

# INTRODUZIONE AI SISTEMI DISTRIBUITI

## Legge di Metcalfe

Il valore di una rete di telecomunicazioni è proporzionale al quadrato del numero degli utenti connessi al sistema.

## Definizioni di sistema distribuito

- 1) Van Steen & Tanenbaum: un sistema distribuito è una collezione di elementi di computazione autonomi che appare agli utenti come un singolo sistema coerente.
- 2) Couloris & Dollimore: un sistema distribuito è un sistema i cui componenti si trovano in computer interconnessi in rete e comunicano tra loro solo scambiandosi messaggi.
- 3) Lamport: un sistema distribuito è un sistema in cui la failure di un componente di cui neanche si sapeva l'esistenza può rendere il proprio computer inutilizzabile.

## Vantaggi di un sistema distribuito

- Permette di condividere risorse.
- Riduce i costi.
- Permette di risolvere problemi di dimensioni maggiori.
- Supporta la Quality of Service (QoS).
- Migliora le prestazioni, la sicurezza, la disponibilità e l'affidabilità.

NB: non confondere la disponibilità con l'affidabilità.

-> **Disponibilità**: è la probabilità di trovare un server attivo (disponibile) nel momento in cui viene contattato per usufruire un servizio:

$$\text{Disponibilità} = 1 - \text{indisponibilità} = 1 - \frac{\text{periodo in cui il servizio non è disponibile}}{\text{periodo totale}}$$

-> **Affidabilità**: è la probabilità che, se il servizio è operante al tempo 0 (ovvero quando viene invocato), continua a essere funzionante anche al tempo t.

## Caratteristiche di un sistema distribuito

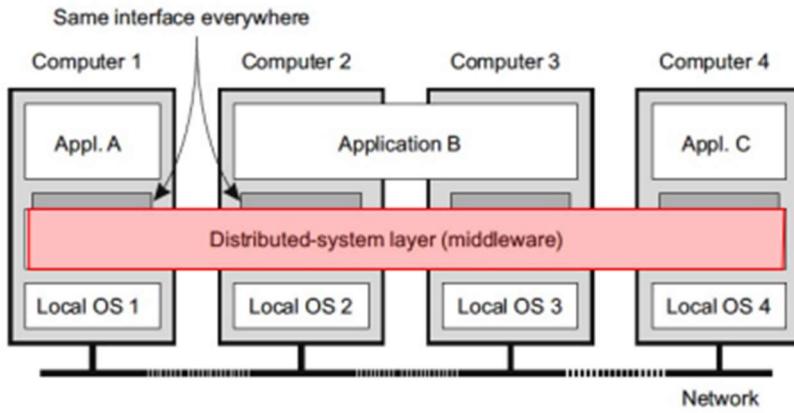
- Concorrenza, che, a differenza dei sistemi entrallizzati, non si può fare a meno di gestire.
- Assenza di un clock globale: piuttosto si hanno tanti clock fisici non necessariamente sincronizzati tra loro.
- Fallimenti parziali e indipendenti ad esempio di qualche nodo o di qualche cavo di rete.

## Challenge per un sistema distribuito

- 1) Eterogeneità: i sistemi possono essere eterogenei:
  - Per la connessione di rete.
  - Per la capacità computazionale.
  - Per il sistema operativo.
  - Per i linguaggi di programmazione usati.
  - A causa di molteplici implementazioni effettuate da parte di più sviluppatori differenti.

Soluzione: introduzione di un **middleware**.

Il middleware è uno strato software al di sopra dei sistemi operativi che fornisce un'astrazione atta a mascherare l'eterogeneità dei servizi sottostanti. Contiene componenti comuni alle diverse applicazioni.



Il middleware inoltre offre diversi servizi:

- Comunicazione.
- Gestione delle transazioni.
- Composizione dell'applicazione.
- Affidabilità.

2) Trasparenza alla distribuzione: è quella proprietà per cui un sistema distribuito appare come un tutt'uno agli utenti. Esistono diversi tipi di trasparenza:

- **Trasparenza all'accesso**: il sistema nasconde le differenze nella rappresentazione dei dati e la modalità con cui le risorse vengono accedute.
- **Trasparenza alla locazione**: il sistema nasconde dove le risorse sono localizzate. Insieme alla trasparenza all'accesso costituisce la network transparency.
- **Trasparenza alla migrazione**: il sistema nasconde il fatto che le risorse possono spostarsi verso locazioni differenti (anche a runtime) senza inficiare sul funzionamento del servizio. Un esempio pratico è la redirezione in http.
- **Trasparenza alla replicazione**: il sistema nasconde l'esistenza di molteplici repliche della medesima risorsa. In tal modo le repliche devono avere tutte lo stesso nome.
- **Trasparenza alla concorrenza**: il sistema nasconde il fatto che le risorse possono essere condivise da più utenti indipendenti.
- **Trasparenza ai fallimenti**: il sistema nasconde i fallimenti e gli eventuali ripristini.

Ambire a una trasparenza completa (soddisfacendo tutti e sei i tipi di trasparenza) spesso risulta eccessivo: ad esempio, le latenze di comunicazione non possono essere sempre nascoste, così come i fallimenti non sono sempre copribili (è difficile distinguere un nodo non funzionante da uno molto lento); inoltre, mantenere i dati replicati perfettamente consistenti e identici tra loro è molto costoso per cui, nella realtà, si effettua piuttosto un trade-off tra consistenza e performance del sistema.

3) Apertura: un sistema è aperto se è in grado di interagire con servizi offerti da altri sistemi indipendentemente dall'implementazione. Perciò, il sistema dovrebbe essere conforme a delle interfacce definite tramite IDL (Interface Definition Language).

Per implementare un sistema distribuito aperto è opportuno separare le politiche dai meccanismi.

- **Esempio di meccanismo**: caching dei dati.
- **Esempio di politica**: per quanto tempo una risorsa può rimanere in cache?

In questo modo però, devono essere configurati molti parametri. A tal proposito, la soluzione consiste nel ricorrere ai sistemi auto-configurabili.

4) Scalabilità: è quella proprietà per cui un sistema distribuito mantiene un livello adeguato di prestazioni anche a fronte di un aumento di:

- Numero di utenti e/o di processi (scalabilità rispetto alla dimensione).

- Distanza massima tra i nodi (scalabilità geografica).
- Numero di domini amministrativi (scalabilità amministrativa).

La scalabilità rispetto alla dimensione (la più sfruttata nella pratica) può essere attuata in due direzioni:

- > **Scalabilità verticale (scale up)**: utilizzo di risorse più potenti (è una soluzione con costo esponenziale).
- > **Scalabilità orizzontale (scale out)**: aggiunta di risorse della stessa capacità di quelle già presenti (è una soluzione con costo lineare).

Vediamo alcune tecniche per scalare un sistema:

- Nascondere la latenza della comunicazione (utilizzando una comunicazione asincrona, che però non è sempre possibile).
- Spostare parte della computazione sul client.
- Partizionare i dati e la computazione su molteplici nodi (principio del divide et impera). Il DNS è un esempio di questo approccio.
- Replicare le risorse e i dati.

### **Tipi di sistemi distribuiti**

- 1) Sistemi di computing distribuito ad alte prestazioni: si classificano a loro volta in:

- > Cluster computing
- > Cloud computing
- > Fog / edge computing

Cluster = insieme di server connessi in una rete ad alte prestazioni, i cui obiettivi sono l'High Performance Computing (HPC) e l'High Availability (HA).

L'architettura tipica del **cluster computing** è quella del master-worker: il nodo master suddivide il carico di lavoro e lo distribuisce tra i nodi worker.

Il cluster computing è una pietra miliare per il **cloud computing**. Il cloud, però, è disponibile a chiunque e su una scala geografica più ampia.

Ad oggi il trend è spostare alcune funzionalità meno onerose ai bordi della rete (**fog / edge computing**):

- > Fog computing: è una piattaforma virtualizzata che fornisce computazione, storage e servizi network tra gli end system e i tradizionali server del cloud. Le risorse si trovano tipicamente (ma non esclusivamente) ai bordi della rete.
- > Edge computing: è simile al fog computing ma è meno integrato al cloud e le risorse sono solo ai bordi della rete.

- 2) Distributed information system: tra questi figurano i **transaction processing system**, che effettuano le operazioni di read e write incapsulate in delle transazioni distribuite (con una politica all-or-nothing), che sono composte da sotto-transazioni eseguite su differenti database server.

La gestione delle transazioni distribuite avviene con un'architettura master-worker, dove il master è il Transaction Processing (TP) Monitor, che è responsabile del coordinamento dell'esecuzione delle transazioni.

- 3) Distributed pervasive system: sono composti da molteplici nodi piccoli, mobili, alimentati a batteria e spesso incorporati in un sistema più grande.

Un esempio di questo tipo di sistemi sono le **reti di sensori**, che possono avere:

- Uno schema centralizzato, in cui i singoli sensori inviano i dati a una locazione centralizzata per processarli.
- Uno schema distribuito, in cui i singoli sensori possono effettuare storage e processamento di dati, seppur in modo limitato.

## INTRODUZIONE AL CLOUD COMPUTING

### Paradigmi di computing

Come si può realizzare un servizio che gestisca milioni di richieste al giorno, faccia fronte a picchi improvvisi di carico e memorizzi exabyte ( $10^{18}$  B) di dati?

Una risposta vecchia e parziale a questa domanda risiede nei primi quattro paradigmi di computing:

- **Utility computing** (forma di servizio informatico in base al quale la società che fornisce il servizio addebita la quantità di utilizzo).
- **Grid computing** (infrastruttura di calcolo distribuito utilizzata per l'elaborazione di grandi quantità di dati mediante l'uso di una vasta quantità di risorse).
- **Autonomic computing** (sistema in grado di autogestirsi senza l'intervento umano).
- **Software as a Service – SaaS** (modello di servizio del software dove il fornitore sviluppa e gestisce un'applicazione web, mettendola a disposizione dei clienti via Internet previo abbonamento).

Nel 2006 però nacque il **cloud computing**, che rappresenta un passo avanti verso la vera soluzione del problema della scalabilità. In particolare, il cloud computing rappresenta lo spostamento di computazione, storage e coordinazione dal singolo server o data center all'Internet.

### Caratteristiche del cloud computing

Esiste una miriade di definizioni di cloud computing, dalle quali è possibile estrarre alcune caratteristiche essenziali:

- **On-demand & self-service**: le risorse cloud possono essere fornite on-demand dagli utenti, senza dover richiedere interazioni col cloud service provider.
- **Ampio accesso alla rete**: le risorse cloud sono in Internet e possono essere accedute indipendentemente dalla piattaforma.
- **Elasticità**: è la capacità di ottenere e rilasciare le risorse cloud al momento del bisogno. A tal proposito, le risorse cloud devono poter essere allocate (scale-out) o deallocate (scale-in) velocemente.
- **Resource pooling**: molti utenti possono usare le stesse risorse hardware (multi-tenancy).
- **Virtualizzazione** delle risorse.
- **Prezzo calibrato all'utilizzo** (modello pay-per-use).
- **Servizio misurato**: molto spesso viene utilizzato il Service Level Agreement (SLA) come metrica di riferimento.

### Modelli di deployment nel cloud

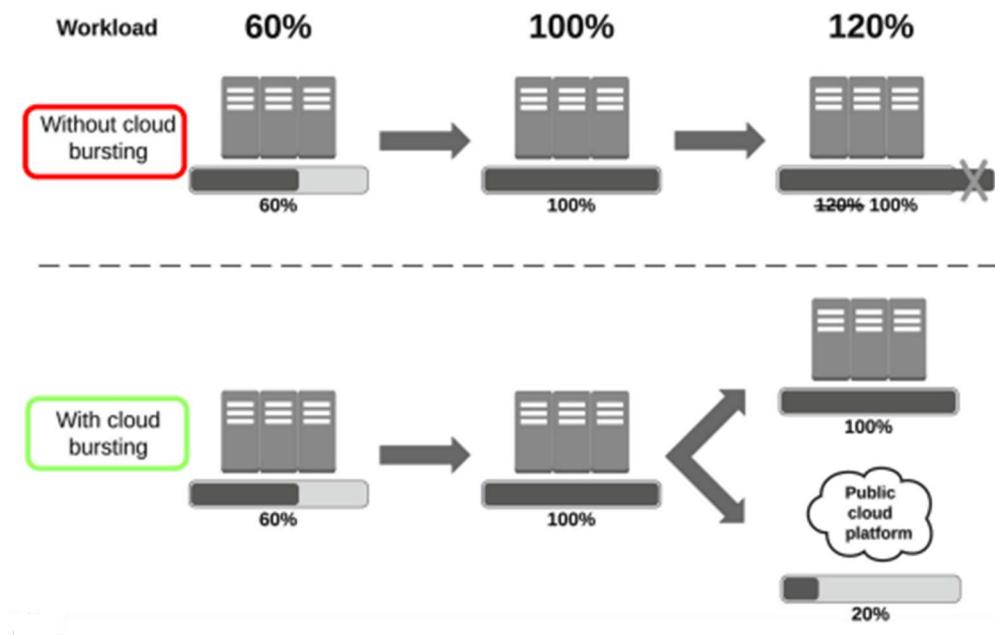
1) Public cloud: l'infrastruttura cloud è resa disponibile da un cloud provider ed è utilizzabile da chiunque. È gestita da un'azienda o un'organizzazione e i suoi servizi possono essere gratuiti oppure a pagamento.

2) Private cloud: l'infrastruttura è ad uso esclusivo di un'organizzazione ed è gestita dall'organizzazione stessa oppure da una terza parte. Offre una maggiore sicurezza e una maggiore facilità di personalizzazione dei servizi. Tuttavia, il suo costo è più elevato e la sua scalabilità è più laboriosa perché le risorse utilizzabili sono limitate a quelle messe a disposizione per l'azienda.

3) Hybrid cloud: l'infrastruttura è data dalla composizione di più infrastrutture (private o pubbliche) distinte. Ha diversi vantaggi:

- Permette il bilanciamento di risorse e costi.
- Permette la differenziazione della privacy, con i dati normali mantenuti nel cloud pubblico e i dati sensibili mantenuti nel cloud privato.
- Migliora la disponibilità del servizio: se va in crash il cloud privato, rimane possibile utilizzare quello pubblico.

- Migliora la scalabilità, mediante l'uso congiunto di cloud privato e pubblico per gestire carichi di lavoro variabili: i picchi insostenibili per il cloud privato vengono gestiti tirando in ballo il cloud pubblico (cloud bursting).



## Modelli di servizio

1) Infrastructure as a Service (IaaS): le risorse fisiche (capacità di processamento, capacità di storage, ecc.) sono gestite dal provider e sono esposte come servizi.

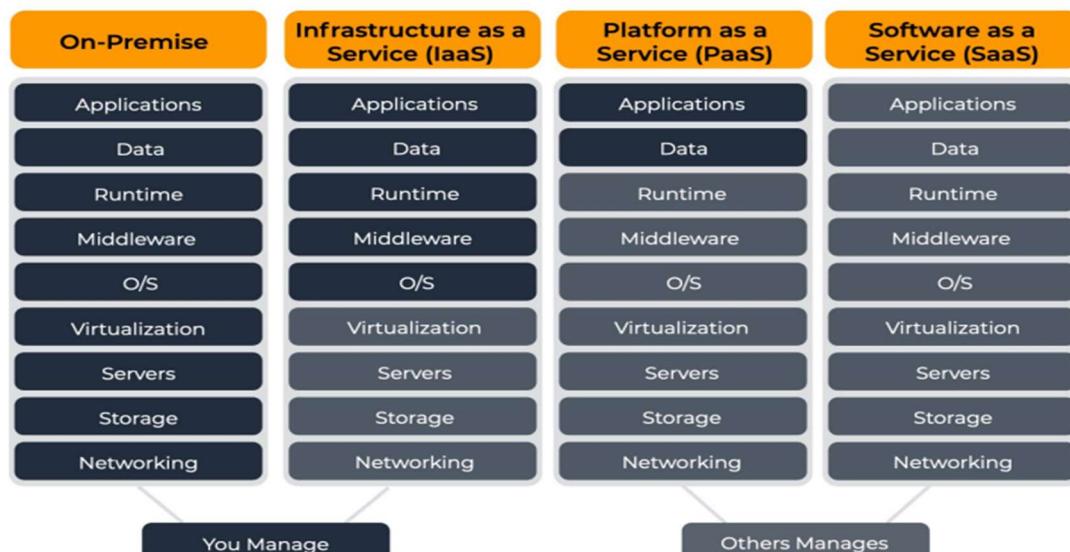
Il customer (ovvero l'utente) non ha controllo sull'infrastruttura cloud sottostante, ma può scegliere l'area geografica, il sistema operativo, la dimensione della macchina virtuale e così via.

2) Platform as a Service (PaaS): consiste in delle piattaforme per sviluppare, eseguire e gestire applicazioni scalabili senza dover preoccuparsi dell'infrastruttura sottostante.

Il customer si occupa solo dello sviluppo e del testing delle applicazioni ed è vincolato ai framework, linguaggi di programmazione e tool offerti dal provider.

3) Software as a Service (SaaS): le applicazioni sono rese disponibili agli utenti tramite Internet.

Il customer non ha alcun controllo sull'infrastruttura e sulle impostazioni dell'applicazione (è tutto controllato dal provider).



## **Modelli di tariffazione**

- 1) Pay-per-use: il prezzo dipende dal livello di utilizzo del servizio.
- 2) Tariffazione fissa: gli utenti devono pagare periodicamente una quota fissa per l'usufruzione del servizio.
- 3) Spot: il prezzo dipende dalla domanda e dall'offerta che il provider sta avendo. Il servizio è interrompibile dal provider in qualunque momento nel caso in cui le risorse disponibili scarseggiano. Perciò, la modalità spot è possibile da utilizzare solo se si devono portare a termine dei compiti non critici (senza vincoli temporali).

## **Multi-cloud**

Consiste nell'uso concorrente di molteplici ambienti cloud. Presenta diversi vantaggi:

- Riduce i rischi di vendor lock-in (situazione di dipendenza tra customer e provider in cui il customer non può richiedere il servizio a provider differenti).
- Migliora i costi.
- Fornisce una maggiore flessibilità.
- Migliora la disponibilità.
- Differenzia la privacy tra dati sensibili e dati normali.
- Migliora la distribuzione geografica.

Tuttavia, il multi-cloud è difficile da gestire perché:

- > Richiede strumenti che automatizzino la divisione del carico di lavoro.
- > Il monitoraggio delle prestazioni deve riguardare molteplici cloud provider.
- > L'utilizzo e i costi delle risorse devono essere riassunti e predetti.

Comunque sia, il multi-cloud può utilizzare l'**Infrastructure as Code (IaC)** per automatizzare la gestione dell'infrastruttura cloud.

## **Capacity planning ed elasticità**

Le applicazioni multi-tier possono essere soggette a variazioni repentine del carico di lavoro richiesto (trattasi di un carico di tipo bursty). Perciò, è necessario dimensionare le capacità delle risorse.

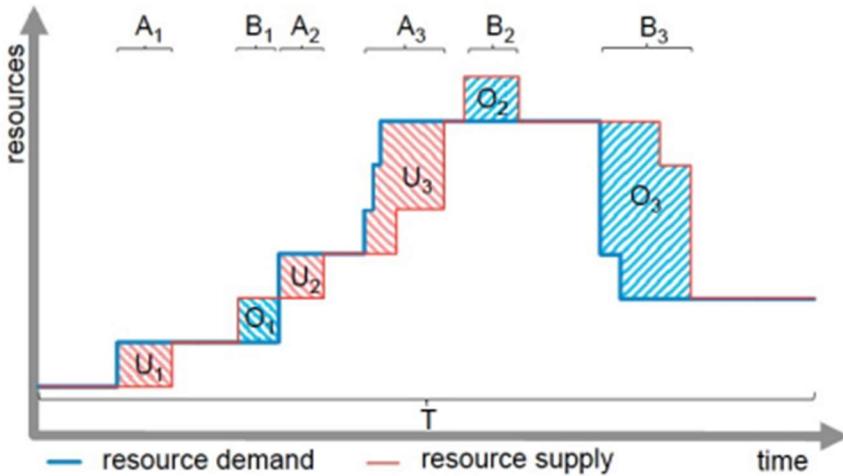
**Capacity planning** = determinazione della dimensione corretta di ogni tier dell'applicazione in termini di:

- Numero di risorse.
- Capacità e dimensioni di ciascuna risorsa.
- Storage, memoria e risorse di rete.

Primo approccio (over-provisioning): consiste nel dimensionamento delle risorse solo in base ai picchi di carico, per cui causa un sottoutilizzo delle risorse e un costo maggiore del necessario.

Secondo approccio (under-provisioning): consiste nel dimensionamento delle risorse solo in base al carico medio, per cui causa un sovraccarico delle risorse e una perdita degli utenti.

Approccio elastico: consiste nell'allocazione e deallocazione delle risorse in modo autonomico (non richiede cioè l'intervento umano) in modo tale che, in ogni istante di tempo, la quantità di risorse allocata corrisponda il più possibile alla reale domanda.



Vediamo due possibili metriche per misurare l'elasticità di un sistema:

- 1) **Accuracy** = somma delle aree di over-provisioning (O) e di under-provisioning (U) durante il periodo di tempo T.
- 2) **Timing** = tempo totale speso in over-provisioning (B) e in under-provisioning (A) durante il periodo di tempo T.

Nella realtà è normale avere dei piccoli momenti di over-provisioning e under-provisioning poiché allocare e deallocare risorse richiede un certo delay (minuti in caso di macchine virtuali, secondi in caso di container).

### Load balancer

Ma come si gestisce la distribuzione delle richieste tra molteplici repliche di server?

Tramite il load balancer, che può essere:

- Centralizzato (single point of failure, collo di bottiglia).
- Distribuito.

Il load balancer può distribuire il carico di lavoro tra più server seguendo più politiche differenti:

-> Scegliendo via via il server in modo randomico.

-> Con uno schema Round Robin.

-> Con uno schema con livelli di priorità.

-> In base alle capacità dei server.

-> In base al livello di carico dei server (problema: molteplici richieste possono essere inoltrate tutte contemporaneamente allo stesso server, che diventa così sovraccarico, dando luogo al cosiddetto effetto gregge; è più indicato individuare un insieme di k server meno carichi e selezionarne randomicamente uno tra questi k).

### Service Level Agreement (SLA)

È l'accordo formale tra il consumatore e il fornitore di un servizio. È composto da:

- Scopo del contratto.
- Scopo del servizio.
- Restrizioni (ovvero casi di non applicabilità).
- Servizi opzionali.
- Tempo di validità del contratto.
- Contraenti.

- SLO (Service Level Objective), che sono indicazioni quantitative sui parametri di qualità di servizio che il fornitore di impegna a garantire (e.g. disponibilità, tempo di ritardo).
- Penalità o compensazioni nel caso in cui qualche SLO non venga rispettato.

Il fatto che un SLO venga rispettato o meno tipicamente deve essere stabilito dall'utente mediante un'attività di monitoraggio, che può incentrarsi su delle metriche orientate al sistema (e.g. utilizzo della CPU, utilizzo del disco, memoria usata) oppure su delle metriche orientate all'applicazione (e.g. tempo di risposta, disponibilità dei componenti).

### **Problemi per i cloud customer**

- Privacy e sicurezza: dove sono localizzati i dati? Chi può accedervi? Vengono protetti critograficamente anche in transito?
- Latenze di comunicazione: è possibile anche avere millisecondi di latenza che, tra l'altro, è possibile dipendano dalla rete e non dal cloud provider. Una possibile soluzione sta nella content delivery network (la più famosa è Akamai), che è un'infrastruttura di reti (con server in tutto il globo) con lo scopo di distribuire i servizi e le risorse.
- Portabilità: capacità di adattare velocemente i dati per passare da un servizio a un altro. Un modo per migliorare la portabilità sta nell'utilizzare container o tool di automazione per il deployment.
- Interoperabilità: capacità di usare più servizi contemporaneamente.
- Supporto misero per la gestione e la negoziazione dei SLA.
- Scalabilità & elasticità.
- Outage dell'infrastruttura: non è così raro avere un importante disservizio.

### **Problemi per i cloud provider**

- Variabilità del livello di servizio richiesto: una possibile soluzione sta nell'utilizzare il modello di tariffazione spot.
- Gestione dei SLA: a volte i cloud provider sono a loro volta utenti per altri servizi; se loro sono vittime di una qualche violazione, si potrebbe generare un effetto a cascata.
- Gestione dei consumi energetici.
- Interoperabilità tra cloud.

# ARCHITETTURA DEI SISTEMI DISTRIBUITI

## Architettura software vs architettura di sistema

- Architettura software = organizzazione logica e interazione dei componenti software.
- Architettura di sistema = instanziazione finale di un'architettura software.

## Stili architetturali

Concentriamoci dapprima sull'architettura software, che può essere definita mediante uno stile architettonico.

Uno stile architettonico è un insieme coerente di decisioni di design riguardanti l'architettura software, ed è formulato in termini di componenti e connettori:

- **Componente** = unità modulare con un'interfaccia definita.
- **Connettore** = meccanismo per collegare più componenti tra loro.

Noi analizzeremo 6 stili architettonici principali.

1) Layered style: i componenti sono organizzati in layer (livelli), col componente al livello  $i$  che invoca tramite messaggi il componente al livello  $j < i$ . I layer tipicamente sono:

- Layer di presentazione (interfaccia).
- Layer di business (logica applicativa).
- Layer di persistenza (talvolta accorpato col layer di business).
- Layer di database.

2) Object-based style: i componenti sono oggetti che encapsulano una struttura dati e offrono API per accedere e modificare i dati. La comunicazione tra le componenti avviene tramite chiamata a procedura remota (RPC) o remote method invocation (RMI).

3) RESTful style: il sistema distribuito è visto come un insieme di risorse che sono gestite in modo individuale dai componenti e possono essere aggiunte, rimosse, recuperate e modificate rispettivamente tramite i metodi PUT, DELETE, GET e POST di http. L'interazione è stateless, per cui lo stato, nel caso in cui ce ne fosse bisogno, viene trasferito dai client verso i server. Le risorse sono identificate da un URI (Uniform Resource Identifier).

Esempio: AWS S3, dove gli oggetti sono memorizzati in dei bucket.

- REST = Representational State Transfer.

Questi tre stili presentano tutti un alto grado di accoppiamento tra i componenti, il che può introdurre delle limitazioni.

Soluzione: far comunicare i componenti indirettamente aggiungendo un intermediario (e.g. un ulteriore livello di indirezione).

Esistono tre diverse proprietà di disaccoppiamento:

-> **Disaccoppiamento spaziale**: i componenti non devono necessariamente conoscere per comunicare e cooperare.

-> **Disaccoppiamento temporale**: i componenti non devono necessariamente essere presenti in contemporanea durante la comunicazione.

-> **Disaccoppiamento rispetto alla sincronizzazione**: i componenti non hanno bisogno di aspettarsi a vicenda e non entrano in uno stato di blocco in attesa di una risposta.

4) Event-driven style: si ha un bus di eventi; nel momento in cui un componente pubblica un evento, quest'ultimo viene notificato ai componenti interessati. Si tratta di un meccanismo che offre disaccoppiamento spaziale; quello temporale, invece, viene soddisfatto se e solo se il bus è in grado di

memorizzare i messaggi.

Esempio: Java Swing.

5) Data-oriented style: la comunicazione è mediata da un'area di memoria condivisa, che può essere attiva o passiva. Si tratta di un meccanismo che offre disaccoppiamento temporale; per quanto riguarda quello spaziale, invece, dipende dall'implementazione.

Le API per lo spazio condiviso sono:

- Write.
- Read.
- Take (= leggi ed elimina).
- Push / notify (API che prevede il polling, ma è disponibile solo nel caso in cui la memoria condivisa è attiva).

Esempio: Linda tuple space.

6) Publish-subscribe style: la comunicazione è mediata da un middleware. I componenti publisher generano eventi, mentre i subscriber si sottoscrivono al middleware in modo da essere notificati in caso di pubblicazione degli eventi a cui sono interessati. Si tratta di un meccanismo che offre tutte e tre le proprietà di disaccoppiamento.

Esistono più varianti di questo schema di sottoscrizione:

-> **Topic-based**: gli eventi sono marcati da un topic, per cui i subscriber possono interessarsi agli eventi di un determinato topic. Tuttavia, poiché i topic sono identificati da delle keyword, l'espressività è limitata.

-> **Content-based**: gli eventi sono classificati rispetto al proprio contenuto (e.g. i metadati). Perciò i subscriber possono sottoscriversi agli eventi utilizzando dei filtri anziché i topic. Tuttavia, si tratta di una variante più complessa da implementare.

D'altra parte, è possibile ricorrere a un'implementazione centralizzata (→ single point of failure, collo di bottiglia) oppure decentralizzata (con un'architettura master-worker, come ad esempio Apache Kafka).

### **Tipologie di architetture di sistema**

Le architetture di sistema possono essere di tre tipi: centralizzato, decentralizzato e ibrido. Analizziamole nel dettaglio.

#### **Architettura centralizzata**

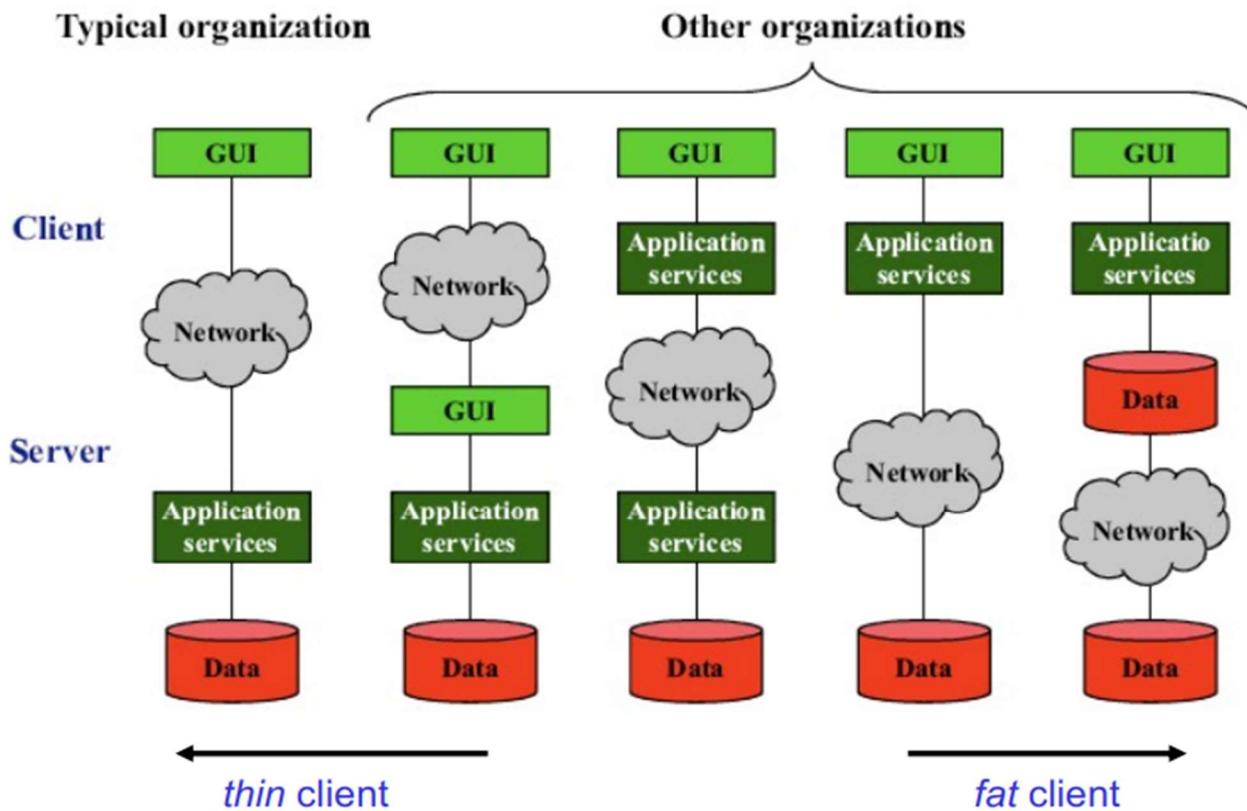
È basata sul modello client-server. La comunicazione tra client e server è basata su scambio di messaggi (il che può rappresentare un collo di bottiglia) ed è spesso sincrona e bloccante. Inoltre, un'operazione nello scambio di messaggi può essere idempotente o meno.

Un servizio idempotente prevede che, a parità di messaggi di richiesta da parte del client, vengano inviati gli stessi messaggi di risposta da parte del server.

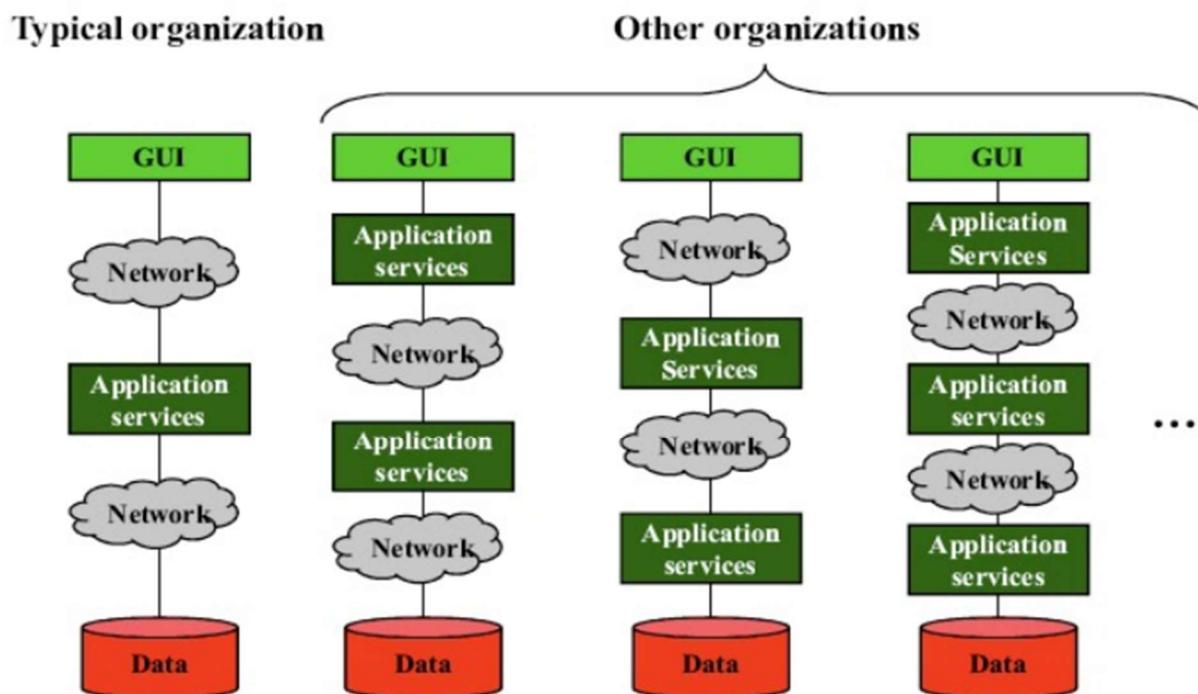
Supponiamo ora di avere un'architettura composta da tre layer (presentation, business e data storage).

Come si possono mappare i livelli logici (layer) sui livelli fisici (tier)?

## 2-tiered client-server architectures



## 3-tiered client-server architectures



Il sistema può essere composto anche da N tier ( $N > 3$ ): in tal caso si avrebbe maggiore flessibilità e disaccoppiamento, ma potrebbe esserci un degrado delle prestazioni dovuto a un maggiore costo di gestione e manutenzione.

Inoltre, un sistema multi-tier può avere una distribuzione:

- **Verticale** se ciascun layer si trova su un determinato server fisico.
- **Orizzontale** se, per ciascun layer, si ha una replicazione dei nodi; in tal caso deve esistere un load balancer che distribuisca le richieste. Questa soluzione contribuisce a migliorare l'elasticità e la disponibilità del sistema.

### **Architettura decentralizzata**

È basata sul modello P2P. I sistemi usano risorse distribuite per fornire servizi in modo decentralizzato. I nodi hanno capacità computazionali simili tra loro e possono ricoprire sia il ruolo di client che il ruolo di server; inoltre, non sono fissi ma possono entrare e uscire improvvisamente e/o frequentemente.

Problemi da affrontare nell'ambito P2P:

- **Eterogeneità** dei peer.
- **Scalabilità**: avere tanti nodi non è una condizione sufficiente per avere un sistema scalabile.
- **Locazione dei dati** e distanza tra i nodi.
- **Tolleranza ai guasti** e agli abbandoni improvvisi dei nodi.
- **Prestazioni**: ad esempio, come si può bilanciare in modo efficiente il carico dei nodi?
- **Free-rider**: è un nodo egoista che richiede delle risorse agli altri nodi ma non mette a disposizione le proprie.
- **Anonimato e privacy**: è possibile introdurre l'onion routing per far sì che la comunicazione sia anonima.
- **Gestione della trustiness** e della reputazione dei nodi.
- **Necessità di difendersi** dagli attacchi.
- **Resilienza ai churn**: è necessario gestire aggiunte e abbandoni continui dei nodi.

Un nodo, quando accede a una rete P2P, esegue tre operazioni principali:

- > Bootstrap: è la fase iniziale in cui il nodo conosce la rete e gli altri nodi da contattare.
- > Resource lookup: è la fase che si occupa di come vengono localizzate le risorse.
- > Retrieval: ottenimento delle risorse.

### **Overlay network**

È una rete logica che interconnette i nodi tra loro; tuttavia, i collegamenti di un'overlay network non corrisponde ai collegamenti fisici tra i nodi.

Qui viene effettuato l'**overlay routing**, che è un meccanismo di instradamento che, al contrario del routing tradizionale, non si incentra sugli indirizzi dei nodi, bensì sulle risorse da recuperare (file, CPU, banda, ecc.). Ciascuna risorsa viene identificata tramite un GUID (Globally Unique Identifier), che in genere deriva da una funzione hash sicura; in realtà, anche i peer hanno un identificatore calcolato mediante una funzione hash. Le overlay network si suddividono in strutturate e non strutturate.

### **Overlay network non strutturate**

Sono overlay network rappresentabili con un grafo qualsiasi: per un nodo che entra in un'overlay network non strutturata è sufficiente contattare un sottoinsieme di nodi già presenti per diventare un loro vicino. Le overlay network non strutturate sono molto utili quando si ha un elevato tasso di churn (connessioni / disconnessioni) e sono altamente resistenti ai guasti; d'altra parte, però, sono caratterizzate da un alto costo di lookup.

Le overlay network sono rappresentabili con dei grafi random di cui ci interessano le seguenti caratteristiche:

- **Coefficiente di clustering  $C_v$  di un vertice  $v$**  = rapporto tra il numero  $L_v$  di archi tra i  $m_v$  vicini di  $v$  e il massimo numero di archi possibili tra loro.

$$C_v = \frac{2L_v}{m_v(m_v - 1)}$$

- **Coefficiente di clustering del grafo** = media dei coefficienti di clustering di tutti i vertici.

- **Lunghezza media del percorso minimo** = media aritmetica tra le lunghezze dei percorsi minimi tra tutte le coppie di vertici del grafo.

- **Diametro del grafo** = lunghezza del percorso minimo più lungo all'interno del grafo.

Noi analizzeremo tre modelli di grafi random.

1) Erdos-Renyi: fissati il numero  $N$  di vertici e la probabilità  $p$  che una coppia di vertici sia connessa con un arco, abbiamo che:

- Il grado di ciascun vertice segue una distribuzione binomiale; in particolare, se  $p_k$  è la probabilità che un vertice  $v$  abbia grado  $k$ :

$$p_k = \binom{N-1}{k} p^k (1-p)^{N-1-k}$$

- Coefficiente di clustering del grafo =  $p$  (è un valore basso, il che è atipico per le reti reali).

- Lunghezza media del percorso minimo =  $\log(N) / \log((N-1)p)$  (è anch'esso un valore basso ma, stavolta, è tipico per le reti reali).

- Il diametro del grafo ha la dimensione di  $\log(N)$ .

2) Watts-Strogatz: è un modello di grafo con un alto coefficiente di clustering e con una lunghezza media del percorso minimo proporzionale a  $\log(N)$ . Quest'ultima proprietà fa sì che i grafi di Watts-Strogatz godano della proprietà dello small world.

In particolare, lo small-world phenomenon (noto anche come 6 degrees of separation) determina che, se sceglieremo due persone casuali sulla Terra, troveremo un path di al massimo 6 persone che le legano (dove un arco tra due persone è relativo alla relazione di "conoscenza").

Tuttavia, a differenza delle reti reali, una rete di Watts-Strogatz non è caratterizzata dalla presenza di hub, che sono nodi con grado molto elevato.

3) Albert-Barabasi: è un modello di grafo noto anche come **scale-free network**. Il grado di ciascun vertice segue una **legge di potenza**; in particolare la probabilità che un certo vertice abbia grado  $k$  è esprimibile nel seguente modo:

$$p_k \sim ck^{-\alpha} \quad \text{where } 2 < \alpha < 3$$

Un esempio pratico di legge di potenza è dato dalla legge di Pareto: "L'80% degli effetti deriva dal 20% delle cause".

La legge di potenza gode della proprietà di **invarianza alla scala**:

$$f(x) \text{ è invariante alla scala se } f(\lambda x) = \lambda^{-\beta} f(x)$$

In altre parole,  $f(\lambda x)$  ha la stessa identica forma di  $f(x)$  ma con una scala diversa.

Inoltre, le distribuzioni che seguono una legge di potenza sono **heavy-tailed**: tendono a 0 molto più lentamente rispetto alle distribuzioni esponenziali. Di conseguenza, i valori che si trovano nella coda hanno una probabilità non trascurabile di verificarsi.

Le scale-free network sono anche caratterizzate dall'**attacco preferenziale** (preferential attachment): quando un nuovo nodo si connette alla rete, sceglie di collegarsi con probabilità maggiore ai nodi con grado più elevato. In tal modo, si forma un numero limitato di hub e il diametro del grafo cresce come  $\log(\log(N))$ : lentissimamente!

Infine, le reti scale-free sono altamente tolleranti ai guasti.

Tornando al discorso sulle overlay network non strutturate, queste possono essere classificate in:

- **Centralizzate**: vi è un directory server che tiene traccia di quali peer posseggono ciascuna risorsa. Sono reti molto semplici ma sono caratterizzate da collo di bottiglia e single point of failure. Un esempio è Napster.
- **Decentralizzate**: non vi è alcun server, per cui le informazioni sono completamente distribuite. Un esempio è Gnutella.
- **Semi-decentralizzate**: vi sono dei super-peer che detengono le informazioni su quali peer posseggono ciascuna risorsa.

Per quanto riguarda le overlay network non strutturate decentralizzate, i peer, per trovare le risorse di cui hanno bisogno, possono seguire più approcci possibili. Per ora vediamone due.

- 1) Query flooding: il peer sorgente invia una richiesta di lookup a tutti i suoi vicini i quali, a loro volta, inoltrano la richiesta ai loro vicini e così via, finché non si trova il possessore della risorsa cercata.
  - Nelle richieste viene aggiunto un TTL (time to live) che proibisce alle richieste stesse di essere inoltrate all'infinito.
  - Nelle richieste viene aggiunto anche un query ID in modo tale che, se un nodo riceve più volte una stessa richiesta, non deve processarla nuovamente.
  - Per inviare un messaggio di risposta contenente la risorsa al peer sorgente, è possibile avviare una comunicazione diretta oppure ricorrere al backward routing (che consiste nel far viaggiare la risposta sullo stesso percorso attraversato dalla richiesta); quest'ultimo approccio è vantaggioso perché non richiede di specificare il peer sorgente nel messaggio di richiesta e le informazioni sulla disponibilità delle risorse si propagano tra i vari peer.

Il query flooding presenta i seguenti svantaggi:

- > Vi è una crescita esponenziale del numero di messaggi.
- > Sono possibili attacchi DoS.
- > Il costo di lookup è elevato:  $O(N)$ , dove  $N$  = numero di nodi nella rete.
- > Non è detto che si riescano a trovare le risorse cercate, anche se esiste un qualche peer che le possiede.
- > Non è detto che i nodi vicini nell'overlay network siano anche vicini fisicamente.

- 2) Random walk: il peer sorgente invia una richiesta di lookup a un solo vicino scelto randomicamente il quale, a sua volta, inoltra la richiesta a un proprio vicino e così via, finché non si trova il possessore della risorsa cercata.

Il problema principale di questo meccanismo è il costo di lookup che cresce ulteriormente: per ovviare a ciò, il peer sorgente (e solo il peer sorgente) può inviare la richiesta a  $k$  suoi vicini, avviando  $k$  random walk simultaneamente.

## Overlay network strutturate

Sono overlay network con una topologia ben definita. Il loro obiettivo è diminuire il costo di lookup, a discapito però del costo di mantenimento della struttura a seguito di ingressi e uscite di peer.

Il routing nelle overlay network strutturate avviene con l'utilizzo di una **Distributed Hash Table (DHT)**, in cui le informazioni sulle coppie (chiave K, valore V) sono distribuite tra i peer. K è la chiave che identifica la risorsa V, corrisponde a un determinato GUID ed è il risultato di una funzione hash (che riceve in input un qualcosa come i metadati di V). Tipicamente anche i nodi sono identificati da un GUID.

Noi vogliamo che le chiavi siano equamente distribuite tra i peer e che, a seguito di un inserimento o abbandono di un nodo, non venga stravolto tutto in modo da dover ridistribuire da capo le chiavi tra i peer.

A tale scopo, si fa uso del **consistent hashing**, che è una speciale tecnica di hashing in cui:

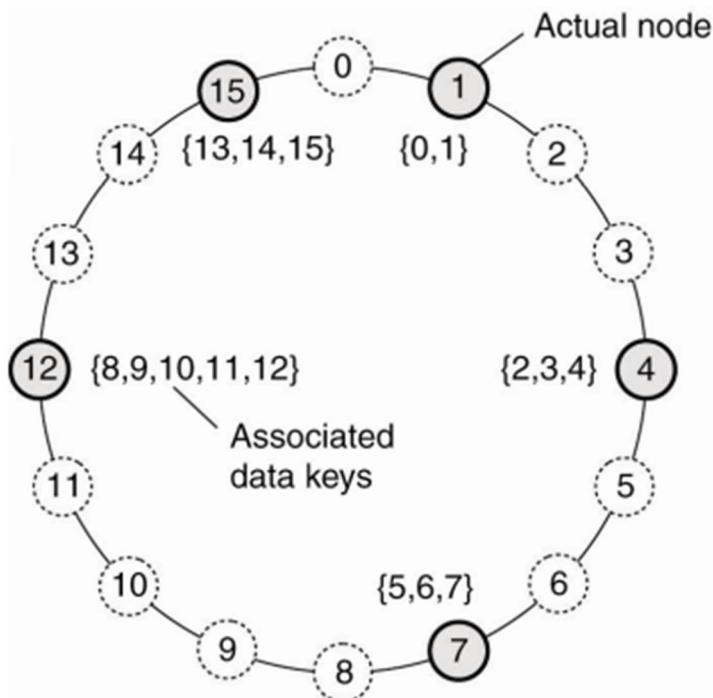
- Sia le risorse che i nodi (i bucket) sono mappati uniformemente sullo stesso spazio di indirizzamento mediante una funzione hash.

- Ciascun nodo gestisce un intervallo di chiavi hash consecutive, non un insieme di chiavi sparse.

Noi analizzeremo due protocolli per le overlay network strutturate: Chord e Pastry.

### Chord

I peer sono organizzati in una struttura ad anello. Ciascuno di essi è responsabile delle chiavi piazzate tra se stesso (incluso) e il nodo precedente (escluso): ciò implica che la risorsa con chiave K è gestita dal peer il cui identificatore è il più piccolo tra tutti quelli maggiori o uguali a K; tale peer è detto successore della chiave K ( $\text{succ}(K)$ ).



- Metrica di distanza tra i nodi = differenza lineare tra i GUID.

#### Operazione di get (routing):

Esistono più soluzioni possibili per effettuare il lookup di una risorsa:

1) Ogni nodo conosce solo il proprio successore: se è in cerca di una risorsa, la richiede a lui (al successore), il quale, eventualmente, inoltra la richiesta al proprio successore e così via. In tal modo, il costo di lookup è  $O(N)$  (piuttosto elevato).

2) Ogni nodo conosce tutti gli altri peer, il che consente di avere un costo di lookup di  $O(1)$ . D'altra parte, però, le operazioni di join e leave richiedono che il nodo che si aggiunge o abbandona la rete informi tutti gli altri.

3) Ciascun nodo conosce bene i nodi a lui vicini e la conoscenza decresce via via che ci si allontana.

In particolare, ogni nodo ha una finger table (FT) in cui sono registrati i peer che conosce. La FT ha m righe

(dove  $m$  = numero di bit del GUID).

If  $FT_p$  is the FT of node  $p$ , then  $FT_p[i] = \text{succ}(p + 2^{i-1}) \bmod 2^m$ ,  $1 \leq i \leq m$

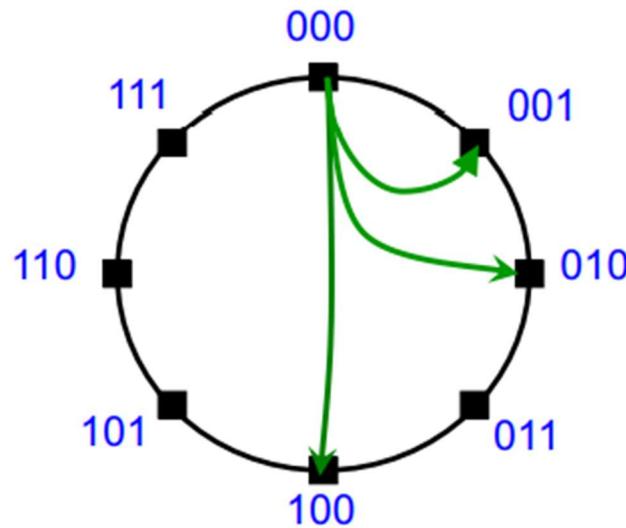
- $\text{succ}(p+1), \text{succ}(p+2), \text{succ}(p+4), \text{succ}(p+8), \text{succ}(p+16), \dots$

In the example  $m=3$

$$FT_0[1]=0+2^0=1$$

$$FT_0[2]=0+2^1=2$$

$$FT_0[3]=0+2^2=4$$



Supponiamo che il nodo  $p$  debba trovare il peer responsabile della chiave  $k$  (ovvero  $\text{succ}(k)$ ).

- Se  $k$  appartiene alla porzione dell'anello gestita da  $p$ , il lookup termina immediatamente.
- Se  $p < k \leq FT_p[1]$  allora  $p$  invia la richiesta al suo successore.
- Altrimenti  $p$  invia la richiesta al nodo  $q$  più lontano da lui tra quelli il cui GUID è minore o uguale a  $k$ :

$$FT_p[j] \leq k < FT_p[j+1]$$

dove  $FT_p[j] = q$ .

In tal modo, il costo di lookup risulta essere  $O(\log N)$ .

#### Operazione di join:

Con l'ipotesi per cui ogni nodo mantenga le informazioni anche sul proprio predecessore, quando un peer  $p$  si inserisce all'interno dell'overlay network, compie i seguenti passi:

- 1) Cerca il nodo  $\text{succ}(p+1)$  in modo da posizionarsi subito prima di lui.
- 2) Avverte il suo successore della sua presenza.
- 3) Inizializza la propria FT.
- 4) Avverte il suo predecessore della sua presenza (in modo da fargli aggiornare la FT).
- 5) Trasferisce su di sé le chiavi di sua competenza (prendendole dal suo successore).

#### Operazione di leave:

Quando un peer  $p$  lascia volontariamente l'overlay network, compie i seguenti passi:

- 1) Trasferisce le proprie chiavi sul successore.
- 2) Avverte il suo successore del suo abbandono (in modo da fargli aggiornare il puntatore "predecessore").
- 3) Avverte il suo predecessore del suo abbandono (in modo da fargli aggiornare la FT).

In realtà, a seguito di un'operazione di join o di leave da parte di un peer, dovrebbero essere aggiornate le FT di tutti i nodi della rete; quello che si fa è effettuare il refresh di tutte le FT periodicamente, in modo da tener conto anche degli abbandoni dovuti a guasti improvvisi.

Le operazioni di join e di leave hanno una complessità di  $O(\log^2 N)$ .

È rimasto ancora un punto in sospeso: quando un nodo crasha, non c'è più nessuno che gestisca le chiavi di cui quel peer era responsabile. Per ovviare a questo problema, si affida un certo numero R di copie delle chiavi ad altrettanti successori del peer; di conseguenza, ogni nodo deve conoscere R successori e non più soltanto uno.

#### Vantaggi di Chord:

- È un protocollo semplice ed elegante.
- Load balancing: le chiavi sono distribuite uniformemente tra i peer.
- Scalabilità: è efficiente nelle operazioni di lookup.
- Robustezza: prevede aggiornamenti periodici delle FT affinché siano consistenti coi cambiamenti che avvengono nell'overlay network.

#### Svantaggi di Chord:

- La vicinanza fisica tra i nodi non viene considerata.
- Le query per intervalli di chiavi sono del tutto inefficienti.

#### **Pastry**

Anche qui i peer sono organizzati in una struttura ad anello. Ciascuna chiave è rappresentata con d cifre, dove ogni cifra è composta da b bit (solitamente b=4).

Pastry è un protocollo basato sul **Plaxton routing**, secondo cui una risorsa V viene memorizzata dal peer il cui GUID ha il prefisso più lungo che matcha col GUID di V. Ciascun nodo p detiene una routing table che viene costruita secondo i seguenti criteri:

- > I GUID dei peer sull'n-esima riga condividono le prime n cifre col GUID del nodo p.
- > L'(n+1)-esima cifra dei GUID dei peer sull'n-esima riga è pari all'indice della colonna della routing table.
- > Esistono entry della routing table che possono essere associate a più di un nodo: ne viene selezionato uno in base a qualche metrica di prossimità (e.g. RTT).

NB: la numerazione delle righe e delle colonne della routing table parte da 0 e non da 1.

#### **Routing table of node 1220**

	0	1	2	3
0	0212	-	2311	3121
1	1031	1123	-	1312
2	1200	1211	-	1231
3	-	1221	1222	1223

$\lceil \log_2 N \rceil$  rows in the table

$2^b - 1$  items in each table row

In Pastry l'operazione di lookup ha una complessità di  $O(\log^b N)$ .

## **Architettura ibrida**

Un'architettura ibrida cerca di combinare i benefici delle architetture centralizzata e decentralizzata.

Due esempi di architetture ibride sono:

- 1) Sistema P2P ibrido con dei super-peer che hanno maggiori funzionalità degli altri peer; il routing avviene solo tra questi super-peer.
- 2) BitTorrent, dove un nuovo utente ottiene il file torrent contenente i puntatori ai tracker (che sono i server che conoscono i peer attivi che forniscono i chunk del file desiderato dall'utente); l'utente poi parla con un tracker che gli restituisce la lista dei peer che posseggono i chunk del file desiderato; infine, si unisce a uno swarm di peer da cui effettua il download dei chunk del file e, nel frattempo, mette a disposizione i chunk che ha già scaricato.

## **Definizioni di middleware**

- 1) Bernstein: il middleware è un insieme di strumenti situati tra le applicazioni e il supporto di basso livello (hardware, sistema operativo, ecc.).
- 2) Couloris & Dollimore: il middleware è uno strato software che fornisce un'astrazione di programmazione che maschera l'eterogeneità delle reti, dell'hardware, dei sistemi operativi e dei linguaggi di programmazione sottostanti.
- 3) Anthony: il middleware è uno strato virtuale tra le applicazioni e le piattaforme sottostanti che fornisce una trasparenza significativa.

## **Tipi di middleware**

- 1) RPC/RMI middleware: offre una funzionalità di comunicazione in remoto e fornisce un Interface Definition Language per definire il contratto di comunicazione. Alcuni esempi sono SUN RPC, Java RMI e gRPC.
- 2) Message-oriented middleware (MOM): fornisce un servizio di comunicazione persistente e asincrona tramite un sistema a code di messaggi. Alcuni esempi sono ActiveMQ, Kafka e RabbitMQ.
- 3) Component-based middleware: qui i componenti vengono eseguiti all'interno di container. Alcuni esempi sono .NET e Java EE.
- 4) Service-oriented middleware: è un middleware per le architetture basate su microservizi.

## **Middleware auto-adattativo**

Utilizzare un middleware che segue uno stile architetturale fisso è indice di poca flessibilità. È per questo che esistono i reflective middleware (che utilizzano il meccanismo di reflection per adattarsi ai cambiamenti del contesto ambientale) e i self-adaptive middleware. Noi ci concentreremo su questi ultimi.

I **sistemi software auto-adattativi** (o autonomici) sono sistemi in grado di adattare le proprie operazioni a run-time in base a sé stessi e all'ambiente (gli interventi umani devono essere assenti o fortemente limitati).

Gli obiettivi di un sistema auto-adattativo sono:

- **Self-optimizing** = capacità di un sistema di ottimizzare le sue performance o l'uso delle sue risorse.
- **Self-healing** = capacità di un sistema di diagnosticare e adottare misure correttive a problemi o guasti.
- **Self-protecting** = capacità di un sistema di mettere in atto strategie di difesa contro possibili attacchi anche in modo pro-attivo.
- **Self-configuring** = capacità del sistema di integrare nuovi elementi senza alterare il suo corrente funzionamento e di impostare determinati parametri di configurazione.

Come si raggiungono questi 4 obiettivi?

-> Il sistema deve essere consapevole del suo stato interno (**self-awareness**) e dell'ambiente in cui si trova

**(self-situation)**, dove:

Ambiente = software non controllabile dal sistema + hardware + rete + contesto fisico + utenti.

-> Il sistema deve poter identificare dei cambiamenti (**self-monitoring**) e adattarsi di conseguenza (**self-adjustment**).

## MAPE

È un'architettura di riferimento per i sistemi auto-adattativi che consiste di 4 fasi: Monitor, Analyze, Plan ed Execute; talvolta, viene considerata anche una quinta componente, Knowledge, per cui l'architettura viene spesso denotata con MAPE-K.

- **Monitor**: riceve dei dati in ingresso, li filtra e li correla per la fase successiva.
- **Analyze**: osserva i risultati del monitoraggio e determina se è necessario attuare delle operazioni di adattamento; se sì, dà il via alla fase successiva.
- **Plan**: pianifica le azioni di adattamento da mettere in atto.
- **Execute**: esegue ciò che è stato stabilito nella fase precedente.
- **Knowledge**: i dati usati dal sistema autonomo vengono memorizzati come conoscenza condivisa tra le 4 fasi appena descritte.

Vediamo ora come avviene la progettazione di alcune di queste fasi.

### Monitor:

- 1) Il monitoraggio deve essere effettuato periodicamente o on-demand?
- 2) Cosa deve essere monitorato (risorse, carico di lavoro, prestazioni)?
- 3) L'architettura del monitoraggio deve essere centralizzata o decentralizzata?
- 4) Il monitoraggio deve essere passivo (mera osservazione) o attivo (scatenando le operazioni da monitorare)?
- 5) Dove e come devono essere memorizzati i dati monitorati?

### Analisi:

- 1) L'analisi deve avvenire periodicamente o a seguito di un determinato evento?
- 2) L'analisi deve essere reattiva o proattiva?

### Plan:

Esiste una gran varietà di metodologie che possono essere adottate nella fase di plan:

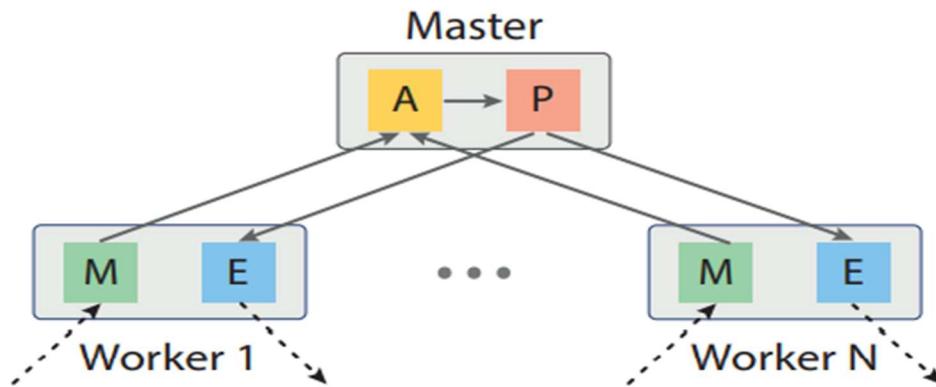
- 1) Formulazione di problemi di ottimizzazione (spesso NP-hard).
- 2) Utilizzo di euristiche che portano a una soluzione sub-ottima ma in modo efficiente.
- 3) Utilizzo di machine learning e reinforcement learning.
- 4) Utilizzo della teoria del controllo.
- 5) Utilizzo della teoria delle code.

E così via.

Il sistema MAPE può essere **centralizzato** (tutte le componenti sono nello stesso nodo) oppure **distribuito**. In particolare, un'architettura distribuita può essere a sua volta **gerarchica** oppure **flat**.

## Pattern MAPE gerarchici

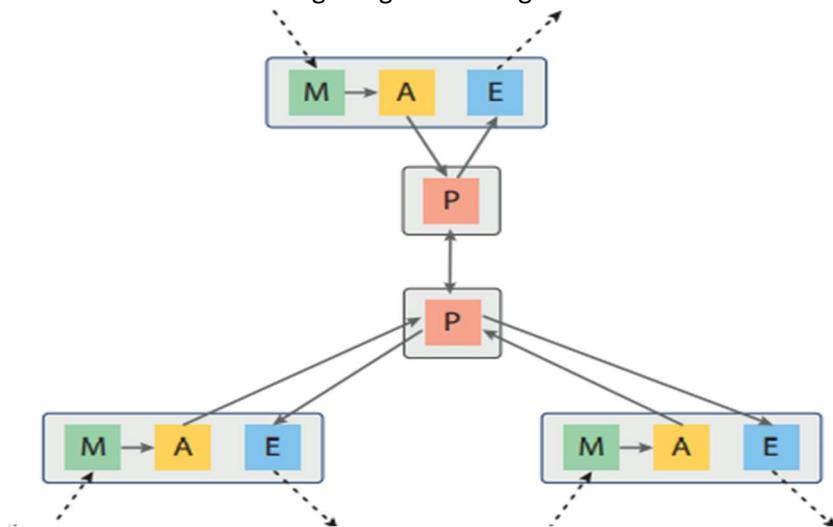
- 1) Master-worker pattern: decentralizza Monitor ed Execute su molteplici nodi worker e mantiene Analyze e Plan centralizzate su un unico nodo master.
  - PRO: il master sa tutto di tutti, per cui può prendere facilmente delle decisioni globali.
  - CONTRO: c'è molto overhead di comunicazione dato che il master comunica con tutti i worker; inoltre, il master potrebbe rappresentare un collo di bottiglia e un single point of failure.



2) Regional pattern: è composto da molteplici regioni debolmente accoppiate, ciascuna delle quali ha una struttura gerarchica a due livelli: il livello superiore comprende Plan, mentre quello inferiore comprende Monitor, Analyze ed Execute.

**PRO:** offre una maggiore flessibilità (le regioni possono stare sotto amministrazioni differenti).

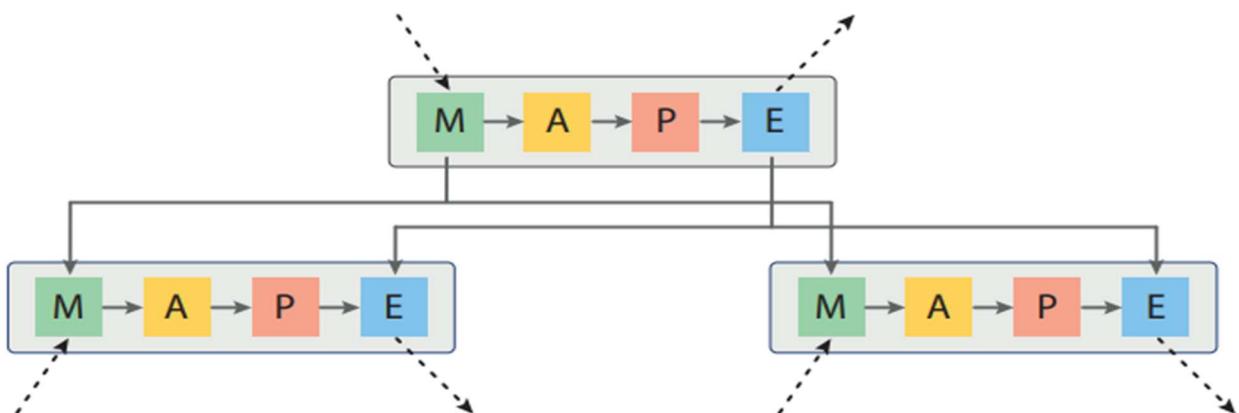
**CONTRO:** è difficile conseguire gli obiettivi globali.



3) Hierarchical control pattern: consiste in tanti nodi che comunicano tra loro (e hanno una visione locale dei dati) e un nodo su un livello superiore che ha una visione più globale. In tutti i nodi vengono effettuate tutte e 4 le fasi.

**PRO:** il nodo di livello superiore può conseguire facilmente gli obiettivi globali.

**CONTRO:** potrebbe essere farraginoso decidere cosa controllare a livello locale e cosa a livello globale.

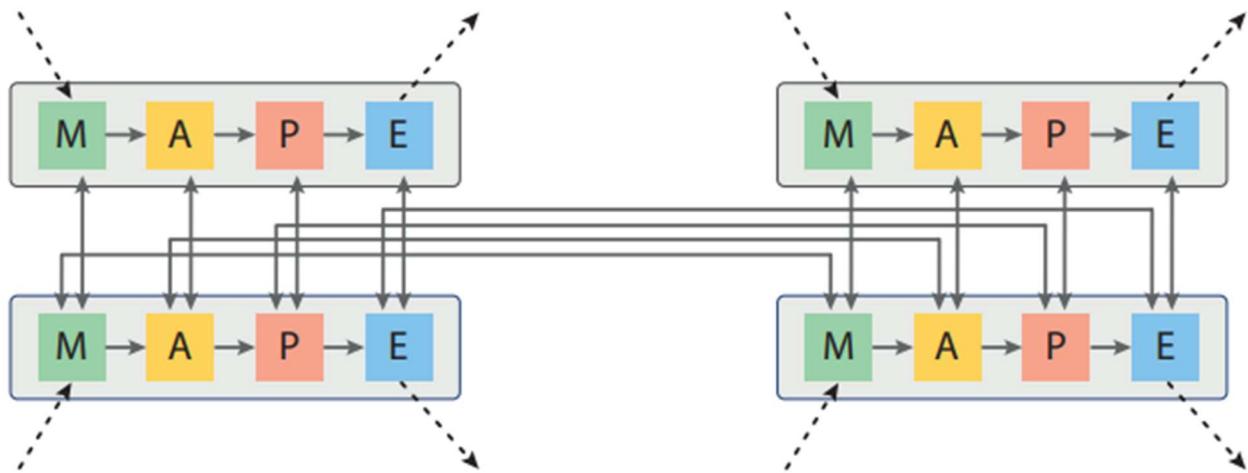


### Pattern MAPE flat

1) Coordinated control pattern: prevede molteplici cicli MAPE situati su altrettanti nodi; ciascuna fase di ogni nodo comunica con la corrispondente fase di tutti gli altri nodi.

PRO: è scalabile.

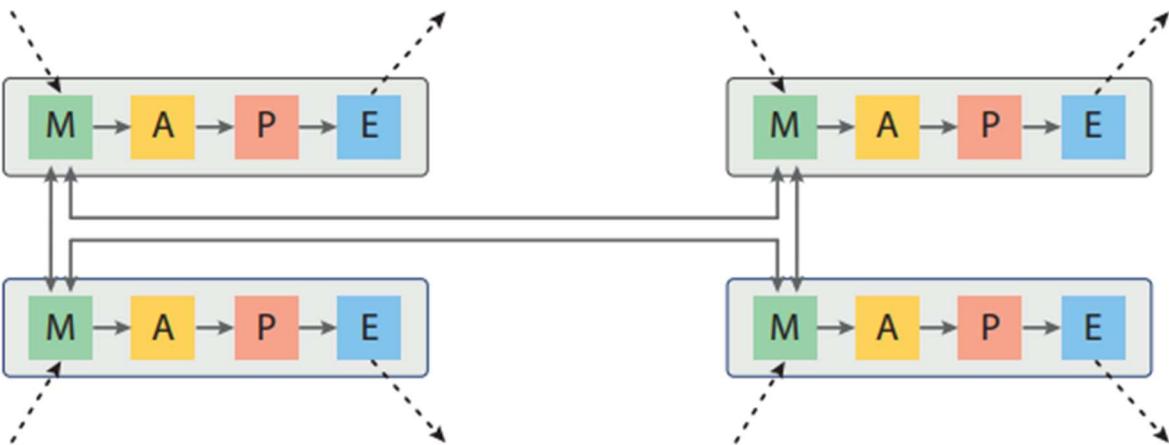
CONTRO: c'è moltissimo overhead di comunicazione.



2) Information sharing pattern: è un caso particolare del coordinated control pattern, in cui solo i componenti Monitor comunicano tra loro.

PRO: è scalabile.

CONTRO: c'è poca coordinazione a causa della mancanza di comunicazione nella fase Plan.



# COMUNICAZIONE NEI SISTEMI DISTRIBUITI

## Tipi di comunicazione

Ricordandoci che i servizi per la comunicazione nei sistemi distribuiti vengono forniti dal middleware, distinguiamo i tipi di comunicazione rispetto a tre assi:

- Persistenza (comunicazione persistente vs transiente).
- Sincronizzazione (comunicazione sincrona vs asincrona).
- Dipendenza dal tempo (comunicazione discreta vs streaming).

### Comunicazione persistente:

Il messaggio viene memorizzato dal middleware finché non viene ricevuto dal destinatario. Ciò consente di avere disaccoppiamento temporale.

### Comunicazione transiente:

Il messaggio viene memorizzato dal middleware solo finché sia il mittente sia il destinatario sono in esecuzione; se la consegna non è possibile, il messaggio viene scartato. Ciò proibisce di avere disaccoppiamento temporale.

### Comunicazione sincrona:

Le operazioni di invio e di ricezione di un messaggio sono bloccanti. Per quanto riguarda il mittente, esistono tre alternative per stabilire fino a quando viene bloccato:

- 1) Finché il middleware destinatario non prende in carico il messaggio.
- 2) Finché il destinatario non riceve il messaggio a livello applicativo.
- 3) Finché il messaggio non viene processato completamente dal destinatario.

### Comunicazione asincrona:

L'operazione di invio di un messaggio è non bloccante, mentre l'operazione di ricezione può essere sia bloccante che non bloccante.

### Comunicazione discreta:

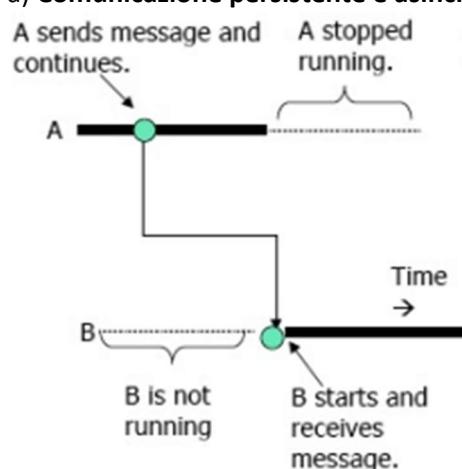
Ciascun messaggio costituisce un'unità completa di informazione.

### Comunicazione basata su streaming:

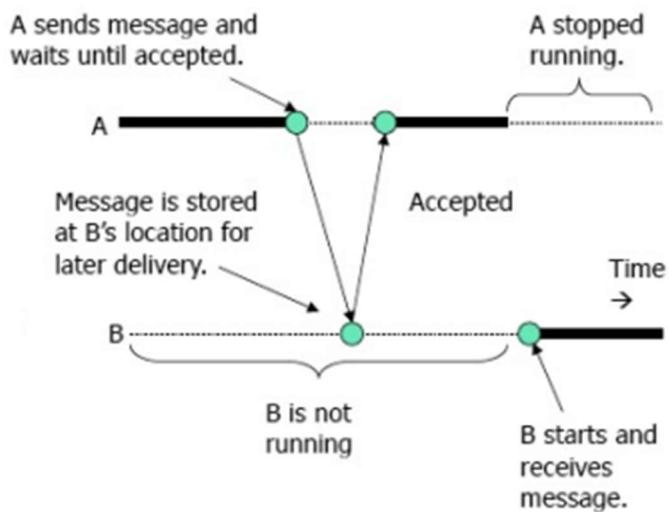
I messaggi hanno una relazione temporale, sono caratterizzati da un ordine di invio e costituiscono uno stream.

Combinando persistenza e sincronia, possiamo ottenere i seguenti scenari di comunicazione:

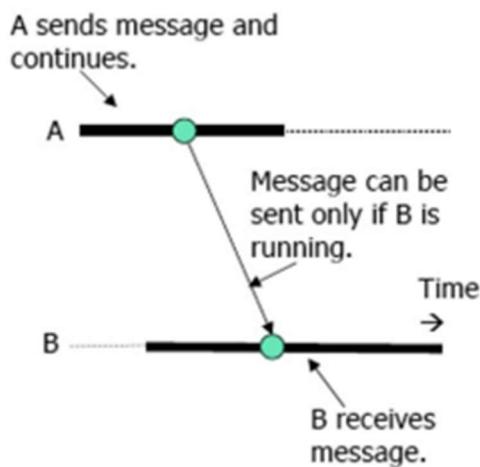
- a) **Comunicazione persistente e asincrona:** è tipica delle e-mail.



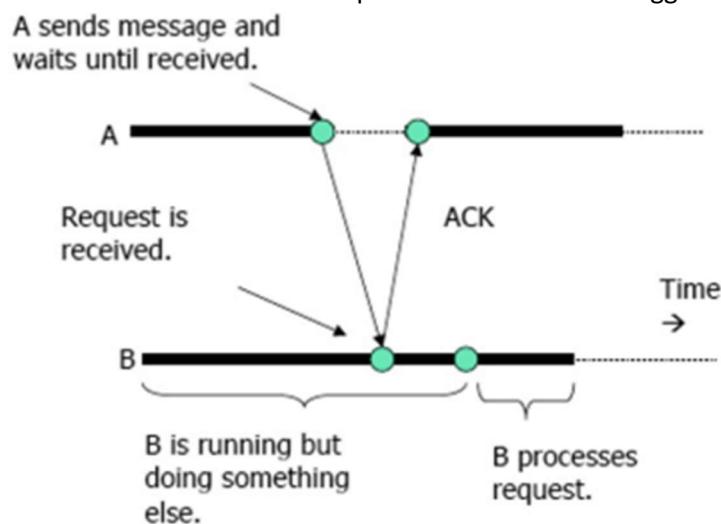
- b) **Comunicazione persistente e sincrona:** il mittente rimane bloccato finché non è sicuro che il messaggio sia stato memorizzato nel middleware lato destinatario.



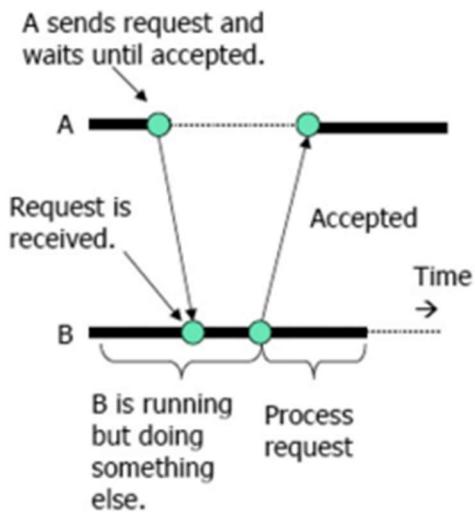
c) **Comunicazione transiente e asincrona:** il mittente non rimane mai in stato di blocco ma il messaggio può andare perso se il destinatario è irraggiungibile.



d) **Comunicazione transiente e “receipt-based synchronous”:** il mittente rimane in stato di blocco finché il middleware destinatario non prende in carico il messaggio.

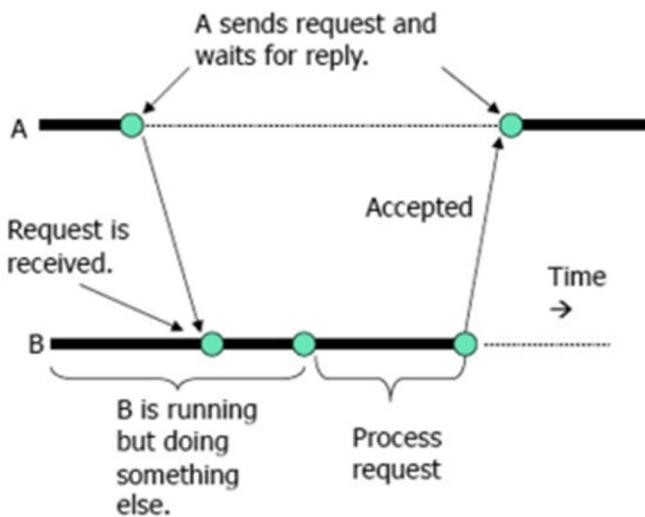


e) Comunicazione transiente e “**delivery-based synchronous**”: il mittente rimane in stato di blocco finché il destinatario non riceve il messaggio a livello applicativo.



#### (e) Delivery-based synchronous communication

f) Comunicazione transiente e “**response-based synchronous**”: il mittente rimane in stato di blocco finché il messaggio non viene processato completamente dal destinatario (e quest’ultimo non invia un messaggio di risposta).



#### (f) Response-based synchronous communication

### Semantica degli errori di comunicazione

Durante la comunicazione tra client e server possono verificarsi diversi errori:

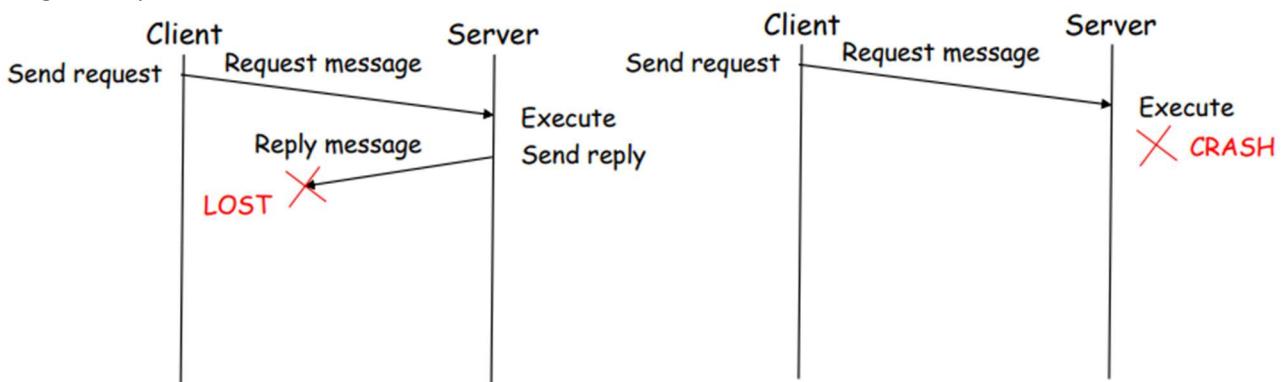
- > Un messaggio può essere perso o ritardato, o la connessione può essere resettata.
- > Il server può crashare (sia prima che dopo aver processato il servizio richiesto).
- > Il client può crashare.

È possibile gestire tali situazioni tramite le cosiddette semantiche di errore, che dipendono dalla combinazione dei seguenti tre meccanismi di base:

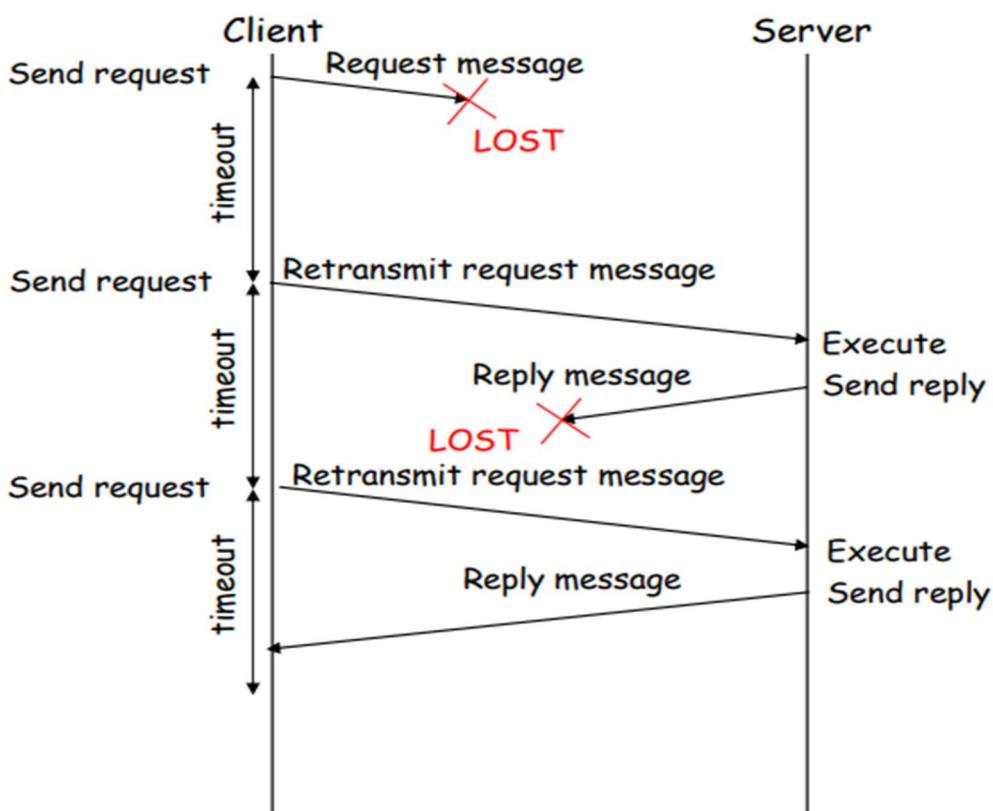
- **Request Retry (RR1)**: il client riprova a inviare il messaggio di richiesta finché non riceve una risposta oppure finché non ha effettuato un numero sufficiente di tentativi per cui può assumere che il server sia andato in crash.
- **Duplicate Filtering (DF)**: il server scarta i messaggi di richiesta duplicati.
- **Result Retransmit (RR2)**: il server memorizza la risposta per ciascun messaggio di richiesta in modo da ritrasmetterla senza calcolarla nuovamente nel caso in cui ricevesse un messaggio di richiesta duplicato.

Analizziamo ora le 4 possibili semantiche di errore.

- 1) Semantica May-be: il client invia il messaggio di richiesta una sola volta e non è sicuro se il server ha eseguito l'operazione o meno. Non viene utilizzato alcun meccanismo tra RR1, DF e RR2.

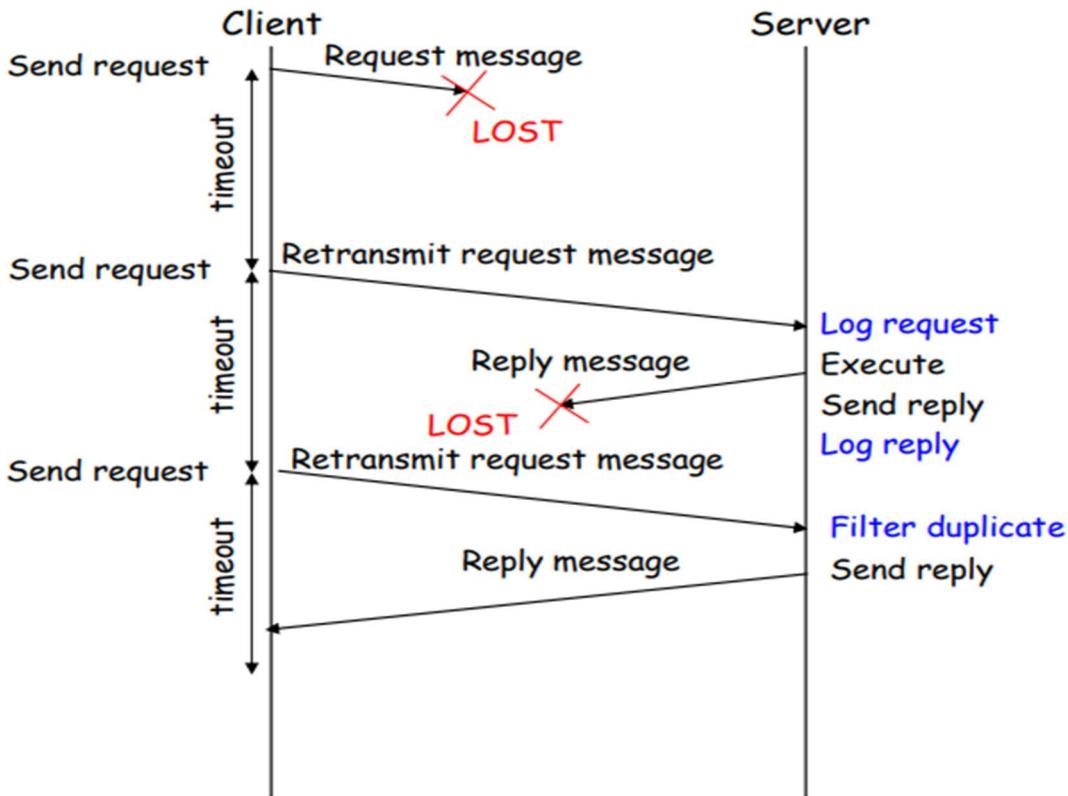


- 2) Semantica At-least-once: se il server è funzionante, processa ciascuna richiesta **almeno** una volta. Di fatto, il client mette in atto il meccanismo RR1, mentre il server non utilizza alcun meccanismo. Si tratta di una semantica compatibile solo coi servizi idempotenti: per un servizio non idempotente, invece, il server rischia di inviare risposte diverse per duplicati del medesimo messaggio di richiesta.



3) Semantica At-most-once: se il server è funzionante, processa ciascuna richiesta **al più una volta**; tuttavia, in caso di fallimento, il client e il server non hanno alcuna informazione sullo stato della controparte (e.g. il client non ha idea se il server ha processato la richiesta). I meccanismi di base RR1, DF e RR2 vengono utilizzati tutti e tre. Si tratta di una semantica compatibile sia coi servizi idempotenti sia con quelli non idempotenti.

Ma come fa il server a riconoscere le richieste duplicate? Basta che il client includa nei messaggi un ID univoco da riutilizzare quando deve inviare una richiesta duplicata.



4) Semantica Exactly-once: se il server è funzionante, processa ciascuna richiesta **esattamente** una volta; ciò vuol dire che, in ogni caso, sia il client che il server conoscono lo stato della controparte. Di conseguenza, i tre meccanismi di base (RR1, DF, RR2) non sono sufficienti: sono richiesti dei meccanismi addizionali per far fronte ai guasti lato server:

- Transparent server replication.
- Write-ahead logging (WAL).
- Recovery.

Per quanto riguarda il WAL, è un pattern che prevede che i cambiamenti di stato del server, prima di manifestarsi, debbano essere scritti nel log.

Il log è un file che si trova su un dispositivo di memoria persistente e viene scritto tramite un'operazione di append (nel nostro caso specifico, viene scritto un gruppo di entry per volta). Nel caso in cui il server subisce un crash, è possibile recuperare il suo stato originale riproducendo le operazioni scritte nel log (recovery).

### RPC (Remote Procedure Call)

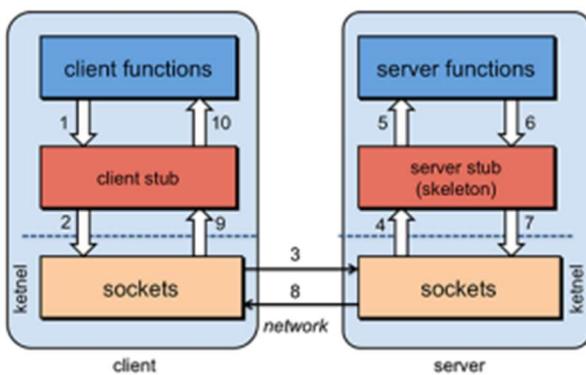
Idea: usare il modello client / server per chiamare procedure in esecuzione su macchine remote, in modo tale che lo scambio dei messaggi venga gestito dal middleware, in modo trasparente al programmatore (il quale deve programmare come se stesse implementando una chiamata a procedura locale, senza ulteriori

complicazioni).

Questa illusione si ottiene aggiungendo due entità intermedie dette **stub** (una lato client e una lato server) che si occupano dei dettagli implementativi relativi allo scambio dei messaggi.

#### Step di base di RPC:

- 1) Il client invoca una procedura locale chiamata **client stub**.
- 2) Il client stub impacchetta il messaggio di richiesta mediante il **marshaling** dei parametri (dove gli argomenti vengono convertiti dal formato locale al formato comune) e chiama il sistema operativo locale.
- 3) Il sistema operativo del client invia il messaggio di richiesta al sistema operativo del server.
- 4) Il sistema operativo del server passa il messaggio di richiesta al **server stub**.
- 5) Il server stub spacchetta il messaggio di richiesta mediante l'**unmarshaling** dei parametri (dove gli argomenti vengono convertiti dal formato comune al formato locale) e invoca la procedura.
- 6) Il server esegue la procedura e restituisce i valori di ritorno al server stub.
- 7) Il server stub impacchetta il messaggio di risposta mediante il marshaling dei valori di ritorno e chiama il sistema operativo.
- 8) Il sistema operativo del server invia il messaggio di risposta al sistema operativo del client.
- 9) Il sistema operativo del client passa il messaggio di risposta al client stub.
- 10) Il client stub spacchetta il messaggio di risposta mediante l'unmarshaling dei valori di ritorno e restituisce il risultato al client.



#### **Problemi legati a RPC**

- 1) Eterogeneità nella rappresentazione dei dati: client e server possono utilizzare rappresentazioni di dati differenti; per gestire questo, esistono quattro soluzioni alternative:
  - > Specificare la codifica nel messaggio.
  - > Fare in modo che il mittente converte i dati nella codifica del destinatario: la conversione è veloce (solo il mittente deve effettuarla) ma ciascun componente deve conoscere le codifiche di tutti gli altri componenti.
  - > Fare in modo che il mittente converte i dati nella codifica standard: la conversione è lenta (sia il mittente che il destinatario devono effettuarla) ma ciascun componente deve conoscere solo la codifica standard.
  - Questo è il meccanismo adottato in RPC e viene realizzato mediante l'utilizzo di **proxy** (che in RPC sono gli stub).
  - > Introdurre un intermediario (detto **broker**) che si occupi della conversione dei dati.
- 2) Passaggio dei parametri: può avvenire secondo una delle seguenti tre modalità:
  - **Call by value**: il valore dei parametri viene copiato sullo stack della funzione chiamata. Eventuali modifiche sul valore dei parametri non si riflettono nel chiamante.
  - **Call by reference**: il riferimento (l'indirizzo) ai parametri viene copiato sullo stack della funzione chiamata, la quale agisce direttamente sui dati del chiamante.
  - **Call by copy-restore**: il chiamante passa alla funzione chiamata una copia dei parametri. Quando la funzione chiamata restituisce il controllo al chiamante, quest'ultimo riporta le eventuali modifiche dei

parametri sui dati originali. Tale meccanismo simula il passaggio dei parametri per riferimento (call by reference).

3) Semantica di errore: RPC utilizza una semantica **At-least-once** oppure **At-most-once**.

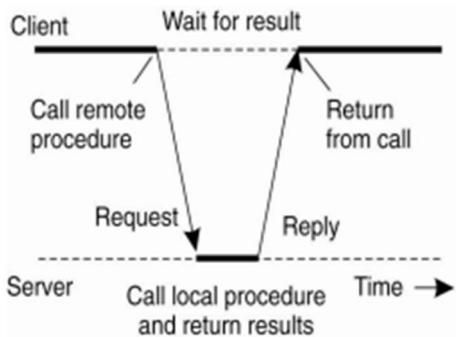
4) Server binding: è un problema in cui ci si chiede come deve fare il client per localizzare il server endpoint e la procedura remota da invocare su di esso. Di base, il binding può essere **statico** (l'indirizzo del server e altre informazioni sono incorporate nel codice, il che non permette trasparenza e flessibilità) oppure **dinamico** (viene effettuato a run-time, il che porta a un maggiore overhead).

Il binding dinamico si articola in due fasi:

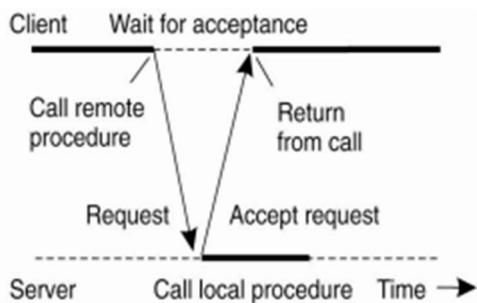
-> **Naming**: è una fase statica, che avviene prima dell'esecuzione. Qui il client specifica quali sono il servizio e il server a cui vuole far riferimento.

-> **Addressing**: è una fase dinamica, che avviene durante l'esecuzione. Qui si ha il binding vero e proprio tra client e server. L'addressing può essere **esplicito** (il client invia una richiesta broadcast o multicast e si connette al primo server che gli risponde) oppure **implicito** (c'è un **name server** che registra i servizi offerti dai server, gestisce una binding table e si occupa dei collegamenti tra i client e i server).

5) Sincronia: RPC può essere sia sincrono che asincrono.



**Synchronous RPC**



**Asynchronous RPC**

6) Trasparenza: RPC in realtà non è davvero trasparente rispetto all'accesso e alla locazione. Infatti, le chiamate a procedura remota su LAN richiedono da 0,1 a 1 ms, per cui sono circa 100.000 volte più lente delle chiamate locali che richiedono 3 ns (per quanto riguarda le reti WAN le cose peggiorano ulteriormente). Inoltre, possono verificarsi dei fallimenti che non si presentano mai in locale; ad esempio:

- Il client potrebbe non riuscire a localizzare il server.
- Il messaggio di richiesta o di risposta potrebbe andare perso.
- Il server e il client potrebbero crashare.

- 7) **Sicurezza**: poiché abbiamo a che fare con una comunicazione che avviene in rete, dobbiamo fare i conti con delle problematiche relative alla sicurezza: In particolare, potrebbe essere necessario:
- > Autenticare il server e/o il client.
  - > Cifrare i messaggi, che altrimenti sarebbero visibili in rete.
  - > Autenticare i messaggi, che altrimenti potrebbero essere modificati.
  - > Far fronte ai replay attack.

### **IDL (Interface Definition Language)**

È un linguaggio che consente al programmatore di descrivere un'interfaccia per la chiamata a procedura remota. In particolare, risolve sia il problema per cui alcuni linguaggi (come C e C++) non concepiscono le chiamate a procedure remote, sia il problema per cui client e server possono girare su ambienti di esecuzione differenti (e.g. client Java che comunica con un servizio Python).  
Esiste inoltre il compilatore IDL (detto **precompilatore**) che genera automaticamente il client stub e il server stub.

### **SunRPC**

Detto anche Open Network Computing (ONC), è un esempio di RPC di prima generazione ed è una suite di prodotti tra cui:

- eXternal Data Representation (XDR) come IDL.
- Remote Procedure Call GENerator (RPCGEN) come compilatore IDL.
- Port mapper come name server.
- Network File System (NFS) come file system distribuito.

#### Step di base per programmare con SunRPC:

1) Definire l'interfaccia RPC definendo un file con formato .x che è scritto col linguaggio XDR ed è suddiviso in tre porzioni:

- Definizione delle costanti e dei parametri di input e output delle procedure.
- Definizione delle procedure stesse con i relativi identificatore (= numero di procedura), numero di versione e numero di programma (tutti questi valori, che sono identificatori, vanno scritti interamente in **UPPERCASE**).
- Definizione di eventuali tipi di dato non predefiniti in XDR.

NB: XDR è un formato binario che usa **implicit typing**, per cui vengono trasmessi soltanto i valori, non i tipi di dato.

2) Compilare il file *nomeProgramma.x* con RPCGEN, che genera:

- > Il client stub (*nomeProgramma\_clnt.c*).
- > Il server stub (*nomeProgramma\_svc.c*).
- > Le routine XDR per convertire i dati dal formato locale al formato comune XDR (*nomeProgramma\_xdr.c*).
- > Il file header che contiene le strutture dati (i tipi di dato) necessarie (*nomeProgramma.h*).

3) Scrivere il file *server.c*, che implementa le procedure remote fornite dal server RPC. Osserviamo che:

- Non c'è alcuna funzione main: piuttosto si trova nel server stub (di fatto è lui a invocare in ultima istanza le procedure).

- Ciascuna procedura ha esattamente due parametri di input e un parametro di output; tutti e tre questi parametri sono dei puntatori. In particolare, il secondo parametro di input è un puntatore al client transport manager, che è utilizzato dal client stub per gestire la comunicazione col server. Inoltre, il parametro di output dev'essere un puntatore a una variabile *static*, cosicché l'area di memoria puntata continui a esistere dopo la terminazione della procedura.

- Il nome delle procedure deve essere del tipo *nomeProcedura\_1\_svc*, dove 1 (o qualunque altro intero positivo) indica il numero di versione.

4) Scrivere il file *client.c*, che implementa il main e la logica necessaria per trovare e invocare la procedura remota desiderata. Osserviamo che:

- Preliminarmente il client deve invocare la seguente funzione:

*clnt\_create (host, NUM\_PROG, NUM\_VERS, PROTOCOL)*      dove PROTOCOL = "tcp" o "udp".

In particolare, *host* è l'hostname del server da contattare.

- In base anche al punto precedente, il client deve conoscere l'hostname del server, il numero di programma, il numero di versione e il nome della procedura remota.

- Il client deve gestire eventuali errori che possono verificarsi durante la chiamata remota, e lo fa tramite le chiamate a *clnt\_pcreateerror()* (in caso di errore nella creazione del client transport manager) e a *clnt\_perror()* (in caso di errore di comunicazione col server).

5) Compilare tutti i file sorgente (client, server, client stub, server stub e routine di conversione) e linkare i file oggetto.

6) Quando il server viene avviato, pubblica il servizio (ovvero registra le procedure remote col name server, detto anche port mapper o RPCBIND).

7) Quando il client viene avviato, trova l'endpoint del servizio (il server) tramite il port mapper.

#### SunRPC features:

-> Un programma tipicamente contiene diverse procedure remote e, per ognuna di esse, esistono molteplici versioni.

-> Il server, di default, è sequenziale: può essere eseguita una sola call per volta, per cui viene garantita la mutua esclusione.

-> Il client, quando contatta il server, entra in stato di blocco in attesa di risposta (comunicazione sincrona).

-> La semantica di errore adottata è At-least-once.

-> La comunicazione tra client e server è transiente (non persistente).

#### Server binding:

Prima della comunicazione, il server RPC registra i programmi RPC nella **port map**, che è una tabella contenente i servizi RPC offerti in quella macchina e gestita da un singolo processo (il port mapper).

Ciascuna entry della port map contiene:

- Il numero di programma.

- Il numero di versione.

- Il protocollo di livello di trasporto (TCP o UDP).

- Il numero di porta su cui contattare il server.

Il numero di procedura non viene specificato, poiché tutte le procedure all'interno di un programma condividono il medesimo transport manager.

Il port mapper è in ascolto sulla porta 111 del server e supporta le seguenti operazioni:

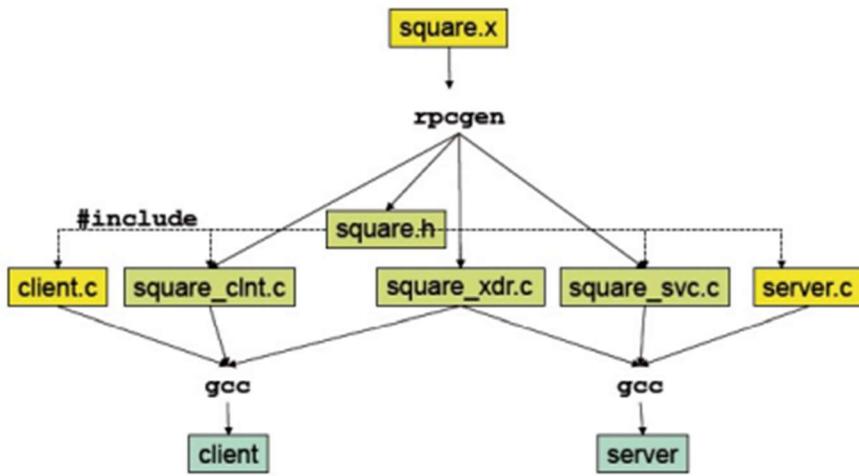
-> Inserimento di un servizio.

-> Eliminazione di un servizio.

-> Ricerca del numero di porta associato a un determinato servizio.

-> Listing dei servizi registrati.

### Development process:



**Lato server:** nella funzione main, il server stub crea una socket e vi collega qualunque porta disponibile. Dopodiché, invoca *svc\_register()*, una funzione della libreria RPC che serve a registrare le procedure col port mapper. Infine, attende le richieste chiamando *svc\_run()*, un'altra funzione della libreria RPC che ha il compito di invocare le procedure in risposta delle richieste provenienti dai client.

**Lato client:** quando viene avviato il programma del client, la funzione *clnt\_create()* contatta il port mapper lato server per trovare il numero di porta per l'interfaccia usata. Questa operazione è detta early binding e viene eseguita una sola volta (non a ogni chiamata a procedura). Dopodiché, sarà il client stub a gestire la comunicazione col server: è lui a implementare il timeout per le richieste e le operazioni di marshaling e unmarshaling.

### Tipi di trasparenza (non) offerti da SunRPC:

- SunRPC è trasparente all'accesso solo parzialmente: è vero che determinati meccanismi per l'invocazione a procedura remota (istanziazione degli stub, marshaling, unmarshaling e così via) non sono oggetto di preoccupazione per il programmatore, ma nel codice ci sono comunque delle differenze rispetto alle chiamate locali: c'è una convenzione sui nomi delle procedure ben precisa, ci sono dei vincoli sul numero dei parametri delle procedure, è necessario gestire molti più errori e il client deve conoscere l'host name del server.
- SunRPC non è trasparente alla concorrenza: abbiamo infatti osservato che il server lavora in maniera sequenziale.
- SunRPC non è trasparente alla replicazione: il port mapper è locale al server che offre il servizio, per cui dovrebbe essere replicato nel caso in cui si abbiano più server.

### **Java RMI (Remote Method Invocation)**

È un esempio di RPC di seconda generazione ed è un insieme di tool, politiche e meccanismi che consentono a un'applicazione Java di invocare i metodi di un oggetto remoto (istanziato su un host differente, tipicamente un server), rendendo possibile la comunicazione tra JVM diverse.

Benché l'oggetto utilizzato sia in esecuzione su un server remoto, il client invoca i metodi remoti dell'oggetto mediante un riferimento locale. Ciò richiede una separazione prima di tutto logica tra l'interfaccia e l'implementazione dell'oggetto (ovvero la classe).

In particolare, nel momento in cui il client si connette con l'oggetto nel server remoto, all'interno del client space viene caricato lo **stub**, che è la copia dell'interfaccia server e svolge un ruolo analogo al client stub di RPC. D'altra parte, le richieste che giungono all'oggetto nel server remoto vengono gestite dallo **skeleton**,

che è un client agent ubicato nel server e svolge un ruolo analogo al server stub di RPC. Lo stub e lo skeleton vengono generati automaticamente.

#### Serializzazione e deserializzazione:

Poiché, grazie al bytecode e alla portabilità del codice Java, gli ambienti di esecuzione di client e server sono simili tra loro, con Java RMI non è necessario eseguire le operazioni di marshaling e unmarshaling dei dati, bensì è sufficiente ricorrere alla serializzazione e deserializzazione, che sono operazioni direttamente supportate da Java.

- **Serializzazione:** trasforma un oggetto in una sequenza di byte; per realizzarla, bisogna invocare il metodo *writeObject* su uno stream di output.

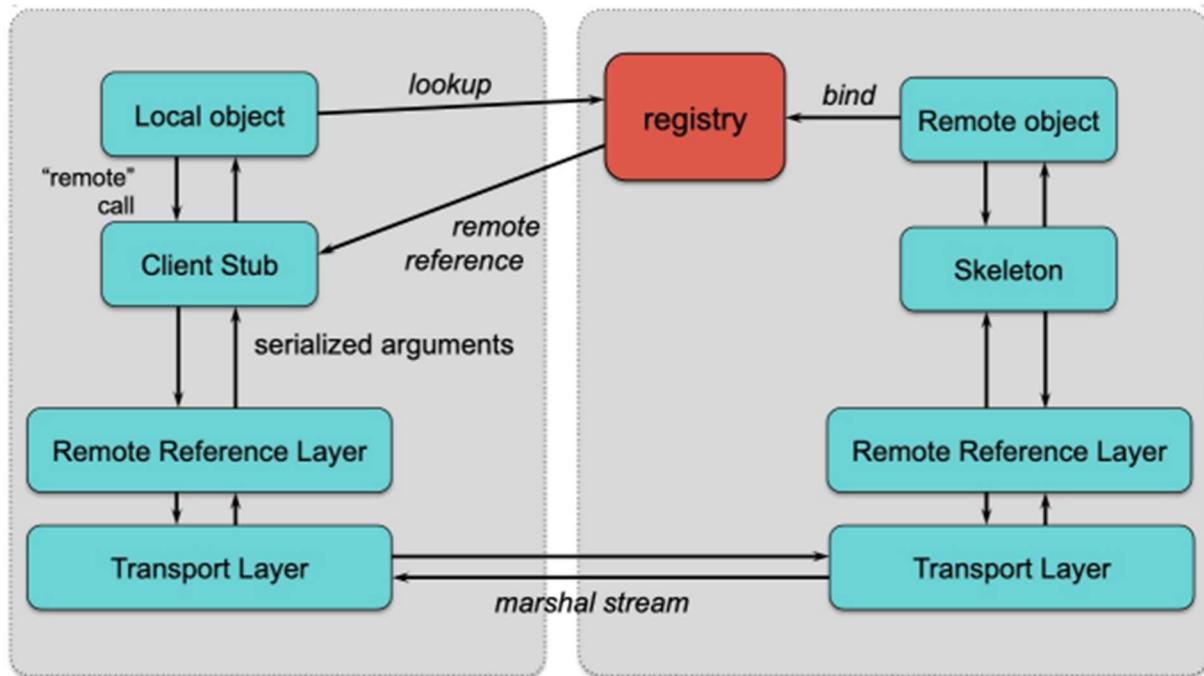
- **Deserializzazione:** decodifica una sequenza di byte e ricostruisce il relativo oggetto; per realizzarla, bisogna invocare il metodo *readObject* da uno stream di input.

Possiamo notare che la serializzazione è solo uno dei passaggi del marshaling.

#### Interazione tra stub e skeleton:

- 1) Il client ottiene un'istanza dello stub.
- 2) Il client invoca i metodi sullo stub.
- 3) Lo stub serializza le informazioni necessarie per l'invocazione (ID del metodo e parametri) e le invia allo skeleton in un messaggio.
- 4) Lo skeleton riceve e deserializza il messaggio, invoca il metodo richiesto, serializza il valore di ritorno e lo invia allo stub in un messaggio.
- 5) Lo stub deserializza il valore di ritorno e lo restituisce al client.

In realtà, sotto a stub e skeleton esistono altri due layer intermedi:



- **Remote Reference Layer:** gestisce i riferimenti remoti e le connessioni stream-oriented.

- **Transport Layer:** gestisce le connessioni tra JVM differenti e può utilizzare diversi protocolli di livello di trasporto, purché siano connection-oriented (come TCP).

Dalla figura notiamo anche la presenza dell'**RMI registry**, che è l'equivalente del port mapper di SunRPC: permette al server di pubblicare i servizi e consente al client di ottenere una copia dello stub necessaria a invocare i metodi remoti. Per motivi di sicurezza, può essere eseguito solo sulla stessa macchina del server, per cui viene meno la trasparenza alla locazione.

## Programmazione con Java RMI:

### **Server-side:**

- 1) Definire un'interfaccia pubblica che estenda `java.rmi.Remote` cosicché i suoi metodi possano essere invocati da altre JVM. Ciascun metodo remoto:
  - > Deve dichiarare di essere in grado di lanciare l'eccezione `java.rmi.RemoteException` per indicare l'occorrenza di un errore di comunicazione o di protocollo.
  - > Accetta un qualunque numero di parametri in input ma può restituire solo un risultato.
  - > Ha una signature indistinguibile dai metodi locali (ad esempio il suo nome non deve seguire un pattern ben preciso, come invece accade con SunRPC).
- 2) Definire la classe che implementi l'interfaccia ed estenda `java.rmi.UnicastRemoteObject`; il termine **unicast** indica che l'oggetto non può essere duplicato. Ciascuna classe dovrebbe:
  - > Dichiare l'interfaccia che sta implementando.
  - > Definire un costruttore.
  - > Fornire un'implementazione per ciascun metodo dichiarato nell'interfaccia.
- 3) Definire il codice del server, in cui:
  - > Venga creata un'istanza dell'oggetto.
  - > Venga registrato l'oggetto tramite l'RMI registry, utilizzando il metodo `bind()` o `rebind()` della classe `java.rmi.Naming` (`rebind()` rimpiazza un'associazione già esistente).Da qui si può notare come la registrazione dei servizi, a differenza di SunRPC, sia a cura non dello stub ma dello sviluppatore.

### **Client-side:**

- 1) Ottenere il riferimento all'oggetto remoto invocando sull'RMI registry il metodo `lookup()` della classe `java.rmi.Naming`.
- 2) Assegnare il riferimento appena ottenuto a una variabile che ha l'interfaccia remota come tipo.
- 3) Invocare i metodi remoti desiderati utilizzando la variabile di cui sopra.

### Java RMI features:

- > Il passaggio dei parametri avviene per valore per i tipi di dato primitivi e gli oggetti serializzabili, mentre avviene per riferimento per gli oggetti remoti.
- > Java RMI offre supporto alla concorrenza; a tal proposito, per proteggere un metodo remoto da accessi concorrenti, esso deve essere definito come "synchronized".
- > Il client, quando contatta il server, entra in stato di blocco in attesa di risposta (comunicazione sincrona).
- > La semantica di errore adottata è At-most-once.
- > La comunicazione tra client e server è transiente (non persistente).

### Garbage collector distribuito:

Quando all'interno del server si ha un oggetto non più usato dai client, è bene dealloccarlo. Affinché ciò sia possibile, l'oggetto deve tener traccia del numero dei client che lo stanno utilizzando. Tuttavia, si tratta di un requisito non semplice a causa di possibili crush dei client e fallimenti della rete, per cui si può affermare che Java RMI non è adatto alle applicazioni su larga scala.

Quel che fa Java RMI per far funzionare correttamente la garbage collection è sfruttare un meccanismo basato sul **lease**. In particolare, la JVM del client:

- Invia periodicamente un messaggio di **dirty** alla JVM del server mentre l'oggetto remoto è in uso.
- Invia un messaggio di **clean** alla JVM del server quando l'oggetto non serve più.

L'intervallo di tempo ogni cui la JVM del client deve inviare il messaggio di dirty è detto **tempo di lease**, scaduto il quale il server può assumere che il client sia andato in crash e, quindi, non stia più referenziando l'oggetto in questione.

Nel momento in cui il server non riceve più alcun messaggio di dirty, può deallocare l'oggetto mediante il garbage collector.

### Confronto tra SunRPC e Java RMI

SunRPC	Java RMI
È process oriented.	È object oriented.
Offre una trasparenza incompleta all'accesso.	Offre una trasparenza migliore all'accesso.
Le entità che possono essere richieste sono le operazioni e le funzioni.	Le entità che possono essere richieste sono i metodi degli oggetti remoti.
Supporta la comunicazione sincrona (quella asincrona è disponibile solo in alcune estensioni di SunRPC).	Supporta solo la comunicazione sincrona.
La semantica di comunicazione è At-least-once.	La semantica di comunicazione è At-most-once.
Il server binding avviene tramite il port mapper.	Il server binding avviene mediante l'RMI registry.
Utilizza XDR come IDL per la rappresentazione dei dati.	Utilizza Java come IDL per la rappresentazione dei dati.
Il passaggio dei parametri avviene tramite il meccanismo di copy-restore.	Il passaggio dei parametri avviene per valore per i tipi di dato primitivi e gli oggetti serializzabili, mentre avviene per riferimento per gli oggetti remoti.
Non supporta né disaccoppiamento spaziale, né disaccoppiamento temporale, e nemmeno disaccoppiamento rispetto alla sincronia.	Non supporta né disaccoppiamento spaziale, né disaccoppiamento temporale, e nemmeno disaccoppiamento rispetto alla sincronia.

### RPC in Go

Go supporta nativamente RPC, tant'è vero che comprende il package `net/rpc`.

Vengono imposti dei vincoli sui metodi RPC:

- Accettano solo due argomenti, di cui il secondo deve essere un puntatore a una struttura che memorizza la risposta da parte del server.
- Devono restituire sempre un errore (che varrà nil se l'invocazione del metodo va a buon fine).

Anche marshaling e unmarshaling sono supportati da un package di Go, `encoding/gob` (gestisce gli stream come valori binari), che tuttavia richiede che sia il client che il server siano scritti in Go. Se non si vuole sottostare a questo vincolo, si possono considerare due alternative:

- > Package `net/rpc/jsonrpc`
- > gRPC

In conclusione, poiché la conversione dati è supportata da un apposito package, non viene utilizzato un IDL specifico.

#### Server:

Lato server è richiesta l'invocazione alle seguenti funzioni:

- **Register** (o **RegisterName**), che serve a pubblicare i metodi che costituiscono una particolare interfaccia: a proposito di ciò, non è necessario fare un uso esplicito di un name server.
- **Listen**, che serve a mettersi in ascolto delle richieste da parte dei client.
- **Accept**, che serve ad accettare le connessioni sul listener restituito dalla Listen. È una funzione bloccante ma, se il server desidera portare avanti altro lavoro subito dopo la sua invocazione, può effettuarne la chiamata in una goroutine (keyword `go`).

### Client:

Lato client è richiesta l'invocazione alle seguenti funzioni:

- **Dial** (o **DialHTTP**), che serve a connettersi al server RPC relativo a uno specifico indirizzo IP e a uno specifico numero di porta.
- **Call**, che serve a effettuare una chiamata RPC sincrona, oppure **Go**, che serve a effettuare una chiamata RPC asincrona (in questo secondo caso la risposta del server viene inviata su un apposito canale).

### Caratteristiche di RPC in Go:

- > Fornisce una trasparenza all'accesso non completa a causa dei vincoli imposti sui parametri dei metodi.
- > Non fornisce trasparenza alla locazione: il client, per connettersi con il server, deve conoscerne indirizzo IP e numero di porta.
- > Viene supportata sia la comunicazione sincrona (funzione Call) sia la comunicazione asincrona (funzione Go).
- > La semantica di errore adottata è At-most-once.
- > La comunicazione tra client e server è transiente (non persistente).
- > Si può avere disaccoppiamento rispetto alla sincronia, ma non quelli spaziale e temporale.

### **gRPC**

È un framework open-source multi-piattaforma, multi-linguaggio e nativamente utilizzabile per i servizi cloud e le applicazioni a microservizi.

Un'architettura a microservizi è costituita da servizi indipendenti, autonomi, debolmente accoppiati e sviluppati da team diversi.

gRPC utilizza il protocollo **http/2** per il trasporto: in particolare, la sua idea di base è trattare le Remote Procedure Call come riferimenti agli oggetti http.

Le principali caratteristiche di http/2 sono:

- Presenza di un **binary framing layer** al di sopra degli strati di TCP e TLS che si occupa di suddividere le richieste / risposte in messaggi più piccoli e convertirle in un formato binario, in modo tale che la trasmissione sia particolarmente efficiente.
- Possibilità di trasmettere stream di dati; a proposito di ciò, distinguiamo gli stream, i messaggi e i frame nel seguente modo:
  - > **Stream** = flusso di byte bidirezionale in una connessione; può trasportare uno o più messaggi.
  - > **Messaggio** = sequenza completa di frame che mappa in una richiesta o in una risposta.
  - > **Frame** = la più piccola unità di comunicazione in http/2.
- Supporto nativo dello streaming bidirezionale.
- Compressione dell'header http per ridurre l'overhead del protocollo.

gRPC utilizza i **protocol buffer** sia come IDL sia per definire l'interfaccia del servizio. Inoltre, è basato sul solito proxy pattern, con stub (= client stub) e server (=server stub) generati automaticamente; a tal proposito, il protocol buffer fa anche da tramite per la comunicazione tra stub e server.

### Step di base per programmare con gRPC:

1) Utilizzare il protocol buffer come IDL per definire in un file .proto il servizio (= collezione di metodi remoti) e i tipi di messaggio che vengono scambiati tra client e server. In particolare, i messaggi definiscono i parametri di input e di output di ciascun metodo, sono fortemente tipati e hanno la seguente sintassi:

```
message Person {
    string name = 1;
    int32 id = 2;
    bool has_ponycopter = 3;
}
```

dove 1, 2, 3 sono gli indici dei singoli campi del messaggio e sono univoci all'interno dello stesso messaggio.

2) Usare il **protocol buffer compiler** (protoc) per generare client e server scritti con linguaggi di programmazione arbitrari (anche diversi tra loro) a partire dal file .proto.

Più specificatamente, vengono generati due file:

- *NomeServizio.pb.go*, che contiene il codice per la gestione dei messaggi.
- *NomeServizio\_grpc.pb.go*, che contiene gli stub di client e server, che servono rispettivamente a invocare i metodi definiti nel servizio e a implementare i metodi definiti nel servizio.

3) Implementare la logica del server e del client usando le API gRPC. In particolare:

-> Il codice del server è composto da due parti: da un lato si ha l'implementazione dei metodi, mentre dall'altro si ha l'invocazione alle seguenti 4 API:

- **Listen**, che serve a mettersi in ascolto delle richieste da parte dei client.
- **NewServer**, che crea un nuovo server stub.
- **RegisterGreeterServer**, che serve a registrare il servizio.
- **Serve**, che attiva il server.

-> Anche il codice del client è composto da due parti: all'inizio si ha l'invocazione delle seguenti 2 API:

- **Dial**, che serve a creare un canale di comunicazione su cui inviare le richieste al server.
- **NewGreeterClient**, che serve a creare lo stub.

Infine, è possibile invocare sul client stub i metodi che compongono il servizio. Notiamo che il primo parametro è un oggetto *context* che ci permette di cambiare il comportamento della Remote Procedure Call, se necessario.

#### Tipi di metodi di gRPC:

- **Simple RPC**: il client invia una richiesta al server e riceve un'unica risposta (è il caso considerato finora).
- **Server-side streaming RPC**: il client invia una richiesta al server e riceve un flusso di risposte.
- **Client-side streaming RPC**: il client invia un flusso di richieste e riceve un'unica risposta.
- **Bidirectional streaming RPC**: sia il client che il server comunicano con flussi di messaggi.

#### Debolezze di gRPC:

- > Poiché utilizza solo il protocollo http/2 per il trasporto, non supporta le chiamate ai servizi gRPC tramite browser. Per mitigare tale problema, è possibile usare gRPC-Web, che però fornisce un supporto ai browser soltanto di alcune funzionalità gRPC.
- > Il formato dei dati per i protocol buffer è binario, per cui non è human-readable; di conseguenza, l'attività di debugging deve essere supportata da particolari tool aggiuntivi.

### **Comunicazione orientata ai messaggi**

Abbiamo visto come RPC provi a fornire trasparenza alla distribuzione per la comunicazione tra componenti remoti; tuttavia, fa in modo che i componenti abbiano un certo grado di accoppiamento. Per questo motivo, introduciamo dei modelli di comunicazione message-oriented, che migliorano il disaccoppiamento e la flessibilità:

- Message Passing Interface (MPI), che offre comunicazione transiente.
- Message Oriented Middleware (MOM), che offre comunicazione persistente.

### **Message Oriented Middleware (MOM)**

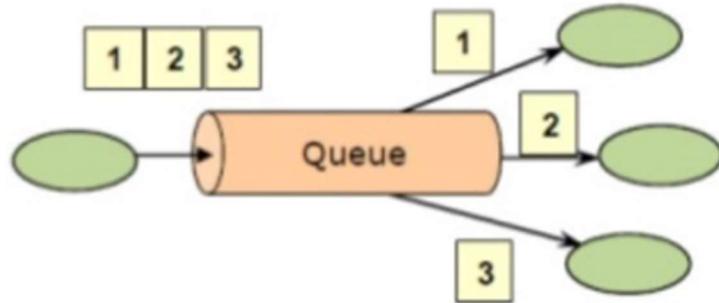
È un middleware di comunicazione che offre disaccoppiamento spaziale e temporale, ma può anche supportare il disaccoppiamento rispetto alla sincronia. Tipicamente viene usato nelle architetture serverless e a microservizi.

Esistono due pattern architetturali per il MOM: **message queue** e **publish-subscribe**.

#### Message queue pattern:

Il mittente invia un messaggio alla coda di messaggi, dove viene memorizzato finché non viene ricevuto dal

destinatario. Molteplici consumer (lettori) possono utilizzare la coda di messaggi ma ciascun messaggio può essere consegnato una sola volta a un solo consumer.



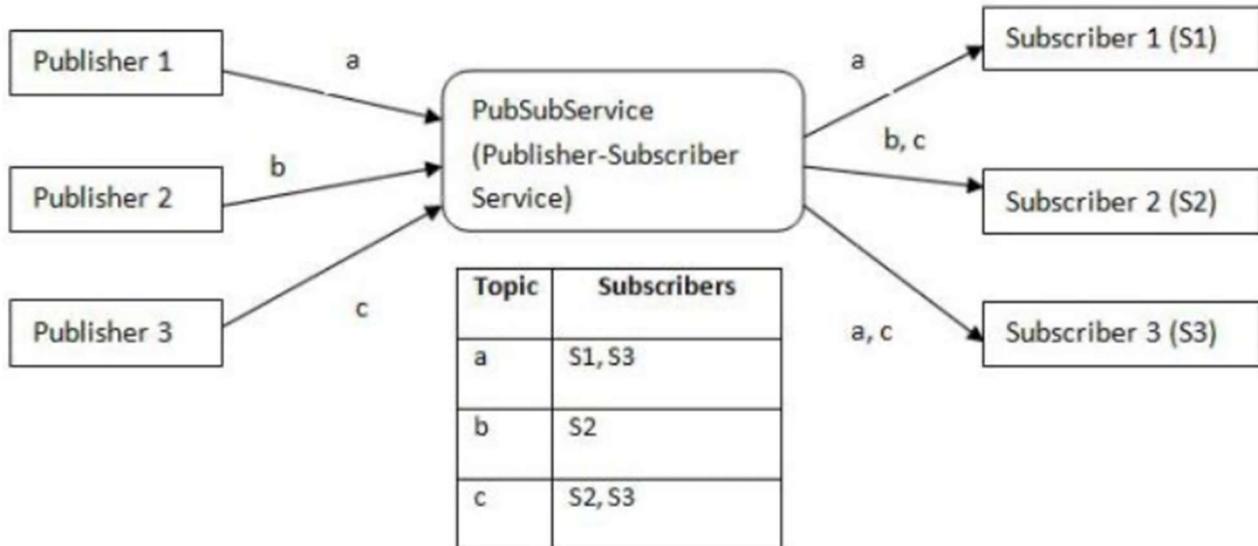
I sistemi a code di messaggi mettono a disposizione le seguenti API:

- **Put**: invio non bloccante di un messaggio alla coda (avviene tramite un'operazione di append).
- **Get**: ricezione bloccante di un messaggio (tipicamente del primo messaggio della coda specificata, ma esistono varianti in cui si può fare ricerca di un messaggio specifico).
- **Poll**: ricezione non bloccante di un messaggio (fallisce nel caso in cui non ci sia alcun messaggio).
- **Notify**: ricezione non bloccante di un messaggio in cui viene installato un handler (una funzione di callback) che verrà chiamato automaticamente quando verrà inserito un messaggio nella coda specificata.

#### Publish-subscribe pattern:

I componenti dell'applicazione possono pubblicare messaggi (i.e. eventi) in modo asincrono e/o dichiarare il loro interesse in determinati topic effettuando una sottoscrizione. Qui, a differenza del message queue pattern, ciascun messaggio può essere inviato a molteplici subscriber (= consumer), dato che più consumer possono sottoscriversi a uno stesso topic.

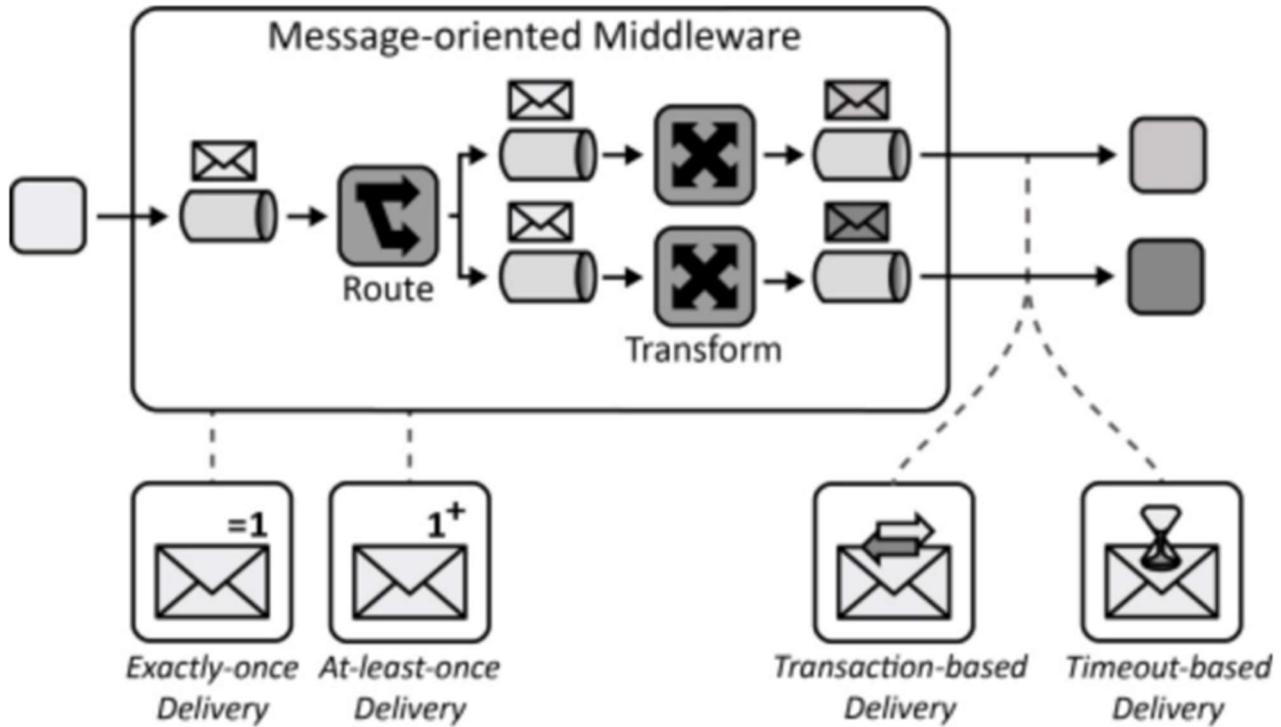
Le sottoscrizioni sono gestite da un componente chiamato **event dispatcher**, che ha la responsabilità di effettuare il routing dei messaggi a tutti i subscriber interessati. Per motivi di scalabilità, l'implementazione dell'event dispatcher è distribuita.



I sistemi publish-subscribe mettono a disposizione le seguenti API:

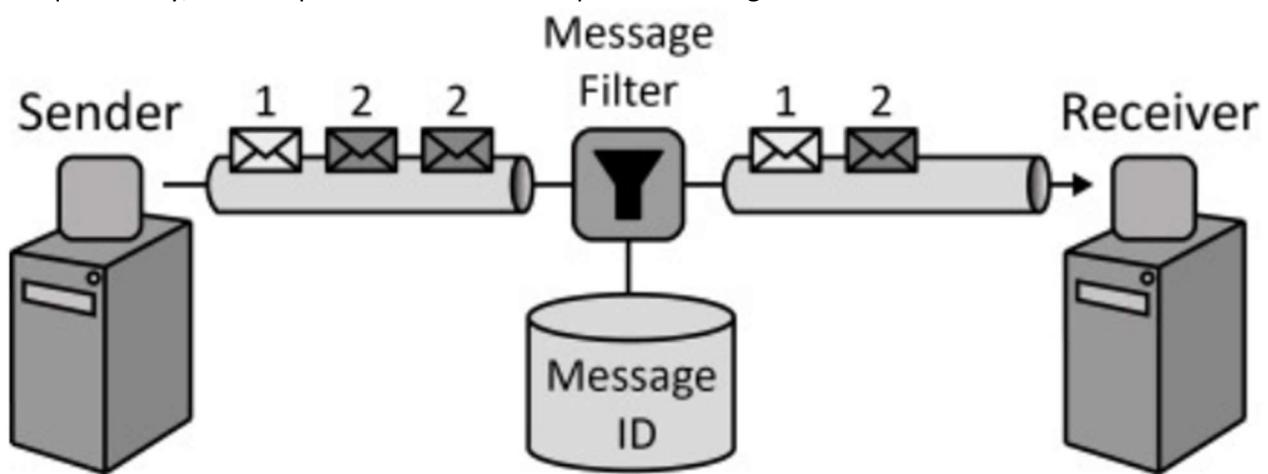
- **Publish**: pubblicazione di un evento.
- **Subscribe**: sottoscrizione agli eventi di interesse.
- **Unsubscribe**: annullamento di una particolare sottoscrizione.
- **Notify\_cb**: invio di un evento ai subscriber interessati (questa API viene utilizzata dal sistema publish-subscribe).

## Funzionalità del MOM



### Semantiche di consegna:

- 1) **Consegna At-least-once**: il consumer riceve il messaggio almeno una volta; in particolare, viene implementato il meccanismo di Request Retry. Ciò richiede che l'applicazione sia idempotente.
- 2) **Consegna Exactly-once**: il consumer riceve il messaggio almeno una volta; stavolta, oltre al meccanismo di Request Retry, viene implementato anche il Duplicate Filtering.



3) **Consegna transaction-based**: affinché si abbia la garanzia che i messaggi vengano eliminati dalla message queue solo dopo essere stati ricevuti correttamente, le operazioni di *read* e di *delete* costituiscono una transazione che assume un comportamento ACID.

4) **Consegna timeout-based**: quando un messaggio viene ricevuto dal destinatario, non viene eliminato subito, bensì viene portato dallo stato **visible** allo stato **invisible**, in modo tale che sia ancora presente nella coda ma non possa essere ricevuto da nessuno; nel frattempo, viene avviato il cosiddetto **timer di visibilità**. A questo punto si possono avere due scenari:

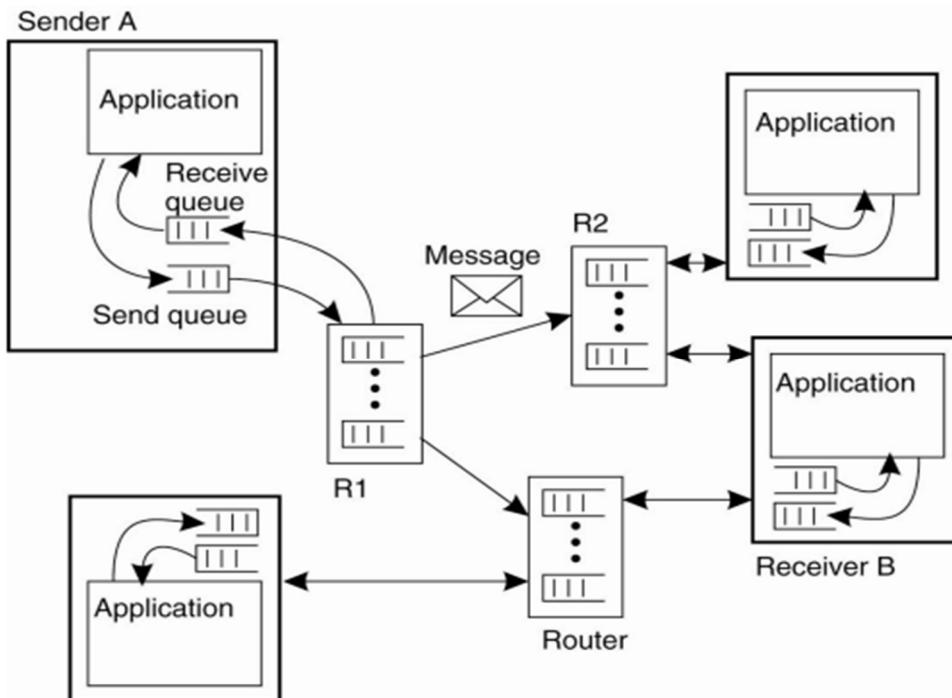
- Il mittente riceve un ack da parte del destinatario → il messaggio viene eliminato dalla coda.
- Scatta il timeout di visibilità → il messaggio viene reso nuovamente visibile.

### Routing di messaggi:

Ciascuna coda di messaggi viene gestita da un **queue manager (QM)** e, in particolare:

- > Un'applicazione può inserire messaggi solo in una coda locale.
- > Un'applicazione può ricevere messaggi solo estraendoli da una coda locale.

Di conseguenza, i QM hanno bisogno di effettuare il routing di messaggi e tutti insieme formano un'overlay network che può essere configurata staticamente oppure dinamicamente.



### Trasformazione di messaggi:

Nel contesto dei MOM, l'eterogeneità nella rappresentazione dei dati viene affrontata mediante l'utilizzo di un **message broker**, il quale:

- Converte i messaggi in ingresso nel formato target fornendo trasparenza all'accesso.
- Gestisce un repository di programmi e di regole di conversione che supportano la trasformazione dei messaggi.
- Può essere implementato in modo distribuito per incentivare la scalabilità e l'affidabilità.

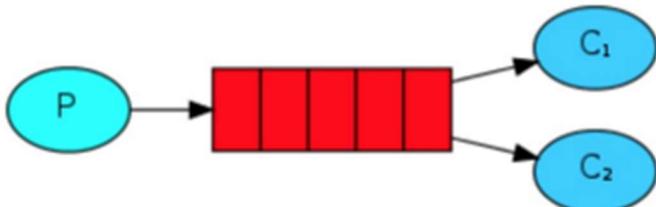
### **RabbitMQ**

È un famoso framework open-source a coda di messaggi che supporta diversi casi d'uso:

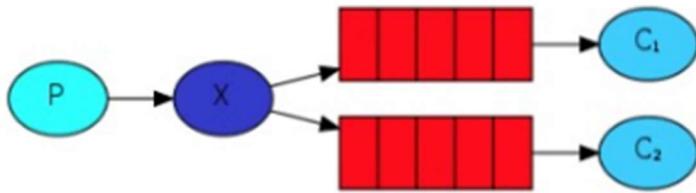
- 1) Store & forward di messaggi che vengono inviati da un producer e ricevuti da un consumer (**message queue pattern**).



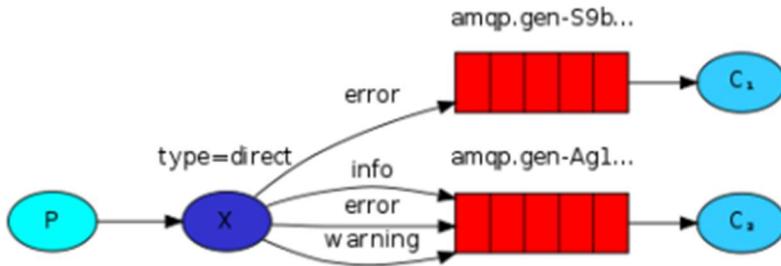
- 2) Distribuzione di task tra molteplici worker (**competing consumers pattern**).



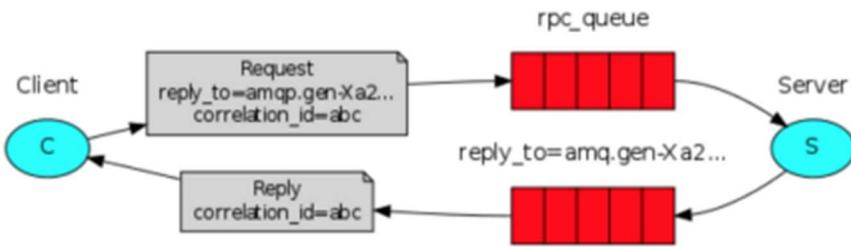
3) Consegnare di messaggi a più consumatori per volta utilizzando un message exchange (**publish-subscribe pattern**).



4) Invio di messaggi a un message exchange, il quale seleziona le code a cui inoltrarli (**ricezione selettiva dei messaggi**).



5) Esecuzione di una funzione su un nodo remoto + attesa del risultato (**request-reply pattern**).



## IBM MQ

Storicamente è la prima tecnologia enterprise per le code di messaggi, e presenta le seguenti caratteristiche:

- Le code risiedono sotto il regime di un queue manager (QM).
- I processi possono inserire i messaggi in code locali oppure in code remote tramite un meccanismo RPC.
- I messaggi vengono trasferiti da una coda all'altra, e il trasferimento dei messaggi richiede un canale.
- In ciascun endpoint del canale vi è un **message channel agent (MCA)**, che è responsabile dell'istanziazione dei canali e di invio, ricezione e cifratura dei messaggi.
- I canali di comunicazione sono unidirezionali.
- È possibile definire un'overlay network in cui la configurazione del routing è manuale.

## Amazon SQS (Simple Queue Service)

È un servizio cloud di code di messaggi basato su polling. Il suo obiettivo è disaccoppiare i componenti di un'applicazione cloud-native, tant'è vero che questi possono essere eseguiti indipendentemente, possono comunicare in maniera asincrona e possono essere sviluppati con tecnologie differenti.

Amazon SQS implementa la consegna timeout-based e può essere combinato con Amazon SNS (Simple Notification Service) per inviare messaggi a molteplici code SQS in parallelo.

Le API offerte da Amazon SQS sono:

- > CreateQueue, ListQueues, DeleteQueue.
- > SendMessage, ReceiveMessage.
- > DeleteMessage.

-> ChangeMessageVisibility.

-> SetQueueAttributes, GetQueueAttributes.

Notiamo che la SendMessage può inserire in una determinata coda solo messaggi di dimensione non superiore ai 256 KB: per messaggi più grandi, si aggiunge nella coda un riferimento (un link) al payload del messaggio il quale invece viene memorizzato in un data storage (Amazon S3).

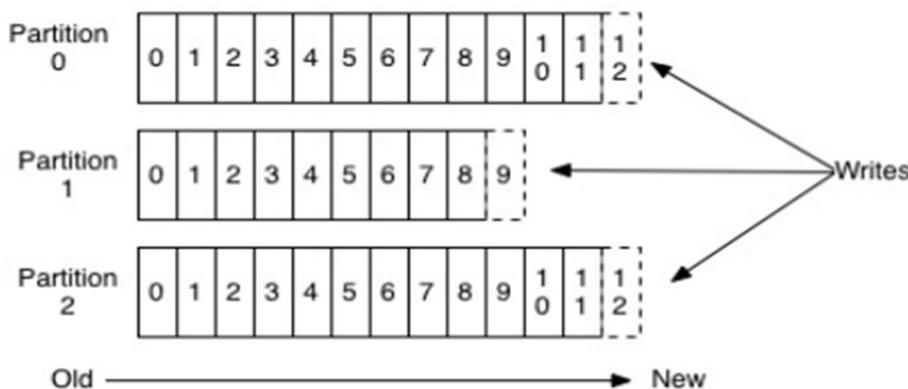
## Apache Kafka

È un sistema publish-subscribe distribuito scritto nel linguaggio Scala e con un throughput di ingestion (velocità nel caricare messaggi) molto elevato.

I messaggi sono suddivisi in categorie chiamate topic. Gli attori in gioco sono i **producer**, che pubblicano i messaggi, i **consumer**, che si sottoscrivono a determinati topic, e il **cluster di Kafka**, che per ogni topic mantiene un log di dati.

Il log è una struttura dati suddivisa in un certo numero fissato di partizioni; ciascuna partizione rappresenta un'unità di parallelismo del topic (infatti il partizionamento consente gli accessi paralleli) e consiste in una sequenza ordinata, numerata e immutabile di record a cui vengono effettuate le operazioni di append. In particolare, ogni record è associato a un sequence number via via crescente chiamato offset.

Si ha una forte garanzia sull'ordinamento dei record soltanto all'interno delle singole partizioni: per avere un ordinamento totale a livello di topic, il producer dovrebbe attuare dei particolari accorgimenti.



Per motivi di scalabilità, le partizioni di ciascun log sono distribuite e gestite da un insieme di server detti broker.

Inoltre, per incentivare la tolleranza ai guasti, le partizioni vengono replicate su un numero configurabile di broker. In particolare, ogni partizione ha un broker **leader** e 0+ broker **follower**; le operazioni di read e write sono gestite dal leader, mentre i follower fungono unicamente da backup. Per aumentare ulteriormente la tolleranza ai guasti, si fa in modo che, a ogni scrittura, il messaggio non venga reso disponibile ai consumer finché tutti i follower della partizione interessata non hanno a loro volta completato l'operazione di write e hanno inviato un ack al leader.

Per distribuire al meglio il carico di lavoro tra i broker, si fa in modo che ciascuno di essi sia leader per alcune delle sue partizioni e sia follower per le altre.

### Producer:

Pubblicano i record su una partizione di un topic a loro scelta (la selezione della partizione può avvenire secondo uno schema round-robin oppure sull'utilizzo di una chiave) e, nel farlo, inviano i dati direttamente (i.e. senza alcun intermediario) al broker che funge da leader per quella partizione.

Più producer differenti possono scrivere sulla medesima partizione.

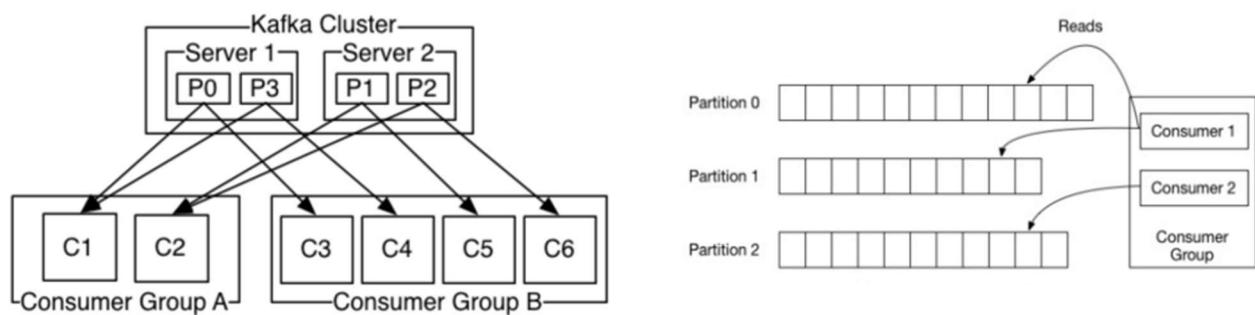
### Consumer:

Possono essere implementati secondo due modelli diversi:

- **Push model:** il broker effettua la push dei messaggi verso i consumer. Il problema è che si potrebbero avere dei consumer che girano a velocità differenti, e il broker deve adattarsi di conseguenza. Inoltre, il broker deve decidere se inviare ciascun messaggio immediatamente oppure inviare i dati solo dopo averne accumulati un po'.

- **Pull model:** il consumer ha la responsabilità di recuperare i messaggi dal broker e, per farlo, deve mantenere un offset che identifica il prossimo messaggio che dovrà processare. Qui si ha maggiore scalabilità (i broker sono meno carichi) e maggiore flessibilità (è più facile gestire consumer che hanno necessità e capacità computazionali diverse). D'altra parte, però, i consumer possono essere soggetti al busy waiting. Comunque sia, questo è il modello utilizzato da Kafka.

Per migliorare la scalabilità e la tolleranza ai guasti, i consumer possono essere raggruppati e condividere un group ID comune, formando così un **consumer group**, il quale è relativo a un subscriber logico.



### Semantiche di comunicazione supportate da Kafka:

- > **At-least-once** (default).
- > **Exactly-once** (non è però un'Exactly-once completa e in più utilizza un protocollo oneroso – il 2 phase commit - per la gestione delle transazioni).
- > L'utente può anche implementare la sematica **At-most-once** disabilitando le ritrasmissioni sul producer e facendo effettuare il commit al consumer prima di processare un messaggio.

### Kafka e ZooKeeper:

ZooKeeper è un datastore di tipo {chiave, valore} e ha un'architettura distribuita.

Kafka usa ZooKeeper per coordinare i producer, i consumer e i broker. In particolare, ZooKeeper memorizza i metadati di Kafka, che comprendono:

- La lista di broker.
- La lista di consumer e i relativi offset da cui riprendere a leggere.
- La lista di producer.

### API di Kafka:

- > **Producer API:** servono a pubblicare nuovi record su qualche topic.
- > **Consumer API:** servono a leggere dei record da qualche topic.
- > **Connect API:** servono a implementare connettori riutilizzabili (per connettere i topic di Kafka con applicazioni esistenti in modo tale da poter spostare grandi insiemi di dati da e verso Kafka).
- > **Stream API:** servono a processare i dati che arrivano (e.g. filtrarli).

### **Protocolli per MOM**

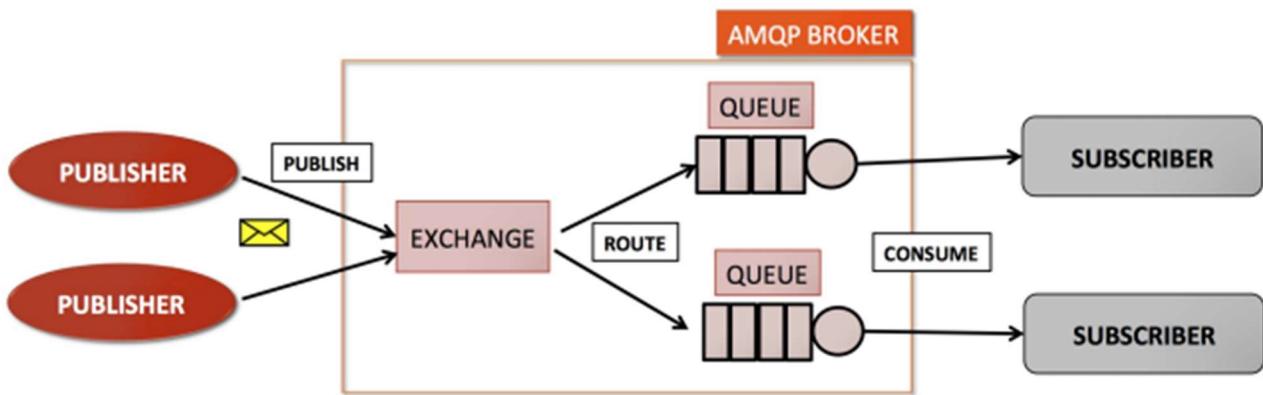
Sono protocolli per sistemi a code di messaggi che garantiscono disaccoppiamento, resilienza ai guasti e gestione dei picchi di carico. Sono agnostici rispetto alla piattaforma utilizzata e sono spesso utilizzati nell'ambito dell'Internet of Things. Tra questi abbiamo:

- AMQP (Advanced Message Queueing Protocol).
  - MQTT (Message Queue Telemetry Transport).
  - STOMP (Simple Text Oriented Messaging Protocol).
- Tra questi introdurremo AMQP.

### AMQP

È un protocollo applicativo di tipo binario che si appoggia su TCP ed è programmabile. Coinvolge tre attori principali: i **publisher**, i **subscriber** e i **broker**. All'interno di un broker si hanno tre entità: le **code**, gli **exchange** e i **binding**.

- > I publisher pubblicano i messaggi sugli exchange (che possiamo vederli come delle mailbox).
- > Gli exchange distribuiscono le copie dei messaggi alle code utilizzando delle particolari regole chiamate binding.
- > È possibile sia che i broker effettuino la push dei messaggi, sia che i subscriber richiedano i messaggi tramite una pull.



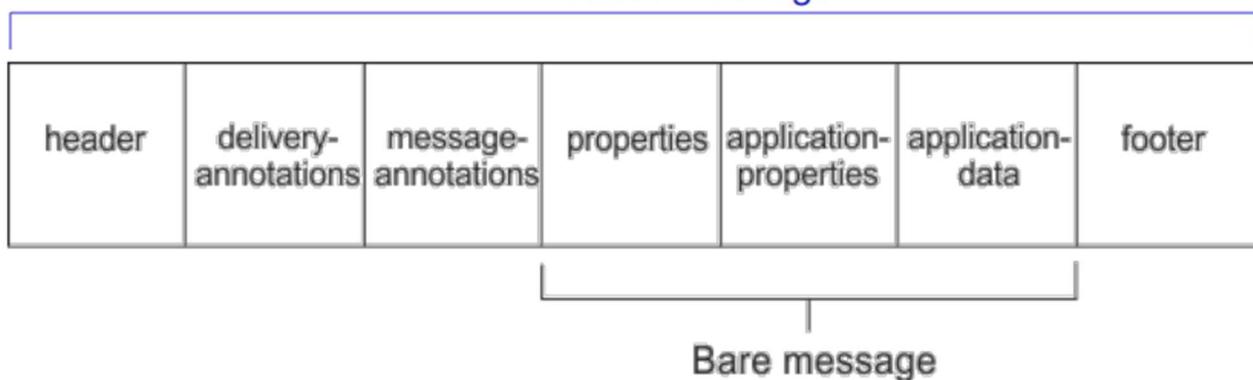
Esistono vari binding disponibili:

- 1) **Direct exchange**: ciascun messaggio viene inoltrato alle code associate a una particolare routing key.
- 2) **Fanout exchange**: ciascun messaggio viene inoltrato a tutte le code collegate all'exchange.
- 3) **Topic exchange**: ciascun messaggio viene inoltrato alle code associate a un particolare topic.
- 4) **Headers exchange**: ciascun messaggio viene inoltrato alle code associate a determinati attributi nell'header.

AMQP, infine, definisce due tipi di messaggi:

- Bare messages, che vengono forniti dal server.
- Annotated messages, a cui vengono aggiunti un header e un footer da AMQP durante il transito. In particolare, l'header specifica i parametri di consegna.

Annotated message



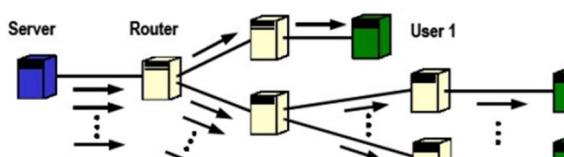
## Comunicazione multicast

È un pattern di comunicazione in cui i dati vengono inviati a più destinatari per volta. Un caso speciale di multicast è il broadcast, in cui i dati vengono inviati a tutti i destinatari possibili.

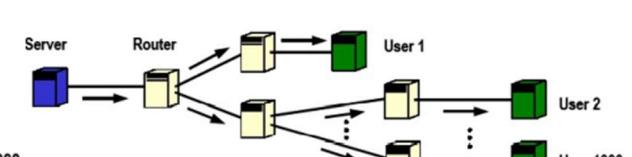
La comunicazione multicast può essere:

- **One-to-many** (e.g. distribuzione di file).
- **Many-to-many** (e.g. tool di conferenza).

La comunicazione unicast, a differenza della multicast, non scala poiché prevede la generazione di n flussi di dati da parte del nodo sorgente.



Unicast of video to 1000 users



Multicast of video to 1000 users

Il multicast può essere realizzato a livello di rete oppure a livello applicativo.

### Network-level multicast:

La replicazione dei pacchetti e il routing sono gestiti dai router.

Il protocollo **IPMC** (IP Multicast) generalizza UDP mediante un comportamento one-to-many ed è basato sui gruppi. Un gruppo è un insieme di host interessati nella stessa applicazione multicast e identificati dal medesimo indirizzo IP (compreso tra 224.0.0.0 e 239.255.255.255). Per inserirsi all'interno di un gruppo si utilizza il protocollo **IGMP** (Internet Group Management Protocol).

IPMC non è supportato su larga scala. Infatti, presenta problemi nel tenere traccia dell'appartenenza al gruppo e viene tendenzialmente disabilitato dalle piattaforme cloud a causa del **broadcast storm problem** (crescita esponenziale del traffico di rete con possibile saturazione).

### Application-level multicast:

La replicazione dei pacchetti e il routing sono gestiti dagli end host.

Qui i nodi sono organizzati in un'overlay network, che può essere:

- **Strutturata**, con dei path di comunicazione esplicativi, che possono essere dei Tree (alberi) o Mesh (maglie).
- **Non strutturata**, con la possibilità di utilizzare un meccanismo di comunicazione basato su flooding, random walk oppure gossiping.

Consideriamo un albero di multicast strutturato in Scribe (dove Scribe è un sistema publish-subscribe con architettura decentralizzata basata sulla DHT Pastry):

- > Il nodo che inizializza la sessione genera un identificatore multicast (**mid**).
- > Il nodo inizializzatore, usando Pastry, cerca **succ(mid)**, che è il nodo responsabile per il multicast mid.
- > Il succ(mid) diventa il root dell'albero multicast.
- > Se il nodo P vuole aggiungersi al multicast, gli invia una richiesta di join.
- > Quando la richiesta arriva al nodo Q, P diventa child di Q; se Q non aveva mai visto la richiesta di join, diventa forwarder, altrimenti non c'è bisogno di inoltrare ulteriormente la richiesta.

## Gossiping

I protocolli basati sul gossiping sono probabilistici e sono anche detti epidemici. L'idea è che il mittente invia un messaggio a un sottoinsieme dei suoi vicini scelti casualmente; dopodiché ciascuno di essi inoltra a sua volta il messaggio a un sottoinsieme dei suoi vicini scelti randomicamente, e così via. Tale meccanismo consente la disseminazione delle informazioni nelle reti a larga scala; la propagazione è definita lazy (non immediata).

Il gossiping è molto utilizzato nei sistemi distribuiti su larga scala poiché presentano i seguenti vantaggi:

- **Semplicità** degli algoritmi di gossiping.
- **Assenza di un controllo centralizzato** (coi relativi bottleneck e single point of failure).
- **Scalabilità**: ciascun nodo invia solo un numero limitato di messaggi indipendentemente dalla dimensione complessiva del sistema.
- **Affidabilità e robustezza** grazie alla ridondanza dei messaggi.

Il gossiping è molto utilizzato da AWS S3, Amazon Dynamo, BitTorrent e Cassandra.

#### Modelli di propagazione:

-> **Pure gossiping** (aka **rumor spreading**): un nodo (P) che ha appena ricevuto un aggiornamento (i.e. è stato appena contaminato) lo inoltra a un certo numero di altri nodi, contaminandoli a loro volta. Supponiamo che tra questi nodi rientri Q. Se Q conosceva già l'aggiornamento, P può perdere interesse nel propagare il gossip e smette di contattare altri nodi con probabilità  $1/k$  (dove k è un parametro che indica il grado di diffusione).

Se denotiamo con s la frazione dei nodi che non sono mai stati aggiornati, è possibile mostrare che:

$$S = e^{-(k+1)(1-s)}$$

Notiamo che, all'aumentare di k, il valore di s decresce.

Comunque sia, se vogliamo davvero assicurarcici che tutti i nodi vengano aggiornati, il pure gossiping da solo non è sufficiente ma dovremmo combinarlo con un altro modello di propagazione: l'anti-entropia.

-> **Anti-entropy**: il suo obiettivo è incrementare la similarità tra gli stati dei nodi, in modo tale da diminuire il disordine (da qui deriva il nome "anti-entropy"). Un nodo (P) seleziona un altro nodo Q randomicamente; l'aggiornamento può avvenire in tre modi differenti:

1) Push: P è l'unico a inviare i dati a Q.

2) Pull: P è l'unico a richiedere gli aggiornamenti da Q.

3) Push-pull: P, Q si inviano gli aggiornamenti a vicenda. Questa è la strategia più veloce e richiede  $O(\log(N))$  round per disseminare i messaggi a tutti gli N i nodi, dove un round (= gossip cycle) è l'intervallo di tempo in cui tutti i nodi hanno preso l'iniziativa di iniziare uno scambio di informazioni.

#### Algoritmo di base del gossiping:

##### **Active thread (node P):**

- (1) **selectPeer**(&Q);
- (2) **selectToSend**(&bufs);
- (3) **sendTo**(Q, bufs);
- (4)
- (5) **receiveFrom**(Q, &bufr);
- (6) **selectToKeep**(cache, bufr);
- (7) **processData**(cache);

##### **Passive thread (node Q):**

- (1)
- (2)
- (3) **receiveFromAny**(&P, &bufr);
- (4) **selectToSend**(&bufs);
- (5) **sendTo**(P, bufs);
- (6) **selectToKeep**(cache, bufr);
- (7) **processData**(cache)

- **selectPeer**: seleziona un vicino randomicamente.

- **selectToSend**: seleziona dalla cache locale alcune entry da inviare al destinatario.

- **selectToKeep**: seleziona quali entry ricevute memorizzare nella cache locale; in particolare, rimuove le entry replicate.

Ora introdurremo due esempi di protocolli di gossiping: **Blind counter rumor mongering** e **Bimodal multicast**.

### **Blind counter rumor mongering**

Da dove viene fuori il nome di questo protocollo?

-> Rumor mongering = atto del diffondere un rumore = gossip.

-> Blind = perdita di interesse nel gossip a prescindere da quale sia il nodo destinatario.

-> Counter = perdita di interesse nel gossip dopo un certo numero di contatti.

Per questo protocollo sono definiti due parametri:

- **B**: numero di vicini che ciascun nodo contatta a ogni iterazione.

- **F**: numero di volte in cui ogni nodo riceve un determinato aggiornamento prima di smettere di propagarlo.

Il meccanismo è il seguente:

-> Il nodo sorgente inizia la comunicazione inviando il messaggio m a B vicini scelti randomicamente.

-> Quando il nodo p riceve m dal nodo q, controlla se, complessivamente, non ha ricevuto m più di F volte.

Se il check passa, allora inoltra m a B vicini tra quelli che teoricamente non hanno mai visto il messaggio.

Almeno per quel che sa p, i nodi che hanno già ricevuto almeno una volta il messaggio sono quelli che lo hanno inviato direttamente a p e quelli che lo hanno ricevuto direttamente da parte di p.

### **Bimodal multicast**

Detto anche **pbcast** (probabilistic multicast), è un protocollo composto da due fasi:

1) Message distribution: un processo invia un messaggio in multicast senza particolari garanzie di affidabilità.

2) Gossip repair: a intervalli regolari, ciascun processo contatta un sottoinsieme di peer per confrontare i loro stati ed eliminare i gap (le differenze) nelle sequenze di messaggi che hanno ricevuto nella prima fase. In particolare, le informazioni scambiate tra i processi in questa seconda fase sono i **digest**, che identificano i messaggi ricevuti in precedenza (senza però includerli). Nel momento in cui un peer realizza di aver ricevuto qualche messaggio in meno del dovuto, invia una sollecitazione al processo con cui ha scambiato i digest affinché esso risponda coi messaggi mancanti originali.

Il termine bimodale deriva da due particolari caratteristiche del protocollo:

- Nella stragrande maggioranza dei casi, le informazioni vengono consegnate o a quasi tutti i processi o a quasi nessun processo.

- Nella stragrande maggioranza dei casi, le informazioni vengono consegnate o con ritardi estremamente piccoli, o con ritardi estremamente grandi.

### **Event matching**

Detto anche **notification filtering**, è una funzionalità centrale dei sistemi publish-subscribe che, ogni qualvolta viene pubblicato un evento, si occupa di:

- Verificare le sottoscrizioni a quel tipo di evento.

- Notificare i subscriber che hanno effettuato tali sottoscrizioni.

Per implementare l'event matching esistono diverse soluzioni architettoniche:

-> Architettura centralizzata: è semplice da realizzare ma non scala.

-> Architettura master-worker (o gerarchica): si ha un unico nodo master che suddivide il lavoro tra più nodi worker; ciascun worker si occupa di un sottoinsieme delle sottoscrizioni. Il lavoro può essere ad esempio partizionato utilizzando l'hashing dei nomi dei topic per mappare le sottoscrizioni e gli eventi sui worker (a patto che il sistema publish-subscribe sia topic-based).

-> Architettura flat: è simile all'architettura master-worker ma prevede l'uso di più server distribuiti (chiamati broker) tra cui il lavoro viene suddiviso, in modo tale da fare a meno del singolo nodo master.

-> P2P unstructured overlay: usa il flooding o il gossiping per propagare le informazioni.

-> P2P structured overlay.

# VIRTUALIZZAZIONE

## Introduzione

La virtualizzazione è un'astrazione delle risorse computazionali che consente all'utilizzatore di avere una vista logica del sistema diversa dalla vista fisica. Ciò avviene disaccoppiando il comportamento e l'architettura delle risorse hardware e software percepiti dall'utente dalla loro realizzazione fisica.

Abbiamo tre componenti fondamentali:

- 1) **Guest** = componente del sistema che interagisce col layer di virtualizzazione (anziché con l'host) e ha una vista virtuale del sistema.
- 2) **Host** = ambiente originale sottostante al layer di virtualizzazione.
- 3) **Layer di virtualizzazione** = strato che ha la responsabilità di ricreare un ambiente virtuale possibilmente diverso da quello reale dove il guest andrà a operare.

Ma cosa è possibile virtualizzare?

- L'ambiente di esecuzione con le relative risorse hardware e software (noi ci focalizzeremo su questo).
- Lo storage (e.g. Storage Area Network).
- La rete (e.g. VLAN, VPN).
- Il data center.

## Vantaggi della virtualizzazione

- > Facilita la compatibilità, la portabilità, l'interoperabilità e la migrazione delle applicazioni e degli ambienti di esecuzione da una macchina all'altra (politica **create once, run anywhere**).
- > Permette il **consolidamento dei server**: è possibile avere dei server software (e.g. http server, application server e DB server) su macchine virtuali diverse ma all'interno dello stesso host fisico, necessitando così di molti meno host. Ciò porta a una riduzione dei costi, dei consumi energetici, dello spazio occupato e dei tempi di downtime (poiché è possibile migrare una VM da un server a un altro mentre è in funzionamento).
- > Permette di isolare componenti malfunzionanti o soggetti ad attacchi di sicurezza, incrementando così l'affidabilità e la sicurezza delle applicazioni.
- > Permette di isolare le prestazioni di diverse VM in esecuzione su uno stesso host tramite uno scheduling delle risorse fisiche tra le VM.
- > Consente di bilanciare il carico sui server mediante la migrazione di una VM da un server a un altro.

## Interfacce di sistema

- User-level ISA: comprende le istruzioni macchina non privilegiate invocate da qualunque programma.
- System ISA: comprende le istruzioni macchina privilegiate, usate per lo più per la gestione delle risorse del sistema.
- ABI (Application Binary Interface): comprende le system call.
- API (Application Programming Interface): è una funzione di libreria.

## Livelli di virtualizzazione

- 1) Livello ISA: il suo obiettivo è emulare una data ISA (e.g. MIPS) a partire dall'ISA della macchina host (e.g. x86); l'emulazione può avvenire tramite **code interpretation** (più lenta poiché lavora istruzione per istruzione) oppure tramite **dynamic binary translation** (più veloce poiché lavora a blocchi di istruzioni).
- 2) Livello hardware (aka virtualizzazione di sistema): il suo obiettivo è virtualizzare le risorse di un computer come i processori, la memoria e i dispositivi I/O. È basato sul **Virtual Machine Monitor (VMM**, detto anche **hypervisor**), il quale gestisce le risorse hardware e le condivide tra molteplici VM, garantendo isolamento e protezione delle VM.

- 3) Livello di sistema operativo (aka container): il suo obiettivo è creare molteplici container isolati.
- 4) Livello di libreria: il suo obiettivo è creare un ambiente di esecuzione per far girare delle applicazioni che altrimenti non sarebbero compatibili con l'ambiente host.
- 5) Livello di applicazione utente (aka process VM): il suo obiettivo è compilare un'applicazione in un codice intermediario e portabile (e.g. Java bytecode) ed eseguirla in una piattaforma virtuale (e.g. JVM); tale piattaforma può eseguire un singolo processo.

### **Virtualizzazione di sistema**

Qui l'**host** è la piattaforma di base sulla quale vengono eseguite le VM, mentre il **guest** rappresenta tutto ciò che si trova all'interno della singola VM.

La virtualizzazione di sistema viene classificata in base a due assi:

- 1) Dove viene posto il VMM?

- **VMM di sistema (o type-1 hypervisor)**: viene eseguito direttamente sull'hardware e offre funzionalità di virtualizzazione integrate in un sistema operativo semplificato. Può avere un'architettura a microkernel oppure monolitica.
- **VMM ospitato (o type-2 hypervisor)**: viene eseguito sul sistema operativo host, accede alle risorse hardware tramite le chiamate di sistema del sistema operativo host ed emula l'ISA dell'hardware virtuale per i sistemi operativi guest. Poiché il sistema operativo host rappresenta un layer di indirezione aggiuntivo, questa soluzione porta a un degrado delle prestazioni (anche se non occorre modificare il sistema operativo guest).

- 2) Come viene gestita l'esecuzione delle istruzioni privilegiate?

- **Virtualizzazione completa**: il VMM espone a ogni VM interfacce hardware simulate funzionalmente identiche a quelle della macchina fisica sottostante; in particolare, intercetta le richieste di accesso privilegiato all'hardware e ne emula il comportamento atteso.

- **Paravirtualizzazione**: il VMM espone a ogni VM interfacce hardware simulate funzionalmente simili (ma non identiche) a quelle della macchina fisica sottostante; non viene emulato l'hardware, bensì viene creato uno strato minimale di software (Virtual Hardware API) per assicurare la gestione delle VM e il loro isolamento.

### Teorema di Popek e Goldberg:

Introduciamo le seguenti definizioni:

- > **Istruzione privilegiata** = istruzione che invia un trap (un'eccezione) e trasferisce il controllo al VMM se viene eseguita in user mode.
- > **Stato privilegiato** = stato che determina l'allocazione delle risorse.
- > **Istruzione sensibile** = istruzione che modifica lo stato privilegiato (control sensitive instruction) oppure lo espone (behavior sensitive instruction).

Il teorema afferma che è possibile costruire un VMM se l'insieme delle istruzioni sensibili è incluso nell'insieme delle istruzioni privilegiate.

In altre parole, è sufficiente che tutte le istruzioni che possono coinvolgere il funzionamento del VMM (ovvero le istruzioni sensibili) inviano un trap e passino il controllo al VMM stesso, il quale verifica la correttezza dell'operazione richiesta e ne emula il comportamento (meccanismo **trap-and-emulate**).

Tuttavia, l'implementazione del trap-and-emulate può essere complessa: infatti, le architetture comuni (come x86, MIPS, ecc.) non sono virtualizzabili secondo il teorema di Popek e Goldberg.

### Problemi nel realizzare la virtualizzazione di sistema:

L'architettura del processore opera secondo almeno due livelli (**ring**) di protezione:

- **Ring 0**: privilegi massimi (supervisor mode).

- **Ring 3**: privilegi minimi (user mode).

Con la virtualizzazione, il VMM (ring 0) opera in supervisor mode, mentre il sistema operativo guest (ring 1) e le applicazioni (ring 3) operano in user mode. Da qui sorgono due problemi:

- > **Ring deprivilegging**: poiché il sistema operativo guest opera in un ring che non gli è proprio, non può eseguire le istruzioni privilegiate.
- > **Ring compression**: poiché le applicazioni e il sistema operativo guest eseguono allo stesso livello, occorre proteggere lo spazio del sistema operativo.

#### Virtualizzazione completa:

Risolve il problema del ring deprivilegging mediante il meccanismo trap-and-emulate, che può essere realizzato in due modi:

- **Hardware-assisted CPU virtualization**: è la soluzione adottata se il processore fornisce supporto alla virtualizzazione; fornisce alla CPU due modalità di operare, chiamate **root mode** e **non-root mode**. Il VMM gira in root mode (Root-Ring 0), mentre il sistema operativo guest esegue in non-root mode nel suo ring originale (Non-Root Ring 0): in tal modo vengono risolti sia il problema del ring deprivilegging che il problema del ring compression.
  - **Fast binary translation**: è la soluzione adottata se il processore non fornisce supporto alla virtualizzazione. Qui il VMM scansiona il codice prima della sua esecuzione per sostituire i blocchi contenenti istruzioni privilegiate con blocchi funzionalmente equivalenti e contenenti istruzioni per la notifica di eccezioni al VMM. Ciò comporta una maggiore complessità del VMM e minori prestazioni.
- > **Vantaggi** della virtualizzazione completa: non occorre modificare il sistema operativo guest e si ha un isolamento completo tra le istanze di VM.
- > **Svantaggi** della virtualizzazione completa: richiede o una collaborazione del processore o un VMM molto più complesso.

#### Paravirtualizzazione:

Si tratta di una soluzione alla virtualizzazione non trasparente, in cui il kernel del sistema operativo guest deve essere modificato affinché invochi le API virtuali esposte dall'hypervisor. In particolare, le istruzioni non virtualizzabili vengono rimpiazzate dalle **hypercall**, che sono trap software diretti verso l'hypervisor.

### **hypercall : hypervisor = syscall : kernel**

Anche qui il problema del ring deprivilegging viene risolto: il sistema operativo guest opera nel ring 0.

- > **Vantaggi** della paravirtualizzazione: l'implementazione è semplice e non richiede una collaborazione del processore.
- > **Svantaggi** della paravirtualizzazione: richiede che il codice sorgente dei sistemi operativi guest sia disponibile; se non lo fosse (come accade in Windows), è necessario ricorrere a dei driver ad-hoc che mappano l'esecuzione delle istruzioni non virtualizzabili sulle API virtuali esposte dal VMM.

#### Architettura del VMM:

Un VMM è composto da tre moduli principali:

- **Dispatcher**: è l'entry point del VMM che inoltra le istruzioni privilegiate invocate dalla VM verso uno degli altri due moduli.
- **Allocator (o scheduler)**: decide come distribuire le CPU virtuali tra le VM in esecuzione.
- **Interpreter**: esegue una routine appropriata quando la VM invoca un'istruzione privilegiata.

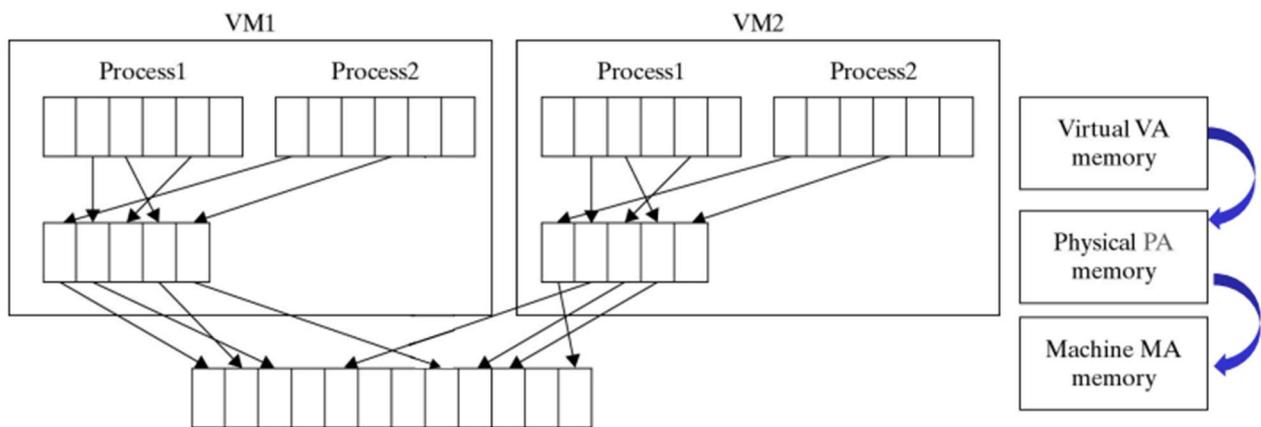
#### **Virtualizzazione della memoria**

Noi sappiamo che in un ambiente non virtualizzato si ha l'**one-level memory mapping**, in cui le tabelle delle pagine contengono le conversioni dagli indirizzi di memoria virtuali a quelli fisici.

In un ambiente virtualizzato, invece, tutte le VM condividono la memoria di uno stesso host e il VMM ha bisogno di partizionare la memoria tra le VM. Per questo motivo, si ha il **two-level memory mapping**, in cui si passa dai **guest virtual address (GVA)** ai **guest physical address (GPA)** agli **host physical address (HPA)**.

In particolare:

- Guest virtual memory = memoria visibile alle applicazioni, che consiste in un address space virtuale composto da indirizzi contigui.
- Guest physical memory = memoria visibile al sistema operativo guest.
- Host physical memory = memoria hardware reale visibile al VMM.



Esistono due soluzioni per evitare un crollo delle prestazioni dovute al two-level memory mapping: la **shadow page table (SPT)** e la **second level address translation (SLAT)**.

#### Shadow page table:

È una tabella mantenuta dall'hypervisor e caricata all'interno del MMU (unità di gestione di memoria) che fa il mapping direttamente da GVA a HPA (senza passare per il GPA). Quando il sistema operativo guest modifica la sua tabella delle pagine, le modifiche vanno riportate nella SPT a seguito di un trap.

Tale soluzione presenta le seguenti problematiche:

- Il sistema operativo guest si aspetta uno spazio di memoria contiguo che inizia dall'indirizzo 0, ma la memoria della macchina sottostante potrebbe non essere contigua: il VMM deve preservare questa illusione.
- L'implementazione della SPT è complessa.
- Le SPT richiedono parecchio spazio di memoria e causano un overhead sul tempo di esecuzione poiché, per essere mantenute, richiedono che le operazioni del sistema operativo guest vengano intercettate.

#### Second level address translation:

È una soluzione hardware-assisted per la virtualizzazione della memoria e, in particolare, per la traduzione dei GVA in HPA. Rispetto all'utilizzo delle SPT, porta a un guadagno delle prestazioni di circa il 50%.

#### Xen

È un VMM di sistema open-source con un design a microkernel. Nonostante sia l'esempio più famoso di paravirtualizzazione, supporta anche la virtualizzazione hardware-assisted. Inoltre, offre ai sistemi operativi guest un'interfaccia virtuale (hypercall API) utile per accedere alle risorse della macchina fisica.

Gli elementi che possono essere paravirtualizzati sono:

- Disco e dispositivi di rete.
- Interrupt e timer.
- Scheda madre simulata.
- Istruzioni privilegiate e tabelle delle pagine.

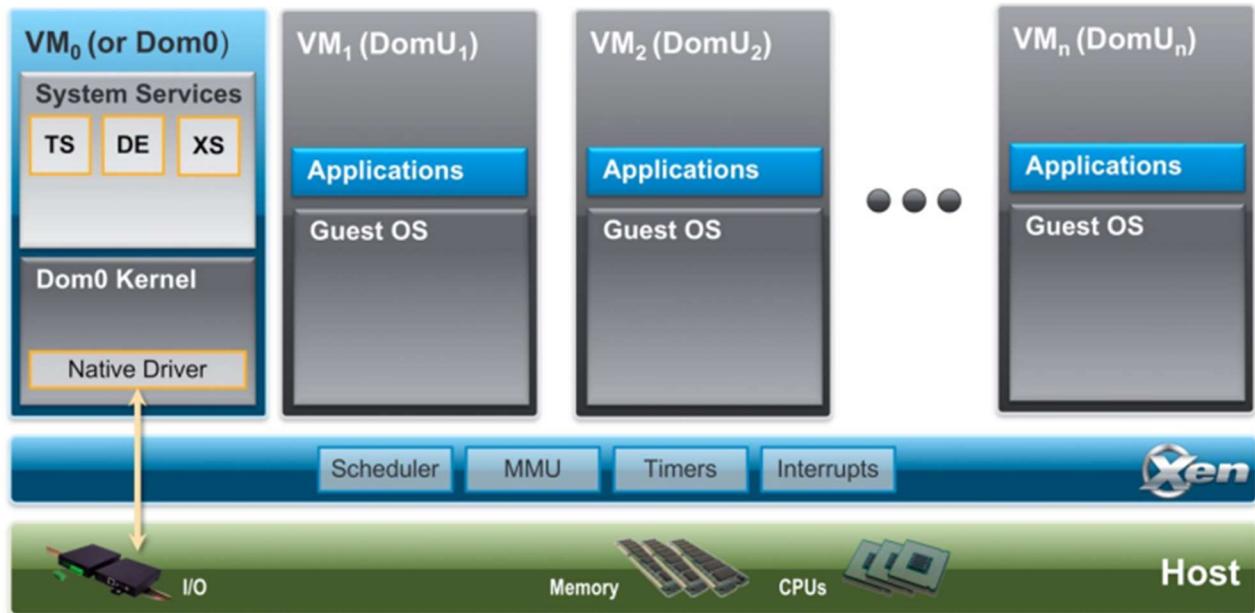
### Vantaggi:

- > È un hypervisor scritto con poche righe di codice (circa 300.000), per cui occupa poco spazio, è scalabile ed è più robusto e sicuro rispetto ad altri hypervisor.
- > Ha un'ampia comunità di sviluppatori ed è in continuo miglioramento.
- > Può essere configurato.
- > Presenta un overhead contenuto.
- > Supporta la migrazione live delle macchine virtuali.

### Svantaggi:

- > L'I/O non è particolarmente efficiente.

### Architettura:



Si hanno diversi domini che distinguiamo in:

- **DomU** (domini non privilegiati): sono le VM guest, che sono totalmente isolate dall'hardware (ad esempio non hanno accesso privilegiato all'hardware).
- **Dom0** (dominio di controllo): è un dominio speciale che si occupa dell'esecuzione delle funzioni di controllo di Xen e delle istruzioni privilegiate. È un dominio obbligatorio, contiene i driver nativi per l'accesso alle risorse, può accedere direttamente all'hardware e può interagire con le VM.

Tra i componenti del dominio di controllo troviamo XenStore e Toolstack.

- > **XenStore**: è uno storage condiviso tra i vari domini e gestito dal demone di sistema Xenstored. Memorizza le informazioni sullo stato e sulla configurazione delle VM. È implementato con un meccanismo a coppie chiave-valore: quando un valore viene modificato all'interno dello store, una particolare funzione notifica i listener (e.g. i driver) interessati alla relativa chiave.
- > **Toolstack**: consente all'utente di configurare e di gestire il ciclo di vita (creazione, shutdown, pausa, migrazione) di una VM. Per creare una nuova VM, l'utente fornisce un file di configurazione dove vengono descritte l'allocazione di memoria e di CPU e la configurazione dei dispositivi; dopodiché Toolstack effettua il parse di tale file e scrive le varie informazioni nello XenStore.

### Scheduling delle CPU:

Uno scheduler dell'hypervisor decide quali CPU virtuali devono essere eseguite su quali CPU fisiche. Xen permette di selezionare uno scheduler tra tutti quelli disponibili. Comunque sia, in generale, gli scheduler concorrono tutti ai seguenti obiettivi:

- Fornire un algoritmo **proportional share**: per ogni CPU virtuale viene fornita una quantità di CPU fisica proporzionale al suo numero di share (ovvero al suo peso); ciò serve a garantire una distribuzione fair delle CPU fisiche.
- Fornire un algoritmo **work-conserving**: le CPU fisiche non devono essere lasciate idle se c'è del lavoro da fare.
- Fornire uno schedulo con **bassa latenza**.

Lo scheduler di default per Xen è il **Credit scheduler**, in cui a ciascun dominio vengono assegnati un weight e un cap:

- > Weight = numero di CPU relative allocate per dominio (default: 256).
- > Cap = limite massimo di CPU fisiche che possono essere assegnate a una CPU virtuale (default: 0, che corrisponde all'illimitatezza).

Lo scheduler converte il weight in credito, che viene consumato dalle CPU virtuali nel momento in cui sono in esecuzione.

Per ciascuna CPU fisica viene mantenuta una coda di CPU virtuali, dove in testa si ha la CPU virtuale con meno credito. Inoltre, viene attuato un meccanismo di load balancing automatico per cui, se una CPU fisica rimane senza CPU virtuali in coda, controlla lo stato delle altre CPU fisiche ed eventualmente preleva da una di loro una CPU virtuale in coda (in tal modo viene anche soddisfatta la proprietà del work-conserving).

### **Confronto tra performance di diversi hypervisor**

A seguito di alcuni studi effettuati nel 2013, è emerso che:

- Per applicazioni CPU-intensive non ci sono particolari differenze tra le prestazioni di diversi hypervisor.
- Per applicazioni I/O-intensive, invece, si hanno delle variazioni significative nelle prestazioni. In ogni caso, la scelta del VMM migliore dipende dal carico di lavoro.

In conclusione, non esiste un hypervisor migliore di altri in assoluto, ma la sua scelta dipende molto dall'applicazione in esame.

### **Portabilità delle VM**

L'immagine di una macchina virtuale è una copia della VM contenente il sistema operativo, i file e le applicazioni; di conseguenza, ha dimensioni molto elevate.

Ma come è possibile importare ed esportare l'immagine di una VM in modo tale da evitare situazioni di vendor lock-in? La soluzione consiste nell'introdurre l'**Open Virtualization Format (OVF)** che consente di rappresentare la VM in modo agnostico rispetto alla piattaforma e, in particolare, introduce due file:

- Uno in formato .ova che contiene l'immagine della VM vera e propria (per cui ha dimensioni elevate).
- Uno in formato .ovx che descrive la configurazione della VM in un file XML (per cui ha dimensioni ridotte).

### **Resizing dinamico delle VM**

Che cosa può essere ridimensionato in modo dinamico all'interno di una macchina virtuale, ovvero senza arrestare e riavviare la VM stessa (warm resize)?

- > Il numero di CPU virtuali.
- > La memoria.

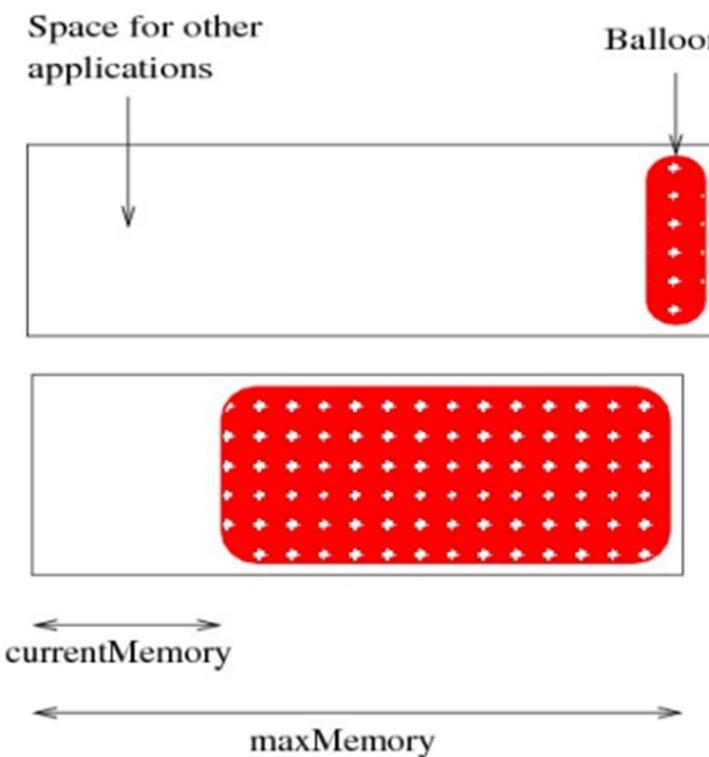
#### Resizing del numero di CPU:

I sistemi Linux supportano il meccanismo di **hot-plug / hot-unplug**, tramite cui è possibile modificare le impostazioni sulle CPU virtuali utilizzate all'interno della directory sys/devices/system/cpu.

#### Resizing della memoria:

È basato sul **memory ballooning**, un meccanismo utilizzato dagli hypervisor in cui viene gestita la dimensione di un "palloncino" all'interno dell'area di memoria relativa a una VM: più si gonfia il palloncino,

meno memoria resta a disposizione della VM. Se l'hypervisor, gonfiando il palloncino, sottrae un'area di memoria che la VM stava usando, i dati lì presenti subiscono uno swap-out su disco affinché non vadano persi.



### **Migrazione dinamica delle VM**

Consiste nel trasferire le VM da una macchina fisica e l'altra durante la loro esecuzione.

#### Vantaggi:

- > Consolida l'infrastruttura.
- > Garantisce un certo grado di flessibilità in caso di fallimenti.
- > Permette di bilanciare il carico.

#### Svantaggi:

- > Non è supportata da tutti i VMM.
- > Comporta un overhead di migrazione non trascurabile.
- > Nell'ambito WAN (Wide Area Network) non è banale.

Prima di avviare la migrazione live, si ha una fase di **setup**, in cui si seleziona l'host di destinazione (ovvero il prossimo host che dovrà ospitare la VM da migrare). Dopodiché si passa alla migrazione vera e propria.

Ma cosa è possibile migrare? Lo storage, le connessioni di rete e la memoria.

#### Migrazione dello storage:

Si hanno due scenari differenti:

- Si ha uno storage condiviso tra host sorgente e host destinazione.
- Non si ha uno storage condiviso: in tal caso, il VMM sorgente salva tutti i dati della VM sorgente in un file di immagine che verrà poi trasferito sull'host destinazione.

#### Migrazione delle connessioni di rete:

La VM sorgente ha un indirizzo IP virtuale e un indirizzo MAC virtuale. Se host sorgente e host destinazione

si trovano nella medesima sottorete IP, è sufficiente aggiornare le tabelle ARP; in caso contrario, si utilizza un IP tunnel tra host sorgente e host destinazione che viene interrotto a fine trasferimento.

#### Migrazione della memoria:

Esistono tre diversi approcci:

1) **Pre-copy:** è conveniente quando l'applicazione è read intensive, e prevede tre fasi.

- Fase di pre-copy: il VMM copia in modo iterativo le pagine da VM sorgente a VM destinazione mentre la VM sorgente è in esecuzione. In particolare, all'iterazione n vengono copiate le pagine modificate durante l'iterazione n-1.

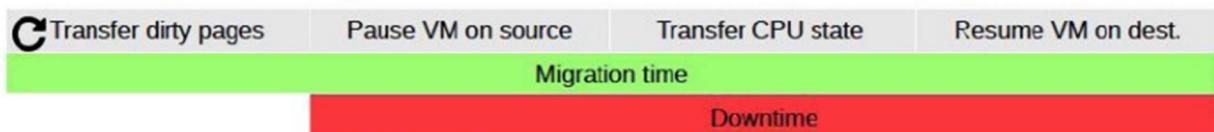
- Fase di stop-and-copy: la VM sorgente viene fermata e vengono copiate le pagine dirty e lo stato della CPU e dei device driver. Durante questa fase si ha un downtime, che può durare da qualche millisecondo a qualche secondo, in funzione della dimensione della memoria, del tipo di applicazione e della banda di rete.

- Fasi di commitment e reactivation: la VM sorgente viene rimossa, mentre la VM destinazione carica lo stato e riprende l'esecuzione.

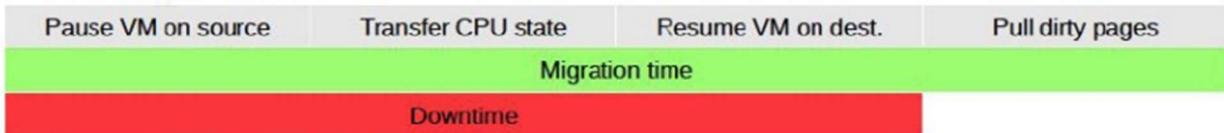
2) **Post-copy**: è conveniente quando l'applicazione è write intensive. Prevede che lo stato della CPU e dei device driver venga spostato sull'host destinazione immediatamente; dopo che l'host destinazione ha acquisito il controllo, le pagine di memoria vengono acquisite on-demand (man mano che le applicazioni le utilizzano). Questo, tuttavia richiede che la VM sorgente rimanga attiva per una quantità di tempo non nota a priori e comporta che, per ogni nuova pagina di memoria acceduta, si verifica un page fault che deve essere risolto in rete (gravando così sulle prestazioni).

3) **Ibrido**: consiste nel fare il post-copy dopo una fase di pre-copy parziale; è un approccio efficiente nel momento in cui, durante il pre-copy, viene trasferito verso l'host destinazione un sottoinsieme delle pagine più accedute in lettura (in modo tale da prevenire una gran quantità di page fault nella fase successiva).

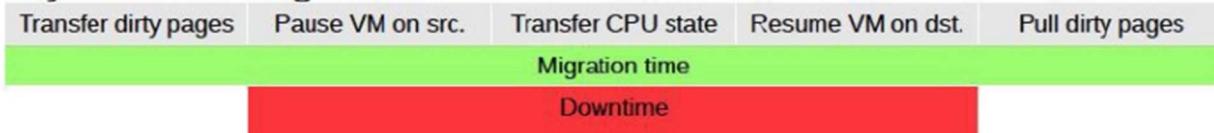
- **Pre-copy Live Migration**



- **Post-copy Live Migration**



- **Hybrid Live Migration**



#### **Migrazione delle VM in ambienti WAN**

Vediamo come è possibile migrare lo storage e le connessioni di rete in un ambiente WAN.

#### Migrazione dello storage:

Può avvenire secondo tre possibili approcci:

-> **Storage condiviso**: il tempo di accesso allo storage potrebbe essere troppo elevato.

-> **Fetching on-demand**: viene trasferito prima un sottoinsieme di blocchi verso l'host destinazione e poi, per i restanti, si procede on-demand; in tal caso è necessario che l'host sorgente sia sempre funzionante.

-> **Pre-copy / write throttling**: viene effettuato un pre-copy del disco sull'host destinazione mentre la VM continua l'esecuzione, poi si tiene traccia delle operazioni di write che avvengono lato sorgente e infine si riportano tali scritture lato destinazione. Se la velocità delle operazioni di scrittura è troppo elevata, la si può ridurre mediante il meccanismo del throttling, in modo tale da agevolare la migrazione.

#### Migrazione delle connessioni di rete:

Anch'essa può avvenire secondo tre possibili approcci:

-> **IP tunneling**: viene installato un IP tunnel tra l'host sorgente e l'host destinazione che serve a inoltrare verso la destinazione tutti i pacchetti inviati verso la sorgente. Una volta che la migrazione è terminata, viene aggiornata la entry DNS opportuna con l'indirizzo IP dell'host destinazione. Infine, l'IP tunnel viene rilasciato una volta che non c'è più alcuna connessione con l'host sorgente.

Chiaramente questo approccio non funziona se la VM sorgente va in crash.

-> **Virtual Private Network (VPN)**.

-> **Software-Defined Networking (SDN)**.

#### **Virtualizzazione di sistema operativo**

Permette di eseguire molteplici ambienti di esecuzione tra loro isolati all'interno di un singolo sistema operativo. Tali ambienti sono detti container, ciascuno dei quali ha il proprio insieme di processi, i propri file system, i propri utenti, le proprie interfacce di rete, le proprie tabelle di routing, le proprie regole di firewall e così via. Tuttavia, i container condividono il medesimo sistema operativo.

Esistono diversi meccanismi per creare un container, tra cui:

- **Chroot** (change root directory): consente di cambiare la root directory per il processo in esecuzione e per tutti i suoi figli.

- **Namespaces**: permette di isolare cosa un insieme di processi vede del sistema operativo.

- **Cgroups**: permette di controllare e/o limitare le risorse utilizzate da ciascun insieme di processi all'interno dell'ambiente di esecuzione.

#### Namespaces:

Esistono 6 tipologie differenti di namespaces:

1) **Mnt**: differenzia la root directory.

2) **Pid**: identifica i processi che possono essere visti nel container.

3) **Network**: permette a ciascun container di avere il suo stack di rete dedicato (con la relativa tabella di routing, gli indirizzi IP, il firewall e così via).

4) **User**: identifica gli utenti e i gruppi del container.

5) **Uts** (Unix timesharing): fornisce un hostname e un domain name dedicati.

6) **Ipc**: fornisce una memoria condivisa dedicata per IPC (inter-process communication).

#### Vantaggi rispetto alla virtualizzazione di sistema:

-> È molto più leggera, per cui minimizza i tempi di startup e shutdown e ha un'immagine più piccola (detta footprint) che, di fatto, non include il kernel del sistema operativo; di conseguenza, a parità di host fisico, possono essere ospitati più container che macchine virtuali.

-> Non porta a gravi perdite di performance: infatti, le applicazioni invocano le system call direttamente, senza indirezione del VMM.

-> Incentiva la portabilità e l'interoperabilità delle applicazioni, che possono essere spostate migrando i container; perciò, le applicazioni all'interno di un container sono indipendenti dal contesto di esecuzione.

-> Consente a container diversi di condividere pagine di memoria; tuttavia, questo può rappresentare anche uno svantaggio dal punto di vista della sicurezza.

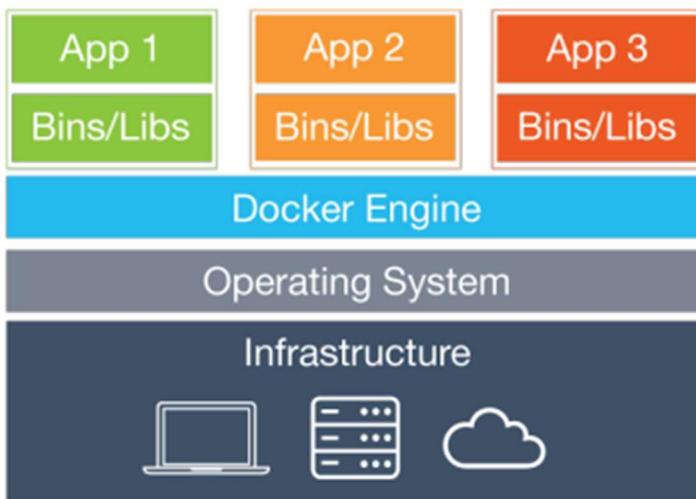
-> È uno strumento fondamentale per DevOps, poiché consente agli sviluppatori di condividere le immagini dei container per effettuare il deployment delle applicazioni verso diverse infrastrutture.

### Svantaggi rispetto alla virtualizzazione di sistema:

- > Porta a una minore flessibilità: ad esempio, non permette di eseguire più kernel contemporaneamente sulla medesima macchina fisica.
- > Le applicazioni devono essere native per il sistema operativo supportato dalla macchina fisica. Altrimenti, si potrebbe installare una VM col sistema operativo desiderato e inserire i container al suo interno; così però si creerebbe un doppio grado di indirezione che porterebbe a un crollo delle prestazioni.
- > Porta a un minore isolamento tra i container (ad esempio è possibile condividere anche porzioni del file system tra i container).
- > Espone i container a una maggiore vulnerabilità poiché la loro superficie d'attacco è il sistema operativo guest, che ha molte più linee di codice di un VMM; inoltre, se viene attaccato il kernel, tutti i container sono compromessi.

### Docker

È un ambiente di virtualizzazione a livello di sistema operativo che offre container che includono le applicazioni e tutte le loro dipendenze.



È scritto in Go e sfrutta i meccanismi di virtualizzazione offerti da Linux (namespaces e cgroups).

### Componenti:

- **Docker daemon:** rappresenta il server e non è altro che un'interfaccia utilizzabile dal client mediante la command line interface (CLI).
- **Docker engine:** è un'applicazione client-server composta dal Docker daemon, dal CLI e dalle REST API, che esportano le funzionalità che i programmi possono usare per interagire col Docker daemon; nella pratica serve a gestire il container a basso livello.
- **Registry:** è un repository di container.
- **Builder:** è un componente che permette di costruire l'immagine del container (che poi verrà eseguita dal Docker engine).

### Docker image:

È un template accessibile in sola lettura usato per creare un container Docker e può contenere tutte le dipendenze e le configurazioni del container stesso.

Docker può costruire un'immagine automaticamente con l'utilizzo di un file testuale (detto **Dockerfile**). Le immagini possono essere caricate o scaricate da registry privati o pubblici. Il nome di ciascuna Docker image è del tipo `[registry/] [/user/] name[:tag]` dove:

- Registry = repository da cui l'immagine è stata scaricata.

- User = utente che ha creato l'immagine.
- Tag = versione dell'immagine (default: latest).

Dopodiché, da una stessa immagine è possibile creare anche più container, ciascuno dei quali rappresenta un'istanza eseguibile della Docker image.

#### Dockerfile:

Un'immagine può essere creata a partire da un Dockerfile e da un parametro context, dove:

- > Il Dockerfile è composto da una sequenza di istruzioni che indicano come deve essere assemblata l'immagine.
- > Il parametro context rappresenta un insieme di file (e.g. applicazioni, librerie).

Vediamo alcune possibili istruzioni del Dockerfile:

- FROM: specifica l'immagine parent (istruzione obbligatoria).
- RUN: esegue un comando in un nuovo layer del container.
- ENV: imposta le variabili d'ambiente.
- EXPOSE: imposta le porte su cui il container si metterà in ascolto.
- CMD: imposta i comandi di default da eseguire all'avvio.

#### Layer in una Docker image:

Ciascuna immagine è composta da una serie di layer. Docker usa degli **union file system** per combinare questi layer in un'unica vista unificata. La stratificazione risulta vantaggiosa poiché i layer possono essere utilizzati per più container (per cui viene risparmiata memoria).

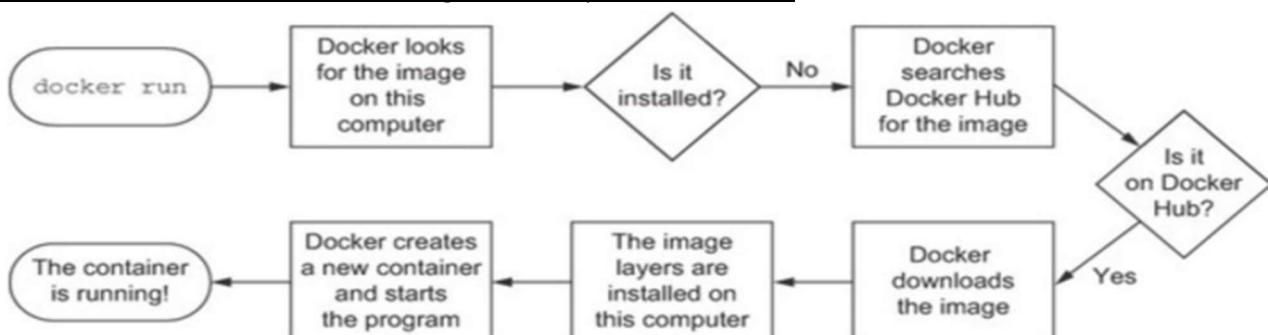
Ciascun layer rappresenta un'istruzione nella Docker image eccetto CMD, che modifica semplicemente i metadati dell'immagine. Inoltre, ciascun layer tranne l'ultimo è read only; l'ultimo è detto **container layer** e viene aggiunto quando il container viene creato. In particolare, qualunque modifica apportata al container (e.g. scrittura su un file) viene riportata sul container layer.

#### Docker volumes:

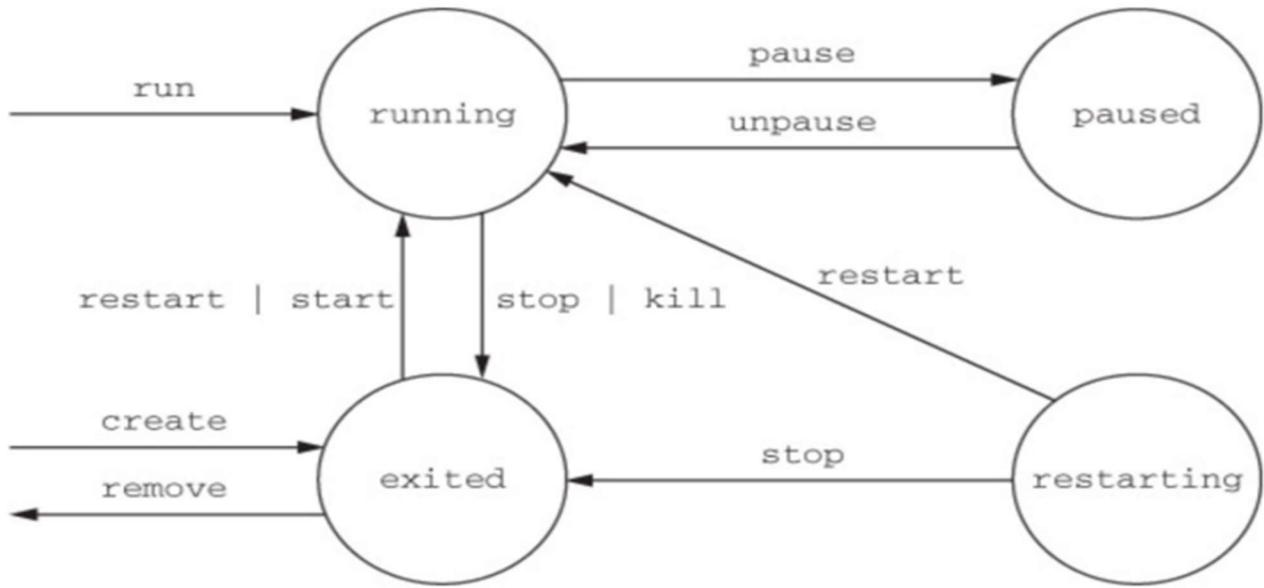
In realtà i container dovrebbero essere stateless. Idealmente, pochissimi dati dovrebbero essere scritti nel container layer, mentre si preferisce l'utilizzo dei Docker volumes esterni al container; tra l'altro, i Docker volumes presentano i seguenti vantaggi:

- > Sono completamente gestiti da Docker.
- > Sono facili da migrare.
- > È facile crearne un back-up.
- > Possono essere gestiti utilizzando la riga di comando CLI Docker oppure le Docker API.
- > Possono essere usati su qualunque sistema operativo.
- > Possono essere condivisi tra più container.
- > Il loro contenuto può essere pre-popolato e/o cifrato.
- > Mentre il container layer svanisce nel momento in cui il container viene disattivato, i dati scritti nei Docker volumes sopravvivono anche dopo.

#### Esecuzione di un container con immagine non disponibile in locale:



### State diagram dei container Docker:



### Applicazioni Docker multi-container:

Sono applicazioni che utilizzano più di un container. Esistono diversi modi per avviare:

- **Definire la rete in modo esplicito** e collegare i container alla rete.
- **Docker Compose** (tool che permette il deployment di un'applicazione multi-container su un'unica macchina).
- **Docker Swarm** (tool che permette il deployment di un'applicazione multi-container su molteplici host).
- **Kubernetes** (altro tool che permette il deployment di un'applicazione multi-container su molteplici host; lo analizzeremo più avanti).

### Docker Compose:

Consente di definire la composizione dei container su un unico Docker engine; ciò viene fatto scrivendo sul file docker-compose.yml.

### Docker Swarm:

Orchestra i container su macchine differenti, e lo fa gestendo un cluster di Docker engine chiamato **swarm**. D'altra parte, un servizio comprende uno o più **task**, ciascuno dei quali rappresenta un container.

Infine, all'interno del cluster di Docker engine si ha un insieme di nodi, che si suddividono in:

- > **Nodi worker**, che eseguono i task.
- > **Nodi manager**, che gestiscono il cluster e schedulano i task tra i nodi worker; tipicamente sono più di uno per incentivare la tolleranza ai guasti, per cui può essere necessario implementare un algoritmo di consenso per prendere delle decisioni.

Le funzionalità di Docker Swarm sono:

- **Scaling** sul numero di task per ogni servizio.
- **State reconciliation**: lo swarm monitora lo stato del cluster e riconcilia qualunque differenza tra lo stato effettivo e lo stato desiderato.
- **Multi-host networking**: Docker Swarm permette di specificare un'overlay network tra i servizi.
- **Load balancing**: lo swarm permette di stabilire come distribuire i container tra i nodi.

### **Resizing dinamico dei container**

È possibile effettuare il resizing dei container modificando le impostazioni per il cgroup "on the fly". Anche qui, come nel caso delle VM, è possibile ridimensionare il numero di CPU e la memoria.

### Resizing del numero di CPU:

Prendendo in considerazione Docker, esistono più opzioni:

- > Numero di CPU da assegnare a ciascun container (vincolo hard).
- > Proporzione su quale porzione di CPU assegnare a ciascun container (vincolo soft).

### Resizing della memoria:

Docker, per ogni container, può imporre due tipi di limitazioni per la quantità di memoria:

- > Limitazione hard (viene imposto un limite che non può mai essere superato).
- > Limitazione soft (viene imposto un limite solo in certe condizioni, ad esempio solo se c'è un alto grado di contesa).

### **Migrazione dinamica dei container**

In modo analogo alle macchine virtuali, per migrare un container è necessario:

- Salvare lo stato.
- Trasferire lo stato, l'immagine e i volumi del container e le connessioni di rete.
- Ripristinare lo stato.

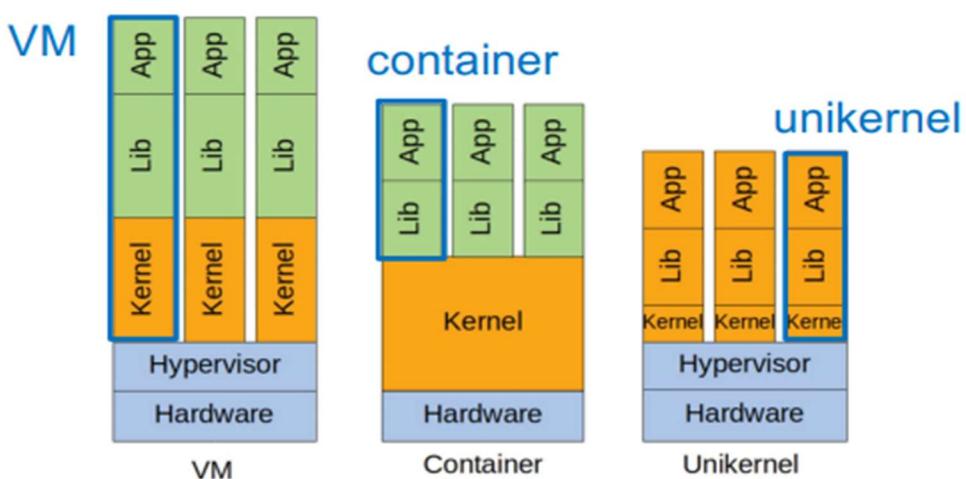
Questi passaggi vengono tuttavia effettuati con le applicazioni "freezeate", per cui si ha comunque un tempo di downtime, seppur minore rispetto a quello delle VM. In ogni caso, il trasferimento può avvenire sia con la tecnica pre-copy, che con la tecnica post-copy.

Un tool che supporta la migrazione live è **CRIU**, e lo fa mediante le tecniche di checkpointing e restore: in pratica inserisce un checkpoint sullo stato del container, il quale viene freezato; dopodiché CRIU raccoglie le informazioni riguardo lo stato della CPU, della memoria e dei processi, le passa all'host destinazione e, infine, il container viene ripristinato a partire dal checkpoint.

### **Unikernel**

Insieme ai sistemi operativi lightweight rappresenta una tecnica di virtualizzazione molto leggera e a basso overhead. Infatti, è utile per i microservizi, il serverless computing, l'IoT e il fog/edge computing.

Più specificatamente, l'unikernel è una macchina virtuale single-purpose e single-language eseguita su un hypervisor e ospitata in un ambiente minimale. Di fatto, il kernel di tale VM, le applicazioni e un insieme minimale di librerie sono collassati all'interno di un unico address space.



### Vantaggi:

- > La sua immagine (footprint) è estremamente leggera.
- > È veloce (non si ha context switching, che consiste nell'interruzione dell'esecuzione di un processo per riprendere quella di un altro processo).

- > È sicuro (la superficie di attacco è molto ridotta).
- > È caratterizzato da avvii veloci (misurati in millisecondi).

Svantaggi:

- > Funziona solo sugli ambienti virtuali basati su hypervisor.
- > Il debugging è molto povero.
- > Supporta un unico linguaggio di programmazione.

Prodotti unikernel:

- > **OS<sup>V</sup>**, che supporta C++, Go, Python, Java, ecc.
- > **IncludeOS**, che supporta C++.
- > **MirageOS**, che supporta Ocaml.

# MICROSERVIZI

## Introduzione

I microservizi rappresentano uno stile architetturale emergente per quelle applicazioni distribuite che vengono sviluppate come una collezione di servizi debolmente accoppiati. Tali servizi sono delle unità piccole e auto contenute con un'interfaccia ben definita, e possono comunicare mediante chiamate RPC o con un sistema di tipo publish / subscribe.

I microservizi, inoltre, possono essere visti come complementari della virtualizzazione basata su container: quello che si fa molto spesso nella pratica è inserire un'istanza di microservizio su ogni container, in modo tale da migliorare la scalabilità, isolare le istanze di microservizi e applicare dei limiti per le risorse sulle istanze di microservizi. Tuttavia, questo meccanismo richiede un'orchestrazione dei container per gestire le applicazioni multi-container.

## Service Oriented Architecture (SOA)

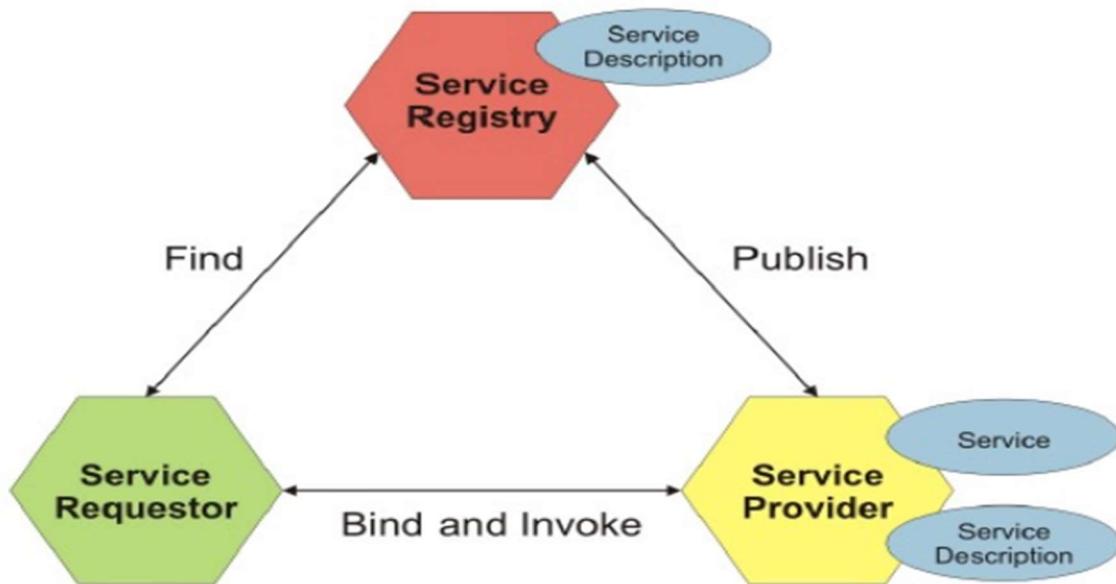
È l'antenato dei microservizi ed è un paradigma architetturale per progettare sistemi software distribuiti i cui componenti devono essere debolmente accoppiati e possono avere amministratori diversi.

Presenta le seguenti proprietà:

- Offre una vista logica dell'applicazione.
- La comunicazione tra i componenti avviene tramite scambio di messaggi.
- È *platform neutral*.

SOA prevede tre entità che interagiscono tra loro:

- > **Service requestor** (o consumer): richiede l'esecuzione del servizio.
- > **Service provider**: fornisce e rende disponibile il servizio.
- > **Service registry**: consente al provider di pubblicare i servizi e permette al requestor di trovare il servizio desiderato.



## Web service:

Rappresenta l'implementazione più nota di SOA ed è un sistema software progettato per supportare l'interazione tra componenti sulla rete.

Qui la comunicazione tra i servizi avviene mediante il protocollo SOAP (Simple Object Access Protocol), che è un protocollo a livello applicativo basato su http.

### Differenze tra SOA e i microservizi:

- Mentre SOA è una tecnologia heavyweight, i microservizi rappresentano una tecnologia lightweight.
- Mentre SOA è associato ai protocolli dei web service (come SOAP), i microservizi tipicamente si basano sui protocolli REST e http.
- SOA è principalmente visto come una soluzione di integrazione tra applicazioni, mentre i microservizi sono tipicamente sfruttati per costruire applicazioni software individuali.

### **Decomposizione di un'applicazione**

Per decomporre un'applicazione monolitica (= applicazione in cui interfaccia utente e codice di accesso ai dati sono combinati in un unico programma da un'unica piattaforma) in molteplici microservizi, è possibile seguire uno dei seguenti pattern:

- > Decomposizione basata sulle *business capability*.
- > Decomposizione basata sui sottodomini (domain driven design – DDD).
- > Decomposizione basata sui casi d'uso.
- > Decomposizione basata sulle entità e sulle risorse.

### **Scalabilità dei microservizi**

Per conseguire la scalabilità dei microservizi, è necessario utilizzare molteplici istanze di uno stesso microservizio e bilanciare le richieste tra le varie istanze. Per migliorare poi la scalabilità, è bene avere dei servizi stateless anziché stateful.

In un servizio stateless lo stato viene gestito da un'entità esterna al servizio, in modo tale da rendere l'applicazione più tollerante ai fallimenti dei servizi, oltre che più scalabile. In tal caso, il load balancing delle richieste può anche essere implementato in un modo semplice (e.g. con una politica random o round-robin); nel caso del servizio stateful, invece, se lo stato è partizionato tra le diverse repliche, è richiesto che il load balancer sia **request aware**, ovvero sia in grado di selezionare esattamente la replica capace di soddisfare una data richiesta, ad esempio mediante un algoritmo di hashing.

### **Integrazione dei microservizi**

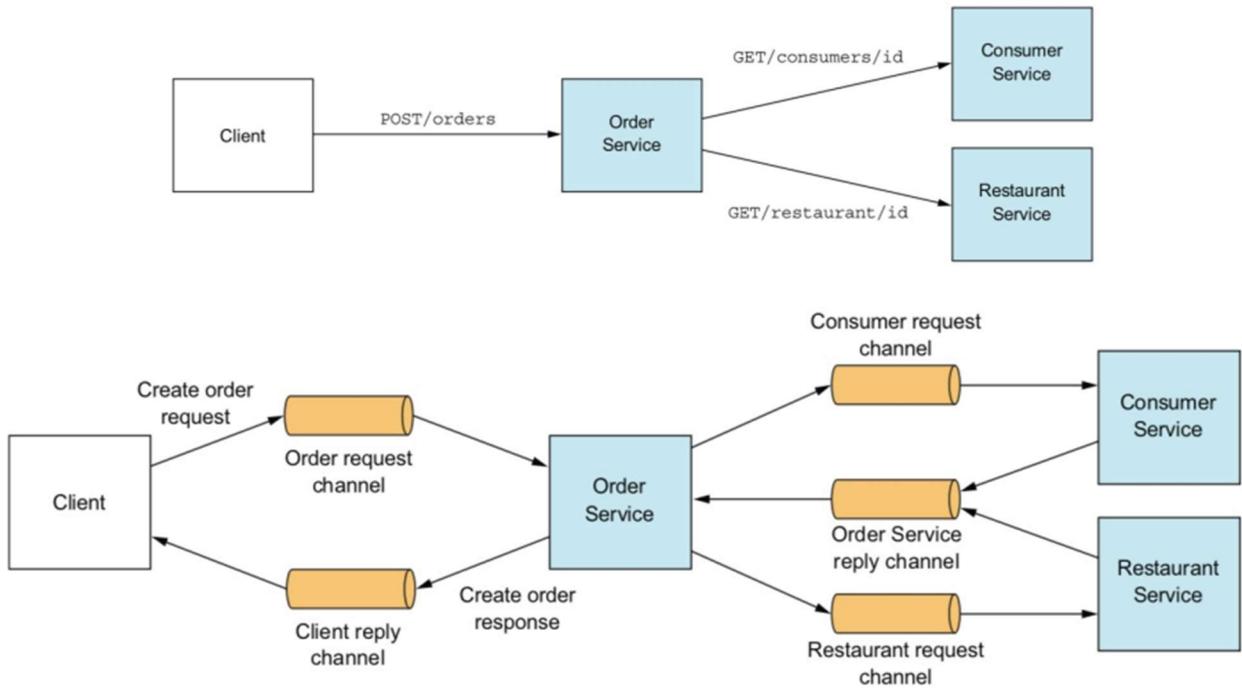
Consideriamo due problemi legati all'integrazione dei microservizi:

- > Comunicazione sincrona vs asincrona.
- > Orchestrazione vs coreografia.

#### Comunicazione sincrona vs asincrona:

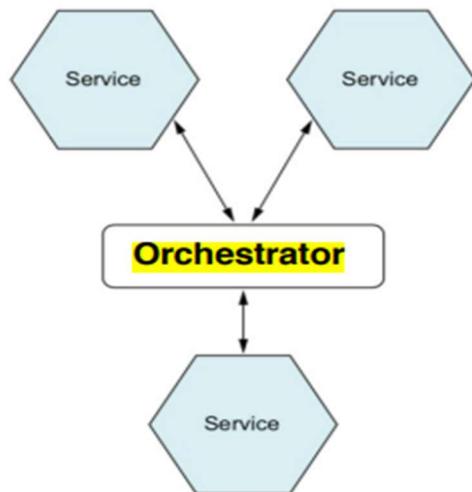
- **Comunicazione sincrona:** è in stile richiesta / risposta (e.g. REST, gRPC) e, in quanto tale, potrebbe presentare una disponibilità limitata.
- **Comunicazione asincrona:** è in stile event-drive ed è basata sul meccanismo a coda di messaggi o su quello publish / subscribe. L'interazione può dunque essere uno-a-uno oppure uno-a-molti.

Nella prossima pagina saranno presentati un esempio di comunicazione sincrona e uno di comunicazione asincrona.

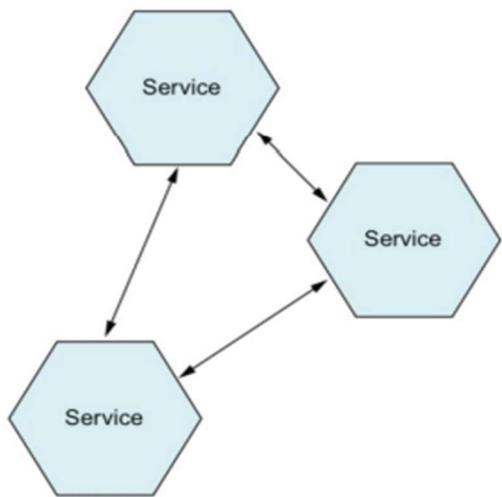


### Orchestrazione vs coreografia:

- **Orchestrazione:** un singolo processo centralizzato (detto orchestratore, conduttore o message broker) coordina l'interazione tra i vari servizi, ed è in particolar modo responsabile dell'invocazione e della combinazione dei servizi stessi, i quali possono non essere consapevoli della loro composizione. Trattasi di un pattern semplice e popolare ma è soggetto a single point of failure, bottleneck, alto accoppiamento tra i servizi ed elevata latenza nella comunicazione.



- **Coreografia:** i microservizi sono maggiormente disaccoppiati ma, se devono comunicare, devono conoscersi tra loro (a meno che utilizzano code di messaggi); la comunicazione avviene attraverso uno scambio diretto di messaggi. Trattasi di un pattern più flessibile dell'orchestrazione ma il monitoraggio dei servizi è più complesso e l'implementazione dei meccanismi (come la garanzia di consegna) è più difficile.



### **Design pattern per i microservizi**

Esaminiamo cinque design pattern: circuit breaker, database per service, saga, log aggregation e distributed request tracing.

#### Circuit breaker:

**Problema:** come evitare che il fallimento di un servizio crei un effetto domino sugli altri servizi?

**Soluzione:** il servizio client invoca il servizio remoto sfruttando un proxy che funziona in modo analogo al breaker di un circuito elettrico: quando si raggiunge un certo numero di fallimenti consecutivi, il proxy “apre il circuito” e rende impossibile l’invocazione del servizio remoto finché non scatta un timeout. Allo scadere del timeout, il proxy mette a disposizione un numero fissato di tentativi per contattare il servizio remoto: se tutti i tentativi vanno a buon fine, il proxy “chiude il circuito” ripristinando così le normali operazioni; in caso contrario, il circuito rimane aperto e il timer viene riavviato.

#### Database per service:

**Problema:** quale architettura per i database selezionare?

**Soluzione:** mantenere i dati di ciascun microservizio privati per quel microservizio (cosicché ogni microservizio possieda un proprio database). In tal modo, si ha un basso accoppiamento tra i servizi e ciascuno di loro può appoggiarsi a una tecnologia di database differente; tuttavia, le transazioni che coinvolgono microservizi differenti sono farraginose da gestire.

#### Saga:

**Problema:** ciascun microservizio ha il proprio database ma alcune transazioni coinvolgono più servizi: come assicurare la consistenza dei dati tra i servizi?

**Soluzione:** implementare ciascuna transazione che coinvolge più servizi come una **saga**. Una saga non è altro che una sequenza di transazioni locali. Ciascuna transazione locale aggiorna il relativo database e pubblica un messaggio che triggerà la transazione locale successiva. Se una qualche transazione locale fallisce, la saga esegue una serie di operazioni compensatorie che effettuano l’undo delle modifiche apportate dalle transazioni locali precedenti: in tal modo, vale il principio all-or-nothing nell’ambito globale. La saga può essere coordinata mediante orchestrazione oppure tramite coreografia.

#### Log aggregation:

**Problema:** come effettuare il monitoraggio dei microservizi?

**Soluzione:** utilizzare un servizio di log centralizzato che aggrega i log dei vari microservizi. Purtroppo, si tratta di una soluzione centralizzata e la gestione di una gran quantità di log richiede un’infrastruttura complessa.

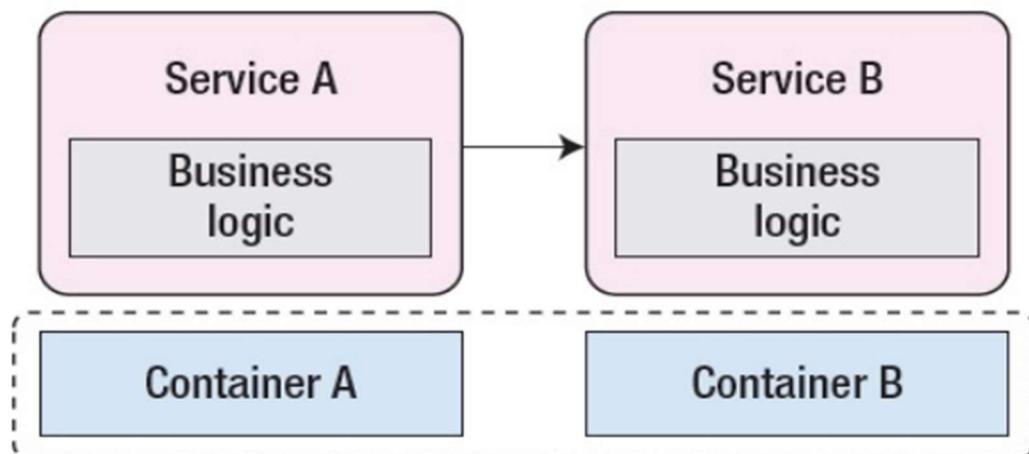
### Distributed request tracing:

**Problema:** come effettuare il monitoraggio dei microservizi?

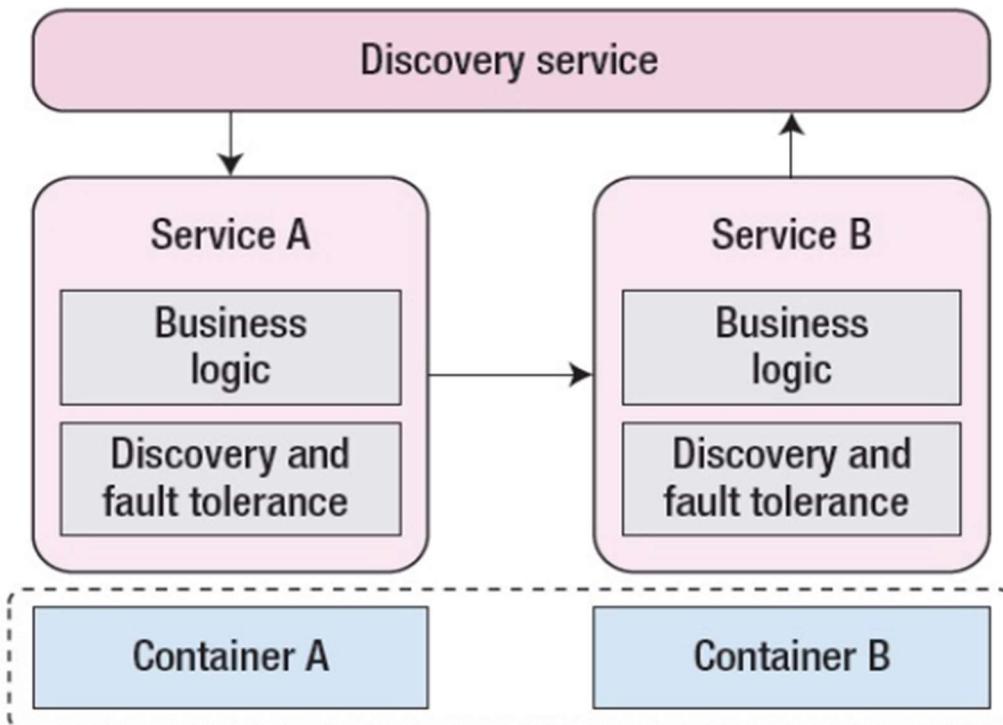
**Soluzione:** assegnare a ciascuna richiesta un ID univoco e aggiungere tale ID alle tracce di esecuzione dei microservizi che sono coinvolti in tale richiesta (ovvero il servizio che l'ha generata e i servizi che hanno effettuato le operazioni). Dalle diverse tracce di esecuzione è possibile ottenere il flusso di esecuzione congiunto; tuttavia, questa operazione, seppur necessaria, richiede un'infrastruttura significativa.

### **Generazioni di architetture a microservizi**

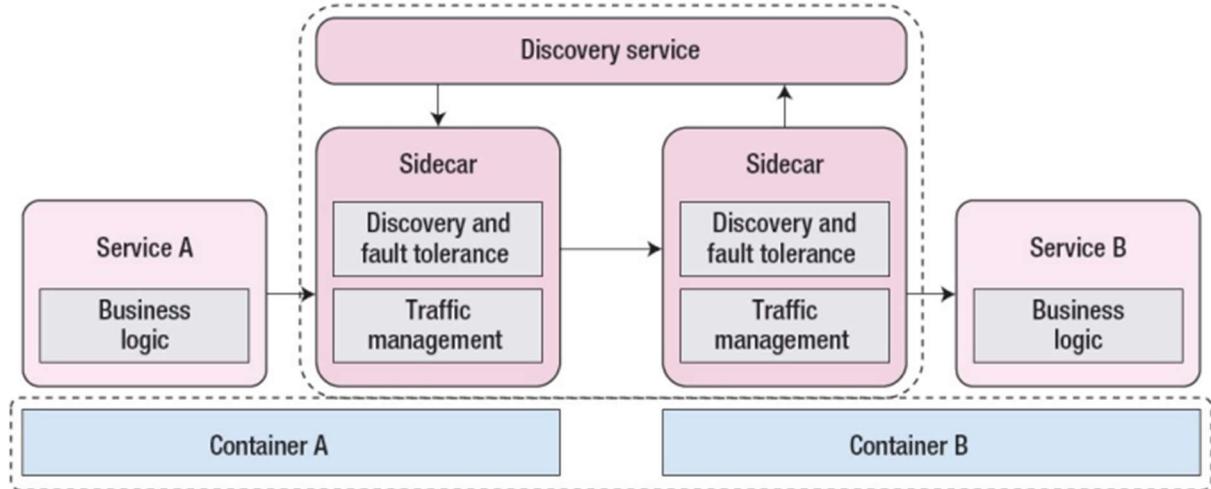
#### 1° generazione:



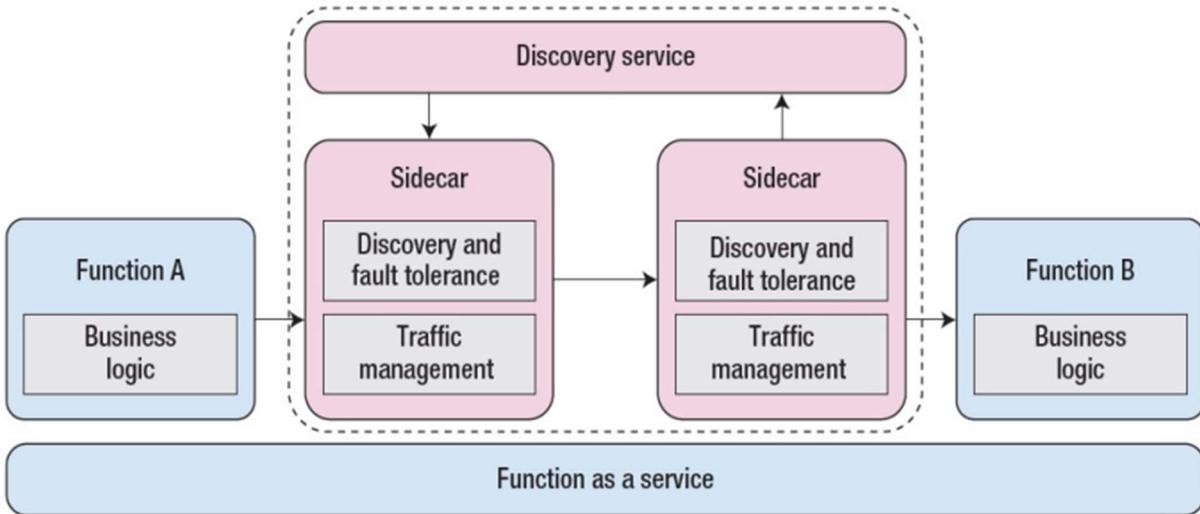
#### 2° generazione:



### 3° generazione:



### 4° generazione (basata su FaaS e serverless computing):



### **Serverless computing**

È un modello di cloud computing che punta ad astrarre la gestione del server e l'infrastruttura a basso livello. I programmati possano sviluppare, eseguire e gestire il codice delle applicazioni (i.e. le funzioni) senza preoccuparsi sul fornimento, sulla gestione e sullo scaling delle risorse computazionali; in altre parole, le funzioni girano in ogni caso su un qualche server ma i programmati non se ne interessano.

Il **Function as a Service (FaaS)** è una piattaforma che offre le funzioni come servizio e spesso viene utilizzato come sinonimo di serverless computing.

Esistono diversi cloud provider che offrono il serverless computing come servizio pienamente gestito. Tra questi abbiamo AWS Lambda, Azure Functions, Google Cloud Functions e IBM Cloud Functions.

### Caratteristiche del serverless computing:

- Le risorse di computing sono effimere, istanziabili on-demand e possono rimanere attive nel corso di una singola funzione. Da qui però sorge il problema del **cold start**: quando un servizio deve essere avviato ma nell'ambiente non sono ancora state istanziate le risorse, il tempo di risposta è particolarmente alto.
- L'elasticità è automatizzata (non deve essere configurata).

- Il modello di pagamento è un true **pay-per-use**: l'utente paga esclusivamente per il tempo di esecuzione vero e proprio, il che rappresenta un risparmio poiché l'applicazione non deve essere perennemente attiva.
- È **event-driven**: quando viene scaturito un evento, viene allocata dinamicamente una porzione di infrastruttura per eseguire il codice della funzione.

#### Sfide e limitazioni del serverless computing:

- Prestazioni non ottimali a causa della latenza dovuta al cold start.
- Necessità di supportare i linguaggi di programmazione con cui sono state scritte le funzioni.
- Limiti sulle risorse computazionali che dovrebbero essere compatibili con le risorse richieste dalle funzioni.
- Assenza di standard consolidati con conseguente rischio di vendor lock-in.

#### Composizione di funzioni serverless:

L'idea generale è quella di scrivere funzioni piccole, semplici e stateless, per poi comporle in un workflow per definire il flusso di esecuzione globale.

Esempio: **AWS Step Functions**, che è un servizio di orchestrazione serverless che consente agli sviluppatori di coordinare molteplici Lambda function in workflow.

Non esiste ancora una soluzione standard consolidata per definire un workflow. Comunque sia, un tentativo è dato da **CNCF Serverless Workflow**, un ecosistema che:

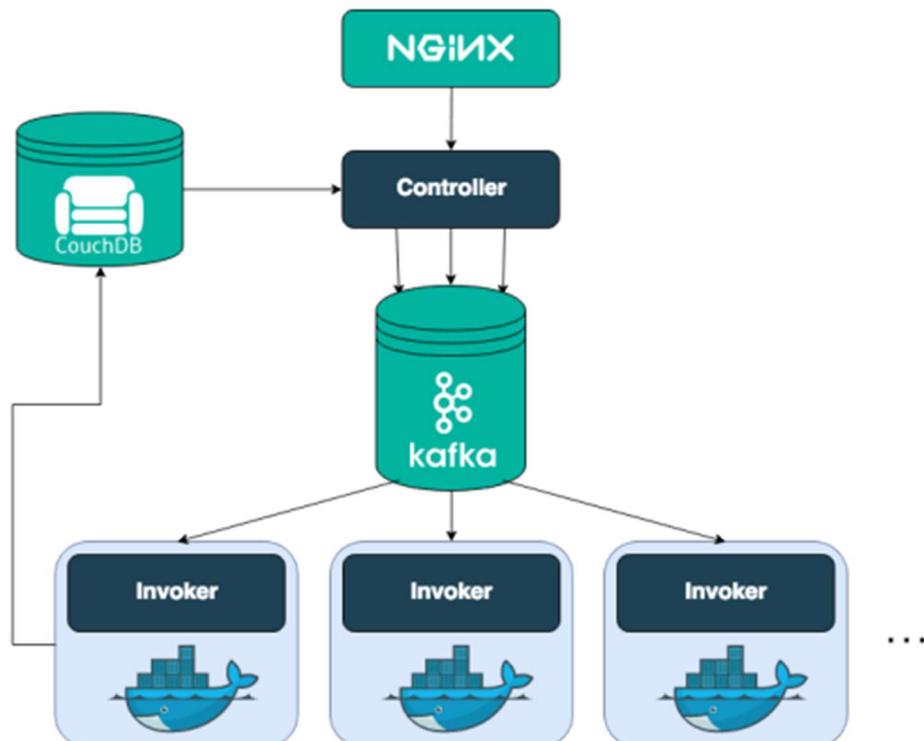
- > È vendor-neutral e open-source.
- > Permette di descrivere i workflow tramite file YAML o JSON.
- > Supporta l'invocazione sia delle funzioni RESTful sia di quelle event-triggered.
- > Supporta molteplici costrutti logici control-flow.

#### **Piattaforme FaaS open-souce**

##### OpenWhisk:

È una piattaforma distribuita che permette di eseguire funzioni ( dette anche **actions**) tramite eventi o REST API. Inoltre, supporta la composizione delle funzioni.

L'architettura interna di OpenWhisk è la seguente:

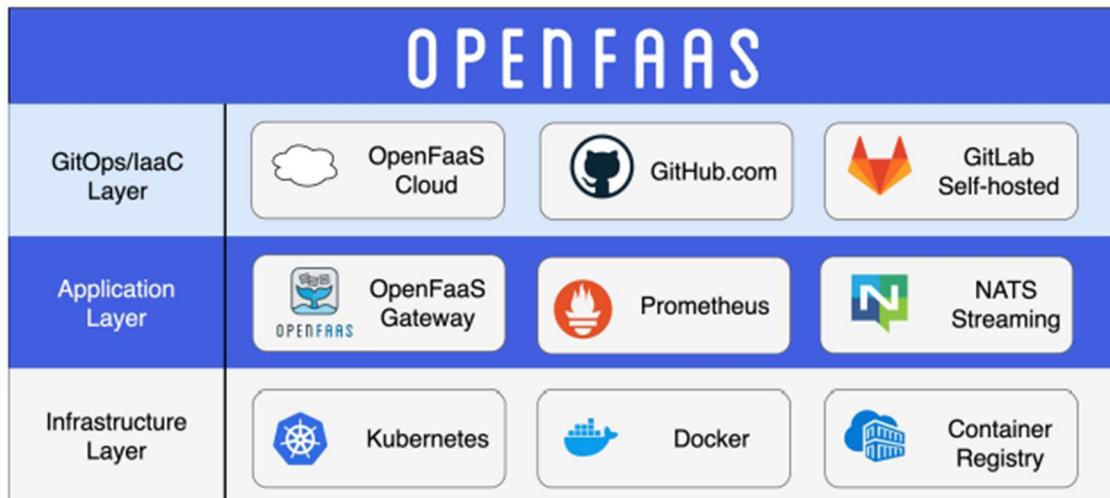


Ciascuna richiesta arriva a **NGINX** che la inoltra al **controller**. Quest'ultimo verifica se la richiesta ha i permessi per essere processata; se sì, i relativi metadati vengono memorizzati su **Kafka**, il quale istanzia dei container detti **invoker** che, infine, su occupano dell'invocazione della funzione richiesta.

#### OpenFaaS:

È un framework che si occupa di buildare le funzioni su Docker e Kubernetes. I componenti del livello applicativo di OpenFaaS sono:

- **OpenFaaS Gateway**, che è il componente che accetta le richieste per invocare un qualche servizio.
- **Prometheus**, che viene usato per l'esecuzione delle funzioni e abilita l'auto-scaling.
- **NATS**, che fornisce l'esecuzione asincrona dei task e delle funzioni di background.



#### **Kubernetes**

È una piattaforma open-source di Google utile per far comunicare e gestire container su macchine fisiche diverse. È portabile, estensibile e soddisfa le proprietà di self-optimization e self-healing.

#### Pod:

È la più piccola unità di schedulazione nonché il più piccolo oggetto di computazione deployabile. Nella pratica è un insieme di container correlati tra loro, che condividono lo storage e la rete.

Kubernetes offre un indirizzo IP a ciascun pod e un unico nome DNS a ogni insieme di pod, in modo tale da poter applicare il load balancing sui pod appartenenti allo stesso insieme.

Gli utenti organizzano i pod utilizzando le label, che sono delle coppie chiave-valore arbitrarie e descrivono una caratteristica o il ruolo di ciascun pod.

#### Architettura:

Kubernetes è organizzato secondo il pattern master-worker, dove il master è rappresentato dal **Kubernetes control plane** (che può essere anche più di uno per migliorare la disponibilità e la tolleranza ai guasti) e i worker sono rappresentati dai **nodi**.

Il control plane è suddiviso nei seguenti componenti:

- > **Kube-apiserver**: è il front-end del control plane che espone le API di Kubernetes.
- > **Etcdb**: storage chiave-valore distribuito ad alta disponibilità che memorizza le informazioni sui nodi e le informazioni sul posizionamento dei pod all'interno dei nodi.
- > **Kube-scheduler**: decide come assegnare i pod ai nodi (quindi come effettuare il posizionamento dei pod all'interno dei nodi).

D'altra parte, i nodi worker (che possono essere macchine fisiche o VM) sono suddivisi nei seguenti componenti:

- > **Kubelet**: agent che si assicura che i pod all'interno del nodo siano in esecuzione e siano healthy.
- > **Kube-proxy**: proxy di rete che gestisce il routing tra i nodi worker.

#### Auto-scaling:

Esistono diverse tecniche di auto-scaling in Kubernetes:

- **Cluster Autoscaler**: aggiusta la dimensione del cluster di Kubernetes, andando a scalare sul numero di server istanziati.
- **Vertical Pod Autoscaler (VPA)**: scala sulla quantità di CPU e memoria associata ai pod, e lo fa tramite un algoritmo che si basa sull'analisi storica dell'utilizzo di CPU e memoria. Questo tipo di scaling, però, richiede che i pod vengano riavviati, il che porta a un tempo di downtime.
- **Horizontal Pod Autoscaler (HPA)**: scala sul numero di repliche di ciascun pod, e lo fa tramite un algoritmo che si basa sull'osservazione dell'utilizzo della CPU da parte dei pod e di eventuali altre metriche customizzabili. La variazione del numero di repliche è trasparente, per cui non ci sono tempi di downtime. La politica precisamente utilizzata per lo scaling è la seguente euristica di tipo best-effort:

$$\text{desiredReplicas} = \left\lceil \text{currentReplicas} \frac{\text{currentMetricValue}}{\text{desiredMetricValue}} \right\rceil$$

Questa politica può portare a delle fluttuazioni, ovvero a un valore di *desiredReplicas* che varia in modo continuo e altalenante. Per evitare ciò, si può definire una finestra di stabilizzazione, in cui si impone che un pod che ha appena subito uno scale in / scale out non può osservare un ulteriore variazione di *desiredReplicas* per alcuni minuti.

#### Tool di Kubernetes:

- > **Kubectl**: è un tool command-line utile per eseguire comandi per deployare le applicazioni, gestire le risorse e visualizzare i log.
- > **Metrics Server**: è un tool che recupera da Kubelets le metriche sui pod e le espone mediante l'utilizzo di API affinché vengano poi utilizzate dal VPA e dall'HPA per lo scaling.

## SINCRONIZZAZIONE NEI SISTEMI DISTRIBUITI

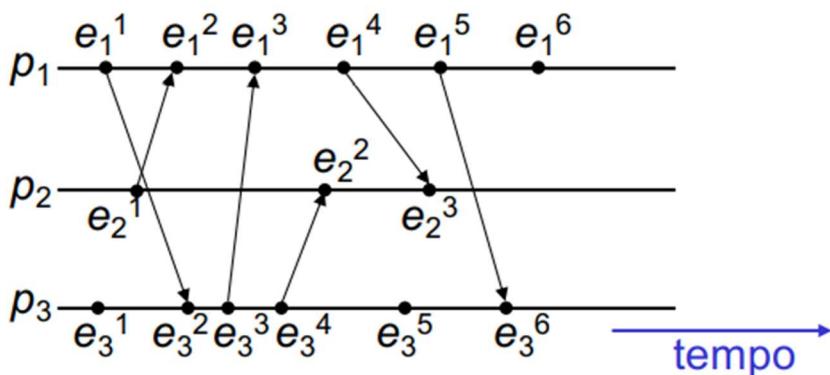
### Introduzione

I processi in un sistema distribuito vengono eseguiti su nodi connessi in una rete e cooperano per portare avanti una computazione. A tale scopo, molti algoritmi richiedono una sincronizzazione tra questi processi, i quali devono avere una nozione comune di tempo.

In un sistema distribuito, a differenza di uno centralizzato, è impossibile avere un unico clock fisico comune a tutti i processi; tuttavia, per numerosi problemi nei sistemi distribuiti è fondamentale risalire al tempo in cui gli eventi sono stati generati, o comunque stabilire l'ordinamento degli eventi stessi. Per far ciò, esistono due soluzioni:

- 1) **Sincronizzazione degli orologi fisici**: il middleware di ogni nodo del sistema distribuito aggiusta il valore del suo clock fisico in modo coerente con quello degli altri nodi o con quello di un clock di riferimento.
- 2) **Sincronizzazione degli orologi logici**: Lamport ha dimostrato come in un sistema distribuito non sia necessaria la sincronizzazione degli orologi fisici, ma lo sia solo l'ordinamento degli eventi.

L'evoluzione della computazione può essere visualizzata con un **diagramma spazio-tempo**:



-  $e_i^k$  = k-esimo evento generato dal processo  $p_i$ .

- Gli eventi possono essere **interni** (se portano a un cambiamento dello stato del processo) o **esterni** (se sono relativi a una send o una receive di un messaggio).

-  $\rightarrow_i$  = **relazione di ordinamento** tra due eventi di uno stesso processo  $p_i$ ; ad esempio,  $e_3^5 \rightarrow_3 e_3^6$  indica che l'evento  $e_3^5$  è accaduto prima dell'evento  $e_3^6$  all'interno del processo  $p_3$ .

### Sistemi distribuiti sincroni vs asincroni vs parzialmente sincroni:

-> In un sistema distribuito **sincrono**, esistono dei vincoli sulla velocità di esecuzione di ciascun processo, sul tempo di trasmissione di ciascun messaggio (sia con upper bound che con lower bound) e sul tasso di scostamento di ciascun clock dal clock reale (**clock drift rate**).

-> In un sistema distribuito **asincrono**, non si hanno vincoli né sulla velocità di esecuzione dei processi, né sul ritardo di trasmissione dei messaggi, né sul clock drift rate.

-> In un sistema distribuito **parzialmente sincrono**, viene soddisfatto uno o due dei vincoli sopra descritti.

### Soluzioni per sincronizzare i clock:

- Una prima soluzione consiste nel tentare di sincronizzare con una certa approssimazione i clock fisici dei processi; in tal modo, ogni processo può etichettare gli eventi col valore del suo clock fisico. In questo caso, si ha un timestamping basato sul tempo fisico (**clock fisico**).

- In un sistema distribuito asincrono non è possibile mantenere limitata l'approssimazione dei clock fisici: in tal caso, si introduce la nozione di clock basata su un tempo logico (**clock logico**).

### Clock fisico

All'istante di tempo reale  $t$ , il sistema operativo legge dal clock hardware  $H_i(t)$  del computer e produce il clock software  $C_i(t) = aH_i(t) + b$ , che approssimativamente misura l'istante di tempo  $t$  per il processo  $p_i$ . Se  $C_i$  si comporta abbastanza bene, può essere usato per il timestamping degli eventi di  $p_i$ . Sicuramente il clock deve avere una risoluzione a grana sufficientemente fine per distinguere una qualunque coppia di eventi rilevanti; in particolare:

$$T_{\text{risoluzione}} < \Delta T \text{ tra due eventi rilevanti}$$

Per i clock fisici introduciamo le seguenti grandezze:

- **Skew** = differenza istantanea fra il valore di due clock.
- **Drift** = fenomeno per cui i clock contano il tempo con frequenze differenti e, quindi, nel tempo divergono rispetto al tempo reale.
- **Drift rate** = differenza per unità di tempo di un clock rispetto a un orologio ideale (e.g. un drift rate di 2  $\mu\text{sec/sec}$  implica che il clock incrementa il suo valore di 1 sec + 2  $\mu\text{sec}$  ogni secondo).

### UTC (Universal Coordinated Time):

È lo standard internazionale per mantenere il tempo. È basato sul tempo atomico, occasionalmente corretto utilizzando il tempo astronomico.

### Sincronizzazione di clock fisici:

-> **Sincronizzazione esterna**: i clock  $C_i$  ( $i = 1, 2, \dots, N$ ) sono sincronizzati con una sorgente di tempo  $S$  (UTC) in modo che, dato un intervallo  $I$  di tempo reale:

$$|S(t) - C_i(t)| \leq \alpha \text{ per } 1 \leq i \leq N \text{ e per tutti gli istanti in } I$$

I clock  $C_i$  hanno **accuratezza**  $\alpha$ , con  $\alpha > 0$ .

-> **Sincronizzazione interna**: due clock  $C_i, C_j$  sono sincronizzati l'uno con l'altro in modo che:

$$|C_i(t) - C_j(t)| \leq \pi \text{ per } 1 \leq i, j \leq N \text{ nell'intervallo } I$$

I clock  $C_i, C_j$  hanno **precisione**  $\pi$ , con  $\pi > 0$ .



Se un insieme di processi è sincronizzato esternamente con accuratezza  $\alpha$ , allora è anche sincronizzato internamente con precisione  $2\alpha$ .

### Correttezza di clock fisici:

Un clock hardware  $H$  è corretto se il suo drift rate è compreso tra  $-p$  e  $+p$ , con  $p > 0$  fissato. In tal caso, l'errore che si commette nel misurare un intervallo  $[t, t']$  di istanti reali è limitato:

$$(1 - \rho)(t' - t) \leq H(t') - H(t) \leq (1 + \rho)(t' - t)$$

Per quanto invece riguarda la correttezza del clock software  $C$ , spesso è sufficiente una condizione di monotonicità:

$$t' > t \text{ implica } C(t') > C(t)$$

Ma quando devono essere sincronizzati i clock fisici?

Consideriamo due clock aventi lo stesso tasso di scostamento da UTC pari a  $\rho$ , e supponiamo che essi si scostino in senso opposto. Si ha che dopo un intervallo di tempo  $\Delta t$  i due clock si sono scostati di  $2\rho\Delta t$ . Per garantire che i due clock non differiscano mai più di  $\delta$ , occorre sincronizzarli ogni  $\delta/2\rho$  secondi.

## Sincronizzazione interna in un sistema distribuito sincrono

Avviene mediante il seguente algoritmo:

- 1) Un processo  $p_1$  manda il suo clock locale  $t$  a un processo  $p_2$  tramite un messaggio  $m$ .
- 2)  $p_2$  riceve  $m$  e imposta il suo clock a  $t+T_{trasm}$ , dove  $T_{trasm}$  è il tempo di trasmissione di  $m$ .  $T_{trasm}$  non è noto ma, essendo il sistema distribuito sincrono,  $T_{min} \leq T_{trasm} \leq T_{max}$ .
- 3) Se  $p_2$  imposta il suo clock a  $t + (T_{max}+T_{min})/2$ , il valore massimo dello skew tra i due clock è pari a:  $(T_{max}-T_{min})/2$ .

L'algoritmo può essere generalizzato per sincronizzare  $N$  processi, con skew massimo pari a:  $(T_{max}-T_{min})*(1-1/N)$ .

Per un sistema distribuito asincrono, però,  $T_{trasm} = T_{min} + x$ , dove  $x$  è un valore non noto, per cui questo algoritmo non funzionerebbe.

## Sincronizzazione mediante time service

Introduciamo ora alcuni algoritmi di sincronizzazione dei clock fisici che fanno uso del **time service** (un servizio che prevede l'uso di un ricevitore UTC o di un clock accurato).

### Algoritmo di Cristian:

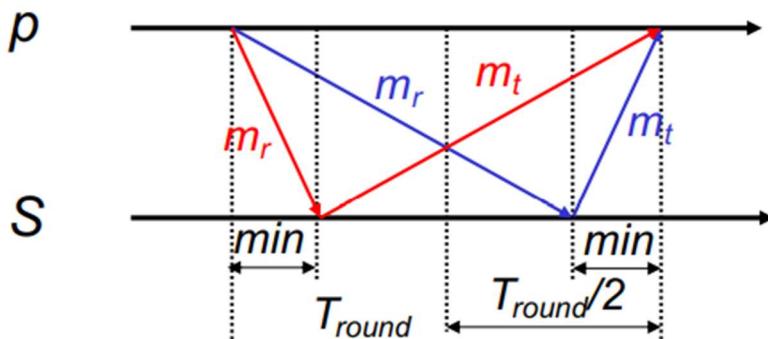
È un algoritmo che fa uso di time service centralizzati e prevede una sincronizzazione esterna.

Il time server  $S$  riceve il segnale da una sorgente UTC. Un processo  $p$  richiede il tempo con un messaggio  $m_r$ , e riceve un tempo  $t$  nel messaggio  $m_t$  da parte di  $S$ . Dopodiché  $p$  imposta il suo clock a  $t + T_{round}/2$ , dove  $T_{round}$  è l'RTT misurato da  $p$ .

Vale la pena fare le seguenti osservazioni:

- Un singolo time server rappresenta un single point of failure, per cui piuttosto si utilizza un gruppo di time server sincronizzati.
- I time server maliziosi non sono gestiti dall'algoritmo.
- L'accuratezza che si ha è ragionevole solo se  $T_{round}$  è breve, per cui l'algoritmo è adatto per lo più alle reti locali con bassa latenza. Infatti, se definiamo con  $\min$  il tempo minimo di trasmissione dei messaggi tra il processo  $p$  e il time server  $S$ , allora l'istante di tempo in cui  $S$  riceve  $m_r$  e invia  $m_t$  è compreso nell'intervallo  $[t+\min, t+T_{round}-\min]$ , la cui ampiezza è  $T_{round} - 2\min$ .

In definitiva, l'accuratezza offerta dall'algoritmo di Cristian è data da  $\alpha \leq T_{round}/2 - \min$ .



### Algoritmo di Berkeley:

È un algoritmo che fa uso di time service centralizzati e prevede una sincronizzazione interna.

Il master  $M$  (che non è altro che il time server) richiede in broadcast il valore dei clock delle altre macchine (worker), incluso se stesso. Dopodiché, per ciascun worker  $i$ , calcola la differenza  $d_i$  tra il suo clock e quello di  $i$  mediante la seguente operazione:

$$d_i = (C_M(t_1) + C_M(t_3))/2 - C_i(t_2) \quad \text{dove:}$$

->  $C_M(t_1)$  e  $C_M(t_3)$  sono i valori del clock sul master; il primo è relativo all'istante in cui il master ha inviato la richiesta in broadcast, mentre il secondo è relativo all'istante in cui ha ricevuto la risposta da parte del worker  $i$ .

->  $C_i(t_2)$  è il valore del clock sul worker  $i$  nell'istante in cui esso ha inviato il messaggio di risposta.

Infine, il master calcola la media dei vari  $d_i$  e invia un valore correttivo ai worker (che corrisponde alla somma tra tale media e il clock di  $M$ ).

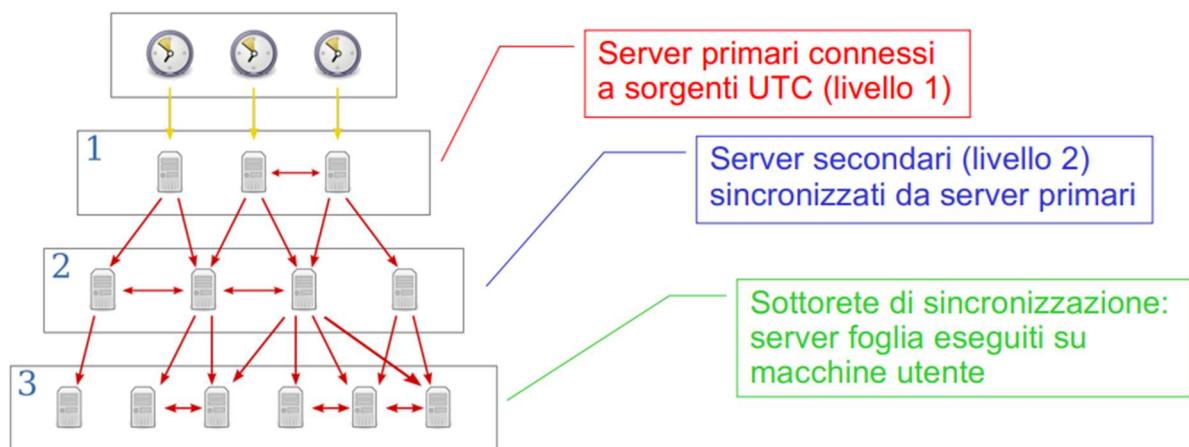
Vale la pena fare le seguenti osservazioni:

- L'accuratezza che si ha dipende da un RTT nominale massimo: il master, nel calcolo della media tra i  $d_i$ , scarta i valori dei clock associati a un RTT superiori al massimo (i.e. scarta gli outlier).
- L'algoritmo è tollerante ai comportamenti arbitrari: il master, nel calcolo della media tra i  $d_i$ , scarta anche i valori dei clock la cui differenza dal suo clock è maggiore di una soglia specificata.
- L'algoritmo è tollerante ai guasti: se il master cade un'altra macchina viene eletta master tramite un algoritmo di elezione.
- Se un worker riceve un valore correttivo che prevede un salto indietro nel tempo, non imposta il nuovo valore, bensì rallenta il clock, onde evitare la violazione della condizione di monotonicità del tempo. Questa soluzione consiste nel mascherare una serie di interrupt che fanno avanzare il clock locale in modo da rallentare l'avanzata del clock stesso; il numero di input mascherati è pari al tempo di slowdown diviso il periodo di interrupt del processore.

#### Network Time Protocol (NTP):

È un algoritmo che fa uso di time service decentralizzati e prevede una sincronizzazione esterna.

I time server sono ridondanti e sono organizzati su una struttura gerarchica:



La sottorete di sincronizzazione viene riconfigurata in caso di guasti. In particolare:

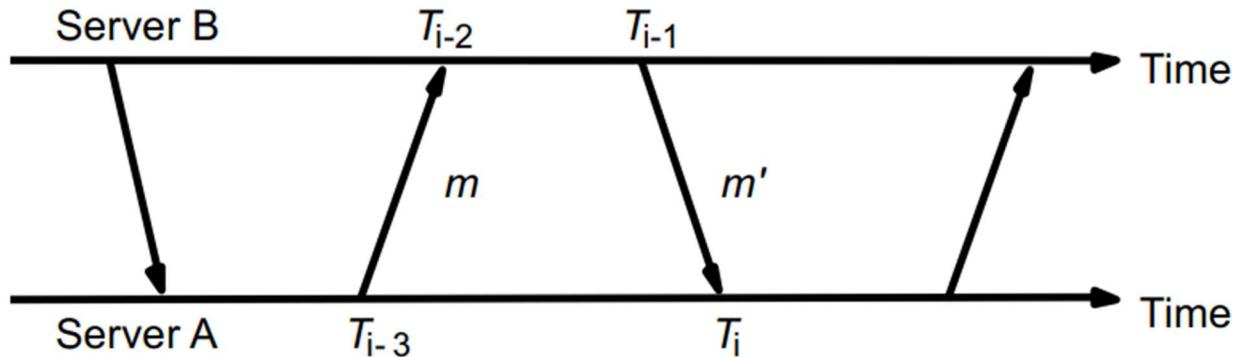
- Un server primario che perde la connessione alla sorgente UTC diventa server secondario.
- Un server secondario che perde la connessione al suo primario (e.g. a causa di un crash del primario) può usare un altro primario.

NTP prevede diversi modi di sincronizzazione tra cui:

- > **Multicast:** un server all'interno di una rete LAN invia in multicast il suo tempo agli altri, i quali impostano il tempo ricevuto assumendo un certo ritardo di trasmissione. L'accuratezza associata a tale modo è relativamente bassa.
- > **Procedure call:** un server accetta richieste da altri (come nell'algoritmo di Cristian). Qui si ha un'accuratezza maggiore rispetto al multicast.
- > **Simmetrico:** coppie di server si scambiano messaggi contenenti informazioni sul timing. Qui si ha un'accuratezza molto alta, per cui si tratta di un modo utilizzato per i livelli alti della gerarchia.

Tutti e tre questi modi usano UDP come protocollo a livello di trasporto. Di seguito approfondiremo il modo simmetrico.

I server si scambiano coppie di messaggi ( $m, m'$ ) per migliorare l'accuratezza della loro sincronizzazione.



Per ogni coppia di messaggi ( $m, m'$ ), NTP stima l'offset  $o_i$  tra i due clock e il ritardo  $d_i$  (pari al tempo totale di trasmissione per  $m$  e  $m'$ ).

Indicando con:

- $o$  = offset reale del clock di B rispetto ad A ( $o = \text{clock}_B - \text{clock}_A$ );
- $t$  = tempo di trasmissione del messaggio  $m$ ;
- $t'$  = tempo di trasmissione del messaggio  $m'$ ;

si ha che:

$$\begin{aligned} \rightarrow T_{i-2} &= T_{i-3} + t + o \\ \rightarrow T_i &= T_{i-1} + t' - o \\ \rightarrow d_i &= t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1} \end{aligned}$$

Se sottraiamo le prime due equazioni otteniamo che:

$$\rightarrow o = o_i + (t' - t)/2, \text{ dove } o_i = (T_{i-2} - T_{i-3} + T_i - T_{i-1})/2$$

Poiché  $t, t' > 0$ , si può dimostrare che:

$$\rightarrow o_i + (-t' - t)/2 = o_i - d_i/2 \leq o \leq o_i + d_i/2 = o_i + (t' + t)/2$$

In conclusione,  $o_i$  è la stima dell'offset tra i due clock e  $d_i/2$  è l'accuratezza di tale stima.

Vale la pena fare le seguenti osservazioni:

- I server NTP considerano di volta in volta le 8 coppie  $< o_i, d_i >$  più recenti, scegliendo come stima di  $o$  il valore di  $o_i$  corrispondente al minimo  $d_j$ .
- I server NTP applicano un algoritmo di selezione dei peer per modificare eventualmente il peer da usare per sincronizzarsi.
- L'accuratezza di NTP è di 10 ms su Internet e di 1 ms su LAN.

### Google's True Time (TT)

La sincronizzazione perfetta tra processi in rete è di fatto impossibile. Tuttavia, Google ci è andato molto vicino con TT, un protocollo per la sincronizzazione utilizzato per sviluppare Spanner, un database distribuito NewSQL. Quando viene effettuata una scrittura sul nodo master, viene propagata ai nodi worker (= le repliche del database) specificando i timestamp in cui dovranno avvenire le scritture sulle repliche: in tal modo le varie scritture avverranno nell'ordine desiderato.

TT è basato sul GPS e su clock atomici (che hanno un drift rate estremamente basso). Inoltre, dispone di una rete in fibra ottica a bassissima latenza che consente una comunicazione molto efficiente tra i nodi. Lo svantaggio di TT sta nel fatto che richiede un hardware speciale e un protocollo di sincronizzazione tra clock avanzato, che sono irrealizzabili per la maggior parte dei sistemi.

## Clock logico

Come abbiamo già accennato, in un sistema distribuito asincrono l'utilizzo di un clock fisico non è sempre possibile: è per questo che si ricorre al clock logico, tramite cui i processi concordano sull'ordine in cui si verificano gli eventi, piuttosto che sul tempo esatto in cui sono avvenuti.

Lamport introduce il concetto di relazione di **happened-before** (anche detta relazione di precedenza o ordinamento causale), che viene indicata con la freccia semplice  $\rightarrow$ .

In particolare, due eventi  $e, e'$  sono in relazione di happened-before ( $e \rightarrow e'$ ) se si verifica uno dei seguenti casi:

- 1)  $\exists p_i \mid e \rightarrow_i e'$
- 2)  $e = \text{send}(m) \wedge e' = \text{receive}(m)$
- 3)  $\exists e, e', e'' \mid (e \rightarrow e'') \wedge (e'' \rightarrow e')$

A questo punto possiamo fare le seguenti osservazioni:

- Applicando i tre casi sopra descritti è possibile costruire una sequenza di eventi  $e_1, e_2, \dots, e_n$  causalmente ordinati, ma non è detto che tale sequenza sia unica.
- La relazione di happened-before rappresenta un **ordinamento parziale**: di fatto, è un ordinamento non riflessivo, antisimmetrico e transitivo.
- Data una coppia di eventi, questa non è necessariamente legata da una relazione di happened-before: in tal caso si dice che gli eventi sono **concorrenti** ( $e \parallel e'$ ).

## Clock logico scalare:

È un contatore software monotonicamente crescente, il cui valore non ha alcuna relazione col clock fisico. Ogni processo  $p_i$  ha il proprio clock logico  $L_i$  e lo usa per applicare i timestamp agli eventi. In particolare, denotiamo con  $L_i(e)$  il timestamp basato sul clock logico applicato dal processo  $p_i$  all'evento  $e$ .

**Proprietà:** se  $e \rightarrow e'$  allora  $L_i(e) < L_i(e')$

Tuttavia non vale il viceversa.

Per implementare il clock logico scalare, si segue l'**algoritmo di Lamport**:

- > Ogni processo  $p_i$  inizializza il proprio clock logico  $L_i$  a zero.
- > Prima di eseguire un evento interno,  $p_i$  incrementa  $L_i$  di 1.
- > Quando  $p_i$  invia un messaggio  $m$  a  $p_j$ , incrementa di 1 il valore di  $L_i$ , allega al messaggio  $m$  il timestamp  $t=L_i$  ed esegue l'evento  $\text{send}(m)$ .
- > Quando  $p_j$  riceve il messaggio  $m$  con timestamp  $t$ , aggiorna il proprio clock logico  $L_j = \max(t, L_j)$ , lo incrementa di 1 ed esegue l'evento  $\text{receive}(m)$ .

Usando il clock logico scalare, due o più eventi possono avere lo stesso timestamp: come si può realizzare un ordinamento totale tra eventi, evitando così che due eventi accadano nello stesso tempo logico?

**Soluzione:** usare, oltre al clock logico, il numero del processo in cui è avvenuto l'evento.

Da qui introduciamo la **relazione di ordine totale** tra eventi (indicata con  $e \Rightarrow e'$ ). In particolare, se  $e$  è un evento nel processo  $p_i$  ed  $e'$  è un evento nel processo  $p_j$ , allora  $e \Rightarrow e'$  se e solo se vale uno dei seguenti due casi:

- $L_i(e) < L_j(e')$
- $L_i(e) = L_j(e') \wedge p_i < p_j$

Infine, il clock logico scalare presenta una limitazione: poiché non è possibile assicurare che  $e \rightarrow e'$  nel caso in cui  $L_i(e) < L_i(e')$ , non si può stabilire, solo guardando i clock logici scalari, se due eventi sono concorrenti o meno. Tale limitazione viene superata introducendo i clock logici vettoriali.

## Clock logico vettoriale:

È un vettore di  $N$  interi, dove  $N$  è il numero di processi attivi all'interno del sistema.

Ogni processo  $p_i$  ha il proprio clock vettoriale  $V_i$  e lo usa per applicare i timestamp agli eventi.

Con il clock vettoriale si catturano completamente le caratteristiche della relazione happened-before:

## **e → e' se e solo se $V(e) < V(e')$**

Dato il clock vettoriale  $V_i$ :

- $V_i[i]$  è il numero di eventi generati da  $p_i$ .
- $V_i[j]$  (con  $i \neq j$ ) è il numero di eventi occorsi a  $p_j$  di cui  $p_i$  ha conoscenza.

Per quanto invece riguarda il confronto tra clock vettoriali:

- $V = V'$  se e solo se  $\forall j: V[j] = V'[j]$ .
- $V \leq V'$  se e solo se  $\forall j: V[j] \leq V'[j]$ .
- $V < V'$  se e solo se  $\forall i: V[i] \leq V'[i] \wedge \exists j | V[j] < V'[j]$ .
- $V || V'$  se e solo se  $\text{not}(V < V') \wedge \text{not}(V > V')$ .

Per implementare il clock logico vettoriale, si segue l'algoritmo qui proposto:

- > Ogni processo  $p_i$  inizializza il proprio clock vettoriale  $V_i$  nel seguente modo:  $V_i[k]=0 \forall k = 1, 2, \dots, N$ .
- > Prima di eseguire un evento interno,  $p_i$  incrementa di 1 la sua componente  $V_i[i]$ .
- > Quando  $p_i$  invia un messaggio  $m$  a  $p_j$ , incrementa di 1 la sua componente  $V_i[i]$ , allega al messaggio  $m$  il timestamp vettoriale  $t=V_i$  ed esegue l'evento  $\text{send}(m)$ .
- > Quando  $p_j$  riceve il messaggio  $m$  con timestamp  $t$ , aggiorna il proprio clock logico  $V_j[k] = \max(t[k], V_j[k])$   $\forall k = 1, 2, \dots, N$ , incrementa di 1 la sua componente  $V_j[j]$  ed esegue l'evento  $\text{receive}(m)$ .

## **Esempi di applicazione del clock logico**

### Multicast totalmente ordinato:

Come si può garantire che scritture concorrenti su un database replicato siano viste nello stesso ordine da ogni replica? Precisiamo che non è fondamentale che l'ordine sia quello corretto, l'importante è che sia il medesimo per tutte le repliche.

La soluzione consiste nell'adottare il multicast totalmente ordinato, che è un'operazione di multicast in cui tutti i messaggi sono consegnati nello stesso ordine a ogni destinatario. A tal proposito, vedremo due possibili algoritmi in cui vengono fatte due assunzioni:

- Comunicazione affidabile (non si ha perdita di messaggi).
- Comunicazione FIFO ordered (i messaggi inviati da  $p_i$  a  $p_j$  vengono ricevuti da  $p_j$  nello stesso ordine in cui  $p_i$  li ha inviati).

**1) Algoritmo centralizzato:** prevede l'utilizzo di un coordinatore centralizzato (detto sequencer). Ogni processo invia il proprio messaggio di update al sequencer il quale assegna al messaggio stesso un sequence number univoco per poi inviarlo in multicast a tutti i processi; dopodiché i processi eseguono gli aggiornamenti in ordine in base al sequence number dei messaggi ricevuti. Tale soluzione però presenta problemi di scalabilità e di single point of failure.

**2) Algoritmo distribuito:** prevede che ogni messaggio abbia come timestamp il **clock logico scalare** del processo che lo invia. Di seguito vengono presentati i passi dell'algoritmo:

- >  $p_i$  invia in multicast (incluso se stesso) il messaggio di update  $msg_i$ .
- >  $msg_i$  viene posto da ogni processo destinatario  $p_j$  in una coda locale  $queue_j$ , ordinata in base al valore del timestamp.
- >  $p_j$  invia in multicast un messaggio di ack della ricezione di  $msg_i$ .
- >  $p_j$  consegna  $msg_i$  all'applicazione se  $msg_i$  è in testa a  $queue_j$ , tutti gli ack relativi a  $msg_i$  sono stati ricevuti da  $p_j$  e, per ogni processo  $p_k$ , c'è un messaggio  $msg_k$  in  $queue_j$  con timestamp maggiore di quello di  $msg_i$  (quest'ultima condizione sta a indicare che nessun altro processo può inviare in multicast un messaggio con timestamp potenzialmente minore o uguale a quello di  $msg_i$ ).

Osserviamo che, per  $N$  processi attivi e per ogni messaggio inviato, vengono inviati in tutti  $N^2$  ack. Di conseguenza, è una soluzione che presenta problemi di scalabilità in sistemi a larga scala. Malgrado ciò, è comunque un meccanismo alla base della **state machine replication**.

La state machine replication è una tecnica software che introduce una replicazione di ciascun componente dell'applicazione, e prevede che ciascuna replica cambi stato in base alle richieste che riceve. Affinché le varie repliche si comportino come un unico server, ciascuna di loro deve soddisfare le richieste nello stesso ordine in modo tale da mantenere il suo stato consistente rispetto alle altre repliche.

#### Multicast causalmente ordinato:

In questo secondo scenario, un messaggio viene consegnato solo se tutti i messaggi che lo precedono causalmente (i.e. secondo la **relazione di causa-effetto**) sono stati già consegnati. Si tratta di un indebolimento del multicast totalmente ordinato.

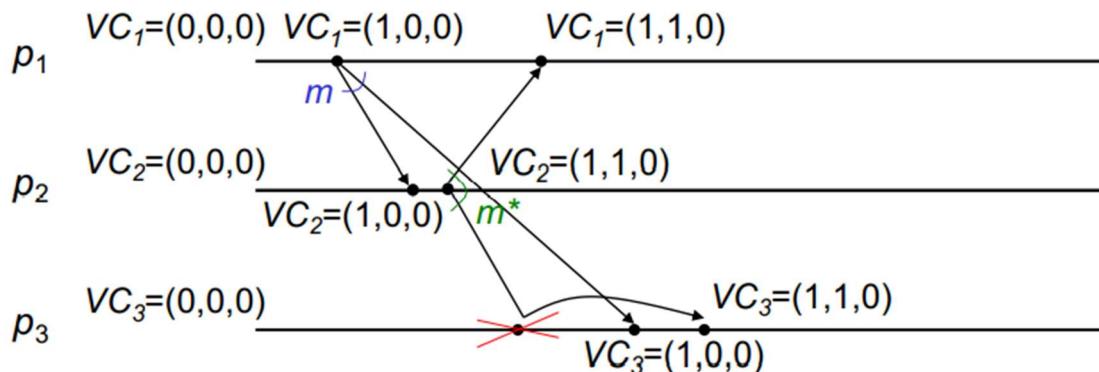
Mantenendo le assunzioni sulla comunicazione affidabile e FIFO ordered, introduciamo un algoritmo che fa uso del **clock logico vettoriale** per risolvere il problema del multicast causalmente ordinato in modo completamente distribuito:

->  $p_i$  invia un messaggio  $m$  con timestamp  $t(m)$  basato sul clock logico vettoriale  $V_i$ , dove stavolta  $V_i[i]$  conta il numero di messaggi che  $p_i$  ha inviato a  $p_j$  (per cui c'è una differenza con l'utilizzo standard dei clock logici vettoriali).

->  $p_j$  riceve  $m$  da  $p_i$  e ne ritarda la consegna al livello applicativo (ponendo  $m$  in una coda d'attesa) finché non si verificano entrambe le seguenti condizioni:

- 1)  $t(m)[i] = V_j[i] + 1$  ( $m$  è il messaggio successivo che  $p_j$  si aspetta da  $p_i$ ).
- 2)  $t(m)[k] \leq V_j[k] \forall k \neq i$  (per ogni processo  $p_k$ ,  $p_j$  ha visto almeno gli stessi messaggi di  $p_k$  visti da  $p_i$ ).

Analizziamo un esempio: consideriamo tre processi  $p_1, p_2, p_3$ , e supponiamo che  $p_1$  invii un messaggio  $m$  a  $p_2, p_3$  e che  $p_2$ , dopo aver ricevuto  $m$ , invii un messaggio  $m^*$  a  $p_1, p_3$ . Se  $p_3$  riceve  $m^*$  prima di  $m$ , l'algoritmo evita la violazione della causalità tra  $m$  e  $m^*$  facendo sì che su  $p_3$   $m$  sia consegnato al livello applicativo dopo  $m^*$ .



#### **Mutua esclusione**

La mutua esclusione nasce nei sistemi concorrenti, in cui  $N$  processi vogliono accedere a una risorsa condivisa e utilizzarla in modo esclusivo senza interferenze da parte degli altri processi.

Ogni algoritmo di mutua esclusione comprende:

- Una sequenza di istruzioni chiamata **sezione critica (CS)** che consiste nell'accesso alla risorsa condivisa.
- Una sequenza di istruzioni chiamata **trying protocol (TP)** che precede la sezione critica.
- Una sequenza di istruzioni chiamata **exit protocol (EP)** che segue la sezione critica.

Inoltre, un algoritmo di mutua esclusione dovrebbe soddisfare le seguenti proprietà:

-> **Mutua esclusione (ME) o safety**: solo un processo per volta può entrare nella sezione critica.

-> **No deadlock (ND)**: se un processo rimane bloccato nella sua trying section, esiste comunque almeno un processo in grado di entrare e uscire dalla sezione critica.

-> **No starvation (NS)**: nessun processo può rimanere bloccato per un tempo indefinito nella trying section, per cui tutte le richieste di ingresso (e uscita) dalla sezione critica sono prima o poi soddisfatte. Trattasi di una condizione di imparzialità (*fairness*) nei confronti dei processi. Notiamo inoltre che NS implica ND.

Due algoritmi storici per la mutua esclusione sono l'**algoritmo di Dijkstra** (ideato per sistemi a singolo processore) e l'**algoritmo del panificio di Lamport** (ideato per sistemi multiprocessore a memoria condivisa). Noi analizzeremo quest'ultimo.

#### Algoritmo del panificio di Lamport:

I processi comunicano leggendo e scrivendo variabili condivise, ciascuna delle quali è proprietà di uno specifico processo  $p$ ; in particolare, solo il processo  $p$  può scrivere la sua variabile, mentre tutti possono leggerla. Tali variabili condivise sono:

- $\text{num}[1, \dots, N]$ : array di interi inizializzati a 0. Deve essere visto come una lista di numeretti assegnati ai vari processi; se un processo ha il proprio numeretto pari a 0, vuol dire che al momento non è interessato a entrare in sezione critica.
- $\text{choosing}[1, \dots, N]$ : array di booleani inizializzati a false. Ciascuna entry indica se il processo corrispondente sta correntemente scegliendo il numeretto o meno.

L'algoritmo è raffigurato qui di seguito:

```
// sezione non critica
// prende un biglietto
choosing[i] = true; // inizio della selezione del biglietto           doorway
num[i] = 1 + max(num[x]: 1 ≤ x ≤ N);
choosing[i] = false; // fine della selezione del biglietto
// attende che il numero sia chiamato confrontando il suo biglietto
// con quello degli altri                                         bakery
for j = 1 to N do
    // busy waiting mentre j sta scegliendo
    while choosing[j] do NoOp();
    // busy waiting finché il valore del biglietto non è il più basso
    // viene favorito il processo con identificativo più piccolo
    while num[j] ≠ 0 and {num[j], j} < {num[i], i} do NoOp();
// sezione critica
num[i] = 0;
// fine sezione critica
```

Relazione di precedenza  $<$  su coppie ordinate di interi definita da:  $(a,b) < (c,d)$  se  $a < c$ , o se  $a = c$  e  $b < d$

Questo algoritmo gode della proprietà di mutua esclusione, della proprietà di no starvation ma anche della proprietà FCFS (infatti, se il processo  $p_i$  entra nella bakery prima che  $p_j$  entri nella doorway, allora  $p_i$  entrerà certamente in sezione critica prima di  $p_j$ ).

L'algoritmo inoltre funziona bene per un singolo sistema multiprocessore, ma è comunque possibile riadattarlo per un sistema distribuito.

Consideriamo dunque un sistema distribuito con  $N$  processi in cui valgono le seguenti assunzioni:

-> Il sistema è asincrono: il ritardo di trasmissione di un messaggio, seppur finito, è impredicibile.

- > I processi non sono soggetti a fallimenti.
- > La comunicazione è affidabile e FIFO ordered.
- > I processi trascorrono un tempo finito nella sezione critica.

A questo punto, comunicazione tra i processi non avviene più mediante l'utilizzo di memoria condivisa, bensì tramite scambio di messaggi. In particolare, ogni processo  $p_i$  si comporta da server rispetto alle proprie variabili  $num[i]$  e  $choosing[i]$  (per cui, se un processo  $p_j$  ha bisogno di accedervi in lettura, invia un messaggio di richiesta a  $p_i$  e attende la risposta).

Si tratta di una soluzione che funziona ma ha un costo totale pari a  $6N$  messaggi. Infatti:

- Per accedere alla sezione critica servono tre scambi di messaggi, di cui uno per la doorway e due per la bakery.
- Per ogni lettura vengono scambiati  $2N$  messaggi.

Notiamo che ogni lettura comporta una latenza pari al tempo impiegato dalla combinazione (canale di comunicazione, processo) più lenta, per cui l'algoritmo è caratterizzato da una scarsa efficienza e scalabilità. Questi problemi derivano dalla mancanza di cooperazione tra processi, che viene introdotta negli algoritmi che vedremo di seguito.

### **Algoritmi per la mutua esclusione distribuita**

Si suddividono in:

- > Algoritmi basati sulle **autorizzazioni**: un processo che vuole accedere a una risorsa condivisa chiede l'autorizzazione. Il meccanismo può essere gestito in modo centralizzato oppure in modo completamente distribuito (algoritmo di Lamport distribuito + algoritmo di Ricart e Agrawala).
- > Algoritmi basati su **token**: tra i processi circola un token, che è un messaggio speciale e unico in ogni istante di tempo; solo chi detiene il token può accedere alla risorsa condivisa. Anche qui il meccanismo può essere gestito in modo centralizzato oppure in modo decentralizzato.
- > Algoritmi basati su **quorum**: un processo che vuole accedere a una risorsa condivisa chiede il permesso solo a un sottoinsieme di processi (algoritmo di Maekawa).

Esistono due criteri principali che vengono usati per valutare gli algoritmi di mutua esclusione distribuita:

- Numero di messaggi scambiati per l'ingresso e l'uscita dalla sezione critica: è una misura indiretta della banda di rete consumata.
- Numero di messaggi scambiati per il solo ingresso in sezione critica: è una misura indiretta del tempo di attesa.

#### Algoritmo basato sulle autorizzazioni centralizzato:

Consideriamo un processo  $p$  che vuole utilizzare una risorsa in mutua esclusione. Esso invia una richiesta di accesso (ENTER) al coordinatore centrale. Se la risorsa è libera, il coordinatore informa il mittente che l'accesso è consentito (GRANTED); altrimenti, accoda la richiesta con politica FIFO e informa il mittente che l'accesso non è consentito (DENIED) o, nel caso di sistema distribuito sincrono, non risponde proprio.

Quando un processo rilascia la risorsa, informa il coordinatore tramite un messaggio RELEASED; dopodiché il coordinatore preleva dalla coda la prima richiesta in attesa e invia il messaggio GRANTED al rispettivo mittente.

Tale algoritmo presenta i seguenti vantaggi:

- Garantisce mutua esclusione, assenza di starvation e fairness (grazie alla coda gestita con disciplina FIFO).
- È facile da implementare e richiede uno scambio di tre soli messaggi (richiesta, risposta e rilascio) per l'ingresso e l'uscita dalla sezione critica.

D'altra parte, però, è caratterizzato dai seguenti svantaggi:

- Il coordinatore può fungere da bottleneck e single point of failure.

- Se fallisce un processo mentre è in sezione critica, si perde il messaggio di rilascio; per mitigare il problema, è possibile introdurre un timeout.

#### Algoritmo di Lamport distribuito:

Ogni processo mantiene un clock logico scalare (che serve da supporto per applicare la relazione di ordine totale =>) e una coda locale per memorizzare le richieste di accesso alla sezione critica. Di seguito vengono presentati i passi dell'algoritmo:

- >  $p_i$ , per richiedere l'accesso alla sezione critica, invia a tutti gli altri processi un messaggio di richiesta  $msg_i$  e pone il suo messaggio  $msg_i$  all'interno della sua coda.
- > Ciascun processo  $p_j$  riceve la richiesta di  $p_i$ , la memorizza nella propria coda e invia a  $p_i$  un messaggio di ack.
- >  $p_i$  accede alla sezione critica se e solo se  $msg_i$  precede tutti gli altri messaggi di richiesta in coda (ossia il suo timestamp  $t$  è il minimo applicando =>) e  $p_i$  ha ricevuto da ogni altro processo un messaggio (di ack o nuova richiesta) con timestamp maggiore di  $t$  (sempre applicando =>).
- > Quando  $p_i$  rilascia la sezione critica, elimina  $msg_i$  dalla sua coda e invia un messaggio di release a tutti gli altri processi.
- > Ciascun processo  $p_j$  riceve il messaggio di release, per cui elimina la richiesta corrispondente dalla sua coda.

Un vantaggio di tale algoritmo è dato dal soddisfacimento delle proprietà di mutua esclusione e no starvation.

Gli svantaggi, invece, sono i seguenti:

- Se un processo fallisce, nessun altro potrà entrare nella sezione critica: occorre aggiungere un meccanismo di failure detection.
- Tutti i processi possono essere collo di bottiglia.

Comunque sia, l'algoritmo richiede  $3(N-1)$  messaggi per accedere alla sezione critica, di cui  $N-1$  messaggi di richiesta,  $N-1$  messaggi di ack e  $N-1$  messaggi di release.

#### Algoritmo di Ricart e Agrawala:

È un'estensione e un'ottimizzazione dell'algoritmo di Lamport distribuito. Infatti, anch'esso è basato sul clock logico scalare e sulla relazione d'ordine totale.

- |  |   |
|--|---|
| <ul style="list-style-type: none"> <li>• Un processo che vuole accedere alla CS manda un <b>messaggio di REQUEST</b> a tutti gli altri contenente:           <ul style="list-style-type: none"> <li>– proprio identificatore</li> <li>– <i>timestamp</i> basato su clock logico scalare</li> </ul> </li> <li>• Si pone in attesa della risposta da tutti gli altri</li> <li>• Ottenuti tutti i <b>messaggi di REPLY</b> entra nella CS</li> <li>• All'uscita dalla CS manda REPLY a <i>tutti</i> i processi nella coda locale</li> </ul> | <ul style="list-style-type: none"> <li>• Un processo che riceve il messaggio di REQUEST può           <ul style="list-style-type: none"> <li>– non essere nella CS e non volervi accedere → manda REPLY al mittente</li> <li>– essere nella CS → non risponde e mette il messaggio in coda locale</li> <li>– voler accedere alla CS → confronta il suo timestamp con quello ricevuto e vince quello minore: se è l'altro, invia REPLY; se è lui, non risponde e mette il messaggio in coda</li> </ul> </li> </ul> |
|--|---|

Di seguito sono riportate le variabili locali di ciascun processo:

- #replies = numero di risposte (REPLY) ricevute (inizializzata a 0).
- State ∈ {Requesting, CS, NCS} (inizializzata a NCS – no critical section).
- Q = coda di richieste pendenti (inizialmente vuota).
- Last\_Req = timestamp del messaggio di richiesta (inizializzato a 0).
- Num = clock logico scalare (inizializzato a 0).

Ogni processo  $p_i$ , per l'esattezza, esegue il seguente algoritmo:

**Begin**

1. State = Requesting;
2. Num = Num+1; Last\_Req = Num;
3. for  $j=1$  to  $N-1$  send REQUEST to  $p_j$ ;
4. Wait until  $\#replies=N-1$ ;
5. State = CS;
6. CS
7.  $\forall r \in Q$  send REPLY to  $r$
8.  $Q=\emptyset$ ; State=NCS;  $\#replies=0$ ;

**Upon receipt of REQUEST( $t$ ) from  $p_j$**

1. if State=CS or (State=Requesting and  $\{Last\_Req, i\} < \{t, j\}$ )
2. then insert  $\{t, j\}$  in Q
3. else send REPLY to  $p_j$
4. Num = max( $t$ , Num)

**Upon receipt of REPLY from  $p_j$**

1.  $\#replies = \#replies + 1$

Una nota positiva dell'algoritmo è che, rispetto a Lamport distribuito, porta a un risparmio di messaggi: infatti, non prevede alcun messaggio di release, per cui richiede solo  $2(N-1)$  messaggi per accedere alla sezione critica, di cui  $N-1$  messaggi di richiesta e  $N-1$  messaggi REPLY. Addirittura, in letteratura esistono ottimizzazioni (non esaminate) che riducono a  $N$  il numero di messaggi.

Per quanto riguarda gli svantaggi, sono perfettamente uguali al caso dell'algoritmo di Lamport.

Algoritmo basato su token centralizzato:

Uno dei processi svolge il ruolo di coordinatore e gestisce il token. Le sue strutture dati sono:

- Reqlist = lista di richieste pendenti, ricevute dal coordinatore ma non ancora servite.
  - V = array di dimensione pari al numero di processi, in cui  $V[i]$  è il numero di richieste di  $p_i$  già servite.
- D'altra parte, ciascun processo  $p_i$  mantiene un suo clock vettoriale  $VC_i$  per il timestamp dei messaggi.

Di seguito vengono elencati i passi dell'algoritmo:

- >  $p_i$ , per richiedere l'accesso alla sezione critica, incrementa  $VC_i[i]$  di 1, invia un messaggio di richiesta al coordinatore avente come timestamp  $T_{pi}$  il proprio clock vettoriale e invia un **messaggio di programma** a tutti gli altri processi sempre col proprio clock vettoriale incluso.
- > Il coordinatore, quando riceve il messaggio di richiesta da parte di  $p_i$ , lo inserisce nella lista delle richieste pendenti.
- > Ciascun processo  $p_j$ , quando riceve il messaggio di programma da parte di  $p_i$ , aggiorna il proprio clock logico vettoriale in modo tale che  $VC_j[k] = \max(VC_j[k], VC_i[k]) \quad \forall k$ .
- > Si dice che la richiesta di  $p_i$  divenga **eleggibile** nel momento in cui  $T_{pi}[k] \leq V[k] \quad \forall k \neq i$ . Quando ciò avviene e il coordinatore ha a disposizione il token, il coordinatore pone  $V$  uguale a  $T_{pi}$  e invia il token a  $p_i$ , il quale entra in sezione critica.
- > Quando  $p_i$  rilascia la sezione critica, restituisce il token al coordinatore.

Tale algoritmo presenta i seguenti vantaggi:

- È più efficiente dell'algoritmo di Ricart e Agrawala in termini di numero di messaggi scambiati, che sono  $3+N-1 = N+2$  messaggi per ogni richiesta di accesso alla sezione critica.
- Vengono garantite anche le proprietà di fairness e ordering.

Tuttavia, si hanno i seguenti svantaggi:

- Il coordinatore è un collo di bottiglia e single point of failure.
- In caso di crash del coordinatore, occorre eleggere un nuovo coordinatore.

#### Algoritmo basato su token decentralizzato:

I processi sono organizzati logicamente in un anello unidirezionale (non c'è alcuna relazione tra la topologia dell'anello e l'interconnessione fisica tra i nodi). Il token viaggia da un processo all'altro in modo tale che, a ogni iterazione, venga passato dal processo  $p_k$  al processo  $p_{(k+1) \bmod N}$ . Solo il processo in possesso del token può entrare in sezione critica; comunque sia, se un processo riceve il token ma non è interessato a entrare in sezione critica, passa direttamente il token al processo successivo.

I vantaggi dati da questo algoritmo sono i seguenti:

- Con l'anello unidirezionale viene garantita anche la fairness.
- Rispetto all'algoritmo centralizzato, presenta un bilanciamento del carico migliore.

D'altra parte, si hanno i seguenti svantaggi:

- Viene consumata la banda di rete per trasmettere il token anche quando nessuno chiede l'accesso alla sezione critica.
- In caso di perdita del token, occorre rigenerarlo; in questo modo, però, è necessario anche un meccanismo per identificare se i token diventano più di uno.
- Se un processo fallisce, occorre riconfigurare l'anello.

#### Algoritmo di Maekawa:

Stavolta, per entrare in sezione critica, il processo  $p_i$  deve raccogliere i voti solo da un sottoinsieme di processi (quorum), che è detto **insieme di votazione**  $V_i$ . I processi appartenenti a  $V_i$  votano per stabilire chi è autorizzato a entrare in sezione critica, e ciascuno di loro può votare solo per un processo.

- |  |   |   |
|--|---|---|
| <ul style="list-style-type: none"><li>• Un processo <math>p_i</math> per accedere alla CS<ul style="list-style-type: none"><li>– Invia <i>request</i> a tutti gli altri membri di <math>V_i</math></li><li>– Attende <i>reply</i> da tutti i membri di <math>V_i</math></li><li>– Ricevute tutte le <i>reply</i> dai membri di <math>V_i</math> usa la CS</li><li>– Al rilascio della CS invia <i>release</i> a tutti gli altri membri di <math>V_i</math></li></ul></li></ul> | <ul style="list-style-type: none"><li>• Un processo <math>p_j</math> in <math>V_i</math> che riceve <i>request</i><ul style="list-style-type: none"><li>– Se è in CS o ha già risposto dopo aver ricevuto l'ultimo <i>release</i>, non risponde e accoda la richiesta</li><li>– Altrimenti risponde subito con <i>reply</i></li></ul></li></ul> | <ul style="list-style-type: none"><li>• Un processo che riceve <i>release</i> estrae <b>una</b> richiesta dalla coda e invia <i>reply</i></li></ul> |
|--|---|---|

Più nel dettaglio, ogni processo  $p_i$  esegue l'algoritmo riportato nella pagina seguente.

## Inizializzazione

state = RELEASED;

voted = FALSE;

## Protocollo di ingresso in CS per $p_i$

state = WANTED;

invia in multicast *request* a tutti i processi in  $V_i$  (incluso se stesso);

wait until (numero di *reply* ricevute =  $K$ );

state = HELD;

## Alla ricezione di *request* da $p_j$ ( $i \neq j$ )

if (state = HELD or voted = TRUE) then

    accoda *request* da  $p_j$  senza rispondere;

else

    invia *reply* a  $p_j$ ; // vota a favore di  $p_j$

    voted = TRUE;

end if

## Protocollo di uscita da CS per $p_i$

state = RELEASED;

invia in multicast *release* a tutti i processi in  $V_i$ ;

if (coda di richieste non vuota) then

    estrai la prima richiesta in coda, ad es. da  $p_k$ ;

    invia *reply* a  $p_k$ ; // vota a favore di  $p_k$

    voted = TRUE;

else

    voted = FALSE;

end if

## Alla ricezione di un messaggio di *release* da $p_j$ ( $i \neq j$ )

if (coda di richieste non vuota) then

    estrai la prima richiesta in coda, ad es. da  $p_k$ ;

    invia *reply* a  $p_k$ ;

    voted = TRUE;

else

    voted = FALSE;

end if

Gli insiemi di votazione godono delle seguenti proprietà:

- >  $V_i \cap V_j = \emptyset \forall i, j$  (in modo tale da garantire la mutua esclusione; infatti, se un quorum accorda l'accesso in sezione critica a un dato processo, nessun altro quorum potrà accordare lo stesso permesso).
- >  $|V_i| = K \forall i$  (in modo tale tutti i processi debbano ottenere lo stesso numero di voti per entrare in sezione critica).
- > Ogni processo  $p_i$  è contenuto in esattamente  $K$  insiemi di votazione (in modo tale che tutti i processi abbiano lo stesso grado di responsabilità).
- >  $p_i \in V_i$  (in modo tale da ridurre il numero di messaggi trasmessi).
- > Si dimostra che la soluzione ottima che minimizza  $K$  è  $K \leq \sqrt{N}$ .

### Esempio: $N=7$

$V_1=\{1,2,3\}$
$V_4=\{1,4,5\}$
$V_6=\{1,6,7\}$
$V_2=\{2,4,6\}$
$V_5=\{2,5,7\}$
$V_7=\{3,4,7\}$
$V_3=\{3,5,6\}$

### Esempio: $N=3$

$V_1=\{1,2\}$
$V_3=\{1,3\}$
$V_2=\{2,3\}$

Un punto di forza dell'algoritmo di Maekawa risiede nelle prestazioni: per accedere alla sezione critica basta un numero di messaggi pari a  $3\sqrt{N}$ , di cui  $2\sqrt{N}$  per l'ingresso in sezione critica e  $\sqrt{N}$  per l'uscita.

Uno svantaggio, invece, consiste nel fatto che possono verificarsi dei **deadlock** (il che si può risolvere introducendo dei messaggi aggiuntivi nell'algoritmo). Ad esempio, consideriamo un insieme di votazione composto dai processi  $p_1, p_2, p_3$ : se tutti e tre richiedono contemporaneamente l'accesso in sezione critica, è verosimile che votino tutti per se stessi, creando di fatto una situazione di deadlock.

### Algoritmi di elezione distribuita

Come abbiamo visto, molti algoritmi distribuiti richiedono che un processo agisca da coordinatore o leader. Per questo motivo, è necessario introdurre degli algoritmi di elezione distribuita che abbiano lo scopo di soddisfare le proprietà di **safety** (secondo cui viene eletto uno e un solo coordinatore) e di **liveness** (secondo cui l'elezione termina con l'accordo di tutti i partecipanti e, in ogni istante di tempo, esiste sempre un nodo coordinatore).

Nella trattazione degli algoritmi di elezione effettueremo le seguenti assunzioni:

- I processi possono essere soggetti a fallimenti di tipo crash (non considereremo i guasti bizantini, in cui un processo continua a funzionare ma assume un comportamento malevolo o comunque arbitrario).
- La comunicazione è affidabile.
- Un processo non indice più di un'elezione per volta.
- Ogni processo ha un id univoco e viene eletto il processo non guasto con l'id più elevato.

Noi tratteremo due algoritmi di elezione distribuita: l'**algoritmo bully** (del prepotente) e l'**algoritmo ad anello di Fredrickson e Lynch**.

#### Algoritmo bully:

Se il processo  $p_i$  rileva l'assenza del coordinatore, promuove una nuova elezione. A questo punto vengono seguiti questi passaggi:

- >  $p_i$  invia un messaggio ELEZIONE a tutti i processi con id maggiore del suo.
- > Se nessuno risponde,  $p_i$  si autoprolama vincitore, invia un messaggio di proclamazione a tutti i processi con id minore del suo e diventa il nuovo coordinatore.
- > Se  $p_k$  (con  $k > i$ ) riceve il messaggio ELEZIONE da  $p_i$ , risponde con un messaggio OK e indice una nuova elezione. Conseguentemente,  $p_i$  riceverà il messaggio OK e abbandonerà il suo proposito di diventare il nuovo coordinatore.

Notiamo che, per appurare che nessuno abbia inviato il messaggio OK,  $p_i$  deve attendere un timeout, ovvero deve conoscere il tempo massimo di trasmissione di un messaggio. Di conseguenza, nella descrizione dell'algoritmo bully è stata fatta l'assunzione secondo cui il sistema distribuito sia sincrono.

Ma qual è il costo dell'algoritmo in termini di numero di messaggi scambiati?

- **Caso ottimo:** il processo non guasto con id più alto è colui che si accorge per primo della failure del coordinatore; esso può eleggersi subito come coordinatore e inviare  $N-2$  messaggi di proclamazione (chiaramente se stesso e il precedente coordinatore che si è guastato non rientrano tra i destinatari). In breve, si hanno  $O(N)$  messaggi.
- **Caso pessimo:** il processo con id più basso è colui che si accorge per primo della failure del coordinatore; esso invia  $N-2$  (o  $N-1$ ) messaggi agli altri processi, ciascuno dei quali a sua volta indice una nuova elezione. Alla fine della fiera si hanno  $O(N^2)$  messaggi.

#### Algoritmo ad anello di Fredrickson e Lynch:

I processi sono organizzati logicamente in un anello unidirezionale, e ciascuno di loro conosce almeno il suo successore (ma più nodi conosce, più l'algoritmo risulta essere tollerante ai guasti di tipo crash).

Se il processo  $p_i$  rileva l'assenza del coordinatore, promuove una nuova elezione. A questo punto vengono seguiti questi passaggi:

- >  $p_i$  invia un messaggio ELEZIONE a  $p_{(i+1) \bmod N}$ .
- > Se  $p_{(i+1) \bmod N}$  è faulty,  $p_i$  idealmente lo salta e passa al processo successivo lungo l'anello, finché non ne trova uno non guasto.
- > Ad ogni passo, il processo che riceve il messaggio di ELEZIONE vi aggiunge il suo id e lo inoltra al successore.
- > Al termine del giro dell'anello, il messaggio di ELEZIONE torna a  $p_i$ , il quale identifica nel messaggio il processo con id più alto e invia sull'anello un messaggio di COORDINATORE per informare tutti su chi sia il nuovo coordinatore.

È facile convincersi che l'algoritmo funziona sia con la comunicazione sincrona che con la comunicazione asincrona; inoltre funziona per qualunque  $N$  (dove  $N$  è il numero di processi all'interno dell'anello) e non richiede ai processi di conoscere il valore di  $N$ .

Questo algoritmo ha sempre un costo di  $O(2N)$  messaggi scambiati (circa  $N$  per lo scambio del messaggio di ELEZIONE e circa  $N$  per lo scambio del messaggio di COORDINATORE). Ciononostante, i messaggi hanno dimensioni maggiori rispetto al caso dell'algoritmo di Bully, dato che devono contenere gli identificatori di uno o più processi.

#### Proprietà degli algoritmi di elezione:

- > Se un processo subisce un crash durante l'elezione, prima o poi qualcun altro dovrà accorgersi del guasto e tenterà di aprire un nuovo processo di elezione. Nel caso della struttura ad anello, potrebbe anche essere necessario implementare dei meccanismi addizionali atti a reconfigurare l'anello stesso.
- > Se va in crash un nodo (o un router) che univa due porzioni del sistema, si va incontro al problema del **network partitioning**: si ottengono cioè due partizioni del sistema funzionanti ma perfettamente isolate tra loro, per cui è facile andare incontro a una situazione con due coordinatori eletti. Di fatto, gli algoritmi che abbiamo analizzato non sono tolleranti al network partitioning.

## CONSISTENZA NEI SISTEMI DISTRIBUITI

### Replicazione

Replicare i dati è un'operazione importante nei sistemi distribuiti poiché ne migliora la disponibilità, la tolleranza ai guasti e la scalabilità. Tuttavia, richiede che le varie repliche siano mantenute consistenti.

Problema: come si possono mantenere aggiornate (e quindi consistenti) le repliche evitando una riduzione significativa delle performance specialmente nei sistemi distribuiti su larga scala? Non dimentichiamo che la latenza di comunicazione è non trascurabile: tra data center differenti va dai 10 ms ai 250 ms, mentre all'interno di uno stesso data center è comunque circa pari a 1 ms.

L'idea di base è che, per mantenere le repliche consistenti, sarebbe necessario assicurarsi che le operazioni conflittuali (siano esse read-write o write-write) su uno stesso dato vengano eseguite nel medesimo ordine ovunque. Tuttavia, non si vuole un ordinamento totale sulle operazioni conflittuali onde evitare un crollo delle prestazioni: piuttosto, si preferiscono dei requisiti di consistenza più deboli che sperabilmente aiutino a ottenere un sistema con un certo grado di consistenza e, al contempo, efficiente.

### Modello di consistenza

È un contratto tra un data store distribuito e i processi, in cui il data store specifica precisamente quali sono i risultati delle operazioni di lettura e scrittura in presenza di concorrenza.

Tutti i modelli di consistenza puntano a restituire l'ultima operazione di scrittura come risultato di un'operazione di lettura; differiscono semplicemente da come l'ultima operazione di scrittura viene determinata / definita.

Esistono due macro-categorie di modelli di consistenza:

- **Modello di consistenza data-centrico:** fornisce una vista rispetto all'intero sistema (ovvero a tutti i client) di un data store consistente.
- **Modello di consistenza client-centrico:** fornisce una vista rispetto a un singolo client di un data store consistente; rispetto al modello data-centrico è più veloce ma prevede una politica di gestione della consistenza meno accurata.

### Notazione:

- >  $W_i(x)a$  = operazione di scrittura da parte del processo  $P_i$  sul dato  $x$  con valore scritto  $a$ .
- >  $R_i(x)b$  = operazione di lettura da parte del processo  $P_i$  sul dato  $x$  con valore letto  $b$ .

### Modelli di consistenza data-centrici



Nota 1: eventual = finale.

Nota 2: i modelli di consistenza stretta, linearizzabile e sequenziale sono gli unici tre a dare a tutti i client l'illusione che il sistema sia composto da un'unica replica.

### Consistenza stretta:

Qualsiasi read su un dato x restituisce il valore della write più recente su x. La write, quindi, deve essere eseguita su tutte le repliche come singola operazione atomica, e deve essere vista istantaneamente da tutti i processi.

In altre parole, la consistenza stretta impone un ordinamento temporale assoluto di tutti gli accessi all'archivio di dati e richiede un clock fisico globale. In un sistema distribuito, ciò si traduce con la necessità di avere una sincronizzazione particolarmente stretta tra i clock fisici. Tuttavia, in generale, la latenza di comunicazione è molto maggiore del tempo che intercorre tra due istruzioni sull'archivio di dati. Perciò, implementare la consistenza stretta in modo efficiente è eccessivamente difficile.

### Consistenza linearizzabile:

Ogni operazione deve apparire come istantanea in un momento tra il suo inizio e il suo completamento: nella pratica, deve avere un timestamp globale dato da un clock sincronizzato (e.g. mediante il protocollo NTP). Perciò, deve esistere un unico **interleaving** (ovvero le operazioni devono apparire nello stesso ordine per tutti i processi) e le operazioni devono essere ordinate in base ai loro timestamp: se  $ts_{OP1}(x) < ts_{OP2}(y)$ , allora OP1(x) deve apparire prima di OP2(y) nell'interleaving.

Una differenza rispetto alla consistenza stretta sta nella possibilità di avere delle operazioni sovrapposte (i.e. con lo stesso timestamp); il modello non impone un ordinamento particolare per queste operazioni: l'importante è che tutti i processi le vedano nello stesso ordine, a prescindere da quale sia quella avvenuta realmente prima nel tempo.

Per implementare la consistenza linearizzabile, è possibile far sì che tutte le scritture e letture vengano effettuate sul medesimo data center (i.e. il principale). In particolare, per quanto riguarda le scritture, il data center principale propaga l'aggiornamento sugli altri data center e non lo mette a disposizione per le letture fin tanto che tutte le repliche non si sono aggiornate correttamente. Chiaramente questo può portare a un abbassamento delle performance, per cui la consistenza linearizzabile ha senso solo in un contesto distribuito in un numero molto limitato di repliche (tipicamente due o tre).

### Consistenza sequenziale:

Il risultato di una qualunque esecuzione è uguale a quello ottenuto dall'esecuzione delle operazioni (di read e write) in un qualunque ordine sequenziale che conservi l'**ordine di programma** di ciascun processo (dove l'ordine di programma è dato dall'ordine in cui un particolare processo osserva le operazioni).

Per una maggiore chiarezza, vediamo due esempi.

-> Esempio di data store sequenzialmente consistente:

P1:	W(x)a
P2:	W(x)b
P3:	R(x)b      R(x)a
P4:	R(x)b      R(x)a

Tale data store ammette più interleaving differenti che soddisfano l'ordine di programma di ciascun processo:

W<sub>2</sub>(x)b R<sub>3</sub>(x)b R<sub>4</sub>(x)b W<sub>1</sub>(x)a R<sub>4</sub>(x)a R<sub>3</sub>(x)a

W<sub>2</sub>(x)b R<sub>4</sub>(x)b R<sub>3</sub>(x)b W<sub>1</sub>(x)a R<sub>4</sub>(x)a R<sub>3</sub>(x)a

W<sub>2</sub>(x)b R<sub>3</sub>(x)b R<sub>4</sub>(x)b W<sub>1</sub>(x)a R<sub>3</sub>(x)a R<sub>4</sub>(x)a

W<sub>2</sub>(x)b R<sub>4</sub>(x)b R<sub>3</sub>(x)b W<sub>1</sub>(x)a R<sub>3</sub>(x)a R<sub>4</sub>(x)a

-> Esempio di data store non sequenzialmente consistente:

P1:	W(x)a	
P2:	W(x)b	
P3:	R(x)b	R(x)a
P4:	R(x)a	R(x)b

P3 and P4 read write operations performed by P1 and P2 in a different order

Di fatto, qui non si ha alcun interleaving ammissibile.

Il modello di consistenza sequenziale è un indebolimento della consistenza stretta e linearizzabile, tant'è vero che non richiede neanche l'utilizzo del clock fisico. È *programmer-friendly*, anche se è difficile da implementare in maniera efficiente. Le opzioni per l'implementazione sono l'uso di un sequencer globale (soluzione centralizzata) e il protocollo multicast totalmente ordinato (soluzione decentralizzata).

#### Consistenza causale:

Le operazioni di write che sono potenzialmente in relazione di causa / effetto devono essere viste da tutti i processi nello stesso ordine. Le operazioni di write concorrenti, invece, possono essere viste in ordine differente da processi differenti.

Per stabilire quali sono le operazioni in relazioni di causa / effetto si applicano le seguenti regole:

- Read seguita da una write sullo stesso processo: la write è (potenzialmente) causalmente correlata con la read.
- Write di un dato seguita da una read dello stesso dato su processi diversi: la read è (potenzialmente) causalmente correlata con la write.
- Si applica la proprietà transitiva: se OP1 è causalmente ordinata con OP2 e OP2 è causalmente ordinata con OP3, allora OP1 è causalmente ordinata con OP3.
- Se due processi scrivono simultaneamente, le due write non sono causalmente correlate (e si dicono **write concorrenti**).

Vediamo anche qui due esempi.

-> Esempio di data store non causalmente consistente:

P1:	W(x)a	
P2:	R(x)a	W(x)b
P3:	R(x)b	R(x)a
P4:	R(x)a	R(x)b

Notiamo che qui W<sub>1</sub>(x)a e W<sub>2</sub>(x)b sono scritture causalmente ordinate per la proprietà transitiva, per cui i processi P3, P4 avrebbero dovuto osservare le due write nel medesimo ordine.

-> Esempio di data store causalmente consistente:

P1:	W(x)a	
P2:	W(x)b	
P3:	R(x)b	R(x)a
P4:	R(x)a	R(x)b

Qui invece le due scritture sono concorrenti, per cui è tollerato che i processi P3 e P4 non vedano le due write nel medesimo ordine.

Il modello di consistenza causale è un rilassamento della consistenza sequenziale e abbatte l'illusione che ci sia una singola replica all'interno del sistema. Comunque sia, la latenza di comunicazione è meno problematica rispetto ai casi precedenti; però l'implementazione risulta non banale poiché richiede un meccanismo per tenere traccia di quali operazioni sono in relazione causa / effetto (i.e. quali processi hanno visto quali scritture). A tal proposito, esistono due diverse soluzioni:

- Costruire un grafo delle dipendenze che mostri quali operazioni dipendono da quali altre operazioni.
- Utilizzare i clock logici vettoriali.

#### Consistenza finale:

A seguito di un aggiornamento, tutte le repliche diventano gradualmente consistenti entro una finestra temporale detta **inconsistency window**: durante tale finestra di tempo sono ammesse inconsistenze.

In assenza di fallimenti, l'ampiezza dell'inconsistency window dipende dalla latenza di comunicazione, dal numero di repliche e dal carico del sistema.

Dal punto di vista pratico, col modello di consistenza finale, le repliche vengono sempre resi disponibili per le letture ma, appunto, potrebbero risultare non consistenti con l'ultima scrittura: a differenza dei modelli di consistenza stretta e linearizzabile, qui si privilegia la disponibilità a discapito della consistenza perfetta.

Tale modello di consistenza presenta i seguenti vantaggi:

- È semplice e poco costoso da implementare.
- Consente letture e scritture veloci sulla replica locale.
- Ha trovato impiego in svariati contesti reali: un esempio importante è DNS, in cui i server non autoritativi mantengono le informazioni memorizzate in cash con un TTL (time to live), in modo tale che, se dovessero diventare inconsistenti, lo sarebbero solo per una finestra temporale limitata. Altri esempi sono dati da Amazon Dynamo, AWS S3, CouchDB, Dropbox, Git e iPhone sync.

Dall'altro lato, la consistenza finale è caratterizzata dai seguenti svantaggi:

- Non offre ai processi l'illusione di avere una singola copia.
- A causa della possibile inconsistenza dei dati dovuta a scritture conflittuali, richiede di risolvere i conflitti tramite un **algoritmo di riconciliazione**.

Per quanto concerne la riconciliazione, può avvenire secondo due strategie diverse:

- > **Last write wins**: i dati vengono contrassegnati con un timestamp dato da un clock vettoriale con lo scopo di catturare le relazioni di causalità tra versioni differenti di dati.
- > Lasciare la risoluzione del conflitto allo sviluppatore.

Anche sul quando effettuare la riconciliazione esistono due possibilità:

- In occasione delle letture: in questo modo sul sistema le write possono essere sempre effettuate; tuttavia, le letture risulteranno lente.
- In occasione delle scritture: la riconciliazione può essere effettuata in modo asincrono rispetto alle write stesse.

Il modello di consistenza finale, tra l'altro, ha riscosso successo a seguito del problema delle partizioni di rete e, in particolare, a seguito dell'introduzione del **teorema CAP**.

#### **Teorema CAP**

In ogni istante di tempo, un qualunque data store condiviso può soddisfare al più due delle seguenti tre proprietà:

- > **Consistency (C)**: tutti i client vedono la stessa vista del data store, anche in presenza di aggiornamenti.
- > **Availability (A)**: tutti i client possono trovare una qualche replica di dati, anche in presenza di fallimenti.
- > **Tolerance to network partitions (P)**: le proprietà del sistema valgono anche se il sistema viene partizionato.

Poiché è desiderabile avere un sistema distribuito che continui a funzionare normalmente anche a seguito di una partizione di rete, fissiamo P. A questo punto, consistenza e disponibilità non possono essere conseguite nello stesso momento quando si verifica una partizione di rete. Quale delle due preferire? È una scelta di design che dipende dall'applicazione che si sta sviluppando.

- **Sistema CP** = sistema in cui la consistenza ha maggiore priorità; in caso di partizioni di rete, non è disponibile per accettare nuovi aggiornamenti.

- **Sistema AP** = sistema in cui la disponibilità ha maggiore priorità; in caso di partizioni di rete, continua ad accettare operazioni di write ma, a seguito di una read, può restituire un risultato che non tiene conto dell'ultima scrittura completata. È anche in questo contesto che possiamo parlare di modello di consistenza finale.

### ACID vs BASE

Sono due approcci che mirano a garantire consistenza e affidabilità dei dati.

-> **ACID** (Atomicity, Consistency, Isolation, Durability): è un approccio pessimistico, che cerca di prevenire i conflitti. Tuttavia, utilizza un modello di consistenza forte, per cui è difficile da scalare.

-> **BASE** (Basically Available, Soft state, Eventual consistency): è un approccio ottimistico, in cui si lascia che i conflitti avvengano e poi si cerca di risolverli con la riconciliazione. Il sistema è disponibile la maggior parte del tempo ma potrebbe esistere un sottosistema momentaneamente non disponibile (Basically Available). I dati non necessariamente sopravvivono a seguito di failure, ma è lo sviluppatore che deve occuparsi della loro persistenza (Soft state). Inoltre, a seguito di un aggiornamento, il sistema può convergere a uno stato consistente dopo una finestra temporale (Eventually consistent).

### Protocolli di consistenza data-centrica

#### Protocolli primary-based:

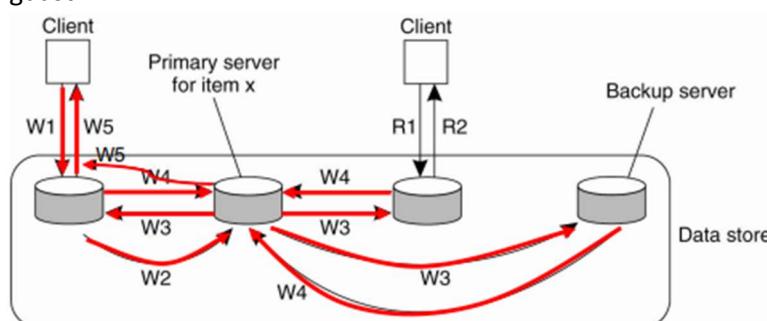
Sono anche detti protocolli primary-backup, di replicazione passiva o leader-based replication.

A ogni dato x è associata una replica primaria (leader) che ha il compito di coordinare le operazioni di scrittura su x sulle repliche secondarie (follower). Per quanto invece riguarda le letture di x, possono essere eseguite su qualsiasi replica (e.g. sulla replica locale al client).

I protocolli primary-based si suddividono a loro volta in due categorie.

-> **Protocolli primary-based di tipo remote-write**: ogni operazione di scrittura di x viene inviata alla replica primaria (eventualmente remota), che poi la inoltra alle repliche secondarie.

Tali protocolli sono usati tipicamente nei database distribuiti (e.g. MySQL), in alcuni data store NoSQL (e.g. MongoDB), nei MQS (e.g. Kafka) e nei file system distribuiti, dove si richiede un grado elevato di tolleranza ai guasti.



W1. Write request

W2. Forward request to primary

W3. Tell backups to update

W4. Acknowledge update

W5. Acknowledge write completed

R1. Read request

R2. Response to read

In figura: protocollo  
remote-write bloccante

Nei protocolli primary-based di tipo remote-write, l'aggiornamento delle repliche può essere effettuato in

modo **bloccante** o in modo **non bloccante** per il client.

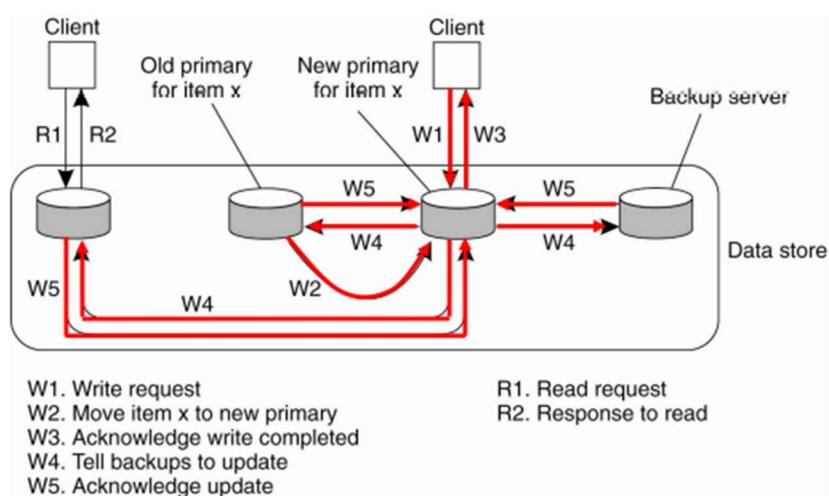
Nel primo caso, la replica primaria invia un ack al client solo quando la scrittura è stata completata su tutte le repliche; di conseguenza, si tratta di un approccio adatto al modello di consistenza linearizzabile, presenta una maggiore tolleranza ai guasti (incluso il crash della replica primaria) ed è caratterizzato da una maggiore lentezza.

Nel secondo caso, invece, la replica primaria invia un ack al client già quando la scrittura è stata completata su di essa; di conseguenza, si tratta di un approccio adatto al modello di consistenza sequenziale, presenta una minore tolleranza ai guasti e comporta una minore attesa per il client (per cui è più adatto quando si hanno molte repliche distribuite geograficamente).

-> **Protocolli primary-based di tipo local-write:** per ogni operazione di scrittura, la copia primaria del dato  $x$  migra verso la replica locale rispetto al client; dopodiché la replica locale inoltra la write alle altre repliche.

Tali protocolli sono molto utilizzati ad esempio in ambito mobile computing.

Stavolta l'aggiornamento delle repliche può essere effettuato esclusivamente in modo non bloccante rispetto al client.



#### Protocolli replicated-write:

Non prevedono un controllo centralizzato delle scritture da parte della replica primaria, bensì considerano scritture eseguite su molteplici repliche.

Anche i protocolli replicated-write si suddividono a loro volta in base a due possibili approcci.

-> **Replicazione attiva (o multi-leader replication):** le letture avvengono sulla replica locale, mentre le scritture avvengono su ogni replica. Onde evitare di ritrovarsi write in ordine differente sulle diverse repliche, si implementa il meccanismo del **multicast totalmente ordinato** (sia esso decentralizzato col clock logico scalare oppure centralizzato col sequencer).

La replicazione attiva supporta il modello di consistenza sequenziale: infatti, prevede che le scritture vengano osservate nello stesso ordine su tutte le repliche, ma tale ordine può anche essere diverso da quello temporale.

-> **Protocolli quorum-based:** le scritture e le letture, per poter essere effettuate, richiedono un particolare quorum di votazioni da parte delle repliche. In particolare, consideriamo  $N$  repliche di un dato  $x$  (quindi un totale di  $N$  voti). Allora, la read di  $x$  richiede un **quorum per la lettura**  $N_R$  per garantire che venga letta l'ultima versione di  $x$ , mentre la write su  $x$  richiede un **quorum per la scrittura**  $N_W$  per completare l'aggiornamento.

a) Se si vogliono impedire conflitti di tipo lettura-scrittura, deve valere  $N_R + N_W > N$ .

b) Se si vogliono impedire conflitti di tipo scrittura-scrittura, deve valere  $N_W > N/2$ .

Se entrambe queste due condizioni sono soddisfatte, allora si garantisce la consistenza sequenziale.

Consideriamo ora alcune configurazioni specifiche per  $N_R$  e  $N_W$ :

- 1)  $N_R = 1$  e  $N_W = N$  (**ROWA – Read Once Write All**): comporta letture veloci e scritture lente.
- 2)  $N_W = 1$  e  $N_R = N$  (**RAWO – Read All Write Once**): comporta letture lente, scritture veloci e la possibilità di avere dei conflitti di tipo scrittura-scrittura.
- 3)  $N_W = N_R = (N/2) + 1$  (**Majority**): comporta letture e scritture relativamente lente, ma anche un'alta disponibilità.

Quello che spesso si fa nella pratica è configurare in maniera arbitraria i quorum per la lettura e per la scrittura in modo tale da poter scegliere se avere una consistenza forte (con una maggiore latenza di comunicazione) oppure un modello di consistenza finale (con una maggiore efficienza). Ad esempio, Cassandra consente la configurazione dei valori di  $N_R$ ,  $N_W$  ( $\rightarrow$  **tunable consistency**).

### Modelli di consistenza client-centrici

L'obiettivo della consistenza client-centrica è fornire garanzie a un singolo client relative alla consistenza degli accessi da parte di quel client a un archivio di dati distribuito.

Ad esempio, consideriamo un archivio di dati distribuito a cui un utente accede tramite un dispositivo mobile, e supponiamo che nella posizione A l'utente acceda alla replica locale dell'archivio, eseguendo letture e scritture. Dopo essersi spostato nella posizione B (dove la replica locale è diversa rispetto alla posizione A), l'utente desidera che i dati scritti e/o letti precedentemente in A risultino allo stesso modo anche in B: in tal caso, l'archivio di dati apparirebbe effettivamente consistente al singolo utente.

Esistono 4 modelli di consistenza client-centrici differenti:

- > **Monotonic-read** (o **read-after-read**): se il client effettua due letture sul dato x, la seconda lettura non può restituire una versione precedente di x rispetto a quella restituita dalla prima lettura (in altre parole, le letture di quel client non possono tornare indietro).
- > **Monotonic-write** (o **write-after-write**): le scritture di uno stesso client non possono tornare indietro.
- > **Read-your-writes** (o **read-after-write**): se il client effettua una scrittura sul dato x e successivamente una lettura su x, deve essere in grado di leggere il valore che egli stesso ha inserito e non il valore di una versione precedente.
- > **Writes-follow-reads** (o **write-after-read**).

## TOLLERANZA AI GUASTI NEI SISTEMI DISTRIBUITI

### Dependability

È la capacità di un sistema di fornire un servizio che può essere considerato fidato in maniera giustificata. In pratica, permette di evitare interruzioni di servizio più frequenti e importanti di quanto si possa tollerare. Vediamo alcune proprietà di un sistema *dependable*.

-> **Disponibilità**: è la probabilità che il sistema sia correttamente operativo nell'istante di tempo t.

È una proprietà che usualmente si misura con la scala dei 9 ed è definita nel seguente modo:

$A = \text{MTTF} / (\text{MTTF} + \text{MTTR})$ , dove:

- MTTF = Mean Time To Failure = tempo medio in cui il sistema riesce a rimanere operativo prima di un fallimento.

- MTTR = Mean Time To Repair = tempo medio necessario per riparare il sistema.

- MTTF + MTTR = MTBF = Mean Time Between Failure = tempo medio che intercorre tra due failure.

-> **Affidabilità**: è la probabilità condizionale che il sistema sia correttamente funzionante in  $[0, t]$  given that il sistema stesso era funzionante all'istante 0.

La metrica dell'affidabilità è MTTF.

Per convincersi che si tratta di una proprietà diversa dall'affidabilità, basti pensare ai due esempi seguenti:

1) Sistema non funzionante per 1 ms ogni ora: ha una disponibilità elevata (a 6 zeri) ma ha affidabilità bassa, essendo MTTF = 1 ora.

2) Sistema che non smette mai di funzionare ma spento per due settimane l'anno: ha una disponibilità bassa (pari al 96%) ma è altamente affidabile.

-> **Safety**: è un parametro che indica quanto un eventuale malfunzionamento del sistema comprometta la sicurezza di persone o impianti relazionati al sistema stesso.

-> **Manutenibilità**: è un parametro che misura la facilità con cui il sistema può essere riparato dopo un guasto.

La metrica della manutenibilità è MTTR.

-> **Integrità**: è un parametro che indica l'assenza di alterazioni improprie del sistema.

### Failure vs error vs fault:

- Failure (fallimento) = comportamento del sistema non conforme alle specifiche (e.g. crash del programma).

- Error (errore) = stato interno non corretto di un componente che può determinare una failure (e.g. bug di programmazione).

- Fault (guasto) = causa di un errore (e.g. dovuto a un programmatore distratto).

## fault → error → failure

### Strumenti per la dependability:

-> **Prevenzione di guasti**: la si può conseguire ad esempio migliorando la progettazione del sistema.

-> **Tolleranza ai guasti**: il sistema continua a funzionare in modo conforme alle sue specifiche anche in presenza di guasti in qualche componente (i quali vengono mascherati).

-> **Rimozione di guasti**: la si può conseguire riducendo il numero e la serietà dei guasti.

-> **Predizione di guasti**: la si può conseguire stimando l'incidenza futura e le conseguenze dei guasti, ad esempio mediante algoritmi di machine learning.

### Tipi di failure dei componenti di un sistema:

- **Crash**: il componente si arresta dopo aver funzionato correttamente fino a quel momento; è il tipo di guasto più innocuo.

- **Omissione:** il componente non risponde a una richiesta (ad esempio perché è sovraccarico).
- **Fallimento nella temporizzazione:** il componente risponde ma il tempo di risposta supera un valore massimo stabilito.
- **Fallimento nella risposta:** la risposta del componente non è corretta.
- **Fallimento arbitrario (o bizantino):** il componente può produrre una risposta arbitraria; tale tipo di guasto può presentare sintomi diversi a osservatori differenti, per cui viene considerato il più grave.

#### Modelli di failure:

Supponiamo che il processo P, nella comunicazione col processo Q, stia attendendo da Q una risposta. In questo scenario, si possono avere i seguenti modelli di failure:

- > Fallimento **fail-stop:** Q ha subito un crash e P può scoprire il fallimento (tramite timeout o preannuncio da parte di Q).
- > Fallimento **fail-silent:** Q ha subito un crash o un'omissione ma P non può distinguere i due casi.
- > Fallimento **fail-safe:** Q ha subito un fallimento arbitrario ma senza conseguenze.
- > Fallimento **fail-arbitrary:** Q ha subito un fallimento arbitrario non osservabile.

#### Failure detection:

Per rilevare un fallimento, si possono seguire tre diverse strategie:

- Failure detection **attiva:** avviene mediante l'invio di un messaggio al quale viene associato un timeout per rilevare se un processo è fallito; è la soluzione più usata ed è adatta per il fallimento fail-stop.
- Failure detection **passiva:** avviene mediante l'attesa di messaggi periodici da parte di un processo: se tali messaggi non arrivano, si può assumere che il mittente abbia avuto un fallimento.
- Failure detection **proattiva:** avviene mediante scambio di informazioni tra vicini (e.g. disseminazione delle informazioni basata su gossiping).

Per quanto concerne il timeout nello specifico, possono esserci difficoltà nell'impostarlo, in particolar modo in un sistema distribuito asincrono (in cui non c'è un upper bound per il tempo di trasmissione dei messaggi). Inoltre, può risultare difficile distinguere un fallimento di un processo da un fallimento all'interno della rete, così come distinguere un crash da un'omissione.

Dunque, come possiamo nella pratica rilevare in modo affidabile che un processo (Q) sia effettivamente andato in crash?

- > Un processo P interroga un altro processo Q.
- > Se Q risponde, viene considerato *alive* da P; altrimenti, se non risponde entro t unità di tempo, diventa sospetto per essere andato in crash (nel caso di un sistema distribuito sincrono il sospetto diviene certezza).
- > Se, nel momento in cui Q è sospetto, Q invia a P un messaggio, allora P smette di sospettare di un crash di Q e incrementa il valore t del timeout.

#### Ridondanza:

È la tecnica principale per mascherare i guasti, e ne esistono diverse tipologie:

- **Ridondanza delle informazioni** (e.g. bit di parità).
- **Ridondanza nel tempo** (un'azione viene eseguita più volte): è una tecnica utile nel caso in cui si abbiano guasti transienti o intermittenti.
- **Ridondanza fisica** (aggiunta di attrezature o processi extra).

Un esempio di ridondanza fisica è dato dal **Triple Modular Redundancy** (TMR – ridondanza a 3 moduli). In questo scenario si hanno tre componenti replicati che eseguono un'operazione; il risultato di tale operazione viene sottoposto a una votazione per produrre un unico output, dove la votazione viene effettuata da parte di tre *voter* replicati (chiaramente anch'essi possono essere soggetti a guasti). Se uno dei tre componenti replicati fallisce, gli altri due possono mascherare e correggere il guasto.

## **Resilienza dei processi**

È la capacità di un sistema distribuito di mantenere un livello di servizio accettabile anche in presenza di guasti e di minacce alla normale attività.

Per ottenere la resilienza, è necessario mascherare i guasti dei singoli processi; per farlo, si replica e distribuisce la computazione di un gruppo di processi identici. La replicazione può essere a sua volta passiva o attiva:

-> **Replicazione passiva**: il gruppo di processi è organizzato in modo gerarchico: una sola replica primaria esegue le azioni sui dati, mentre le altre repliche (passive) fungono esclusivamente da back-up. La ridondanza può essere **calda** (se tutte le repliche sono sempre consistenti e allineate con il master) oppure **fredda** (se le repliche secondarie non sono perfettamente aggiornate, per cui si sfrutta una tecnica di checkpointing per salvare via via lo stato delle repliche stesse). Se la replica primaria subisce un crash, le altre repliche eseguono un algoritmo di elezione per individuare il nuovo coordinatore. Un esempio di replicazione passiva è dato dai protocolli primary-based per la consistenza.

-> **Replicazione attiva**: il gruppo di processi è organizzato in modo flat: si hanno tante repliche attive che si coordinano tra loro. Un esempio di replicazione passiva è dato dai protocolli replicated-write per la consistenza.

### Mascheramento di guasti all'interno dei gruppi:

Consideriamo un gruppo flat composto da N processi tutti identici che eseguono i comandi nello stesso ordine (in modo da essere certi che i processi facciano esattamente la stessa cosa). Il gruppo è detto **k-fault tolerant** se può mascherare k guasti concorrenti, dove k è il grado di tolleranza ai guasti.

Ma quanto deve essere grande un gruppo k-fault tolerant?

- Se il fallimento è fail-stop o fail-silent →  $N \geq k+1$ : infatti, nessun processo del gruppo produrrà un risultato errato, per cui il risultato di un solo processo non guasto è sufficiente.
- Se il fallimento è arbitrario e il risultato del gruppo è definito tramite un meccanismo di voto →  $N \geq 2k+1$ : infatti, c'è bisogno di almeno  $k+1$  processi non guasti affinché il risultato corretto possa essere ottenuto con un voto a maggioranza.

## **Consenso**

Consideriamo un gruppo composto da processi non identici, ovvero un gruppo dove c'è una computazione distribuita. Il nostro obiettivo ora è fare in modo che i processi non guasti del gruppo raggiungano un consenso unanime su uno stesso valore (e.g. il prossimo comando da eseguire) in un numero finito di passi, nonostante l'eventuale presenza di processi guasti; i guasti possono essere sia non bizantini (e.g. crash, omissioni) che bizantini.

Il gruppo e la relativa comunicazione possono avere caratteristiche diverse:

- > **Processi**: possono essere sincroni o asincroni. Se sono sincroni, operano in modalità lock-step, ovvero esiste un  $c \geq 1$  tale per cui, se un processo ha eseguito  $c+1$  passi, tutti gli altri processi hanno eseguito almeno un passo.
- > **Ritardo nella comunicazione**: può essere limitato o illimitato.
- > **Ordinamento dei messaggi**: i messaggi possono essere consegnati nello stesso ordine in cui sono stati inviati oppure senza un ordine preciso.
- > **Trasmissione dei messaggi**: può essere unicast o multicast.

Lo schema raffigurato nella pagina successiva mostra quali sono le condizioni necessarie per poter raggiungere il consenso in un sistema distribuito soggetto a guasti.

		Message ordering				Communication delay
		Unordered		Ordered		
Process behavior	Synchronous	X	X	X	X	Bounded
	Asynchronous			X	X	Unbounded
Process behavior	Synchronous				X	Bounded
	Asynchronous				X	Unbounded
		Unicast	Multicast	Unicast	Multicast	Message transmission

#### Teorema FLP (Fischer, Lynch, Patterson):

In un modello asincrono con un ritardo dei messaggi non limitato ma finito, dove un solo processore potrebbe andare in crash, non c'è alcun algoritmo distribuito che risolva il problema del consenso in tempo finito.

La buona notizia è che gli algoritmi di consenso nel caso descritto dal teorema terminano comunque in un tempo finito nella stragrande maggioranza dei casi, rendendo la probabilità che l'esecuzione non termini mai praticamente trascurabile.

Di fatto, in un ambiente asincrono, il consenso risulta difficoltoso a causa delle seguenti criticità:

- È impossibile rilevare in modo affidabile le failure poiché le velocità dei processi e i ritardi del canale non hanno un upper bound.
- Un processo *faulty* smette di inviare messaggi ma possono comunque arrivare al mittente dei messaggi "lenti" che potrebbero confondere il destinatario sullo stato del processo *faulty*.

Nella pratica, i sistemi sono **parzialmente sincroni**: nella maggior parte del tempo possono essere considerati sincroni (nel senso che prevedono l'uso del tempo e dei clock), ma bisogna tener conto che, ad esempio, non si hanno dei bound temporali sul tempo di trasmissione dei messaggi.

Comunque sia, è possibile effettuare la detection dei fallimenti di tipo crash attraverso il meccanismo basato sul "sospetto" analizzato precedentemente.

Per concludere, esamineremo tre algoritmi di consenso: **Paxos**, **Raft** e **Byzantine generals**.

### **Paxos**

#### Assunzioni:

- > Il sistema distribuito è parzialmente sincrono (ma può essere anche asincrono).
- > Sono in esecuzione N processi e il sistema può tollerare fino a k guasti non bizantini (per cui  $N \geq 2k+1$ ).
- > I processi comunicano tra loro mediante scambio di messaggi.
- > La comunicazione tra i processi può essere non affidabile (e.g. i messaggi possono essere duplicati o andare persi).
- > I processi operano a velocità arbitrarie, possono essere riavviati in caso di failure e, al momento del riavvio, possono "ricordare" delle informazioni accedendo a uno storage persistente che sopravvive ai guasti.

#### Idea:

Lo scopo è far raggiungere a un insieme di processi il consenso su un solo valore tra quelli proposti. L'idea è implementare un meccanismo quorum-based, in cui si ha un sottoinsieme di processi votanti che votano una qualche proposta; una proposta è approvata nel momento in cui viene accettata da una

maggioranza (i.e. un quorum). Un processo che deve considerare più proposte approverà quella col numero di versione (o **proposal number**) più alto.

#### Requisiti:

Paxos ha tre requisiti di safety:

- Può essere scelto esclusivamente un valore tra quelli proposti.
- Può essere scelto un unico valore.
- Un processo non può imparare che un valore sia stato scelto finché non è stato veramente scelto.

D'altra parte, ha due requisiti di liveness:

- Prima o poi dev'essere scelto un valore proposto.
- Quando un valore viene scelto, un processo può imparare tale scelta.

Tuttavia, ricordando il teorema FLP, è impossibile garantire sia la safety che la liveness in un algoritmo di consenso asincrono. In particolare, in Paxos si preferisce conservare la safety e non dare garanzie certe sulla liveness: ciò vuol dire che l'algoritmo di Paxos potrebbe non terminare mai ma, fortunatamente, si tratta comunque di una situazione molto rara.

#### Ruoli dei processi:

Ciascun processo può ricoprire uno, due o tutti i seguenti ruoli:

- > **Proposer**: propone un valore da scegliere per conto di un client.
- > **Acceptor**: è il votante che decide quale valore scegliere.
- > **Learner**: impara quale valore è stato scelto dalla maggioranza degli acceptor e riporta tale decisione al client.

Chiaramente, se esistono  $m$  acceptor, il quorum da raggiungere per la scelta del valore deve essere strettamente maggiore di  $m/2$ .

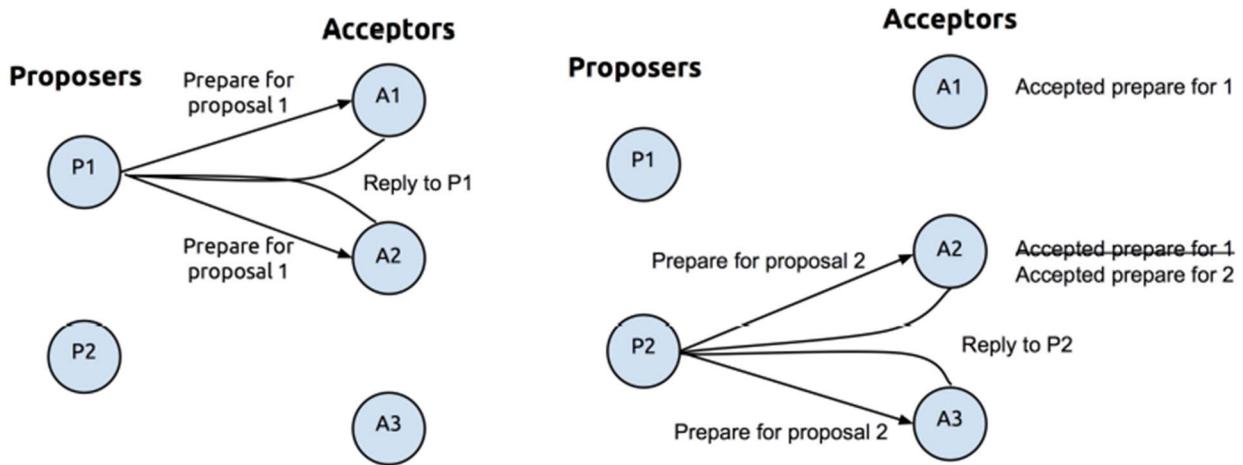
#### Funzionamento:

Paxos si articola in più round, ciascuno dei quali è articolato in due fasi: **prepare** e **accept**. Comunque sia, per comprendere al meglio il funzionamento di Paxos, analizziamolo con un esempio.

Supponiamo di avere due processi proposer  $p_1, p_2$  e tre processi acceptor  $a_1, a_2, a_3$ .

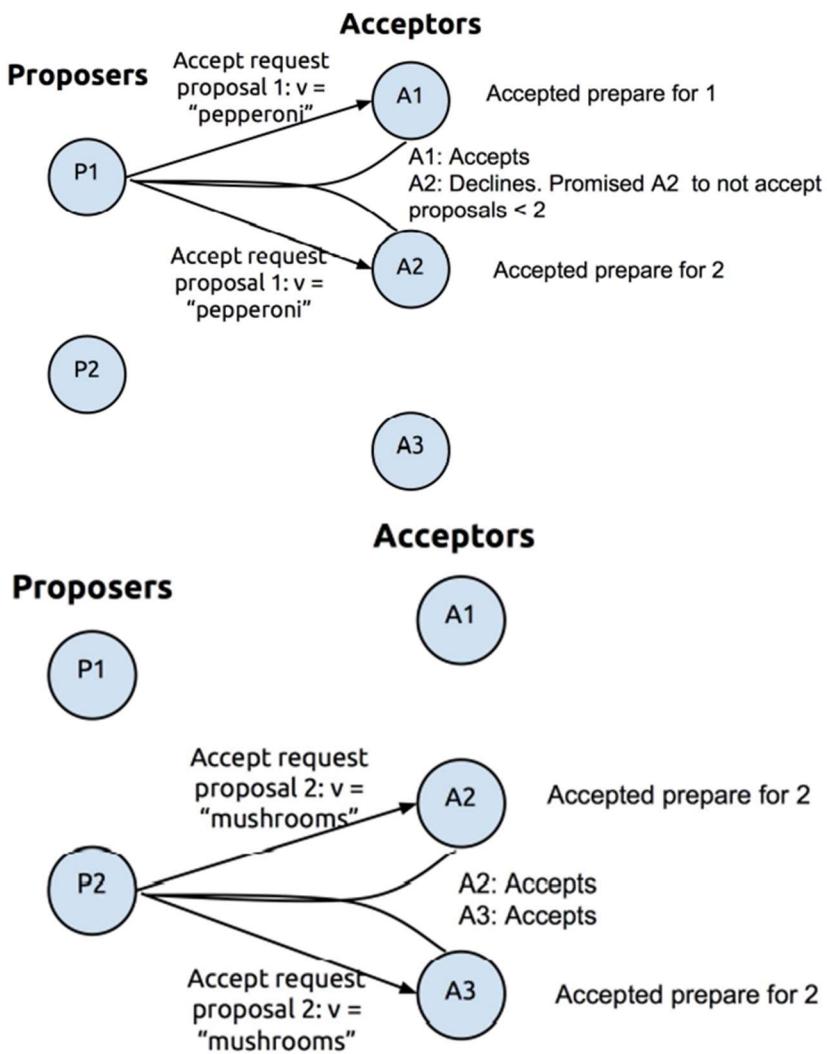
-> **1° round, fase di prepare**:  $p_1$  invia una **prepare request** (i.e. un messaggio composto esclusivamente dal proposal number e non dal valore proposto, che può ad esempio essere il tipo di pizza da ordinare) con proposal number pari a 1 ai processi  $a_1, a_2$ . Dopodiché, i due acceptor rispondono a  $p_1$ , promettendogli di non accettare in futuro richieste con proposal number minori di 1; in realtà, nel messaggio di risposta dovrebbe esserci anche l'eventuale valore precedentemente accettato relativo al proposal number più alto ma, poiché siamo ancora al primo round, tale valore ancora non esiste e  $p_1$  avrà la possibilità di decidere il valore da proporre.

Poi  $p_2$  invia una prepare request con proposal number pari a 2 ai processi  $a_2, a_3$ . In modo analogo a prima, i due acceptor rispondono a  $p_2$ , promettendogli di non accettare in futuro richieste con proposal number minori di 2.

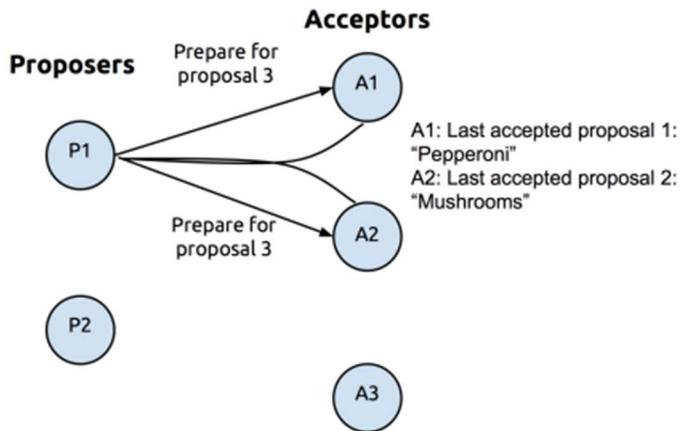


Notiamo che l'acceptor  $a_2$ , dopo aver ricevuto il secondo prepare proposal, non potrà più accettare la proposta da parte di  $p_1$ .

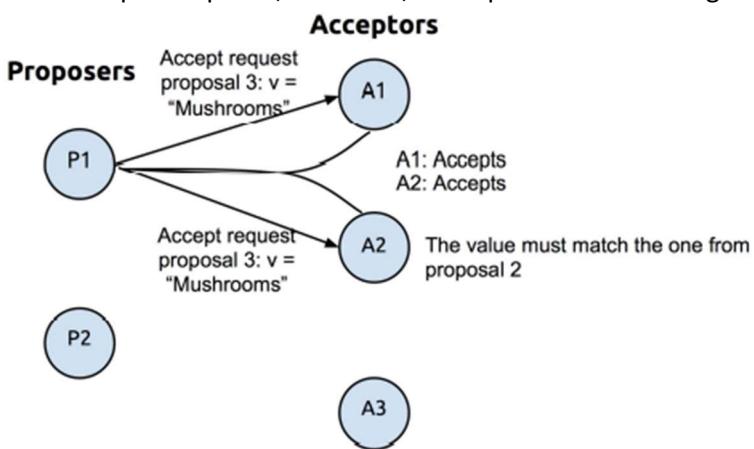
-> **1° round, fase di accept:**  $p_1$  invia un'**accept request** (i.e. un messaggio composto dal proposal number e dal valore proposto) con proposal number pari a 1 e valore “salame” agli stessi acceptor di prima, ovvero  $a_1, a_2$ . Dopodiché,  $a_1$  accetta la richiesta, mentre  $a_2$  no.  
 Poi  $p_2$  invia un’accept request con proposal number pari a 2 e valore “funghi” agli stessi acceptor di prima, ovvero  $a_2, a_3$ . Così, entrambi gli acceptor accettano la proposta. Poiché  $\{a_2, a_3\}$  costituisce una maggioranza di acceptor, la proposta di  $p_2$  è quella che viene scelta: in pratica, è stata votata la pizza coi funghi.



-> **2° round, fase di prepare**:  $p_1$  invia una prepare request con proposal number pari a 3 ai processi  $a_1, a_2$ . Dopodiché, i due acceptor rispondono a  $p_1$ , promettendogli di non accettare in futuro richieste con proposal number minori di 3. Inoltre, nelle risposte degli acceptor viene specificato il valore accettato precedentemente relativo al proposal number maggiore: "funghi".



-> **2° round, fase di accept**:  $p_1$  invia un'accept request con proposal number pari a 3 e valore "funghi" agli stessi acceptor di prima, ovvero  $a_1, a_2$ . Dopodiché entrambi gli acceptor accettano la proposta.



Per quanto riguarda la propagazione delle informazioni ai learner, esistono tre modalità possibili:

- 1) Ciascun acceptor informa tutti i learner sul valore scelto; tale approccio, però, richiede che venga inviata un'enorme quantità di messaggi.
- 2) Gli acceptor informano un *distinguished learner* (che solitamente è il proposer), il quale poi propagherà il risultato tra gli altri learner; tuttavia, tale approccio presenta un single point of failure.
- 3) Compromesso tra i due approcci precedenti: disporre di un numero limitato di *distinguished learner*, in modo tale da mitigare sia il problema del flooding dei messaggi, sia il problema del single point of failure.

Il funzionamento di Paxos così com'è stato descritto presenta ancora una falla: se ci sono più proposer che in modo concorrente propongono valori differenti, potrebbero mandare in stallo il protocollo (vedi il teorema FLP). Facciamo un esempio dove vengono coinvolti due processi proposer  $p, q$ :

- >  $p$  completa la fase 1 per il proposal number  $n_1$ .
- >  $q$  completa la fase 1 per il proposal number  $n_2 > n_1$ .
- > Nella fase 2, l'accept request di  $p$  viene ignorata, poiché  $n_1$  non è stato il proposal number più elevato tra quelli emersi nella fase 1.
- > A questo punto  $p$  anticipa l'invio dell'accept request da parte di  $q$  mandando una prepare request con proposal number  $n_3 > n_2$ ; di conseguenza, anche l'accept request di  $q$  dovrà essere ignorata.
- > E così via, all'infinito.

Di fatto, è stato dimostrato che Paxos garantisce liveness se soltanto uno dei proposer viene eletto come leader, dove il leader è l'unico proposer attivo (i.e. l'unico a inviare le prepare request): gli altri proposer inviano le loro richieste solo quando il leader attuale crasha e c'è bisogno di eleggerne uno nuovo.

#### State machine replication e Paxos:

Ricordiamo che la state machine replication costituisce un approccio generale per costruire sistemi tolleranti ai guasti e basati su server replicati.

Con un algoritmo di consenso, si vuole che ciascuna macchina a stati (i.e. ciascuna replica) processi la stessa lista di comandi, in modo tale da produrre la medesima serie di risultati e da percorrere la medesima sequenza di stati. Perciò, anche in questo contesto si ricorre a Paxos, in cui i comandi da eseguire e il loro ordinamento rappresentano i valori da concordare; tuttavia, in tal modo, ciascun comando costituisce un'istanza di Paxos, rendendo il meccanismo particolarmente inefficiente (si avrebbero svariate istanze di Paxos in esecuzione simultaneamente).

Per questo motivo, si introduce una variante di Paxos, detta **Multi-Paxos**, in cui molteplici round dell'algoritmo vengono gestiti da uno stesso processo leader. In particolare, il leader percorre sia la prepare phase che l'accept phase solo nel primo round, mentre successivamente si avrà soltanto l'accept phase (si dice che un proposer che invia solo accept request sia in **galloping mode**). La galloping mode viene interrotta nel momento in cui il leader dovesse crashare: in tal caso, dovrà essere eletto un nuovo leader.

#### Sistemi distribuiti noti che utilizzano Paxos:

- Petal (dischi virtuali distribuiti).
- Frangipani (file system distribuito scalabile).
- Chubby (servizio di lock distribuito di Google).
- Spanner (database distribuito di Google globalmente distribuito).
- XtreemFS (file system distribuito tollerante ai guasti per ambienti WAN).
- Mesos (progetto per la gestione di cluster di computer).

#### **Raft (Replicated And Fault Tolerant)**

È un algoritmo di consenso pensato per essere più semplice di Paxos ma, allo stesso tempo, più legato ai casi d'uso e ai problemi di implementazione che si possono riscontrare nel mondo reale. In particolare, rispetto a Paxos:

- > Il modello è molto simile: anche qui si assumono una comunicazione inaffidabile con messaggi ritardati o persi e la possibilità di avere dei guasti non bizantini.
- > È equivalente (a Multi-Paxos) per quanto riguarda le performance e la tolleranza ai guasti.
- > Ha la stessa efficienza per quanto riguarda la log replication (un meccanismo che serve a replicare i dati tra nodi differenti).
- > A differenza di Paxos, tende a decomporre ciascun problema in sotto-problemi relativamente indipendenti (i.e. **leader election** e **log replication**).

Raft è implementato su un cluster di server, ciascuno dei quali detiene una macchina a stati, un log contenente gli input dati alla macchina virtuale e il protocollo Raft. Un server viene eletto come leader, mentre gli altri fungono da follower.

Trattasi di un protocollo per la state machine replication, il cui obiettivo è mantenere la consistenza della macchina a stati replicata sui vari server (per cui i comandi devono essere eseguiti nello stesso identico ordine su tutte le repliche).

Ciascun server può appartenere a uno dei seguenti tre stati:

- **Leader** (può essercene al massimo uno per ogni term, dove un term è un periodo di tempo).
- **Follower**
- **Candidate** (se un follower rileva un crash da parte del leader, si candida per diventare lui stesso leader).

### Leader election:

Ciascun server mantiene il numero del term corrente all'interno di una variabile locale (non se ne ha una vista globale); tale valore cresce monotonicamente nel tempo.

Di base, il leader dovrebbe inviare periodicamente l'**heartbeat** ai follower, e comunque entro il cosiddetto **election timeout** (o heartbeat timeout). Se per un determinato follower questo timeout scade, esso assume che il leader sia andato in crash e indice una nuova elezione, votando per se stesso e inviando agli altri server un messaggio di *RequestVote*; i server che ricevono tale messaggio, replicano se e solo se non hanno già votato nel term corrente. Al termine della votazione, il server che ha ricevuto il voto da parte di una maggioranza diventa il nuovo leader.

Tuttavia, le cose potrebbero non andare bene se l'election timeout è pressoché identico per più follower che, in caso di crash del leader, indirebbero l'elezione concorrentemente: in tal caso si può andare incontro al fenomeno dello **split vote**, in cui nessun server raggiunge la maggioranza. Per questo motivo, si prendono due provvedimenti:

- L'election timeout è randomizzato tra 150 ms e 300 ms per ciascun follower.
- Si introduce un nuovo timeout (relativo all'elezione) che, se scade prima della proclamazione di un nuovo leader, porta all'apertura di una nuova elezione.

### Log replication:

Il client invia i comandi soltanto al server leader, il quale effettua un'append di ciascun comando nel proprio log e invia un messaggio di *AppendEntries* a tutti i follower con lo scopo di riportare il cambiamento di stato su tutti i server. In ogni caso, un messaggio di *AppendEntries* viene comunque mandato periodicamente da parte del leader con il ruolo di heartbeat.

Dopo aver ricevuto *AppendEntries*, i follower inviano un acknowledgement al leader. Dopo che il leader ha ricevuto gli ack dalla maggioranza dei server, esegue davvero il comando sulla sua macchina a stati e restituisce il risultato al client; a tal punto, la entry del log è considerata **committed**.

Dopodiché il leader notifica i follower dell'avvenuto commit in un altro messaggio di *AppendEntries*. Quindi i follower eseguono il comando di commit nelle proprie macchine a stati.

Il term prosegue in questo modo finché un follower non smette di ricevere l'heartbeat e non indice una nuova elezione.

### Requisiti:

Raft ha tre requisiti di safety:

- Può essere eletto al più un leader in ogni term.
- I log devono essere mantenuti consistenti.
- Soltanto i nodi con un log aggiornato possono diventare leader.

D'altra parte, ha un requisito di liveness:

- Se le failure avvenute sono sufficientemente poche, il sistema dovrà processare e rispondere a tutti i comandi dei client.

Inoltre, Raft è tollerante alle partizioni di rete.

## **Byzantine generals**

### Assunzioni:

- > Si hanno N processi sincroni e il sistema può tollerare fino a k guasti bizantini (per cui  $N \geq 3k+1$ ).
- > La comunicazione è unicast.
- > I messaggi vengono consegnati nel medesimo ordine con cui vengono inviati.
- > Il ritardo nella trasmissione dei messaggi è limitato.

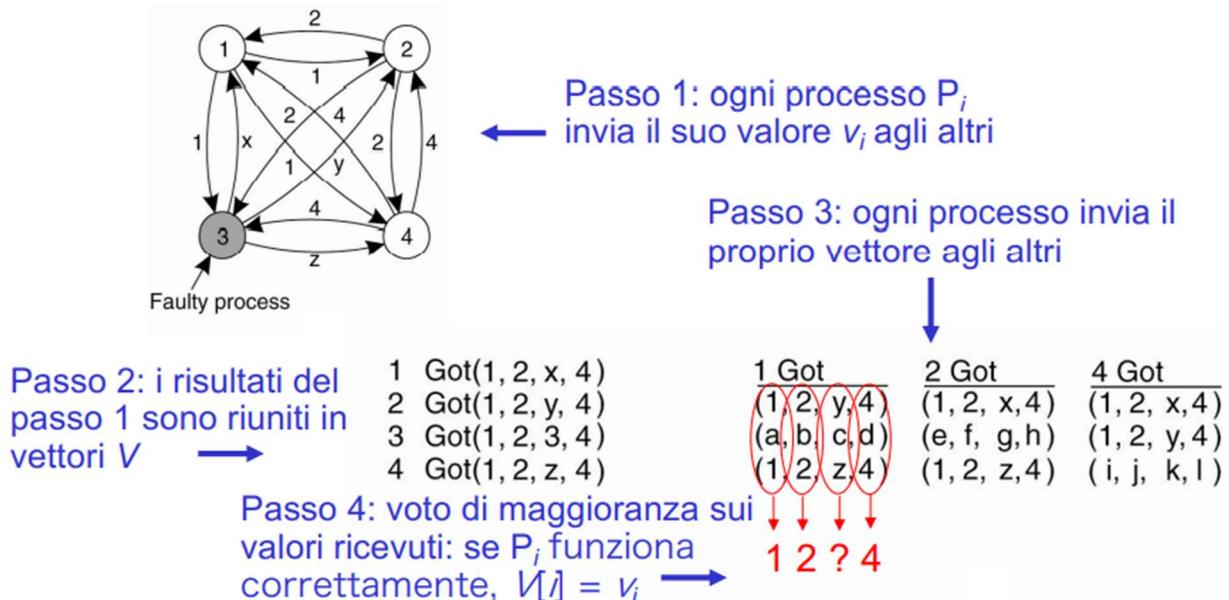
### Idea:

Lo scopo è comunicare un particolare valore tenendo conto che possono esserci fino a k processi traditori

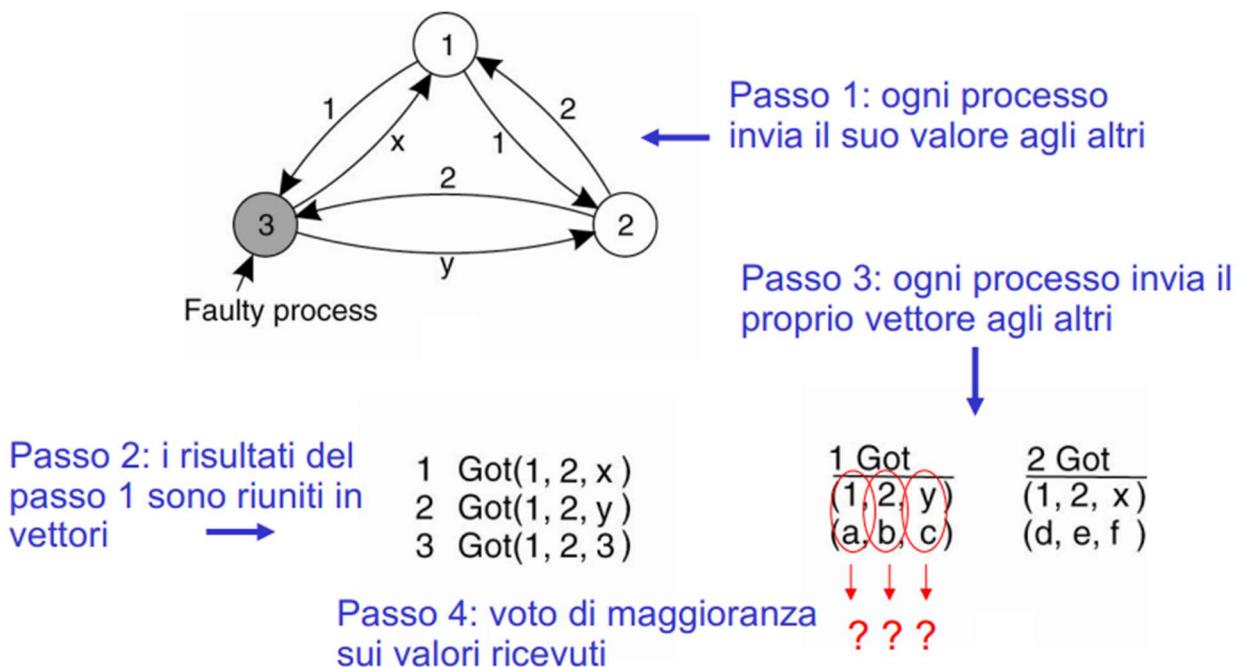
(che comunicano valori totalmente arbitrari). In particolare, possiamo immaginare N generali (ovvero processi), e ciascun generale i fornisce un valore  $v_i$  agli altri, dove  $v_i$  può ad esempio rappresentare la forza della truppa del generale i. Noi vogliamo far costruire a ogni processo un vettore V di dimensione N tale che  $V[i] = v_i$  se i è non guasto, mentre  $V[i]$  è non definito altrimenti.

#### Problema dell'accordo bizantino:

Come si può raggiungere l'accordo nel caso di tre processi che funzionano correttamente e uno fraudolento ( $N=4, k=1$ )? Vediamolo con un esempio, dove per semplicità assumiamo  $v_i = i \forall i$ :



Ma come mai con due processi correttamente funzionanti e uno fraudolento ( $N=3, k=1$ ) non si riesce a raggiungere l'accordo? Verifichiamolo con un altro esempio:

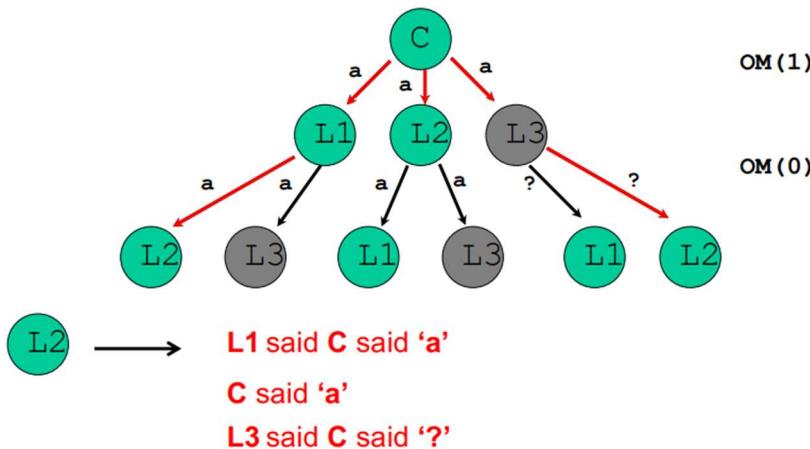


### Algoritmo di Lamport per l'accordo Bizantino:

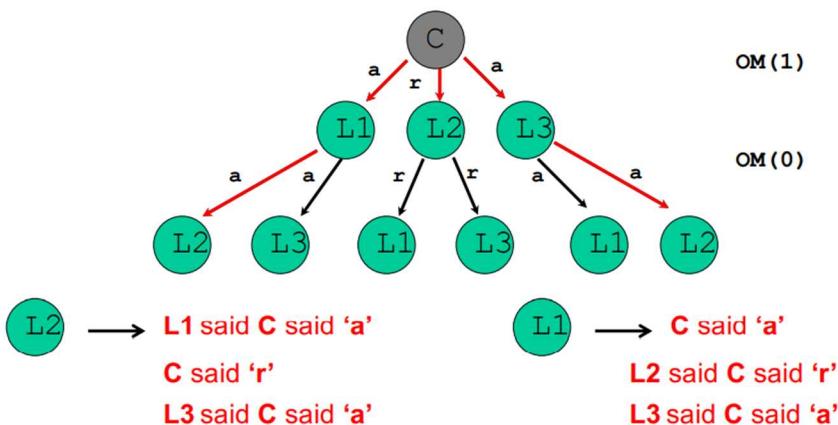
Noto anche come algoritmo OM(k), dove k è il numero massimo di guasti bizantini tollerati, si articola nei seguenti passaggi:

- 1) Si ha un processo comandante, che invia il suo valore a ciascun tenente (i.e. a ciascuno degli altri processi).
- 2)  $\forall i$  sia  $v_i$  il valore che il tenente  $i$  ha ricevuto dal comandante (nel caso in cui non abbia ricevuto alcun valore,  $v_i = \text{RETREAT}$ ). A tal punto il tenente  $i$  agisce da comandante e invia il valore  $v_i$  a ciascuno degli altri tenenti. Tale passaggio viene ripetuto  $k$  volte.
- 3)  $\forall i \forall j \neq i$  sia  $v_j$  il valore che il tenente  $i$  ha ricevuto da parte del tenente  $j$  nello step 2 (anche qui, nel caso in cui non abbia ricevuto alcun valore,  $v_j = \text{RETREAT}$ ). In tal modo, il tenente  $i$  ottiene il vettore  $(v_1, \dots, v_N)$  e utilizzerà il valore maggioritario (i.e. che compare più volte) all'interno del vettore.

Vediamo ora due esempi, dove nel primo il tenente L3 è il traditore e nel secondo il comandante C è il traditore.



**Result:  $\text{majority}(a, a, ?) = a$**



**L2 result:  $\text{majority}(a, r, a) = a$ ; L1 result:  $\text{majority}(a, r, a) = a$**

Per quanto riguarda le prestazioni dell'algoritmo, non sono un granché. Infatti, impiega  $k+1$  round asincroni e prevede lo scambio di un numero esponenziale di messaggi:  $O(N^k)$ .

Effettivamente, i protocolli tolleranti ai guasti bizantini sono stati considerati a lungo troppo onerosi per essere usati in pratica. Tuttavia, nel 1999 venne proposto **PBFT (Practical Byzantine Fault Tolerance)**, che ha dato un cambio di rotta offrendo delle prestazioni decisamente migliori.