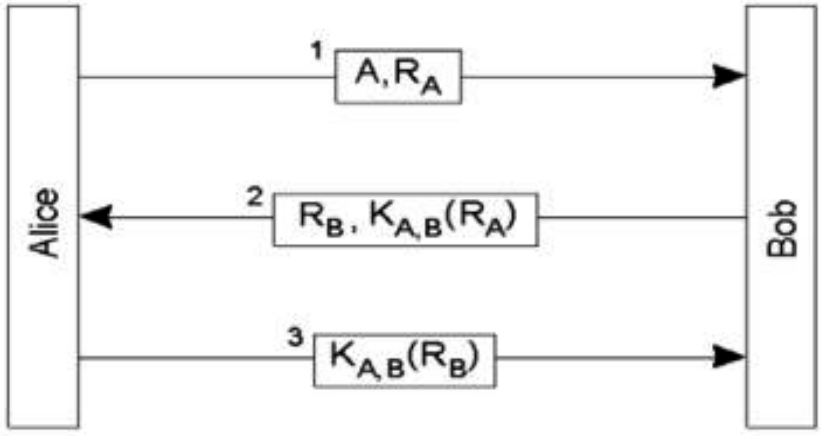
Come giudicare se una funzione di hash è buona?

one-way: dato un hash H deve essere molto complesso trovare il messaggio di partenza.
weak collision resistance: dato H e M, deve essere difficile trovare un altro M che abbia H come hash.
strong collision resistance: data una funzione di hash, deve essere difficile trovare due messaggi diversi che generano lo stesso hash.
Una comune funzione di hash è MD5: partendo dai bit dei file ottengo delle modifiche al messaggio usando traslazioni e funzioni predeterminate varie.
Dal concetto di crittografia nasce la ***firma digitale***, un sistema per cui il mittente usa la propria chiave privata per crittare l'hash del documento, chiunque in possesso della chiave pubblica potrà verificare il documento

➔ ***Autenticazione***

Canali di comunicazione realmente sicuri devono fornire protezione contro intercettazioni e modifiche, usando algoritmi che permettono di sapere da chi arriva il messaggio, attraverso metodi di autenticazione basati sullo scambio di informazione condivisa tra mittente e destinatario.

- **Challenge-Response**: viene scambiata una chiave segreta condivisa, se eseguito in 3 passaggi è suscettibile a *Reflection Attack* (un attaccante può richiedere al nodo la risposta ad un challenge e così autenticarsi). Un altra strategia di attacco, usata se la chiave scambiata è sempre la stessa, è il *Key Wearing*:un attaccante sollecita l'uso della chiave per scoprirla (tipo WEP del wireless).



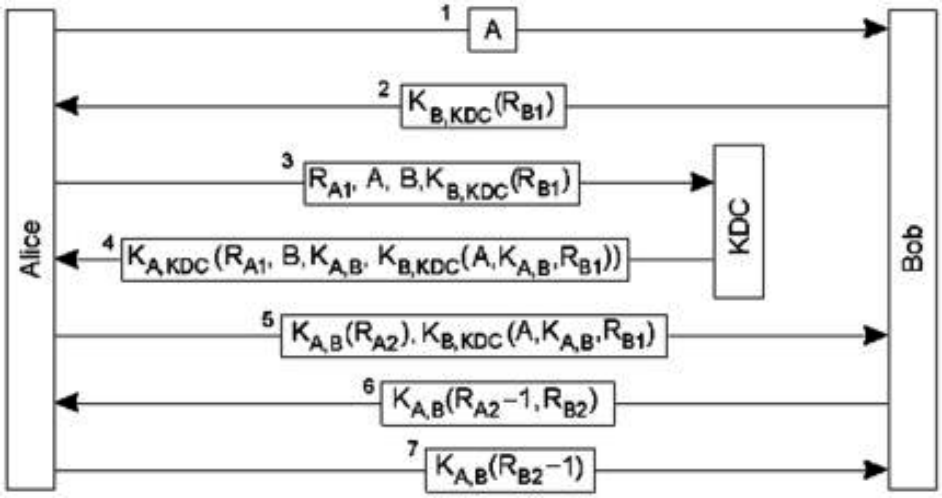
I nodi si scambiano numeri casuali Ra ed Rb, li crittano usando le proprie chiavi condivise per autenticarsi a vicenda.

Sono necessari 5 scambi di messaggi per evitare Reflection Attack.

Challenge-Response insicuro.

Dopo l'autenticazione, su un canale sicuro si stabilisce una chiave simmetrica di sessione per fornire integrità e confidenza dei messaggi. La session key temporanea permette di limitare il key wearing della chiave di autenticazione. Essa viene distrutta alla fine della sessione.

- **Key Distribution Center**: nel caso di un sistema distribuito non si può conservare la chiave di ogni nodo, quindi si usa un Key Distribution Center che fornisce *ticket* che permettono lo scambio di messaggi autenticato tra 2 nodi.
  Per evitare *MITM* (tra nodo e KDC) e *Replay Attack* è necessario usare il protocollo Needham-Schroder, che stabilice di crittografare la risposta del KDC e generare numeri casuali che modifichino ogni messaggio evitando eventuali replay.



I primi due passaggi sono necessari per evitare Replay attack e stabilire un numero random Rb1 con cui verificare la comunicazione successiva

- **Autenticazione con crittografia a chiave pubblica**: sicura in 3 passaggi, i primi 2 crittografati usando la chiave pubblica del destinatario, il terzo usando la chiave di sessione.

- **Kerberos**: l'autenticazione utilizza il protocollo Needham-Schroeder.
  Il KDC è diviso in due parti: Authentication Server (AS e Ticket Granting Service (TGS).
  I nodi si autenticano su una workstation con login-password. La workstation verifica comunicando con AS se la password è corretta, a quel punto ottiene dal TGS un ticket e lo consegna al nodo richiedente.

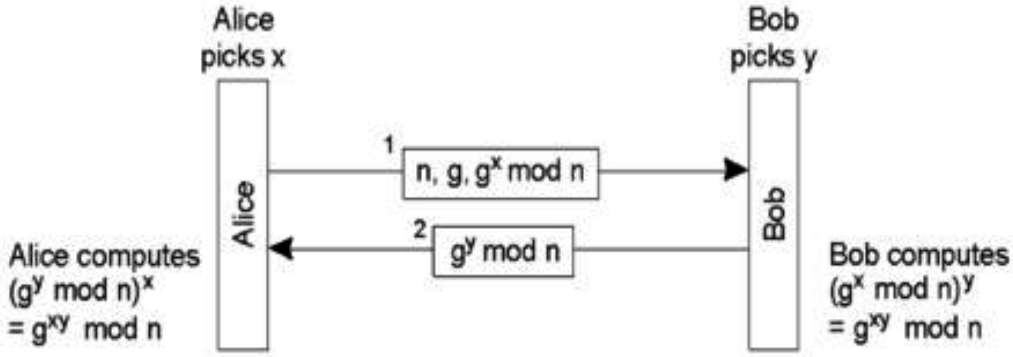
- **Secure Group Comunication**: per rendere sicura la comuncazione tra gruppi di nodi:
  - Crittografia simmetrica (una chiave per ogni coppia di partecipanti: n^2 chiavi);
  - Crittografia asimmetrica: è richiesto parecchio overhead computazionale;
  - Crittografia simmetrica con chiave singola S (una chiave in tutto).

  Il problema principale da gestire è rappresentato dalle operazioni di join/leave da un gruppo.
  Si richiedono *Backward & Forward Secrecy*: chi entra nel gruppo non può leggere i messaggi emessi prima della sua ammissione, e chi esce dal gruppo non deve più poter leggere i messaggi pubblicati dal gruppo.
  Soluzione: si cambia la chiave del gruppo, crittandola adeguatamente.
  Come scegliere la nuova chiave? Se la sceglie il server, ho il problema della distribuzione della nuova chiave. Se la scelgono i partecipanti, il problema diventa l'agreement sulla chiave tra i membri del gruppo.

  Come implementare una distribuzione efficiente della chiave?
  - Utilizzando una *topologia gerarchica di chiavi logiche*: nelle foglie ho i membri con le loro chiavi, mentre in root ho la Data Encryption Key (DEK). Ciascun membro conosce le Key Encryption Keys fino alla root. Dopo una leave, si cambiano tutte le chiavi conosciute dal membro che esce dal gruppo: si cambia DEK e tutte le chavi presenti sul percorso dal membro uscente al nodo. Le nuove chiavi si possono distribuire in modo efficiente sfruttando i sottoalberi stabili (che non vengono mutati).
  - Usando una *Centralized Flat Table*: ciascun nodo ha una chiave per ogni bit del proprio ID. Se esce un membro, si guarda il suo ID e si cambiano tutte le chiavi associate ai bit della codifica dell'ID. Per quanto riguarda DEK, si critta con tutte le altre chiavi. In questo modo tutti tranne chi è uscito possono decrittare la nuova chiave.

- **Secure replicated Servers**: dei client si connettono in modo trasparente ad un server replicato. Bisogna filtrare fino a *k* risposte corrotte da un intruso. In una soluzione semplice, si usano $2k+1$ server replicati, e ciascuno firma la propria risposta. Il client verifica la firma e decide a maggioranza. Così però il client deve conoscere l'identità e la chiave pubblica di tutti i server.
  Soluzione più usata: *Schemi a soglia (n,m)*: si divide un valore "segreto" in *m* pezzi, ne bastano *n* per ricostruire il segreto.
  Altro problema: Come distribuire le chiavi iniziali in modo sicuro?

- **Algoritmo Diffie-Hellman** (metodo per scambiare chiavi simmetriche su un canale insicuro):



n e *g* sono numeri conosciuti pubblicamente, la sicurezza si basa sull'intrattabilità dei logaritmi discreti.

DH non funziona contro attacchi attivi, tramite cui è possibile un attacco MITM.

Diffie Hellman non assicura autenticazione, permette a due nodi sconosciuti di scambiare in modo sicuro una chiave simmetrica che poi verrà usata per la comunicazione. Per evitare attacchi attivi come MITM servirebbe autenticazione ed integrità su entrambi i messaggi DH.

DH è molto usato per scambiare chiavi pubbliche tra nodi, ma come posso autenticare le chiavi pubbliche? Tramite certificati.
Certificati: sono collezioni di informazioni sull'identità che permettono di autenticare le chiavi pubbliche. Ogni certificato è dotato di una chiave pubblica ben conosciuta e firmati da una Certification Authority (le chiavi pubbliche delle CA sono incluse nel sistema operativo!)
Sono possibili modelli di sicurezza gerarchico (CA di autorità centrali, utenti sulle foglie) e PGP (utenti autenticano altri utenti firmando la loro chiave pubblica).

➔ ***Controllo degli accessi e altro...***

Il controllo degli accessi è reso possibile da un *reference monitor* che media le richieste tra soggetti che accedono ad oggetti, autorizzandoli.

Sono possibili varie implementazioni, che influenzano il modo in cui si accede al monitor:
- *AC Matrix e List*: una matrice di utenti e oggetti con i relativi permessi (matrice sparsa).
  Si può anche implementare un ACList per ogni oggetto, per limitare lo spreco di spazio della ACM. Per risparmiare ulteriore memoria è possibile introdurre il concetto di gruppo per avere una gerarchia nell'ACL. La gestione viene molto semplificata anche se il lookup rimane costoso.
- *Role-Based AC*: ogni utente ora è associato ad uno o più ruoli, che vengono assegnati agli utenti al login. I ruoli possono essere cambiati dinamicamente a runtime, definiscono i vincoli sui dati.
- Esempio *Amoeba*: permette delega dei propri permessi ad altri processi, basandosi su proxy formati da certificato+chiave.

Codice mobile

Nel caso di codice mobile è necessario proteggere l'*agente* (data e codice) dalle alterazioni (ma troppa protezione sarebbe eccessivamente restrittiva) e l'*host*, facendo in modo che il codice

scaricato abbia diritti molto limitati per evitare eventuali danni.

Per proteggere l'agente si usa *Append Log*: si implementa in modo che i dati possano essere solamente aggiunti, non cancellati oppure modificati. Nel log si inserisce inizialmente un checksum.

Per proteggere l'host si usa:
Sandbox: si controlla l'esecuzione delle singole istruzioni di codice mobile. In Java si usa TCL (Trusted ClassLoader), Byte Code Verifier, Security Manager.
Playground: si usa una macchina dedicata per eseguire il codice mobile, implemento diversi livelli di protezione.
Signed Code: uso di policy differenti in base alla autenticazione o meno del codice.

Con Java è possibile usare tecniche avanzate, dalle ultime versioni è stato introdotto un file di policy.

Pagamenti elettronici

Nei pagamenti elettronici è necessario trasferire con sicurezza messaggi tra il rivenditore, la banca e l'acquirente. Si usano metodi come:

- E-Cash: il rivenditore controlla autenticità del pagamento verificando la firma della banca sul prelievo effettuato dall'acquirente. Non c'è privacy, banca e rivenditore sono in contatto;
- SET: adatto per pagamenti con carta di credito, consente di avere privacy usando una doppia signature. L'acquirente invia alla banca e al rivenditore una richiesta uguale ma decrittabile solo nella parte che li riguarda. Se i due destinatari accettano la propria parte di messaggio, avviene l'acquisto.

10) Peer to Peer

Con peer to peer si intende una rete di computer senza client o server fissi ma composta di un numero di nodi "alla pari", equivalenti, che fungono sia da client che da server verso gli altri nodi.

E' in antitesi al modello client/server, promuove la condivisione di risorse e servizi attraverso uno scambio tra pari, permette grande scalabilità, dinamismo, fault tolerance, ottimizzando le performance e l'uso delle risorse.
Ogni nodo è indipendente dagli altri, con capacità molto diverse tra loro, software e hardware differenti, l'unico punto in comune è un insieme di regole di comunicazione, un protocollo.

Il problema fondamentale nei sistemi p2p è la ricerca dei dati e la loro localizzazione. In base alla scelta presa dai diversi protocolli distinguiamo in:

➔ Database centralizzato (Napster)

Napster è stata la prima applicazione di file-sharing P2P.
Pur non essendo un sistema peer-to-peer puro, poiché usava server centralizzati per indicizzare i contenuti, Napster consentiva agli utenti di scambiare indipendentemente file senza intermediari.
Era semplice e rapido, ma la centralizzazione dei dati portava all'esistenza di un single point of failure e non consentiva la ricerca distribuita (e l'ha reso punibile penalmente).

Operazioni base:
- JOIN dei nodi sul server
- PUBLISH dei propri files sul server
- SEARCH sul server dei dati
- FETCH dai nodi "alla pari" dei file

➔ Query Flooding (GNUtella)

Gnutella non ha autorità centralizzate, usa un algoritmo di ricerca di tipo flooding, ogni query è inoltrata ai client vicini e così via fino ad un determinato numero di HOP.

- JOIN: all'avvio di ogni nodo avviene il bootstrap che consente di individuare almeno un nodo *anchor* sulla rete, tramite web caches o UDP host. A quel punto si inviano messaggi PING ai dispositivi più vicini e ogni nodo si crea una propria rete con i nodi che rispondono PONG.
- niente PUBLISH dei propri files
- SEARCH tramite flooding sui nodi vicini
- FETCH dei dati dai nodi "alla pari"

Gnutella è quindi totalmente decentrato, la funzione di ricerca è distribuita ma necessita di flooding (elevato overhead, bassa efficienza e visibilità dei file), la rete è instabile a causa dei frequenti cambiamenti di topologia.

Nella seconda versione di Gnutella si è creata una gerarchia nella rete, introducendo *Ultrapeer* che agiscono come proxy e riducono l'overhead e *Leaf*, cioè i normali host che interagiscono solo con gli Ultrapeer, evitando un flooding su tutta la rete.

➔ ***Intelligent Query Flooding (Kazaa)***

Kazaa è un'applicazione peer-to-peer che usa il protocollo FastTrack, senza server centralizzato ma basato sui supernodi.
Ogni peer invia query al supernodo più vicino che cerca di localizzare il contenuto richiesto, in caso di fallimento inoltra la richiesta tramite flooding ad altri supernodi, e così via. Ogni nodo che esegue Kazaa può diventare supernodo, se ha una connessione veloce ed è sufficientemente stabile.
Un supernodo indicizza i file dei client vicini, agendo quindi come un server centralizzato temporaneo e risolvendo il problema del flooding di Gnutella.

Per abilitare l'avvio del sistema sono memorizzati una lista di numeri IP dei supernodi più comuni, da cui ottenere una lista più aggiornata dei supernodi attuali e verso cui mandare la propria lista dei file.

JOIN contattando supernodi predefiniti
PUBLISH della propria lista su un supernodo
SEARCH verso un supernodo che eventualmente inoltra ad altri
FETCH dai nodi che possiedono i dati, simultaneamente

Per distinguere file simili ma non identici viene usata una funzione di hash detta UUHash (veloce ma non molto sicura), che permette di verificare la correttezza delle sottoparti dei files scaricati dai client.

FastTrack cerca di mediare tra una rete totalmente distribuita e l'efficienza dovuta all'introduzione di supernodi che agiscono come server centralizzati temporanei, in modo da avere una topologia di rete più stabile e ottimizzare il trasferimento di messaggi di query.
Purtroppo il flooding non da' garanzie sui tempi e modi di ricerca, inoltre il protocollo è proprietario.

- ***ALTRE APPLICAZIONI***

WinMX: client che si connette a più server contemporaneamente e poi si muove in un contesto decentralizzato in cui supernodi agiscono da server di index dei file.

Edonkey/EMule: usa uno schema simile a Napster ma usando diversi server e ricerche multiple.

Direct Connect: usa *hubs* pubblici o privati a cui si connettono un certo numero di client che comunicano e scambiano dati. E' una via di mezzo tra una chat e una vera applicazione P2P.

➔ ***Swarming (bittorrent)***

Bittorrent è un protocollo P2P che permette a molti client di scaricare lo stesso file senza rallentare il download a causa delle troppe richieste. Questo è possibile perchè ogni nodo scarica porzioni del file e contemporamente esegue un upload di porzioni già scaricate e richieste da altri nodi.
Ogni file viene quindi scomposto in frammenti di dimensione fissa che verranno poi ricomposti e controllati con l'algoritmo SHA1.

Il protocollo bittorrent ha come obiettivo un fetch efficiente e rapido dei file, ma non si occupa della ricerca dei file, che è possibile solo dopo aver scaricato dei piccoli meta-file .torrent, che

contengono i checksum di controllo e l'indirizzo del tracker.
I *trackers* sono dei server che memorizzano per ogni file il numero di *seed* (peer che possiedono tutto il file) e *leech* (peer che possiede porzioni del file).
(Swarm significa seed+leech)

Un limite di bittorrent deriva dal fatto che non è pensato per *condividere* ma più che altro per *diffondere* file, infatti i dati poco scaricati tendono a non essere più reperibili, appena l'ultimo nodo che ne possiede una copia si disconnette dalla rete. Questo causa la "morte" dei file più rari o di vecchia data, mentre permette enormi velocità per i file più recenti e famosi, perchè la banda a disposizione è elevatissima.

Più un client esegue upload verso certi nodi, più ottiene banda da questi stessi nodi... per evitare che nodi senza file da condividere o appena entrati in rete siano bloccati, si usa un meccanismo di *choking* che riserva una parte di banda da assegnare casualmente a peer che ne fanno richiesta (ogni 30sec).

Purtroppo bittorrent non si può considerare completamente distribuito, perchè è necessario un server tracker centralizzato che effettua il bootstrap iniziale, indirizzando i peer verso i nodi che possiedono il file torrent specificato.

➔ ***Unstructured Overlay Routing (Freenet)***

Freenet è un'applicazione P2P costruita per assicurare libertà di comunicazione su internet, infatti rende possibile a chiunque pubblicare e leggere informazioni in modo completamente anonimo.

Freenet si basa su un controllo decentralizzato, scalabile tra centinaia di migliaia di utenti e resistente ai guasti o ad attacchi sui singoli nodi della rete.

JOIN verso i client conosciuti, si ottiene un id unico.
PUBLISH dei file verso il nodo che possiede file con id più vicino.
SEARCH dei file in base all'id, usando un algoritmo simile alla ricerca negli hash
FETCH verso il nodo con i file.

La rete è disegnata in modo tale che le informazioni sono crittografate e replicate su molti nodi in contemporanea, a seconda della frequenza delle richieste. Ogni nodo comunica solo con i propri vicini senza gerarchie, non è possibile conoscere da chi proviene una richiesta, mentre l'inoltramento dei messaggi dipende solo dalla chiave che identifica il file e dalla cache che associa ID al nodo a cui inoltrare.

Il protocollo usato da Freenet è quindi basato sulle chiavi in modo simile a quello che avviene con le hash table distribuite, a parte che i nodi in questo caso usano un algoritmo di routing euristico (che quindi non garantisce il risultato).
Cercando un file il client inoltra la richiesta al nodo che si ritiene sia "più vicino" ai dati, basandosi sul fatto che la pubblicazione dei file avviene in modo ragionato (file con id simile sono vicini).
Se il nodo che riceve la richiesta non trova nulla, esegue a sua volta la ricerca e così via fino ad un numero predeterminato di hops. La pubblicazione dei file funziona allo stesso modo, in modo tale che ricerca e pubblicazione puntino alla stessa posizione, partendo da un determinato ID del file.
La ricerca in FreeNet si basa sullo studio Milgram (bastano pochi 5-10 hop per raggiungere qualsiasi persona nel mondo), gran parte del traffico passa su nodi con molti vicini, nodi che hanno un'ottima conoscenza della topologia di rete.

Appena un nodo si unisce alla rete non ha nessuna tabella di routing e quindi inoltra casualmente i

file che ha intenzione di pubblicare. Con il passare del tempo le informazioni inizieranno ad organizzarsi autonomamente, rendendo FreeNet una rete auto-organizzante.
La presenza di crittografia assicura l'anonimato, gli hash permettono di controllare la validità dei dati, esistono anche meccanismi di validazione per autenticare i file.

Anche se le garanzie di sicurezza non sono ancora state provate, FreeNet usa un routing intelligente che riduce l'overhead limitando al massimo il flooding e rendendo altamente efficiente e rapida la comunicazione.

➔ ***Structured Overlay Routing (DHT)***

Distributed Hash Table sono una classe di sistemi distribuiti decentralizzati che suddividono tra più nodi diversi set di chiavi in modo analogo alle normali tabelle di hash.
E' un modello sviluppato dai ricercatori accademici per raggiungere tempi di ricerca ridotti e scalabilità su milioni di nodi.

I nodi della rete sono collegati formando una toplogia di rete strutturata in modo da ottenere sia la decentralizzazione (tipo FreeNet) che l'efficienza (tipo Napster e i sistemi con server di index).

JOIN su un nodo di bootstrap, il nodo ottiene un ID e viene inserito nella struttura dati.
PUBLISH dei file a seconda della propria chiave.
SEARCH di un file in base alla sua chiave
FETCH dal nodo che ha il file oppure si ottiene indirizzo IP e si effettua una normale richiesta.

- esempio: **Chord**
I nodi e le chiavi sono organizzati in un anello logico, i file con chiave *k* sono salvati nei nodi con l'identificatore *id* più vicino a *k*.
I nodi non cercano un file inoltrando i messaggi per tutto l'anello, ma si usano meccanismo come *finger table* per limitare l'overhead, tenendo memorizzato in una tabella *m* valori che permettono di "saltare" nelle zone della rete più vicine al file con chiave specificata (male che vada ad ogni hop si dimezza la distanza verso il file).
A seguito di un join di un nodo *n* è necessario aggiornare la topologia della rete, i riferimenti dei nodi prima e dopo di *n*, così come le tabelle di finger.

La ricerca avviene nell'ordine di log(N), ma è piuttosto fragile e non tiene conto della topologia fisica della rete, con tutti i problemi di latenza ad essa correlato. Non è semplice supportare la ricerca per parole chiave (mentre è ottima quella in base ad ID).

- esempio: **Kademlia**
Crea una struttura di rete che si basa sulla distanza tra due nodi, calcolata tramite OR esclusivo tra gli ID di due nodi (non distanza geografica quindi).

- esempio: **CAN**
Content Addressable Network è costruita basandosi su uno spazio cartesiano virtuale multidimensionale, partizionato tra tutti i peers in modo che ognuno abbia un proprio spazio logico identificato da coordinate.
Il routing inoltra i messaggi ai nodi che sono logicamente più vicini al destinatario, seguendo le coordinate.

- esempio: **Pastry**
Diversamente dalle altre implementazioni Pastry tiene conto della topologia fisica dei nodi, che

vengono comunque legati in un anello logico. I peer sono identificati da un ID di 128 bit, mentre i file sono riconosciuti dal proprio hash + chiave pubblica del possessore + numeri random.

Sia FreeNet che DHT supportano un meccanismo di ricerca basato sulla conoscenza delle chiavi dei file di cui si ha bisogno, così che la classica ricerca in base a parole chiave deve essere implementata "sopra" i protocolli, in modo meno efficiente dei modelli P2P più usati.

11) Jini tutorial

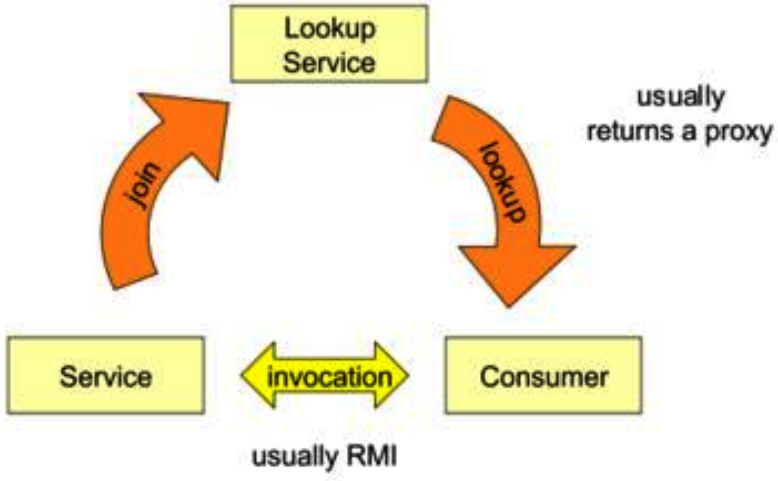
Jini è un'architettura di rete adatta a costruire sistemi distribuiti nella forma di servizi modulari cooperanti, ha come obiettivo la dinamicità dei propri componenti.
Jini offre diversi servizi tra cui binding, configurazione, sicurezza, coordinamento (javaspace, mailbox...) e la tolleranza ai guasti (transaction manager...)

L'implementazione è presentata come JavaStarterKit, costruito sopra la tecnologia J2SE standard:

Lanciando tutti i servizi di Jini tramite lo script Launch-All, i servizi attivi più importanti sono:
- Lookup Service (reggie): registazione ai servizi.
- Persistence Service: Javaspaces
- Transaction Service (mahalo).

➔ Discovery

E' necessario "scoprire" i servizi a run-time, esiste un registry centralizzato detto Lookup Service, su cui ci cercano (lookup) e registrano (join) servizi.



Un servizio effettua join su Lookup Service, il consumatore attiverà il servizio contattando a sua volta il servizio di lookup.

Ogni client ottiene un riferimento al Lookup Service nella fase di bootstrap del sistema tramite richieste UDP broadcast, usando la classe di aiuto Discovery Class.

esempio:

```
LookupLocator l = new LookupLocator(''jini://localhost'');
ServiceRegistrar ls = l.getRegistrar(); // ottengo un ''proxy'' al servizio di Lookup
```

La registrazione dei servizi avviene tramite interfacce: un processo registra le interfacce di cui fornisce implementazione sul registry, un client ottiene queste specifiche e le invoca tramite RMI.

Se non cononsco la posizione del LookupService posso usare classi di aiuto che permettono di specificare più gruppi e/o più indirizzi:

```
// LookupDiscovery, LookupLocatorDiscovery, LookupDiscoveryManager
LookupDiscovery ld = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
ld.addDiscoveryListener(new DiscoveryListener() {
  // devo implementare 2 funzioni
  public void discarded(DiscoveryEvent arg) {  //chiamato se perdo LS
  }
  public void discovered(DiscoveryEvent arg) {
       ServiceRegistrar ls = arg.getRegistrar();  // siamo qui se trovo un LS
  }
}
```

➔ ***Come usare i servizi***

Dopo aver ottenuto un ServiceRegistrar è possibile registrare o cercare servizi:

```
ls.register(new ServiceItem(id,service,null),10000L);
// id: id del servizio o null se è la prima volta,
// service: oggetto da registare o stub del servizio remoto
// null: eventuali attributi del servizio
// longnumber: tempo di lease


ls.lookup(new ServiceTemplate(id, new Class[] {Interface.class}, null);
// id del servizio, interfaccia del servizio, attributi di descrizione del servizio.
```

esempio:

- interfaccia *nome* con un metodo *wassup()*:

```
public interface nome extends Remote {
       public void wassup() throws RemoteException;
}
```

- classe nomeImpl che implementa l'interfaccia e stampa

```
class nomeImpl implements nome { ...  }
```

- aggiungiamo, dopo la getRegistrar del server, la registrazione del servizio:

```
Remote servizio = UnicastRemoteObject.toStub(new nomeImpl());
r.registrar (new ServiceItem(null, service, null, 60000L)); // lease di 60 secondi
```

E' necessario specificare nel codebase la directory in cui trovare l'interfaccia (in run as... aggiungo riga di codebase con la path specificata).

- nel client posso fare lookup verso quel servizio:

```
nome n = (nome) r.lookup(new ServiceTemplate(null,new Class[] {nome.class}, null));
n.wassup(); // eseguo il metodo remoto
```

12) CORBA

CORBA significa Common Object Request Broker Architecture, definisce un framework aperto per sviluppare tecnologie distribuite OO, permette agli sviluppatori di creare applicazioni distribuite come una serie di oggetti cooperanti.

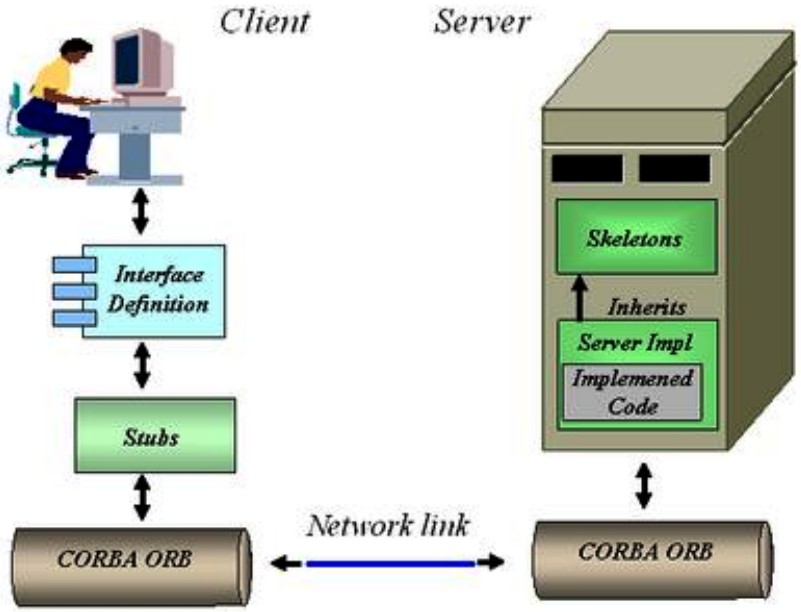
CORBA facilita lo sviluppo di sistemi distribuiti fornendo:

- Una infrastuttura per permettere la comunicazione fra oggetti distribuiti e indipendenti
- Un set di servizi utili
- Un supporto che permette ad applicazioni, implementate usando vari linguaggi, di interoperare

Esistono diverse implementazioni delle interfacce standard disponibili, scritte seguendo un Interface Definition Language (IDL), un linguaggio che specifica in quale modo si debbano costruire le API degli oggetti CORBA nei diversi linguaggi di programmazione.
I vari componenti comunicano attraverso il ***broker ORB***, che può essere visto come l'intermediario. Gli oggetti sono "presentati" al broker attraverso la scrittura di un'interfaccia nel linguaggio IDL.

➔ Architettura

L'architettura di CORBA consiste nel broker ***ORB***, il cuore del sistema, che permette ai diversi oggetti di comunicare tra loro e usufruire dei servizi (CORBA Service, nella loro implementazione particolare).
Al suo interno l'ORB definisce un protocollo standard (IIOP), che permette a *client* e "server" (parte dell'oggetto remoto CORBA, detto *servant*) di comunicare tramite un'interfaccia.



L'immagine non tiene conto dell'esistenza del POA per semplicità.

In realtà ogni server contiene un POA che inoltra le chiamate verso i diversi *servant* locali, che rappresentano gli oggetti.

Un client deve avere un riferimento al servant, verso cui può invocare metodi remoti con passaggio di parametri per copia o per "riferimento" (il valore viene modificato dal servant e poi restituito al client, avviene una copy-return).

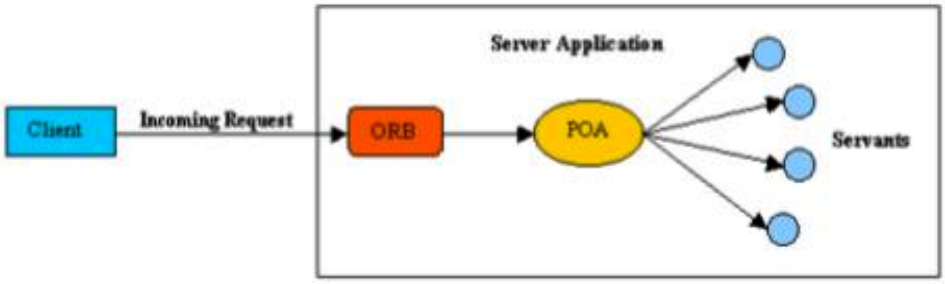
I due nodi comunicanti devono avere stub e skeleton (generati a tempo di compilazione), scritti in

IDL, che effettuano il marshalling dei parametri. E' anche possibile avere una comunicazione dinamica tra oggetti che non conoscono i rispettivi stub e skeleton, usando determinate interfacce e repository distribuiti.

Un ***Object Adapter*** (classa astratta su cui costruire il servant) media la comunicazione tra l'oggetto remoto e l'ORB, incapsulando i principali meccanismi e regole utili per gestire i servant, creare i riferimenti agli oggetti e gestire l'invocazione di servizi.
Gli Object Adapter permettono di accedere ai servizi forniti dall'ORB, ma è necessario implementare le funzioni standard, non disponibili in CORBA.

I protocolli usati per connettere server e client sono *GIOP* e *IIOP*.

La gestione degli oggetti servant avviene nel Portable Object Adaptor (***POA***), situato tra l'ORB e i vari servant. Questo è possibile perchè i messaggi dai client contengono l'indirizzo del POA oltre a quello del servant a cui fare dispatch.
Il POA esegue l'unmarshalling, crea i riferimenti agli oggetti che poi saranno esportati tramite Naming service e controlla che le richieste dei client siano rivolte a oggetti che hanno un servant.



I servizi CORBA sono forniti solo come interfaccia, ogni vendor deve implementare la propria versione. I principali servizi sono il *Naming* (il servant associa una stringa ad un oggetto), servizi di *Event* e *Notify* (supportano comunicazione many-to-many asincrona, usati per publish-subscribe), il servizio *Transaction* (supporto alle transazioni)...

[Le implementazioni free migliori sembrano essere TAO e JacORB.]

➔ ***Come sviluppare applicazioni***

J2SE fornisce un broker ORB, un compilatore IDL — Java e un demone per il servizio di Naming.
JacORB della SUN fornisce diversi servizi ed è integrato in Java.

Il linguaggio IDL permette di definire interfacce indipendenti dal linguaggio di programmazione usato. Il client invocherà funzioni remote solo usando l'oggetto IDL del servant.

Le operazioni in IDL supportano parametri *in*, *out* e *inout*, possono lanciare eccezioni, contengono tipi built-in (string, int, any...) e tipi complessi (struct, array...).
Il "tipo" principale in IDL è *interface*, definisce l'interfaccia di un oggetto CORBA come una serie di metodi:

```
[oneway] <resType> <nome> (par1,...,parN)   [raises (eccez1...)]    [context (c1,...,cN)]
```

Esempio di file NomeX.idl:

```
module  Nome {

       struct Dettagli {
         ...
       };

       exception NomeEx {... };
       interface NomeInt {
         void deposita (in double soldi, out double bilancio);

  // esegue sia GET che SET a seconda dei parametri con cui viene chiamato nomeFunz
         readonly attribute Dettagli nomeFunz;
       };
};
```

Per compilare un file .idl eseguo:

```
> idlj -fall NomeX.idl
```

Il comando *jdlj* genera stub/skeleton, varie classi ed interfacce utili in una directory Nome:

- una classe NomeX che implementa la struttura dati estendendo NomeXOperation;
- una classe interfaccia NomeInt con i metodi astratti e i diversi parametri;
- una classe NomeEx che estende l'interfaccia Exception
- XXXHelper per ogni classe/eccezione che permette di effettuare cast dinamici (.narrow)
- XXXHolder per ogni classe/eccezione che permette di gestire il passaggio di parametri *inout* e *out*, cioè passati per copy-return, cosa che in Java non è supportata. In pratica NomeHolder è una classe contenente i parametri e viene usata per passare "per riferimento" i vari valori *inout*.
- NomeXOperation è un interfaccia contenente i metodi definiti nell'IDL.
- NomeXPOA fornisce le funzionalità base di CORBA

Il servant dovrà implementare le funzionalità richieste dall'interfaccia NomeX.
Creo una classe NomeXImpl che estende NomeXPOA e definisce il metodo *deposita()*.

Il "server" conterrà un main che istanzia il servant, lo inizializza tramite *init()* e ottiene il POA:

```
ORB orb = ORB.init(args, null);  // passo porta e informazioni come parametro
POA rootpoa = POAHelper.narrow(orb.resolve_initial_references(''RootPOA''));
```

A questo punto attivo il gestore POA e creo il riferimento a me stesso:

```
rootpoa.the_POAManager().activate();
... rif = rootpoa.servant_to_reference(nomeXImpl);
... oggetto =orb.resolve_initial_references(''NamingService'');
```

Sul client:

```
NomeX nome = NomeXHelper.narrow(...)
```

Per eseguire l'applicazione eseguo il demone *orbd* specificando la porta:

```
orbd -ORBInitialPort QUALSIASI
```

Eseguo quindi il server (passando la porta e l'indirizzo) che istanzia e fa binding del servant, poi avvio il client (porta+indirizzo).

Ancora su riferimenti e naming...

E' possibile passare riferimenti agli oggetti via rete, trasformandoli in oggetto usando metodi come

servant_to_reference().

Ogni oggetto CORBA ha un identificatore unico chiamato IOR che si può usare come riferimento.

Il NamingService di CORBA fornisce un mapping tra il nome di un oggetto al suo riferimento IOR. Può risolvere o inserire un IOR associato al nome.