

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

# SISTEMI DISTRIBUITI E CLOUD COMPUTING

## PARTE 1

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

## Sommario

<b>1. INTRODUZIONE AI SISTEMI DISTRIBUITI.....</b>	<b>5</b>
<b>1.1. Definizioni sistema distribuito:.....</b>	<b>6</b>
<b>1.2. Utilità del sistema distribuito .....</b>	<b>6</b>
<b>1.3. Differenze tra sistema distribuito e sistema centralizzato .....</b>	<b>7</b>
<b>1.4. Caratteristiche di un sistema distribuito.....</b>	<b>7</b>
<b>1.5. Errori nella progettazione di un sistema distribuito .....</b>	<b>9</b>
<b>1.6. Tipi di sistemi distribuiti .....</b>	<b>10</b>
<b>2. INTRODUZIONE AL CLOUD COMPUTING .....</b>	<b>11</b>
<b>2.1. Definizioni cloud computing .....</b>	<b>12</b>

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

2.2.	Paradigmi di computazione .....	13
2.3.	Caratteristiche principali dei cloud computing: .....	13
2.4.	Modelli di deployment del cloud computing .....	14
2.5.	Tipologie di servizi cloud offerte da cloud provider.....	16
2.5.1.	Confronto IaaS / PaaS .....	18
2.6.	Modelli di pricing.....	18
2.7.	Elasticità .....	18
2.7.1.	Approcci di Capacity Planning:.....	19
2.8.	Service Level Agreement .....	20
2.9.	Tipi di servizi.....	21
2.10.	Processo di sviluppo di applicazioni cloud .....	21
2.11.	Fog computing .....	21
2.11.1.	Edge computing vs Fog computing:.....	22
2.12.	Riassunto dei principali concetti .....	22
3.	ARCHITETTURE DEI SISTEMI DISTRIBUITI .....	23
3.1.	Stili architetturali dei sistemi distribuiti .....	23
3.1.1.	Stile architetture a livelli .....	24
3.1.2.	Object Based Style .....	24
3.1.3.	RESTful style .....	24
3.2.	Disaccoppiamento .....	25
3.2.1.	Event driven architecture .....	26
3.2.2.	Data oriented style .....	26
3.2.3.	Publish/subscribe style: .....	27
3.3.	Tipi di architetture .....	28
3.3.1.	Architettura di sistema centralizzata .....	28
3.3.2.	Architettura di sistema decentralizzata .....	29
3.3.3.	Architetture ibride .....	42
3.4.	Middleware .....	43
3.4.1.	Tipologie .....	43
3.5.	Sistemi software auto adattativi .....	44
3.5.1.	Pattern MAPE (Monitor, analyze, plan, execute) .....	45
3.5.2.	Esempi di sistemi auto-adattativi.....	46
4.	COMUNICAZIONE NEI SISTEMI DISTRIBUITI .....	49

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

<b>4.1. Tipi di comunicazione .....</b>	49
<b>4.1.1. Comunicazione persistente o transiente.....</b>	50
<b>4.1.2. Comunicazione sincrona .....</b>	50
<b>4.1.3. Comunicazione asincrona .....</b>	50
<b>4.1.4. Comunicazione discreta o streaming .....</b>	51
<b>4.2. Combinazioni dei tipi di comunicazione .....</b>	51
<b>4.2.1. Combinazioni tra persistenza e sincronizzazione .....</b>	51
<b>4.2.2. Combinazioni tra transienza e sincronizzazione.....</b>	52
<b>4.3. Semantica di comunicazione in un sistema client-server.....</b>	53
<b>4.3.1. Meccanismi di base per la realizzazione di semantiche di comunicazione .....</b>	53
<b>4.3.2. Tipi di semantiche.....</b>	53
<b>4.4. Programmazione di applicazioni di rete .....</b>	56
<b>4.4.1. Programmazione di rete esplicita .....</b>	56
<b>4.4.2. Programmazione di rete implicita.....</b>	57
<b>4.4.3. Gestione dell'eterogeneità nella rappresentazione dei dati .....</b>	57
<b>4.4.4. Remote Procedure Call (RPC) .....</b>	58
<b>4.5. LEZIONI SU GO .....</b>	77
<b>4.6. Comunicazione message oriented .....</b>	78
<b>4.6.1. MOM .....</b>	80
<b>4.6.2. Multicast applicativo .....</b>	86
<b>5. VIRTUALIZZAZIONE .....</b>	90
<b>5.1. Macchina virtuale .....</b>	90
<b>5.2. Vantaggi della virtualizzazione .....</b>	91
<b>5.3. VIRTUALIZZAZIONE A LIVELLO DI SISTEMA (ottenuta grazie ad un VMM) .....</b>	93
<b>5.3.1. Virtualizzazione completa o paravirtualizzazione .....</b>	94
<b>5.3.2. Problemi per realizzare la virtualizzazione di sistema .....</b>	95
<b>5.3.3. Virtualizzazione completa: soluzioni .....</b>	95
<b>5.3.4. Paravirtualizzazione .....</b>	97
<b>5.4. Architettura di riferimento VMM .....</b>	99
<b>5.5. Virtualizzazione della memoria .....</b>	99
<b>5.5.1. Differenze tra approcci pre realizzare esecuzione istruzioni privilegiate: .....</b>	100
<b>5.5.2. Mappatura a due livelli .....</b>	103
<b>5.6. Caso di studio: Xen .....</b>	103

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

5.6.1.    Pro e contro .....	104
5.6.2.    Architettura Xen .....	104
5.7.    MIGRAZIONE E RESIZING .....	106
5.7.1.    Resizing .....	107
5.7.2.    Migrazione .....	108
5.8.    Virtualizzazione a livello di SO .....	111
5.8.1.    Docker .....	114
5.9.    Lightweight operating systems e Unikernel .....	117
5.9.1.    Orchestrazione dei container .....	120
5.9.2.    Kubernetes .....	120

## 1. INTRODUZIONE AI SISTEMI DISTRIBUITI

**Legge di Metcalfe** → il valore di una rete di telecomunicazione è proporzionale al quadrato del numero utenti connessi al sistema. Per esempio, potere economico di Facebook, Google, Instagram.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

Un'altra **evoluzione tecnica** è la **capacità computazionale**, grazie alla quale i computer sono diventati sempre più

- Piccoli
- Economici
- Efficienti da punto di vista computazionale
- Veloci.

Esempi di sistemi distribuiti:

- Internet e Web
- Sistemi Cloud
- Sistemi peer-to-peer
- **Internet of Things** (IoT)

### **Sistemi distribuiti e intelligenza artificiale AI**

Si hanno tool e strumenti di machine learning di uso comunque, grazie a sistemi distribuiti sottostanti su cui queste metodologie possono essere implementate in modo efficiente. Una delle idee che viene sfruttata per avere sistemi e applicazioni di AI efficienti è la tecnica del **divide et impera**:

*ho un problema di grandi dimensioni, molto oneroso da punto di vista computazionale, e lo suddivido in sotto problemi che sono tra loro collegati ma più facilmente gestibili.*

## **1.1. Definizioni sistema distribuito:**

Abbiamo 3 definizioni

1. Insieme di elementi di computazione fra di loro autonomi (che esistono in modo individuale), che all'esterno appaiono come se fossero un singolo sistema coerente (grazie ad un layer software chiamato "Middleware"). Gli elementi autonomi spesso sono chiamati "nodi" e possono essere hardware o software.
2. Sistema in cui componenti comunicano e coordinano le loro azioni scambiandosi dei messaggi(in modalità sincrona o asincrona).
3. Un sistema distribuito è un sistema in cui il fallimento di un nodo di cui neanche sappiamo l'esistenza, può rendere il servizio inutilizzabile.

## **1.2. Utilità del sistema distribuito**

- Per **condividere risorse** (nodo di computazione, dello spazio di storage, della rete, applicazione ...)
- Per migliorare **qualità** servizio offerto
  - la posso misurare in base a diversi attributi
    - **prestazioni**

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

- voglio migliorare prestazioni sistema, per esempio ridurre tempo risposta sperimentato da utenti
- migliorare **disponibilità** del sistema distribuito
  - misuro disponibilità come % del tempo rispetto al tempo totale di funzionamento per cui il sistema è **disponibile** (*up and running*)
- Migliorarne la **sicurezza**
  - Per esempio, un sistema distribuito poiché è composto da molteplici componenti è più tollerante a attacchi di sicurezza in quanto se i componenti di controllo del sistema sono distribuiti, invece che centralizzati, per far sì che sistema non funzioni più (attacco di sicurezza abbia successo) bisogna compromettere un elevato numero di nodi del sistema ( i nodi fondamentali per il controllo).
- Permette di **distribuire componenti del sistema a larga scala** (anche su base geografica)
  - Posso avere un server in Italia, una replica del server in Germania, una in Francia ..
  - In modo da redirigere utenti verso replica a loro più vicina, in modo da colmare distanze geografiche
- Permette di **mantenere autonomia**
  - Dipende da design del sistema distribuito, che dovrebbe evitare la presenza o ridurre molto il numero di componenti centralizzate, che rappresentano un collo di bottiglia per quanto riguarda le prestazioni, sia un unico punto di fallimento (single point of failure)

### 1.3. Differenze tra sistema distribuito e sistema centralizzato

- **CONCORRENZA** - In un sistema distribuito bisogna per forza gestire la concorrenza in quanto si tratta di un sistema decentralizzato. In caso centralizzato invece è una scelta di design (posso fare un'applicazione single threaded).
- **MANCANZA DI CLOCK GLOBALE** - In un sistema distribuito ci sono tanti clock fisici, non necessariamente sincronizzati fra loro. Si ricorre al senso logico per risolvere i conflitti. In un sistema centralizzato sono invece in grado di sincronizzare thread rispetto ad eventi di sistema.
- **INDIPENDENZA** - Il fallimento di un thread comporta il fallimento dell'eventuale intero sistema centralizzato, a differenza di un sistema distribuito nel quale il fallimento di un nodo compromette solo parzialmente il tutto.

### 1.4. Caratteristiche di un sistema distribuito

- **Eterogeneità** - un sistema distribuito è fatto da numerosi nodi, ovvero differenti reti, sistemi operativi, linguaggi di programmazione o addirittura diversi sviluppatori. Per affrontare questo problema di eterogeneità aggiungiamo un livello software chiamato middleware, una sorta di sistema operativo del sistema distribuito, che fornisce un'astrazione di programmazione che maschera l'eterogeneità al di sotto di lui. Il middleware fornisce un servizio di comunicazione fra i nodi del sistema distribuito, tramite:

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

- Remote Procedure Call (RPC)- chiamata a procedura remota, che ci permette di eseguire un metodo su un nodo remoto senza doverci occupare dei livelli di comunicazione
- Remote Method Invocation (RMI)- invocazione remota di metodo (in generale per linguaggi ad oggetti tipo java)
- Message Oriented Middleware (MOM)- orientati ai messaggi in cui viene adottata una comunicazione persistente, quindi i due che comunicano non devono essere sempre esistenti nella comunicazione, perché il messaggio viene salvato.

- **Trasparenza** - Si vuole evitare che gli utenti possano vedere risorse e processi del sistema.

Tipi di trasparenza di distribuzione:

- All'accesso: nasconde le differenze nella rappresentazione dei dati e nel modo in cui le risorse sono accedute
- Alla locazione: viene nascosto il dove la risorsa sia effettivamente localizzata, l'esempio è l'URL che nasconde l'indirizzo IP che può essere associato alla locazione. Se un sistema ci nasconde sia accesso che locazione si parla di trasparenza di rete
- Alla migrazione: le risorse possono essere spostate da una locazione all'altra senza compromettere l'operatività del sistema.
- Alla replicazione: gli utenti non devono accorgersi che ci sono molteplici repliche della stessa risorsa
- Alla concorrenza: deve essere nascosto il fatto che le risorse vengono usate in modo condiviso da tanti utenti in maniera concorrente.
- Ai fallimenti: vogliamo che i fallimenti siano nascosti, ovvero che l'utente non si accorga se una risorsa va in crash e poi torna a funzionare

Per motivi di costi, in un sistema distribuito non vengono fornite contemporaneamente tutte queste trasparenze.

EX: Le latenze di comunicazione non possono essere sempre nascoste, tra Roma e NY la latenza di comunicazione è almeno 23 millisecondi, non impercettibile. In caso di tanti nodi, nodi su larga scala è impossibile nascondere i fallimenti. Non possiamo distinguere un nodo che è sovraccarico da uno che ha un crash, e quindi ha fallito. Se un client comunica con un server e il server fallisce, il client non può essere sicuro che il server abbia finito il compito prima di crashare.

- **Apertura**- un sistema distribuito aperto significa che è in grado di interagire con altri sistemi aperti, ovvero di operare a prescindere dall'ambiente sottostante. I sistemi distribuiti quindi devono avere delle interfacce ben definite per interoperare con altri sistemi aperti, devono garantire portabilità di applicazioni e devono essere facilmente estensibili. Affinché un sistema distribuito risulti aperto e flessibile è opportuno separare le politiche dai meccanismi; Il sistema fornisce i meccanismi, e poi una serie di politiche li gestiscono.
  - EX: Un browser web mette a disposizione meccanismi fra cui il caching dei dati. Su questo si realizzano diverse politiche per la sua gestione, per esempio salvo solo risorse pubbliche e non le private, o ne salvo solo certi tipi.
- **Scalabilità**- proprietà di mantenere adeguati livelli di performance a fronte di aumenti di
  - Numero utenti o processi (scalabilità dimensionale)

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

- Massima distanza fra i nodi (scalabilità geografica)
- Numero di domini amministrativi (scalabilità amministrativa)

Vediamo solo le soluzioni tramite scalabilità dimensionale:

- SCALABILITÀ VERTICALE (scaling up) - cambiamo le nostre risorse tramite altre più potenti.
- SCALABILITÀ ORIZZONTALE (scaling out) - aumento la quantità di risorse di stessa capacità (creo più copie dello stesso server meno potente ad esempio).

Più server uguali aumentano i costi linearmente, potenziarne uno esponenzialmente, quindi per andare incontro a costi, difficoltà di gestione e tolleranza ai guasti è molto meglio usare la seconda (non a caso è la più usata).

#### TECNICHE DI SCALABILITÀ'

- Nascondere la latenza di comunicazione, usando comunicazione asincrona laddove possibile
- Spostare la computazione o erogazione del servizio quanto più vicina possibile ai client
- Partizionare dati e computazione in parti più piccole distribuendole fra risorse più piccole (divide et impera)
- Replicare risorse e dati del sistema distribuito rendendole disponibili sulle diverse macchine.

#### PROBLEMI PER LA SCALABILITÀ'

- Dobbiamo essere capaci di gestire le repliche per essere in grado di evitare problemi di inconsistenza; Modificare una delle repliche di un file non deve causare inconsistenza per le altre. La sincronizzazione costa molto, e preclude l'avere soluzioni scalabili a larga scala. Per questo si cerca di tollerare un certo grado di inconsistenza.

## 1.5. Errori nella progettazione di un sistema distribuito

- Si assume che la rete sia affidabile (possono invece avvenire fallimenti)
- Si assume latenza 0 (non è mai 0, dipende dalla velocità della luce)
- Si assume larghezza di banda infinita (non lo è mai)
- Si assume che la rete sia sicura
- Si assume che la topologia della rete non cambi (cosa vera finché si rimane in laboratorio, ma poi cambia)
- Si assume che l'amministratore sia sempre uno solo (ne potranno servire di più per gestire i vari nodi)
- Si assumono nulli i costi di trasporto (l'impostazione e la messa in pratica di una rete ha sempre un costo)
- Si assume l'ambiente della rete omogeneo

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

## 1.6. Tipi di sistemi distribuiti

- **Ad alte prestazioni**

Suddivisi in:

- **Cluster:** gruppo di server interconnessi da una rete ad alte capacità con gli obiettivi principali di: HPC (High Performance Computing) e/o HA (High Availability). L'architettura dei cluster è omogenea ed il loro hardware è pressoché identico. Sono organizzati secondo un'architettura master worker, dove il master controlla i guasti e gestisce i worker che lavorano (utile a tal proposito la libreria MPI che permette la comunicazione fra essi tramite scambio di messaggi). I cluster server possono essere controllati da elementi software specifici che li gestiscono tramite un unico sistema. Nei cluster, i server sono comuni macchine (e non ad alte prestazioni), quindi i fallimenti avvengono di frequente. I cluster possono sfruttare tramite il loro sistema operativo un meccanismo di "migrazione" che permette di egualizzare il carico tra i diversi nodi (applicabile a diversi livelli di astrazione) o a spegnere dei server per ridurre il consumo energetico.
- **Cloud computing:** Il cluster computing compone i data center del cloud computing, pertanto compongono questa tecnologia. Il cloud computing a differenza del cluster computing è disponibile per chiunque (o almeno chi sia disponibile a pagarne i servizi). La scala di utilizzo del cluster computing era una rete LAN locale, mentre la scala di utilizzo del cloud computing è geografica. L'estensione del cloud computing è volta alla decentralizzazione, spostando al fog computing e poi agli edge devices i dati.

- **Sistemi distribuiti informativi**

**Transazione:** unità di lavoro indipendente caratterizzata dalle proprietà ACID:

- **Atomic:** La transazione avviene in modo atomico, quindi vengono eseguite tutte le operazioni in essa contenuta o nessuna.
- **Consistent:** Si previene che il sistema possa raggiungere uno stato inconsistente.
- **Isolated:** Le transazioni devono essere indipendenti fra loro.
- **Durable:** Una volta effettuato il commit della transazione, tutti i cambiamenti dello stato apportati diventano permanenti, salvati sui dispositivi di memorizzazione stabili.

Studieremo le **transazioni distribuite**, ovvero transazioni annidate che possono essere eseguite in modo parallelo su molteplici server. Importante è la presenza del **Transaction Processing Monitor** responsabile di coordinare l'esecuzione della transazione distribuita gestendo la concorrenza fra i nodi.

- **Sistemi distribuiti pervasivi:**

Sistemi distribuiti i cui nodi sono spesso di piccola dimensione e mobili, alimentati da batterie, che sono parte di un sistema più grande.

Abbiamo diverse tipologie fra cui:

- Sistemi di **Mobile Computing** (nodi mobili)
- **Reti di sensori**

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

- Essi ascoltano cosa avviene nell'ambiente raccogliendo dati di monitoraggio. I sensori hanno limitate capacità di memoria, computazione e comunicazione, oltre al fatto che trattandosi di dispositivi più semplici, i fallimenti sono più frequenti.

Si possono presentare 2 situazioni estreme:

- **Poca distribuzione:** ogni nodo nella rete di sensori acquisisce informazioni e le invia verso un unico nodo centralizzato che acquisisce info e effettua il processamento (nodo quindi centralizzato che se fallisce compromette tutto, collo di bottiglia che diventa un one point failure)
- **Decentralizzazione totale:** ciascun sensore è in grado di comunicare con gli altri e computare la sua parte di dati

## 2. INTRODUZIONE AL CLOUD COMPUTING

**Problema:** Vogliamo realizzare una app di video playback, che sia scalabile rispetto al numero di utenti che la sta utilizzando.

### **Soluzione centralizzata:**

Una soluzione classica è quella di realizzare un'applicazione **multithreaded** per avere più fili di esecuzione che servono parallelamente specifici utenti ognuno.

I problemi relativi ad un'applicazione multithreaded sono:

1. Scalabilità limitata dal numero di thread che posso lanciare in modo concorrente basate sul sistema operativo e dalla capacità computazionale del nodo considerato.
2. Inoltre, poiché più thread devono condividere tra loro uno stato, servono anche meccanismi di sincronizzazione fra thread concorrenti.
3. In caso di guasto e fallimento di un singolo thread potrebbe avere un impatto su molteplici utenti mandando in crash l'intera applicazione.

L'approccio classico risolutivo quindi ha dei limiti. Ovviamo a questi limiti con una soluzione nativa per il cloud.

### **Soluzione distribuita:**

Pensiamo un'applicazione **singlethreaded** che serve un utente alla volta, istanziata a richiesta. Questo sistema così realizzato è quindi più robusto. Quindi:

1. È Scalabile rispetto al numero di container che hanno l'applicazione single threaded.
2. Non ho problemi di gestione sincronizzazione (poiché ho un sistema singlethreaded).
3. Ho migliore tolleranza ai guasti, perché se fallisce un video player, solo l'utente in questione ne risente, mentre tutti gli altri utenti possono continuare ad utilizzare la loro applicazione. Infatti, l'esecuzione su una macchina virtuale o su un container isola fra di loro le diverse applicazioni concorrenti.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

Il cloud computing si occupa di realizzare sistemi che siano in grado di servire milioni di richieste al giorno, facendo fronte a un carico di lavoro/traffico di richieste in ingresso variabile (all'ordine di exabytes di dati da memorizzare).

## 2.1. Definizioni cloud computing

È il coordinamento di tutte le attività inerenti alla computazione.

Sosteremo nel cloud piattaforme, come ad esempio lo sviluppo di applicazioni web, servizi di machine learning, servizi per computazione per calcolo su grandi quantità di dati (big data analytics) ...

Alcune definizioni

1. “Il cloud computing si riferisce sia alle applicazioni fornite come servizi su Internet sia ai sistemi hardware e software nei data center che forniscono tali servizi applicativi”. Tali servizi sono stati denominati come **Software as a Service (SaaS)**, mentre l’hardware e il software dei data center vengono denominati con **Cloud**.
  - Il Cloud computing ha le seguenti caratteristiche:
    - Illusione di avere a disposizione quantità infinita di risorse di elaborazione
      - Illusione ottenuta tramite uso efficiente della virtualizzazione, che consente di condividere stesse risorse hardware tra diverse componenti andando a isolare tra loro macchine virtuali e container.
    - L’eliminazione di un impegno iniziale da parte degli utenti cloud
      - Utente dovrebbe acquisire risorse hardware (sceglierle, configurarle, installarvi software applicativi o e di base, e effettuare manutenzione durante esecuzione di applicazione)
      - Questi costi vengono eliminati, ma diventano a carico del fornitore dei servizi cloud
    - Possibilità di pagare per risorse utilizzate (pay per use)
      - Pagare in base a utilizzo ha granularità fine ma non finissima, nel momento in cui alloco macchina virtuale, essa sarà a me riservata per 1 ora di tempo, io pagherò per 1 ora di tempo di utilizzo, a anche se la vorrei usare solo per 10 secondi
      - Con serverless computing l’utente paga per effettivo tempo di utilizzo delle funzioni che va ad eseguire
2. “Il cloud computing è un modello per consentire un accesso di rete ubiquo, conveniente e on demand a un pool condiviso di risorse di elaborazione configurabili (ad esempio, reti, server, archiviazione, applicazioni e servizi) che possono essere rapidamente allocate/fornite e rilasciate con uno sforzo di gestione minimo o con l’interazione con il fornitore di servizi (ossia sono risorse elastiche)”.
  - **Ubiquo** → da qualunque dispositivo stia usando l’utente
  - **Conveniente**
  - **On demand** → quando a un utente servono servizi, li richiede, non deve allocarli all’inizio

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

- **Elasticità** → ottengo risorse nel momento in cui ne ho bisogno. Se aumenta il carico di lavoro, aumento la quantità di risorse allocate, se diminuisce de-alloco le risorse allocate (tutto questo può anche avvenire in moto automatico).

3. “I cloud sono un ampio insieme di risorse virtualizzate facilmente utilizzabili e accessibili (come hardware, piattaforme di sviluppo e / o servizi)”. Queste risorse possono essere riconfigurate dinamicamente per adattarsi a un carico variabile (scalabilità e elasticità), consentendo anche un utilizzo ottimale delle risorse. Questo insieme di risorse è tipicamente sfruttato da un modello pay per use, dove l’utente paga in base all’utilizzo delle risorse.

## 2.2. Paradigmi di computazione

- **Cluster computing**
  - Alto accoppiamento poiché la computazione avviene localmente in LAN
  - Omogeneità dell’hardware
- **Distributed computing**
  - Idea di eseguire la computazione su un insieme di server in rete, non più localmente distribuiti e legati da rete LAN, ma con maggiore disaccoppiamento.
  - Applicazione gestita da singolo amministratore
- **Grid computing**
  - Più data center distribuiti geograficamente che ampliano la scala di distribuzione
  - Gestione distribuita (più amministratori)
- **Cloud computing**
  - “Cloud” → la nuvola che usiamo come simbolo per rappresentare internet.
  - “Computing” → include il concetto di computazione, storage e coordinamento delle attività

Il cloud computing consiste nel passaggio del suddetto concetto di “computing” da un singolo server o data center verso internet.

## 2.3. Caratteristiche principali dei cloud computing:

- **On demand self-service**
  - Le risorse cloud possono essere fornite su richiesta dagli utenti (l’utente ottiene i servizi di cui ha bisogno nel momento in cui ne ha bisogno), senza richiedere interazioni con il provider di servizi cloud (limitata/nulla interazione diretta con il cloud provider).
- **Ampio accesso alla rete**
  - Risorse cloud accessibili tramite Internet utilizzando meccanismi di accesso standard che forniscono un accesso indipendente dalla piattaforma

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

- Servizi cloud accessibili tramite API ben definite (interfaccia per accesso ai servizi pubblicata e resa disponibile)

- **Rapida elasticità**

- Elasticità = capacità per i clienti (= utenti dei servizi cloud) di richiedere, ricevere e successivamente rilasciare rapidamente tutte le risorse necessarie, considerando solamente la quantità di risorse di cui hanno effettivamente bisogno.
- Risorse cloud possono essere ottenute da utenti in modo rapido ed elastico.
- Possiamo effettuare uno scale out/scale in delle risorse allocate in base alla domanda, all'effettivo utilizzo delle risorse che dovremo fare (Le risorse cloud possono essere rapidamente ridimensionate / introdotte in base alla richiesta).
  - Scalabilità orizzontale può andare
    - scale out → aumento numero server che sto usando
    - scale in → riduco numero di server, ma server che alloco hanno caratteristiche simili tra loro, sono omogenei.
  - Scalabilità verticale → cambio caratteristiche hardware virtualizzate della risorsa che sto utilizzando

- **Pool di risorse**

- Risorse che vengono usate da utenti sono allocate su macchine condivise tra molteplici utenti. Le risorse cloud vengono quindi raggruppate per servire più utenti utilizzando la multi-tenancy.
- Multi-tenancy: più utenti serviti dallo stesso hardware fisico

- **Virtualizzazione delle risorse**

- Le risorse che vengono virtualizzate sono ambiente di esecuzione, archiviazione, elaborazione, memoria, larghezza di banda di rete e persino data center.

- **Pay per use come modello di pricing**

- Permette di trasformare costi in capitale in costi operativi.
- Nessun costo di acquisizione iniziale elevato.

- **Servizio misurato**

- L'utilizzo delle risorse cloud viene misurato e l'utente andrà a pagare in base a una metrica specifica.
- Servizi cloud caratterizzati da service level agreement

## 2.4. Modelli di deployment del cloud computing

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

- **Cloud pubblica**

- Infrastruttura cloud fornita dal provider dei servizi cloud (Amazon web service, Google...) affinché possa essere utilizzata dal pubblico (multi-tenancy). L'utilizzo dell'infrastruttura e dei servizi cloud è quindi aperto a tutti, al pubblico generale. Tutti questi utenti dei servizi cloud andranno a condividere tra loro le stesse risorse hardware non accorgendosi di ciò (o comunque in minima parte)
- Posseduta, gestita e operata da un'organizzazione di business, compagnie e aziende come Amazon, Google, Aruba.. Anche l'università potrebbe mettere a disposizione un cloud per studenti e personale
- Le risorse cloud esistono on premise (su pretesa) del cloud provider, ossia vengono gestite, acquistate, manutenute dal fornitore dei servizi cloud.
- Servizi cloud messi a disposizione possono essere gratuiti o a pagamento

- **Cloud privata**

- Infrastruttura cloud fornita per l'uso esclusivo da una organizzazione che ha molteplici utenti.
  - Per esempio, una azienda mette a disposizione cloud privata, le risorse a disposizione sono disponibili solo per il personale dell'azienda.
- Infrastruttura posseduta, gestita e operata da organizzazione, da una terza parte, o da una combinazione delle due.
  - Per esempio, una azienda potrebbe decidere di non condividere il proprio data center con i propri dipendenti con un modello di cloud privato, ma può pagare cloud provider affinché metta a disposizione risorse in modo esclusivo per azine Cloud pubblica organizzazione paga un provider cloud che metta a disposizione risorse in modo esclusivo per azienda
- Infrastruttura esiste on premise dell'azienda o no
- **Vantaggi** rispetto alla pubblica:
  - Maggiore sicurezza → non essendoci condivisione delle risorse (multitenancy) viene garantita sicurezza ai dati e alle applicazioni.
  - Si può offrire a utilizzatori u maggior grado di personalizzazione dei servizi e dei service level agreement.
- **Svantaggi:**
  - Minor vantaggio economico, soprattutto se infrastrutture hardware e software gestite da azienda (costi per risorse hardware e software).
  - Scalabilità non è infinita → non abbiamo a disposizione quantità infinita di risorse.

- **Cloud ibrido**

- Modello cloud più diffuso
- Infrastruttura cloud non unica, ma deriva da composizione di più infrastrutture cloud distinte (pubbliche o private) che rimangono entità uniche, ma sono legate tra loro da una tecnologia standardizzata o proprietaria che consente la portabilità dei dati e delle applicazioni.
- Uso combinato di cloud privati e pubblici.
- **Vantaggi:**
  - Bilanciare risorse e costi
  - Differenziare la privacy (mantenere i dati sensibili nel cloud privato)
    - Maggiore sicurezza e privacy, poiché i dati sono gestiti dai cloud server, memorizzati su infrastruttura di storage del cloud provider. Potrebbero

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

esserci dati sensibili soggetti alla privacy a seconda del paese, dei proprietari dei dati

- Migliora la disponibilità: Avere un cloud public permette il disaster recovery in caso di interruzione imprevista (crash non atteso di grandi dimensioni dell'infrastruttura del cloud privata; i servizi erogati dal cloud privato vengono migrati verso il pubblico).
- Migliora le prestazioni: Cloud Bursting
  - Utilizza un cloud ibrido dinamico (cloud privato + pubblico) per gestire il carico di lavoro variabile quando la capacità del cloud privato è insufficiente.
- All'inizio uso cloud privata, per fornire l'applicazione con un carico di lavoro medio.
- Se aumenta carico inviato a applicazioni eseguite su cloud privata à ottengo maggiore scalabilità integrando cloud privata con pubblica in modo dinamico, in modo oche eccesso di carico sia gestito da cloud pubblica, quindi utilizzo un cloud pubblico per gestire i picchi di carico che il cloud privato non è in grado di sostenere.

- **Evoluzione: Multicloud**

- Utilizzo simultaneo di più ambienti Cloud (ibrido, 2 o più solo privati, 2 o più solo pubblici).
- Vantaggi di usare più cloud provider
  - Migliora efficienza dei costi
  - Aumenta flessibilità
  - Soddisfa i vincoli di archiviazione dei dati (ossia i dati archiviati fisicamente in un determinato paese)
  - Migliora la distribuzione geografica
- Il fornitore dell'applicazione cloud con la sua applicazione deve soddisfare dei requisiti in termini di privacy per i dati sensibili, ovvero garantire che dati siano fisicamente memorizzati all'interno dei server in n determinato paese.
- Riduce il rischio di vendor lock -in (rischi del vincolo del fornitore):
  - Ogni fornitore cloud ci mette a disposizione una interfaccia non standardizzata. Ciò vincola gli sviluppatori delle applicazioni cloud alla specifica API esposta dal cloud provider, in quanto diventa molto complicato gestire la portabilità dell'applicazione verso un altro fornitore una volta realizzata l'infrastruttura.

## 2.5. Tipologie di servizi cloud offerte da cloud provider

Caratterizzati da stratificazione in 3 livelli:

- **Infrastruttura (IaaS - Infrastructure as a Service)**
  - I servizi offerti sono quelli di calcolo, archiviazione e risorse di rete, ossia i servizi che ci permettono di usare direttamente risorse di computing, quindi lanciare macchine virtuali, container o utilizzare servizi di storage per la memorizzazione dei file.
  - **Capacità del cliente**
    - Sfruttare l'elaborazione, archiviazione, reti e altre risorse informatiche fondamentali per distribuire ed eseguire software arbitrario (inclusi sistemi operativi e applicazioni)
  - **Controllo/gestione del cliente**

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

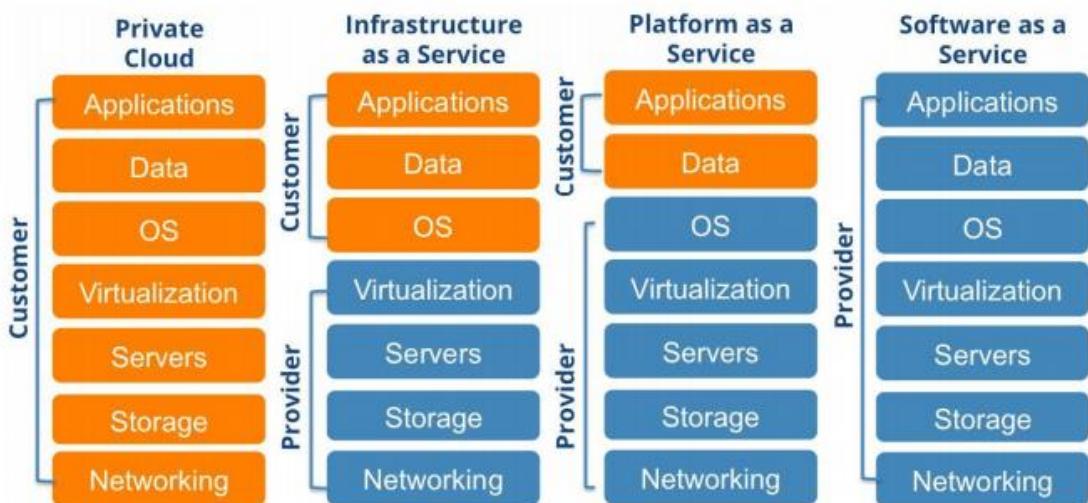
- Utente non controlla direttamente infrastruttura sottostante, ma può decidere dove istanziare macchina virtuale, le caratteristiche hardware, il tipo di sistema operativo, il software di base. Può installare su essa le librerie, i programmi, i linguaggi di programmazione, le applicazioni di cui ha bisogno per propria applicazione cloud. Utente ha controllo forte su applicazioni che andrà a sviluppare e su cui andrà a effettuare il deployment.
  - Esso potrà inoltre controllare alcune componenti di rete (per esempio può configurare le porte accessibili da esterno per le proprie macchine virtuali).
  - Utente controlla e gestisce direttamente la sua applicazione, ha maggior controllo su di essa. Il controllo si riduce man mano che saliamo di livello.
- Esempi:
    - Amazon Web Service
    - Alibaba cloud
    - Google cloud Platform
- 
- **Piattaforma (PaaS - Platform as a Service)**
    - Vengono offerte agli utenti delle piattaforme, sulle quali i clienti possono sviluppare, eseguire e gestire applicazioni scalabili, senza la complessità di costruire e mantenere l'infrastruttura sottostante.
    - **Capacità del cliente**

I clienti possono sviluppare, distribuire e testare sul cloud le applicazioni create/acquisite dai consumatori, realizzate usando linguaggi di programmazione, framework applicativi, strumenti di sviluppo supportati dal provider PaaS.
    - **Controllo/gestione del cliente**
      - Nessun controllo sull'infrastruttura cloud sottostante (rete, server, sistemi operativi, archiviazione) → svantaggio
      - Controllo sulle applicazioni distribuite e possibilmente sulle configurazioni dell'ambiente di hosting dell'applicazione
- 
- **Software (SaaS - Software as a Service)**
    - Software e applicazioni messe a disposizione dei clienti su Internet → il mercato diventa ancora più ampio.
    - **Capacità del cliente**
      - Utilizza applicazioni del provider SaaS in esecuzione su un'infrastruttura cloud.
      - Applicazioni accessibili da vari dispositivi client tramite Web o API del provider
    - **Controllo/gestione del cliente**
      - Nessun controllo sull'infrastruttura cloud sottostante
      - Funzionalità di rete, server, sistemi operativi, archiviazione o anche singole applicazioni
      - Possibile eccezione: impostazioni di configurazione dell'applicazione specifiche dell'utente limitate
    - **Gmail, Google drive sono esempi di SaaS**

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

### 2.5.1. Confronto IaaS / PaaS

- **IaaS:** Abbiamo risorse visualizzate pure, quindi CPU, storage e capacità di rete. Il sistema operativo è tipicamente già incluso; Solitamente vengono messe a disposizione delle taglie già configurate di risorse acquistabili.
- **PaaS:** Risorse virtualizzate sul framework già in esecuzione. L'autoscaling (politica che gestisce l'aggiunta o la rimozione di macchine virtuali a disposizione) ed il load balance (politica in base alla quale viene gestito il carico di lavoro) non vengono configurati dall'utente a differenza di prima. Nel PaaS, esiste l'elastic bean stalk che si occupa autonomamente delle suddette problematiche. Questa tipologia inoltre impone dei vincoli sull'architettura dell'applicazione e quella dei dati, costringendo il developer ad utilizzare il linguaggio di programmazione scelto dal vendor, di fatto aumentando il rischio di vendor lock in.



## 2.6. Modelli di pricing

**Pay per use:** Gli utenti pagano in base all'utilizzo che fanno delle risorse cloud (usualmente avviene su fascia oraria).

**Fixed:** Gli utenti pagano una quantità prestabilita al mese per l'utilizzo delle risorse cloud.

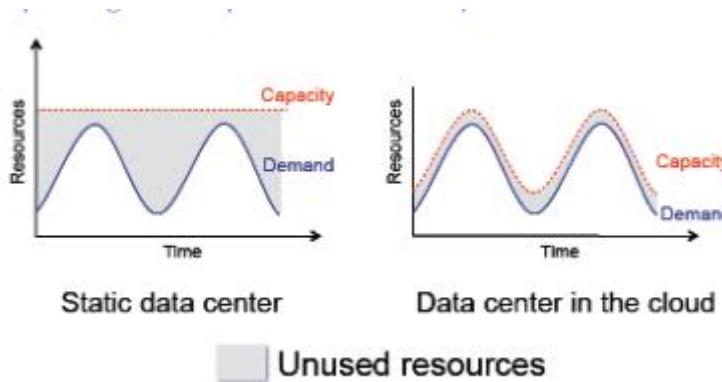
**Spot:** Il prezzo delle risorse cloud è variabile; è determinato in base alla domanda e all'offerta. La modalità è la seguente: l'utente fissa un prezzo d'asta; Le risorse offerte dal provider restano a disposizione del developer finchè la domanda resta inferiore a quello che ha scommesso (e quindi pagato). Il provider può infatti toglierle la risorsa all'utente in modo improvviso qualora la domanda superasse la soglia d'asta.

## 2.7. Elasticità

Il cloud computing è l'infrastruttura di back-end utilizzata dalle più grandi aziende per fornire servizi Internet popolari e di successo che sono scalabili, elastici, economici, affidabili e sicuri.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

**Elasticità:** Variare il numero di risorse e le loro capacità dinamicamente a seconda del carico di lavoro. Questo concetto è molto importante in quanto il traffico internet risulta molto variabile. I picchi improvvisi di traffico prendono il nome di "burst"; A fronte di questo è importante il **capacity planning**, ovvero il come

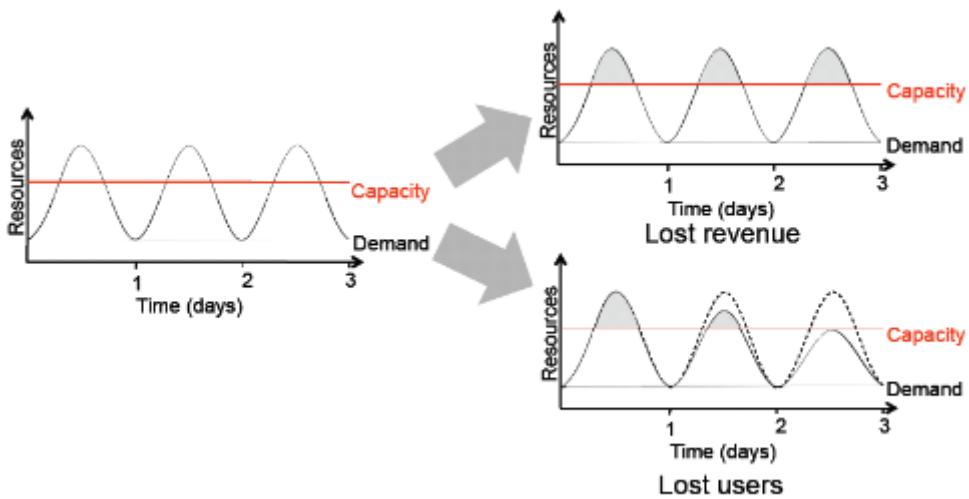


dimensionare correttamente le risorse messe a disposizione dall'applicazione.

### 2.7.1. Approcci di Capacity Planning:

- **Algoritmi di predizione:** Si cerca di predire in qualche modo la domanda in maniera da gestire in anticipo queste variazioni di carico. Questo si definisce un approccio "pre-attivo".
- **Over provisioning:** Approccio tradizionale che garantisce una capacità sempre sopra i picchi di richiesta di risorse massima. Questo corrisponde tuttavia ad un eccesso di capacità, poiché quando la domanda scende crea uno spreco economico ed energetico non indifferente. L'obiettivo da ottenere diventa quindi quello di far variare la quantità di risorse allocate come la curva della domanda, affinchè la quantità di risorse sprecate sia la minima possibile. L'utilizzo di un approccio tradizionale richiede tempo per istanziare e deallocare le risorse, cosa notevolmente più celere nel cloud computing (il che permette molto più facilmente di inseguire la curva della domanda).
- **Under provisioning:** Approccio che mette a disposizione una capacità attorno al valore medio della richiesta, il che porta a condizioni di sovraccarico dove la domanda eccede la capacità messa a disposizione. Ciò riduce le prestazioni dell'applicazione, il che scoraggerà gli utenti portandoli verso altre applicazioni, riducendo la domanda e creando un danno economico.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.



L'elasticità quindi descrive come il sistema si adatta alle variazioni di carico aumentando o diminuendo le risorse a disposizione in maniera automatica, in modo tale di avvicinare il più possibile la capacità messa a disposizione alla domanda.

L'elasticità è basata su due parametri:

- **Accuratezza:** Somma delle aree di over provisioning e di under provisioning per la durata del periodo di osservazione.
- **Timing:** tempo speso nell'under provisioning o nell'over provisioning rispetto al tempo totale di osservazione.

### Load balancer

Il load balancer serve per gestire le distribuzioni del carico fra le varie repliche delle macchine a disposizione. Gli obiettivi delle tecniche di load balancing sono quelle di massimizzare la quantità di risorse utilizzate, minimizzare il tempo di risposta e massimizzare il throughput.

## 2.8. Service Level Agreement

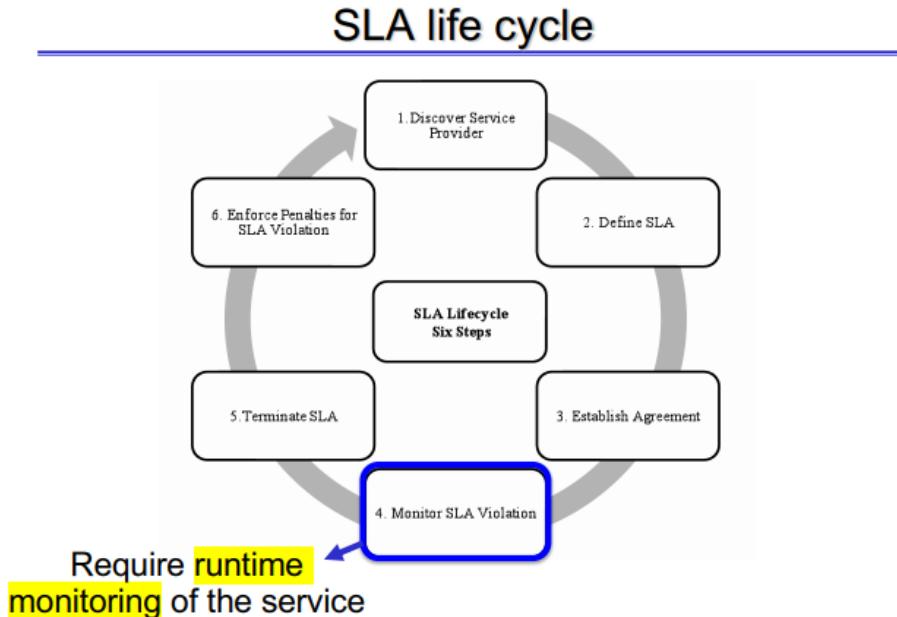
Si tratta di un accordo formale sul livello di servizio, stipulato fra provider e consumer che specifica gli "obiettivi del livello di servizio". Questi obiettivi si definiscono SLO (service level objective), i quali impongono una condizione sulla misura di qualità di uno specifico servizio. Un service level agreement può specificare i tempi medi di un servizio, o i tempi massimi per esempio.

### Problemi inerenti alle SLA

- Non ci sono garanzie in termine di performance: La rete non è controllata dal cloud provider, quindi esso non può dare garanzie sulla velocità di download o sui tempi di risposta.
- Le macchine virtuali garantiscono solo disponibilità ma non la raggiungibilità, che dipende dalla rete. Qualora un sistema cloud non rispetti le disponibilità concordate, esistono dei crediti solo per futuri pagamenti nello stesso servizio che vengono assegnati.
- Responsabilità di monitoraggio a carico dell'utente
- Difficoltà nel confrontare i SLA tra diversi provider poiché manca una standardizzazione

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

**Cloud monitoring:** Obiettivo di monitorare l'utilizzo delle risorse cloud per tenere traccia dello stato di salute delle applicazioni e dei servizi di cui abbiamo effettuato il deployment nel cloud



## 2.9. Tipi di servizi

- **Stateless:** Servizio che non mantenendo uno stato su le successive invocazioni, risulta molto semplice da replicare come tipo.
- **Stateful:** Servizio con stato, per esempio contatore. Lo stato puo essere memorizzato sul servizio stesso o all'interno di un servizio di data store.

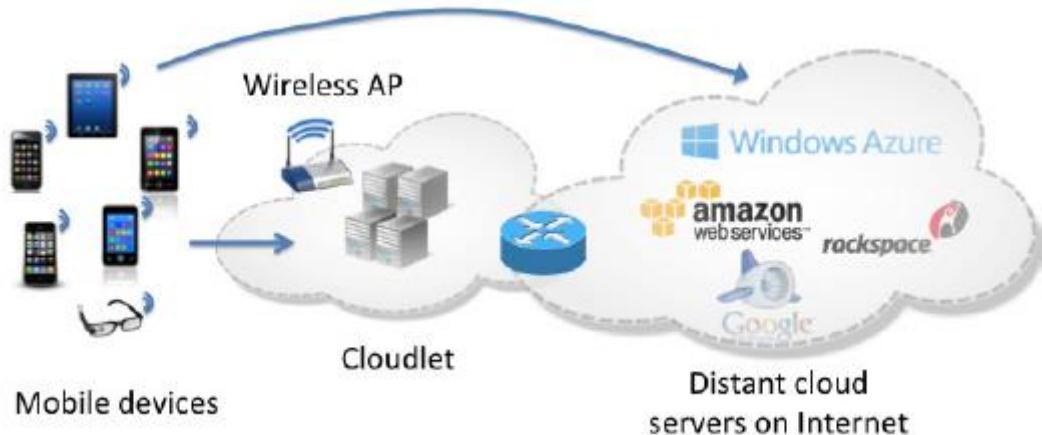
## 2.10. Processo di sviluppo di applicazioni cloud

- **Deployment design:** si decide quali sono la tipologia e la capacità delle risorse in ogni cloud, le interconnessioni, il bilanciamento del carico e le strategie di replicazione.
- **Valutazione delle prestazioni:** si verifica che l'applicazione soddisfi le prestazioni attese. Include anche il monitoraggio del carico dell'applicazione e le performance dei parametri.
- **Deployment refinement:** il deployment può essere raffinato durante la creazione dell'applicazione, considerando varie alternative sulla scalabilità, componenti di interconnessione, il bilancio del carico e le strategie di replica. In poche parole, in questa fase si possono modificare le decisioni prese nella fase iniziale.

## 2.11. Fog computing

Si tratta del futuro scenario del computing, questo perchè le soluzioni solo cloud computing possono essere non praticabili per via della latenza. Si punta a spostare la computazione dal cloud a delle risorse di storage e computing localizzate ai bordi della rete, più vicino all'utente. Parte della computazione si potrà trovare nella “**cloudlet**” (mini-cloud che sarà già presente nell'access point). Questi mini data center

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.



potranno anche essere mobili (i cloudlet mobili sono argomento di studio odierno, per esempio le macchine a guida autonoma).

**Fog computing:** Piattaforma altamente virtualizzata che provvede a computare, offrire storage e servizi di rete fra device degli utenti e i tradizionali data center.

**(Def 2)** Architettura orizzontale a livello di sistema che distribuisce computazione, storage, controllo e funzioni di rete più vicini agli utenti, per offrire un cloud-to-thing continuum.

### 2.11.1. Edge computing vs Fog computing:

Entrambi hanno radici comuni (nella rete) ed architettura decentralizzata. L'unica differenza sta nel fatto che il fog risulta più integrato nel cloud rispetto all'edge. Considerarli sinonimi non è del tutto erroneo.

## 2.12. Riassunto dei principali concetti

Principali **benefici** del cloud:

- **INFORMATION TECHNOLOGY benefits**
  - Scalabilità, elasticità, flessibilità
  - Accessibilità temporale e spaziale, in qualunque momento e ovunque, visto che i servizi sono accessibili tramite internet.
  - Gestione semplice in quanto gli aggiornamenti vengono rilasciati e l'utente deve solo aggiornare l'applicazione.
  - Agilità di organizzazione e operazione
- **BUSINESS benefits**
  - I costi capitali vengono trasformati in costi operativi.
  - Incremento della produttività
- **ENVIRONMENT benefits**
  - meno impatto ambientale

Principali **problemi** del cloud:

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

- Latenza di comunicazione
- Portabilità delle applicazioni, ovvero la migrazione tra cloud provider di una app cloud native con i suoi dati risulta complicato
- Interoperabilità; Il multicloud cerca di far cooperare contemporaneamente più cloud provider. Servono pertanto degli standard per l'operabilità ed interoperabilità del cloud.
- Poco supporto per la definizione e la gestione dei SLA. Manca la possibilità di negoziazione dei SLA, poiché sono già pronti e possono essere solo accettati da un developer non potendoli definire.

Principali **problemi** dei cloud providers:

- Incertezza nella domanda di servizio
- Gestione delle SLA
- Requisiti dei clienti Cloud
- Gestione Energetica
- Cloud interoperability

## 3.ARCHITETTURE DEI SISTEMI DISTRIBUITI

L'architettura di un sistema distribuito si divide in

**Architettura software:** definisce l'organizzazione logica del sistema distribuito e l'interazione fra i diversi componenti e l'ambiente (tra cui l'utente).

**Architettura di sistema:** definisce la realizzazione fisica dell'architettura software. I componenti del sistema distribuito vengono realizzati.

**Pattern:** soluzione che viene comunemente adottata per risolvere una classe di problemi.

Un pattern architettonico riguarda la progettazione dell'architettura. Lo stile architettonico specifica come far interagire fra di loro i componenti e come connetterli.

**Componenti:** unità modulari con interfacce ben definite. Sono rimpiazzabili nel loro ambiente.

**Connettori:** elementi che collegano fra loro due o più componenti del sistema distribuito. Un connettore media l'attività di comunicazione e connessione fra i componenti.

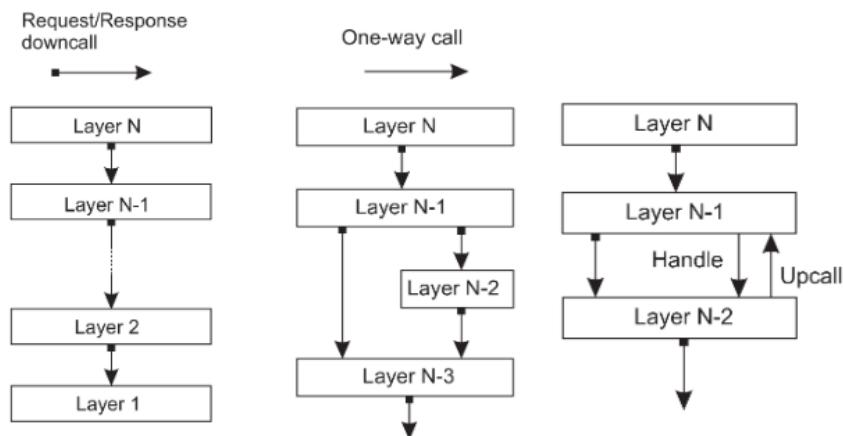
### **3.1. Stili architetturali dei sistemi distribuiti**

- A LIVELLI
- ORIENTATO AGLI OGGETTI
- BASATO SU INTERFACCE REST
- BASATO SU EVENTI
- ORIENTATO AI DATI
- PUBLISH/SUBSCRIBE

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

### 3.1.1. Stile architetturale a livelli

I componenti sono organizzati a livelli, dove il componente livello i-esimo invoca il componente del livello i-1 esimo. La comunicazione fra i diversi componenti è basata su scambi di messaggi. Se il layer N è quello dell'utente, le richieste scendono verso il basso e le risposte salgono verso l'alto. Ci sono diverse configurazioni di architetture a livelli, per esempio si possono introdurre dei livelli intermedi di cache così da non dover scendere fino al livello più basso, poiché risulta possibile trovare i dati in una cache intermedia. Posso anche permettere delle chiamate asincrone fra livelli.



Il modo tradizionale di vedere l'architettura a livelli è

- **Application-interface layer:** comprende la parte dell'architettura responsabile di interfacciare il sistema con l'utente e con le applicazioni esterne.
- **Processing layer:** contiene le funzioni di un'applicazione senza dati specifici.
- **Data layer:** contiene i dati che il client ha bisogno di manipolare tramite i componenti dell'applicazione.

### 3.1.2. Object Based Style

Basata su mapping fra componente ed oggetto. Quest'ultimo incapsula la struttura dati con una API ben definita per accedere e modificare i dati. Ciò permette di avere encapsulamento, information hiding e wrapping dei componenti legacy. La comunicazione fra gli oggetti avviene tramite chiamata a procedura remota o tramite invocazione di metodo remoto.

### 3.1.3. RESTful style

Il sistema distribuito è visto come una collezione di risorse gestite dai componenti. REST (representational state transfer) è un sistema basato sull'idea che le risorse che fanno parte del sistema distribuito possono essere recuperate tramite metodo HTTP. Le risorse sono identificate infatti tramite URI (uniform resource identifier). Questo servizio è stateless poiché per rendere il sistema più leggero lo stato deve essere trasferito dal client sui server. REST permette delle operazioni fra cui GET, PUT, POST e DELETE.

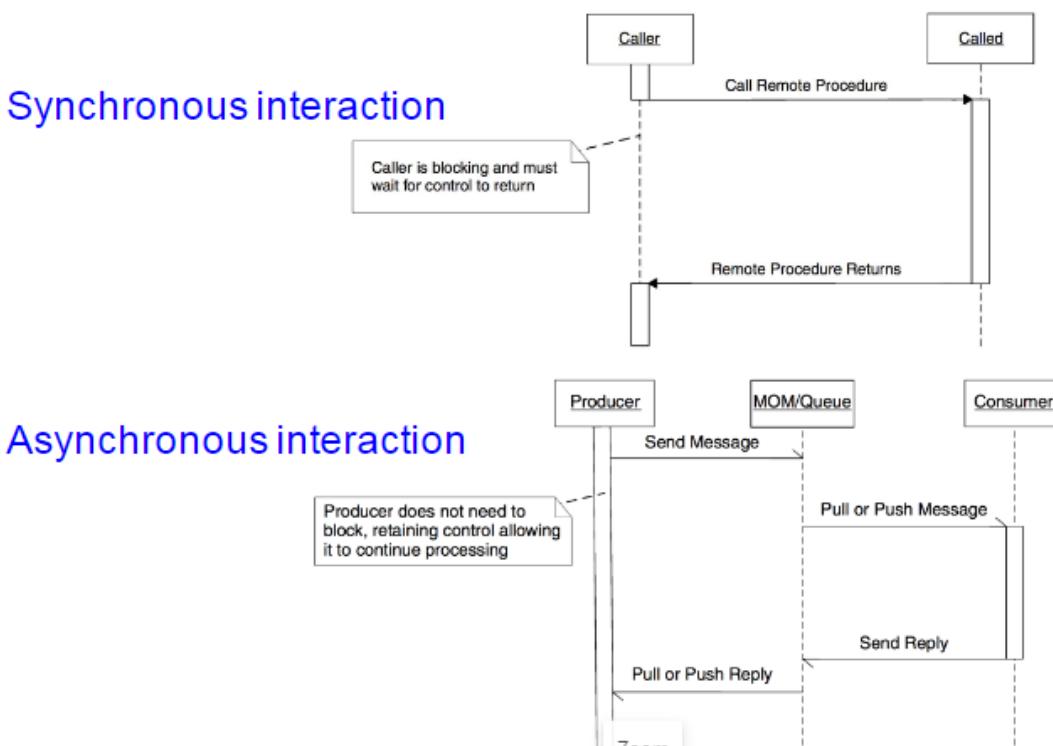
Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

## 3.2. Disaccoppiamento

In ogni stile di quelli visti sopra la comunicazione fra i componenti coinvolti è diretta. Per avere un maggior grado di flessibilità di comunicazione, bisogna introdurre un disaccoppiamento, ovvero far comunicare in modo indiretto il componente n con il componente n-1 tramite un intermediario. Questo concetto di aggiungere un grado di interazione è fondamentale nello sviluppo dei sistemi informatici. Il disaccoppiamento permette di definire stili architetturali che possono sfruttare al meglio distribuzione, scalabilità ed elasticità (architettura micro-service serverless).

Esistono 3 tipi di disaccoppiamento:

- **Disaccoppiamento spaziale:** i diversi componenti dell'applicazione distribuita non devono necessariamente conoscersi fra loro e possono essere anonimi.
- **Disaccoppiamento temporale:** i componenti del sistema distribuito che interagiscono fra loro non devono essere contemporaneamente presenti e attivi nel momento in cui la comunicazione ha luogo. La comunicazione aggiunge un livello di indirezione tramite una coda di messaggi; Il server consulterà la coda quando ritornerà attivo, e manderà la risposta al client mettendo in un'altra coda la risposta. Quindi client e server potranno comunicare senza che l'altro sia attivo in momenti diversi (asincronamente).
- **Disaccoppiamento di sincronia:** i componenti che comunicano fra loro non devono aspettarsi a vicenda; Il client non rimane bloccato in attesa che il server gli mandi la risposta.



L'interazione quindi può essere quindi sincrona o asincrona:

**Sincrona** se il chiamante resta bloccato mentre aspetta una risposta prima di riprendere il flusso di esecuzione.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

**Asincrona** se il produttore continua a fare operazioni dopo aver richiesto qualcosa al consumatore. Tutto funziona tramite deposito di richieste in code che vengono lette e processate. Non resta bloccato.

Grazie al disaccoppiamento un sistema distribuito può essere più tollerante ai guasti garantendo sistemi più elastici. Ha però un lato negativo, ovvero aggiunge un overhead (costo) aggiuntivo; Questo overhead può essere espresso in termini di risorse, latenza o altro.

**Pro:** I componenti possono esser replicati o migrati su altri nodi del sistema, poichè grazie al **disaccoppiamento spaziale** abbiamo una flessibilità che ci permette di rimpiazzarli (i componenti non si conoscono tra loro). Il **disaccoppiamento temporale** ci permette di avere un tasso di volatilità, ovvero i mittenti non sono sempre presenti nel sistema, ma vanno e vengono; Potranno pertanto venire e andare via a piacimento, così come il destinatario del messaggio. Il disaccoppiamento temporale ci permette quindi di gestire in modo flessibile l'aspetto temporale (mittente e destinatario non devono essere per forza connessi nello stesso momento per comunicare). Il **disaccoppiamento di sincronia** offre il vantaggio di non bloccare i componenti in attesa che gli altri terminino il loro task.

**Contro:** overhead in termine di prestazioni e di gestione (più accortezze dal punto di vista implementativo e di design del sistema).

L'innovazione del disaccoppiamento fa sì che i componenti del sistema distribuito comunichino non in modo diretto ma indiretto. Questo mediare la comunicazione ci fa ottenere i vantaggi visti sopra.

Ci sono 3 architetture che realizzano il disaccoppiamento:

### 3.2.1. Event driven architecture

È basata sulla comunicazione dei componenti tramite la propagazione di eventi. Un evento è un cambio significativo dello stato, che viene generato dall'interazione dell'utente (o dell'ambiente) con il sistema; Quando avvengono questi eventi, si vuole che il sistema reagisca in alcuni modi prestabiliti.

I componenti ricevono notifiche degli eventi tramite un event bus. A tal riguardo ci sono dei componenti dedicati al pubblicare gli eventi sul bus; Tutti gli altri ricevono una notifica quando viene pubblicato un evento al quale sono interessati. In breve, i componenti sottoscrivono gli eventi di cui sono interessati a ricevere la notifica. A mandare la notifica ci pensa l'event bus, e a dire quale evento è successo all'event bus sono sempre altri componenti il quale scopo è proprio comunicare cosa è successo. La comunicazione avviene quindi in modo mediato tramite il bus degli eventi ed è basata su scambio di messaggi, asincrona, anonima e multicast (uno pubblica e molti ricevono). In questo tipo di stile architetturale manca il disaccoppiamento temporale perchè non c'è uno storage che memorizzi in maniera persistente gli eventi pubblicati. Il componente sottoscritto riceve la notifica dell'evento solo se è in ascolto in quel momento, altrimenti lo perde.

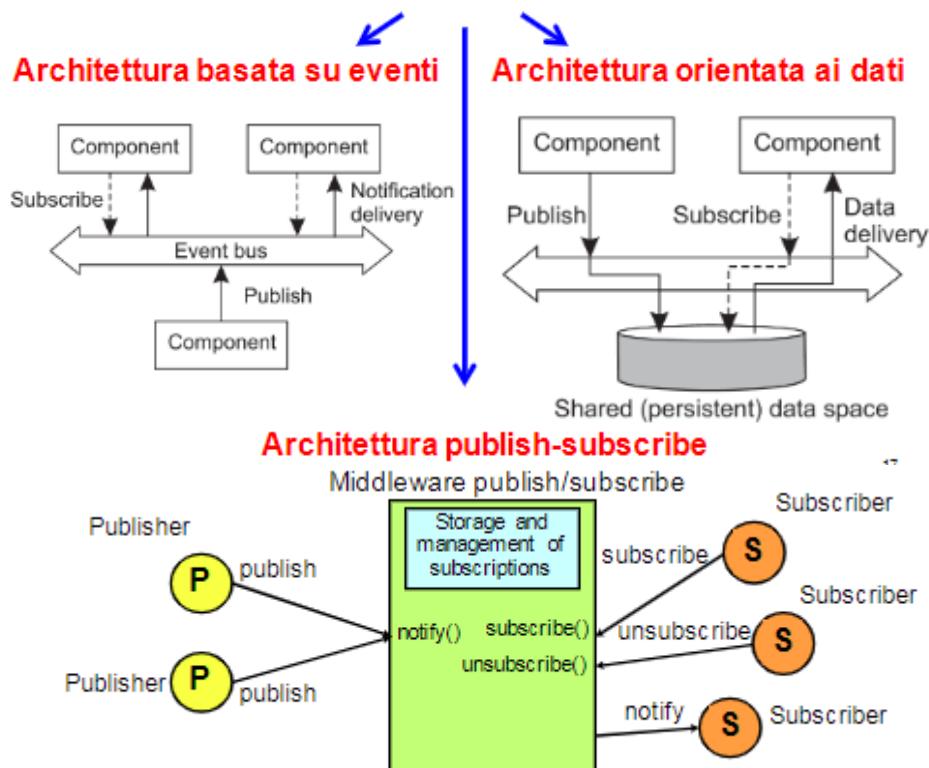
### 3.2.2. Data oriented style

La comunicazione tramite i componenti avviene tramite uno spazio di memorizzazione persistente che può essere condiviso. Questo spazio di archiviazione è tipicamente passivo e non invia notifiche. Nello stile

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

architetturale dedicato ai dati quindi devono essere i componenti a contattare lo spazio di archiviazione per ottenere le informazioni. I componenti interagiscono fra di loro in questo stile architettonico tramite un mediatore, una lavagna (blackboard model), dove chi arriva scrive i dati sull'archiviazione che viene vista come lavagna appunto, così che quando qualche altro componente interessato passa, lo vede e preleva il dato. L'API fornita a questa architettura comprende operazioni quali write, take, read, in caso si voglia mantenere o cancellare il dato letto. Se lo spazio dei dati è anche attivo, in quel caso l'API contiene anche il metodo di notify o push. Questo stile architettonico ci garantisce sempre disaccoppiamento temporale; Un componente si connette, pubblica un evento e si disconnette; un altro si connette prende il dato e si disconnette. Abbiamo sempre disaccoppiamento spaziale in quanto i componenti non si conoscono fra loro. Possiamo avere inoltre disaccoppiamento di sincronia solo se lo spazio dei dati è attivo (se un componente fa pull, è un'azione di polling che lo mette in attesa che lo storage gli fornisca il dato, e quindi non c'è disaccoppiamento di sincronia). Se il data space è attivo e manda notifiche allora nessuno deve fare azione di polling in quanto riceve notifiche, quindi in tal caso avremmo anche disaccoppiamento di sincronia. Possiamo utilizzare delle memorie di tipo associativo (cache) per far sì che ogni componente possa controllare solo nella cache se all'indirizzo dove deve cercare c'è un dato per lui; Così solo in caso affermativo scende nella persistenza e preleva cosa gli serve.

### 3.2.3. Publish/subscribe style:



In questo caso abbiamo dei componenti che svolgono il ruolo di publisher e pubblicano all'interno del sistema i dati (li mettono in memoria e se ne dimenticano). I subscriber si registrano effettuando l'operazione subscribe e vengono notificati nel momento in cui l'evento ha un'occorrenza all'interno del sistema. Qui abbiamo sempre un disaccoppiamento completo di tutti e tre i tipi: componenti anonimi(spaziale), middleware come coda di messaggi realizzato come una memoria(temporale), nessuno resta in attesa dell'altro in quanto il sistema è di tipo attivo e vengono mandate notifiche (di sincronia).

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

Ci sono due tipi di varianti di questo stile:

**Topic-based:** I subscriber si sottoscrivono agli specifici topic, identificati tramite keyword. Questo ha un'espressività limitata, perché è possibile solo specificare le keyword associate al topic. C'è poca flessibilità nella ricerca dei dati

**Content-based:** Gli eventi sono classificati tramite il loro contenuto e quindi ho metadati associati agli eventi. Quando i consumer sottoscrivono lo fanno tramite dei filtri che gli permettono di cercare eventi e dati di interesse nel sistema. Questa soluzione è più elastica della precedente.

L'implementazione più semplice di un sistema publish subscriber è un sistema centralizzato dove abbiamo un componente detto **broker degli eventi** (intermediario, implementato tramite un singolo nodo) che mantiene un repository delle sottoscrizioni ed effettua il matching quando i publisher mettono il dato, così da inviare le notifiche a seconda del topic o del content (dipende dal tipo di sistema). Il vantaggio è che l'applicazione ha una struttura semplice, ma gli svantaggi sono la scalabilità (posso avere solo quella verticale) e la mancanza di tolleranza ai guasti poiché ho un single point of failure (il broker).

Usando un'implementazione distribuita posso avere un broker multiplo su diversi server risolvendo il problema del *single point of failure*. Posso implementare questi event broker cooperativi in una rete *peer to peer*. Maggiore è il grado di distribuzione di questi event broker, maggiore diventa la complessità dell'architettura.

Nello scegliere quale stile architettonico utilizzare dobbiamo studiare i requisiti extra-funzionali:

- Costi
- Scalabilità ed elasticità
- Performance
- Affidabilità
- Tolleranza ai guasti
- Usabilità e gestibilità.

## 3.3. Tipi di architetture

Con architettura di sistema intendiamo l'istanziazione a runtime del sistema distribuito (ovvero il suo deployment a runtime: quali sono componenti, come interagiscono tra loro, dove si posizionano all'interno del sistema).

Esistono 3 tipi di architettura: centralizzata, decentralizzata ed ibrida.

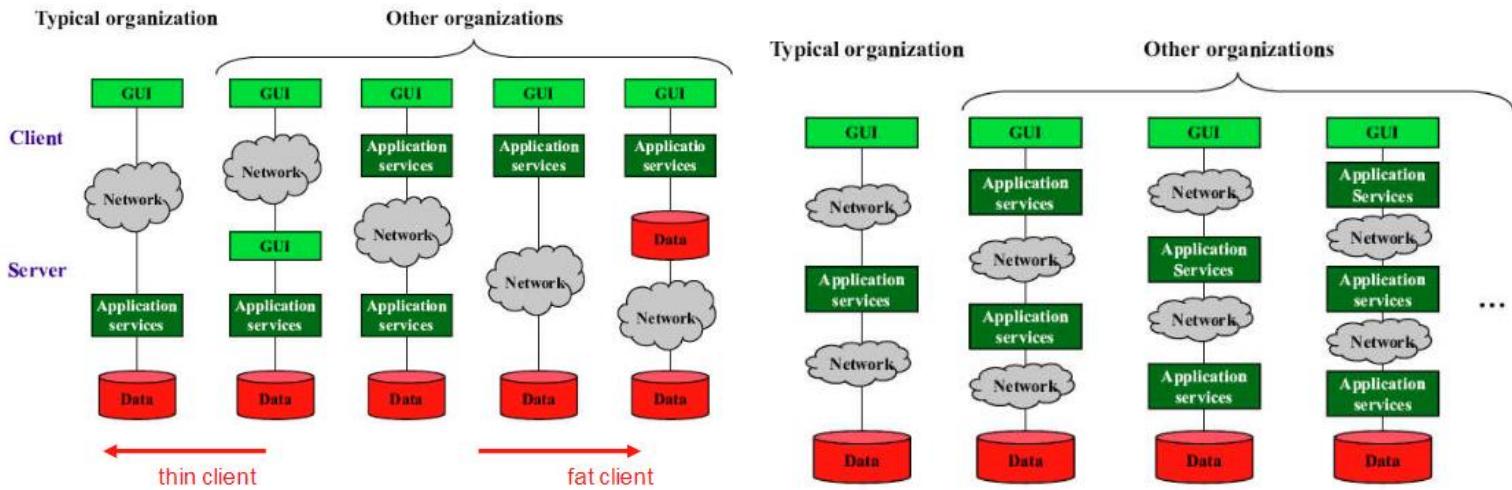
### 3.3.1. Architettura di sistema centralizzata

#### 3.3.1.1. Modello client-server

Modello **client server**: Abbiamo due entità che interagiscono fra di loro che possono essere su macchine differenti ovvero client e server. Il tutto si basa su domanda e risposta, dove il client richiede uso servizio offerto dal server. La comunicazione è basata su scambio di messaggi. Le operazioni possono essere con stato e senza stato (sono operazioni idempotenti, perché ogni volta che rispondono danno lo stesso stato). La comunicazione tra client e server è tipicamente sincrona e bloccante. L'architettura è caratterizzata quindi da stretto accoppiamento perché i componenti comunicano fra loro.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

### Mappare i livelli logici (layer) sui livelli fisici (tier) nel deployment dell'architettura:



I sistemi più frequenti sono le architetture 2 tier (2 livelli fisici) e 3 tier (3 livelli fisici). Il deployment può avere diverse organizzazioni (livello di presentazione, livello di logica e livello dei dati). Il tutto è composto da un livello fisico lato client e un livello fisico lato server; A seconda delle funzionalità gestite si può avere un client più o meno consistente.

Aggiungere un tier (passare da 2 a 5 livelli fisici per esempio) migliora flessibilità e distribuzione ma aumenta il degrado delle prestazioni in quanto aumentano costi di comunicazione e complessità di gestione ed organizzazione e gestione dell'intera struttura. Da un'architettura multi tier possiamo avere una distribuzione di tipo verticale o di tipo orizzontale.

## 3.3.2. Architettura di sistema decentralizzata

### 3.3.2.1. Architetture peer-to-peer (P2P)

#### Peer to peer (P2P)

Classe di sistemi ed applicazioni che usano risorse distribuite per eseguire una funzionalità in modo decentralizzato. Un peer è una entità con capacità simili alle altre nel sistema.

L'obiettivo dei sistemi P2P è quello di condividere risorse, intese come cicli di CPU e spazio di memorizzazione.

#### 3.3.2.1.1. Caratteristiche sistemi P2P

- Tutti i nodi (peer) del sistema hanno stesse capacità e responsabilità
  - Nodi **indipendenti** e localizzati ai bordi (**edge**) della rete
  - Ogni peer ha funzioni di client e server e condivide risorse e servizi
    - **Servent** = server + client
  - Possono anche essere presenti nodi con funzionalità diverse rispetto agli altri (**supernodi**), infatti i sistemi P2P non hanno una vera e propria architettura decentralizzata, ma

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

un'architettura ibrida, in cui possono essere presenti dei nodi che hanno delle funzionalità superiori rispetto agli altri.

- Sono sistemi altamente distribuiti, poiché il numero dei nodi che compongono la rete può essere dell'ordine delle centinaia di migliaia.
- I nodi sono altamente dinamici ed autonomi, ovvero un nodo può entrare (*join*) e uscire (*leave*) dalla rete P2P in ogni momento.
  - Un aspetto cruciale è come gestire queste operazioni di ingresso e di uscita (join/leave) dalla rete P2P, poiché i nodi sono altamente dinamici quindi vanno e vengono dal sistema P2P, ed è quindi opportuno che le risorse all'interno della rete siano ridondanti, ovvero non siano possedute da un unico nodo ma da più nodi, in modo tale che se un nodo che possiede la risorsa non è in quel momento presente all'interno della rete, la risorsa (ad esempio il file) può essere in ogni caso recuperata da altri nodi che lo possiedono.
  - Questa ridondanza delle informazioni permette da una parte di gestire questo elevato grado di dinamicità dei nodi della rete P2P, dall'altra parte ha il vantaggio che rende reti P2P particolarmente robuste in caso di guasti.

### 3.3.2.1.2. Principali problemi da affrontare nelle reti P2P

- **Eterogeneità**
  - Eterogeneità hardware: troppi modelli con architetture diverse
  - Eterogeneità del software: differenze tra sistema operativo e ambiente software
  - Eterogeneità di rete: differenti connessioni di rete e protocolli
- **Scalabilità**
  - Correlata a prestazioni e larghezza di banda
- **Località**
  - Posizione e la località dei dati
  - Prossimità di rete
  - Interoperabilità dei nodi
- **Tolleranza ai guasti**
  - Gestione dei guasti
- **Prestazioni**
  - Routing efficiency
  - Load balancing
  - Auto-organizzazione
- **Evitare free-riding**
  - **Free-riding:** peer egoisti e non disposti a condividere le risorse (nocivi per il funzionamento della rete poiché rendono difficile il bilanciamento del carico di lavoro fra i nodi)
- **Anonimità e privacy**
  - **Onion routing** usato come tecnica di anonimizzazione delle comunicazioni in una rete; Si basa sull'idea di incapsulare i messaggi in strati di criptografia i quali verranno rimossi uno per volta per ogni nodo attraversato. Il cammino da nodo A al nodo B quindi prosegue in quanto ognuno conosce solamente il nodo che lo precede e quello che lo succede nella

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

catena. In questo modo il nodo di partenza e quello di destinazione non si conoscono. Il mittente del messaggio rimane quindi anonimo (anonimizzazione della privacy).

- Gestione della fiducia e reputazione di un nodo
  - Mancanza di fiducia tra peer che sono estranei gli uni agli altri
- Gestione delle minacce della rete
- Resilienza ai churn
  - I peer vengono, se ne vanno e addirittura falliscono randomicamente

### 3.3.2.1.3. Operazioni fondamentali di un nodo

Un nodo di un sistema P2P svolge le seguenti operazioni di base:

- **Bootstrap**: fase di entrata di un nodo nella rete P2P. Per entrare può avere una configurazione statica, può usare informazioni relative a precedenti nodi della rete (cache preesistenti), oppure utilizzare dei nodi della rete attivi il cui indirizzo è noto (nodi well-known).
- **Resource lookup**: una volta entrato, esso cercherà le risorse. Come vengono localizzate le risorse nella rete?
- **Retrieval**: il nodo infine deve prelevare la risorsa in qualche modo.

### 3.3.2.1.4. Overlay network

All'interno di una rete P2P si trova una struttura definita "overlay network", ovvero una rete virtuale che interconnette fra di loro i diversi peer, basata su una rete fisica sottostante. I nodi sono costituiti dai processi. I collegamenti di questa rete virtuale rappresentano i canali di comunicazione. Questa overlay network serve per il lookup delle risorse. Il routing sull'overlay network avviene a livello applicativo (overlay routing), e ciò denota il problema che due nodi collegati direttamente nell'overlay network (rete astratta) possono non essere direttamente collegati nella rete fisica.

- Oltre all'instradamento delle richieste alle risorse, un overlay network deve anche eseguire:
  - Inserisci ed elimina risorse
  - Aggiungi e rimuovi nodi
  - Identifica risorse e nodi
    - Identifico queste risorse utilizzando il **Global Unique Identifier (GUID)**, identificatore globalmente univoco ottenuto applicando un algoritmo di crittografia su dei metadati riguardanti la risorsa in questione.
- Distinguiamo due tipi di overlay network per la gestione delle risorse e dei nodi:
  - Structured overlay network
  - Unstructured overlay network

#### 3.3.2.1.4.1. Overlay network non strutturate

La topologia della rete è un **grafo random** non strutturale

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

- Nessuna struttura particolare sulla rete overlay in base alla progettazione (non ad anello, cubo o mesh)
- Ogni nodo si unisce alla rete seguendo alcune semplici regole locali
- Un nodo che si unisce contatta un insieme di vicini, selezionati (più o meno) casualmente
- Tempi di lookup maggiori rispetto alle reti strutturate (impredicibili in quanto non si conosce la forma della rete e risulta ignoto quanto to ci vorrà a trovare la risorsa).
- Nessun controllo sulla topologia di rete o sul posizionamento delle risorse sui nodi
  - **Pro:** l'inserimento e l'eliminazione di nodi e risorse sono facilmente gestibili
  - **Contro:** l'ubicazione delle risorse è complicata dalla mancanza di struttura -> prestazioni imprevedibili
- Esempi di rete non strutturata sono: Gnutella, FastTrack, eDonkey, ...

\* \* \*

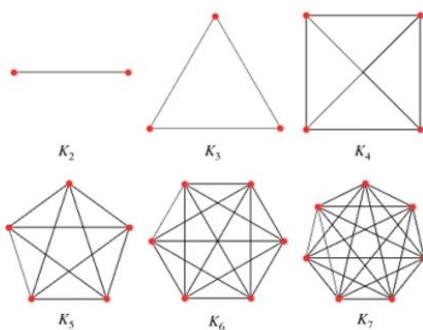
### 3.3.2.1.4.1.1. Modelli di grafi random

3 modelli principali per la creazione di grafi random.

1. **Erdos-Renyi**
2. **Small-world networks**
3. **Scale-free networks**

Proprietà principali di un grafo random

- **Coefficiente di clustering di un vertice di un grafo:** è una misura del grado con cui i nodi del grafo tendono a essere connessi tra di loro. In particolare, è il numero di archi tra i vicini di quel vertice diviso il numero massimo di archi possibili tra loro.
  - Se un vertice ha  $m$  vicini, il valore al denominatore sarà  $m \binom{m-1}{2}$
  - (esso infatti corrisponde al numero di nodi in un grafo completamente connesso).



- **Coefficiente di clustering di un grafo:** il valor medio del coefficiente di clustering su tutti i vertici. Tanto più il coefficiente di clustering è piccolo, tanto più i nodi sono poco connessi tra di loro. Tanto più il coefficiente di clustering è elevato, tanto più i nodi sono connessi tra di loro.
  - Nella maggior parte dei grafi che troviamo nel mondo reale, come ad esempio nelle reti sociali Facebook, Instagram, LinkedIn, ..., le persone (i nodi) tendono a creare dei grafi che

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

sono fortemente uniti tra di loro e hanno una densità di collegamenti abbastanza elevata. Il coefficiente di clustering di reti reali quindi tende a essere maggiore di coefficienti di clustering ad esempio delle reti generate con il modello Erdos-Renyi, il quale quindi non ci non ci permette di creare dei grafi reali.

- **Lunghezza media del percorso più breve:** all'interno di un grafo andiamo a calcolare il cammino minimo tra tutte le coppie possibili di nodi del grafo e ne calcoliamo il valore medio.

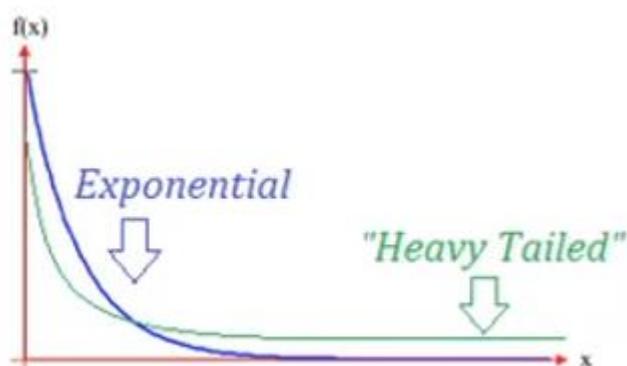
### 3.3.2.1.4.1.1.1. Modello Erdos – Renyi

Secondo questo modello, un grafo random si costruisce come segue:

Definiamo un numero di vertici fissato, definiamo la probabilità che vi sia un vertice tra due nodi, il grado di un vertice seguirà la distribuzione binomiale. Il grado di un vertice infatti è il numero di archi entranti e uscenti da quel vertice, e quindi  $P_k$  (probabilità che il nodo v sia connesso con un nodo k) segue la distribuzione binomiale.

Questo modello Erdos – Renyi ha un basso coefficiente di clustering, che non è una proprietà presente nelle reti reali. Un'altra caratteristica di questi è che il valor medio del percorso minimo, quindi la media di tutti i percorsi minimi tra tutte le possibili coppie di grafi, ha un basso valore.

Inoltre, il grafico di distribuzione di probabilità che viene generato con questo modello ha la caratteristica che la coda decade in modo esponenziale, mentre modelli reali di grafi hanno una coda pesante.



La coda esponenziale indica che la rete che viene generata è omogenea, ovvero senza hub, che invece è una caratteristica (oltre al grado di clustering abbastanza elevato) che troviamo nelle reti reali.

### 3.3.2.1.4.1.1.2. Modello Watts-Strogatz

I grafi creati con questo modello hanno un elevato coefficiente di clustering e godono della cosiddetta proprietà dello small-world: ovvero, presa la distanza media tra due nodi, essa dipende in modo logaritmico dal valore di  $n$ , dove  $n$  è il numero di nodi interni alla rete. Anche in questo modello vediamo assenza di hub, caratteristica non reale, che comparirà invece nel terzo modello di Albert-Barabasi.

#### Proprietà small-world:

Questo è un termine che viene anche spesso chiamato dei 6 ° di separazione, che deriva da un famoso esperimento sociale condotto negli Stati Uniti. L'idea è che presi due individui qualunque sulla terra, è possibile ricreare un percorso di contatti fra conoscenti pari a 6 che li congiunge. Presi quindi due nodi in una rete reale, la distanza tra questi risulta piccola, al massimo 6 ° di separazione. Le reti generate con il modello di Watts-Strogatz sono caratterizzate da un cammino piccolo ed un elevato coefficiente di

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

clustering. Queste sono due proprietà che troviamo nei grafi reali, però questo modello non tiene conto della formazione degli hub, che invece troviamo nelle reti reali.

### 3.3.2.1.4.1.1.3. Modello Albert-Barabasi

Il grado dei vertici segue la distribuzione della **legge di potenza** (*power law*).

NB: nel modello di Erdos – Renyi il grado di un vertice segue una distribuzione binomiale.

Legge di potenza:

La frequenza di un evento varia in base alla potenza di alcuni attributi dell'evento stesso.

- Questo modello definisce la legge di **invarianza di scala**, ossia una caratteristica di oggetti o di leggi fisico-matematiche che indicano che non viene cambiata la forma della rete anche se scaliamo le lunghezze di un fattore comune.
- Le distribuzioni di probabilità della legge di potenza sono a **coda pesante** (*heavy-tailed*)
  - Coda più pesante rispetto alla distribuzione esponenziale
- I nuovi nodi si attaccano a quelli esistenti secondo una logica preferenziale
  - Più un nodo è connesso alla rete, più è probabile che riceva nuovi collegamenti (per esempio gli influencer sui social network, che connettono prima i nuovi utenti)
  - Ci sono alcuni hub, ma il loro numero diminuisce in modo esponenziale (poiché la legge con cui si modella la creazione del grafo è la legge di potenza)
- Scale-free (senza scala): diametro del grafo  $\approx \ln(\ln n)$ 
  - Al crescere di  $n$  (numero di nodi nella rete), il diametro cresce molto lentamente, poiché è il logaritmo del logaritmo (à proprietà invarianza di scala)
  - Le reti senza scala sono altamente resilienti contro componenti difettosi: interconnessione ideale anche per il cloud computing
  - NB: il diametro di un grafo è la più grande lunghezza del cammino minimo (in termini di numero di archi traversati) tra tutte le coppie di nodi.

\* \* \*

### 3.3.2.1.4.1.2. Reti peer to peer non strutturate: Routing

Classifichiamo le reti non strutturate in base al livello di distribuzione dell'indice risorse-peer.

- **Reti non strutturate centralizzate**: directory centralizzata
- **Reti non strutturate decentralizzate pure**: directory completamente distribuita
  - Diffusione dei messaggi basata sui meccanismi di flooding, gossiping o random walk
- **Reti non strutturate ibride**: directory semi-centralizzata
  - Flooding limitato ai super-peer (super nodi che possiedono delle parti dell'indice, ossia informazioni riguardo i peer che coordinano)

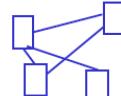
Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.



### 3.3.2.1.4.1.2.1. Reti centralizzate

Reti caratterizzate da un indice che tiene traccia di quali sono i peer che possiedono determinate risorse. Esso è mantenuto all'interno di una rete di un unico nodo o di un cluster di nodi.

- Un nodo centralizzato (directory server) è responsabile del mapping risorse-peer (indice), fornendo un servizio di discovery dei peer e di ricerca delle risorse.
- Il routing è molto semplice, infatti se l'indice di risorse peer è conosciuto da un nodo solo o da un sottoinsieme di nodi, basterà interrogare quel nodo su chi possiede la risorsa cercata.
- Problemi di questa architettura:
  - Gestione costosa della directory centralizzata
  - Collo di bottiglia costituito dal nodo centrale in termini di prestazioni (scalabilità limitata)
  - Single point of failure (motivi tecnici e legali)



### 3.3.2.1.4.1.2.2. Reti decentralizzate pure: flooding

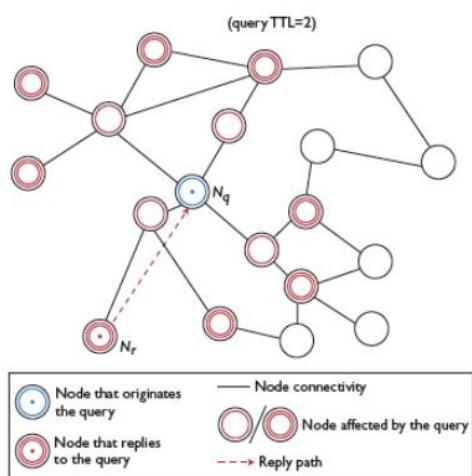
Ogni nodo ha una conoscenza soltanto locale delle risorse (i suoi vicini).

- Per effettuare il lookup delle risorse all'interno della rete peer decentralizzata pura si utilizza l'approccio del flooding (inondazione)
  - Ogni peer interessato a localizzare la risorsa propaga (flood) la richiesta ai peer vicini, che a loro volta inviano la inoltrano ai loro vicini, escludendo quello da cui hanno ricevuto la richiesta (la rete viene quindi inondata di messaggi). Questo avviene fino a che la richiesta è risolta, oppure fino a quando viene raggiunto un massimo numero di peer interrogati, definito Time To Live (TTL), il quale limita il raggio della ricerca evitando che il messaggio sia propagato all'infinito all'interno della rete
    - Il TTL viene inizializzato a un valore e a ogni inoltro della richiesta viene decrementato
  - Per evitare che una richiesta circoli più volte per lo stesso nodo in uno stesso ciclo di lookup, viene assegnato alla richiesta un ID univoco, dove ogni nodo tiene traccia dell'ID delle richieste che ha visto precedentemente, così da eliminare una richiesta qualora l'avesse già inoltrata.
- Costo di look-up:  $O(N)$ , con  $N$  numero di nodi nella rete
  - Poiché le reti P2P possono essere delle reti anche a larghissima scala (centinaia di migliaia di nodi), il costo di lookup è molto elevato, poiché nell'effettuarlo non possiamo utilizzare nessuna conoscenza sulla struttura della rete, cosa che invece viene fatta nelle reti P2P strutturate. Nelle reti strutturate il costo sarà ridotto (lineare in termini di  $n$ ).
- Una volta che è stato identificato il nodo che possiede la risorsa, ci sono due modi per mandare la risposta indietro alla query di look-up:
  - Diretto: il peer che possiede la risorsa può rispondere direttamente
  - Backward routing
    - La risposta può seguire il percorso a ritroso della richiesta
    - Si usa il GUID per individuare il backward path

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

- Il vantaggio di questo approccio è che si diffonde in un sottoinsieme dei nodi la conoscenza sull'indice delle risorse, perché quelli che vengono attraversati durante il cammino a ritroso possono tenere memorizzata in cache l'informazione contenuta nella risposta.

- Esempio di flooding di rete non strutturata



Si suppone che il nodo cerchiato in azzurro sia il nodo che ha originato la richiesta e che il TTL sia 2 (una volta attraversati due hop la richiesta viene cancellata). Il nodo in celeste al primo passo inoltra la richiesta a tutti i nodi che conosce, ovvero i suoi vicini (1,2,3,4). Nessuno di loro possiede la risorsa (i nodi che possiedono una risorsa a cui Nq è interessato sono i nodi identificati dal doppio cerchio in rosso). Al passo successivo il TTL viene decrementato da 2 a 1, perché nel frattempo ha attraversato un hop. La richiesta viene inoltrata da ciascun nodo che l'aveva ricevuto a tutti i suoi vicini. Una volta che Nr riceve il messaggio, risponde a Nq. Esso può rispondere in modo diretto che possiede la risorsa oppure seguire lo schema del backward routing. Arrivati ai nodi cerchiati due volte in rosso raggiunti dal TTL=2, la richiesta non viene ulteriormente prepagata perché il TTL si azzerà.

## Problemi flooding

- Crescita esponenziale del **numero di messaggi**
  - Possibilità di attacchi di tipo *denial-of-service*
  - Nodi black-hole in caso di congestione (buchi neri all'interno della rete)
    - Può succedere che i nodi diventino sovraccarichi (accumulano messaggi ma non riescono a smaltirli) quindi si ha appunto una sorta di buco nero
  - Non è realistico esplorare tutta la rete
- Costo della ricerca ( $O(N)$ )
  - Le risposte dovrebbero essere accessibili in tempi ragionevoli
  - Problema relativo al determinare il raggio di ampiezza del flooding (ovvero il TTL)
    - Si vuole mantenere un valore di TTL abbastanza basso, per evitare l'inondazione della rete di messaggi, ma se è troppo piccolo non è detto che il nodo che possiede risorsa venga raggiunto
- Non è garantito che vengano interrogati (**tutti**) i nodi che posseggono la risorsa
- **Traffico di query elevato**
  - Messaggi senza risultato occupano comunque banda e risorse di rete
- **Mancanza di relazione tra topologia di rete** virtuale e fisica
  - Quanto sono distanti i peer "vicini"?
    - La circolazione dei messaggi avviene sulla base dell'overlay network, ma come accennato specialmente nelle reti P2P non strutturate, l'overlay network è costruita in modo random e non segue le caratteristiche della rete fisica. Pertanto, due peer che

**Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.**

sono tra di loro vicini nell'overlay network potrebbero essere piuttosto distanti in termini poi di rete fisica.

### Random walk – soluzione alla congestione di rete

Per contrastare la crescita esponenziale dei messaggi inviati nel flooding, esiste la tecnica del random walk, dove ogni nodo sceglie a caso fra i suoi vicini a chi mandare la richiesta. Questo riduce drasticamente i messaggi inviati, con lo svantaggio di un aumento significativo del tempo di lookup. Una evoluzione del random walk è effettuare più cammini random anzichè uno. Più percorsi vengono effettuati, più è probabile di trovare prima il nodo desiderato, ma più ci si avvicina alla tecnica del flooding (con tutti i suoi svantaggi); Va trovato il giusto equilibrio.

#### **3.3.2.1.4.2. Overlay network strutturate**

La topologia della rete non emerge in modo random ma è ben definita seguendo algoritmi deterministici. Assegnando risorse ai nodi la topologia viene quindi mantenuta. Esistono numerose soluzioni di reti di overlay strutturate che differiscono per la topologia; Alcune di queste sono: anello, albero, ipercubo, ...

##### **Goal:**

Migliorare l'efficienza nella localizzazione delle risorse minimizzando il numero di messaggi inviati nella fase di lookup.

##### **Svantaggio:**

Le operazioni di inserimento e rimozione di un nodo però diventano più costose rispetto a prima in quanto per mantenere la topologia l'inserimento deve sottostare ad alcune regole.

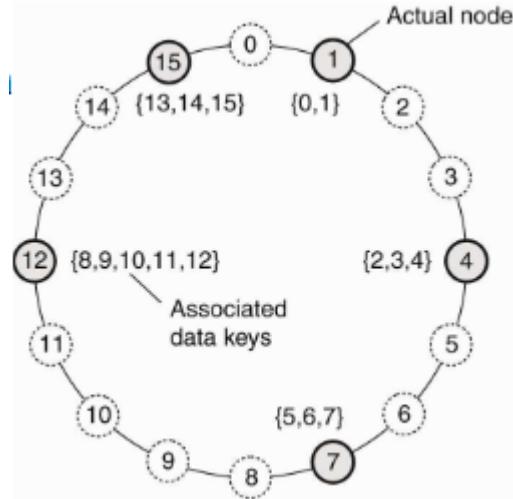
Il routing nelle strutturate si basa su una Hash Table Distribuita (DHT - struttura dati che ci consente di effettuare la ricerca efficiente delle risorse). Nelle reti P2P ad ogni risorsa viene assegnato un identificativo univoco (GUID). Lo spazio degli identificatori per peer e risorse è lo stesso. Il GUID viene prelevato grazie ad una funzione hash (SHA-1, secure hash 1, che crea un digest della risorsa o peer a partire dallo stesso). L'idea del routing è che peer contigui (dal punto di vista del GUID) possiedano lo stesso numero di risorse. Per instradare una richiesta di una risorsa verso il peer ad esso associata, occorre mappare univocamente il GUID della risorsa in quello di un peer, in base ad una metrica di distanza (la richiesta viene instradata al peer con il GUID più vicino alla risorsa cercata). Questo sistema sfrutta in modo intelligente la rete strutturata riducendo notevolmente il tempo di lookup.

##### **3.3.2.1.4.2.1. Topologia ad anello (algoritmo di Chord)**

Chord è un algoritmo, protocollo per effettuare il lookup in reti P2P strutturate. La topologia predefinita è ad anello, ed il mapping viene effettuato sfruttando una funzione specifica di hashing definito consistent hashing. Questa funzione viene molto usata (è stata adottata in Akamai, la principale content delivery network di Chord). Ogni nodo è responsabile delle risorse mappate tra il suo ID e quello del nodo precedente.

**Esempio:** Il nodo 4 della figura è responsabile delle risorse 2, 3 e 4. Il successore rappresenta l'ID successivo alla risorsa; il successore di 2, 3 e 4 è il nodo 4 (vale anche sè stesso). In questo caso si parla di metrica di distanza lineare. L'unica accortezza è usare il modulo per la chiusura dell'anello.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

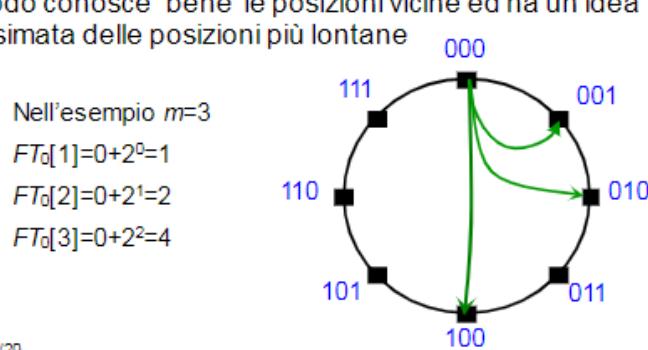


In Chord la funzione di hashing usata per mappare nodi e risorse in un ID è il **consistent hashing**: i nodi e le risorse sono uniformemente distribuite nello stesso spazio identificativo, lo stesso cerchio, usando una hash function. Usare il consistent hashing ci dà **vantaggi quali robustezza** (in caso di ridimensionamento tabella di hash, se vengono aggiunti o rimossi nodi), il mapping delle risorse sui nodi non cambia in modo significativo. Altro vantaggio è che **l'assegnazione delle risorse ai peer avviene in modo bilanciato**, ovvero non ci sono peer che gestiscono molte più risorse di altri peer, equa distribuzione del carico legato alla gestione delle risorse stesse.

In una hash table il contenuto è distribuito fra i diversi nodi; Nel caso di Chord ogni nodo mantiene le informazioni di routing che conosce in una tabella chiamata finger table. Quest'ultima ha una struttura ben definita: posto  $m$  il numero di bit dell'identificatore univoco per i peer e le risorse (il GUID), la finger table avrà dimensione pari a  $m$ .

### Esempio:

- Se  $FT_p$  è la FT del nodo  $p$ , allora  $FT_p[i] = \text{succ}(p + 2^{i-1}) \bmod 2^m$ ,  $1 \leq i \leq m$ 
  - $\text{succ}(p+1)$ ,  $\text{succ}(p+2)$ ,  $\text{succ}(p+4)$ ,  $\text{succ}(p+8)$ ,  $\text{succ}(p+16)$ , ...
- **Idea della finger table**
  - Ogni nodo conosce “bene” le posizioni vicine ed ha un’idea approssimata delle posizioni più lontane



Supponendo che la chiave sia composta da 3 bit, vediamo che abbiamo 8 possibili identificatori nel sistema; la finger table posseduta da ciascun nodo ha quindi 3 righe. Su ogni riga è presente l'ID di nodi successivi

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

(che gestiranno una determinata risorsa), e ciò serve per il lookup. Ogni nodo quindi conosce la porzione di anello prossima a lui, e ha solo un'idea di cosa accade lontano da lui.

### 3.3.2.1.4.2.1.1. Routing nelle finger table

Presi una chiave  $k$  della risorsa si vuole individuare il successore. Il processo si basa su un algoritmo, dove fondamentalmente si cerca di inoltrare la richiesta al nodo con l'ID più vicino possibile alla chiave cercata. Il costo di lookup è basso,  $O(\log n)$ , e questo garantisce che si raggiungono velocemente le vicinanze del punto cercato andando via via di salti sempre più piccoli.

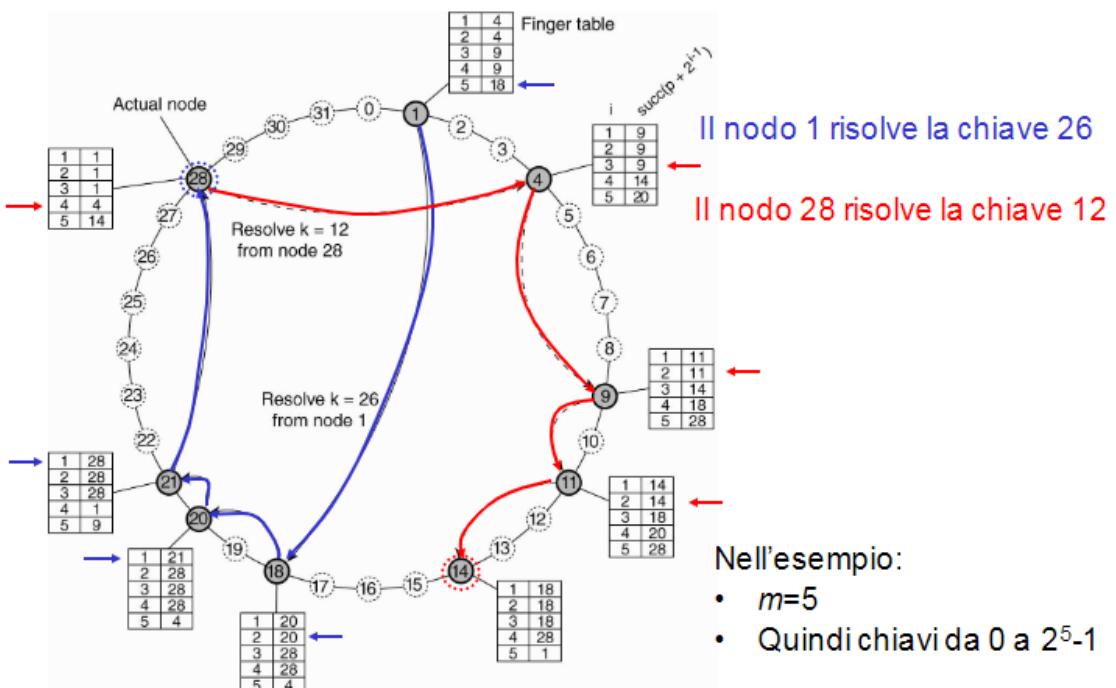
- Come risolvere la chiave  $k$  nell'identificatore di  $\text{succ}(k)$  a partire dal nodo  $p$  (**algoritmo di routing**):
  - Se  $k$  è nella zona di competenza di  $p$ , la ricerca termina
  - Se  $p < k \leq FT_p[1]$ ,  $p$  inoltra la richiesta al nodo successore
  - Altrimenti,  $p$  inoltra la richiesta al nodo  $q$  con indice  $j$  in  $FT_p$

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

$q$  è il nodo più lontano da  $p$  la cui chiave viene prima della (o è uguale alla) chiave  $k$

#### Esempio:

Come avviene il routing in Chord in una rete di 32 nodi (2 alla 5 ovvero ogni FT ha 5 righe).



Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

Prendiamo come esempio la FT del nodo 1. La prima riga prende il successore del nodo  $2^0$  dopo 1, ovvero il successore di 2. La seconda riga prende il successore del nodo  $2^1$  dopo 1, ovvero il successore di 3. Poi la terza riga prenderà il successore del nodo  $2^2$  dopo 1, ovvero il successore di 5, e così via.

### 3.3.2.1.4.2.1.2. Ingresso e uscita dei nodi in una rete Chord

Dobbiamo tener conto che non bisogna cambiare la struttura topologica nel processo, quindi non è un procedimento banale randomico come nelle reti non strutturate. L'operazione di lookup viene portata a termine in  $O(\log n)$ , quella di ingresso/uscita in  $O(\log(\log n))$ . Ogni nodo mantiene il puntatore al successore nella prima riga della sua FT. Per semplificare le operazioni di join e leave, ciascun nodo conosce anche il suo predecessore (non riportato nella FT).

- Al **join** nell'overlay network, il nodo  $p$ :
  - Contatta un nodo arbitrario a cui chiede la ricerca di  $\text{succ}(p+1)$
  - Si inserisce nell'anello
  - Inizializza la sua FT chiedendo  $\text{succ}(p + 2^{i-1})$ ,  $2 \leq i \leq m$
  - Aggiorna la FT del nodo che lo precede sull'anello
  - Trasferisce dal suo successore su se stesso le chiavi di cui è responsabile
- Alla **leave** dall'overlay network, il nodo  $p$ :
  - Trasferisce al suo successore le chiavi di cui è responsabile
  - Aggiorna nel nodo che lo precede il puntatore al nodo che lo segue sull'anello
  - Aggiorna nel nodo che lo segue il puntatore al nodo che lo precede sull'anello

Per mantenere aggiornati i valori delle FT in quanto inserimento e rimozione cambia i valori dei nodi, ma non delle risorse, periodicamente viene effettuato refresh della FT.

### 3.3.2.1.4.2.1.3. Vantaggi e svantaggi del Chord

**Vantaggi:**

- **Bilanciamento del carico:** Chord distribuisce uniformemente le chiavi fra i nodi
- **Scalabilità:** Le operazioni di lookup sono efficienti ( $O(\log n)$ )
- **Robustezza:** Chord aggiorna periodicamente le finger table dei nodi per riflettere i mutamenti della rete

**Svantaggi:**

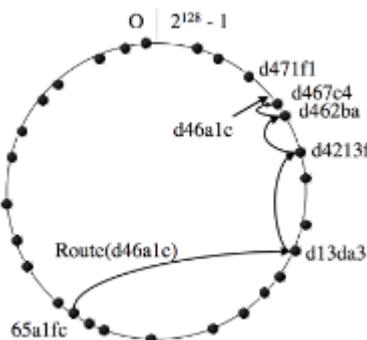
Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

- Manca una nozione di prossimità fisica
- Supporto costoso per ricerca senza matching esatto

### 3.3.2.1.5. RETE P2P Pastry

Pastry fornisce un **middleware** che ha inspirato lo sviluppo di altre applicazioni P2P con scopi differenti. La soluzione specifica del routing è basata su un meccanismo detto **Plaxton routing** (meccanismo di distribuzione di risorse all'interno di una rete). L'idea di base è nella metrica di distanza utilizzata, che non è più lineare, ma basata sul matching dell'ID che identifica la risorsa o il peer. L'idea del plaxton routing è che una risorsa verrà memorizzata sul nodo della rete che ha con la risorsa stessa il prefisso (porzione di ID) in comune più lungo possibile. In Pastry, oltre questo viene mantenuto anche un insieme di foglie (**leaf set**), ovvero ha conoscenza dei nodi più vicini a lui nello spazio bidirezionale degli ID.

Quindi mentre nel caso di Chord la metrica di distanza era lineare per identificare il nodo che possiede una risorsa, in Pastry ci si basa sul matching del prefisso più lungo. Questo tipo di rete vanta un'organizzazione più gerarchica dell'indirizzo rispetto a Chord, in quanto l'organizzazione delle chiavi è simile (sempre mappate su un anello), con la caratteristica aggiuntiva che sono anche rappresentate da d simboli di b bit. Ad ogni passo di routing effettuato nel lookup, il nodo coinvolto nel trasferimento del messaggio inoltra la query verso il nodo che ha ID più vicino a quello della risorsa cercata. Il nodo quindi guarda la sua tabella di routing e cerca quello che ha il matching su più simboli del prefisso possibile.



### 3.3.2.1.6. Plaxton routing (esempio di funzionamento)

Si prenda una chiave composta da  $b=2$  bit con  $d=4$  ovvero ogni simbolo può assumere 4 diversi valori.

**Tabella di routing del nodo 1220**

Regole di composizione della tabella di routing:

- Gli ID dei nodi sulla riga  $n$ -esima condividono le prime  $n$  cifre con l'ID del nodo corrente.
- La  $(n+1)$ -esima cifra degli ID sulla riga  $n$ -esima è il numero di colonna

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

$\lceil \log_2 N \rceil$  righe nella tabella  
 $2^b - 1$  elementi in ogni riga

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

Il Plaxton routing tiene conto maggiormente rispetto a Chord della prossimità dei nodi che si trovano nell'overlay network e della posizione nella rete fisica. Nel caso in cui ci siano più nodi che corrispondono al matching quindi si sceglie il nodo in base ad una legge di prossimità come per esempio il round trip time o il numero di hop che separano i due nodi.

La query di lookup è inoltrata quindi in base al meccanismo di longest prefix matching, verso il nodo che condivide con quello di destinazione almeno una cifra in più del nodo corrente, e se non esiste, al nodo numericamente più vicino.

Tabella di routing del nodo 1220

Routing verso 0321	0212	-	2311	3121
Routing verso 1106	1031	1123	-	1312
Routing verso 1201	1200	1211	-	1231
Routing verso 1221		1221	1222	1223

### 3.3.3. Architetture ibride

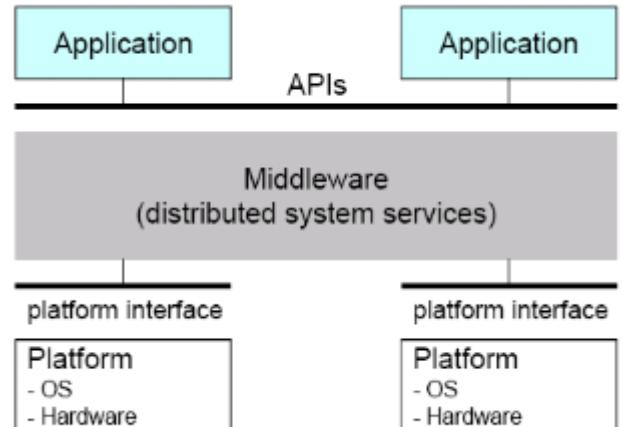
- **P2P con super peers** (peer con migliore connessione o maggiore capacità hardware): le richieste vengono passate fra i super peer per poi essere inoltrate al peer desiderato quando si arriva al super peer che lo gestisce.
- **BitTorrent**: il file da scaricare viene suddiviso in chunk, così che più nodi possano scaricarli in parallelo, e ogni volta che un peer acquisisce un nuovo chunk lo mette a disposizione per gli altri.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

## 3.4. Middleware

Il middleware è una sorta di sistema operativo di un sistema distribuito. Esso mette a disposizione una serie di servizi che permettono di costruire al di sopra delle applicazioni. Il middleware ha il compito di mascherare tutte le differenze fra i nodi del sistema distribuito, fornendo un livello di astrazione che agevola la realizzazione del sistema stesso.

- **DEF 1:** è un servizio di tipo general purpose in the middle (tra applicazioni e piattaforma).
- **DEF 2:** è uno strato software che fornisce un'astrazione di programmazione che permette di mascherare l'eterogeneità sottostante.
- **DEF 3:** Layer virtuale fra le applicazioni e le piattaforme che fornisce un grado significativo di trasparenza.



### 3.4.1. Tipologie

- **Object-oriented middleware**

Si tratta di middleware che permettono di rappresentare i componenti dell'app distribuita come oggetti con identità propria, ed espongono interfacce con metodi pubblici. La comunicazione tra i componenti è sincrona; gli oggetti comunicano fra loro tramite invocazione di metodo remoto.

- **Message-oriented middleware**

Supporta la comunicazione asincrona e persistente. Permette di realizzare app distribuite dove i componenti sono poco accoppiati fra di loro. Vanta numerose implementazioni basate su sistema a code di messaggi. Le code offrono un tipo di memorizzazione persistente, e possono rendere asincrona la comunicazione fra produttore e consumatore.

- **Middleware a componenti**

È un'evoluzione del precedente che supporta comunicazione sincrona e asincrona. I componenti vivono all'interno di container (application server) che sono in grado di gestire la configurazione e la distribuzione dei componenti e fornire ad essi funzionalità di supporto.

- **Service-oriented middleware**

Pone l'enfasi sul far interoperare componenti eterogenei, sulla base di protocolli standard aperti. Sfrutta comunicazione sincrona e asincrona e persistenza della comunicazione.

In molti casi il middleware segue uno specifico stile architetturale, il che offre semplicità progettuale, ma allo stesso tempo scarsa adattabilità e flessibilità.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

È preferibile un middleware che possa adattare comportamento e proprietà rispetto all'applicazione specifica e all'ambiente. Introduciamo a tal proposito i self-adaptive systems.

## 3.5. Sistemi software auto adattativi

L'idea è quella di avere un sistema software in grado (a runtime) di adattare il suo comportamento rispetto a modifiche nel sistema stesso o sull'ambiente in cui opera (sistema autonomico: paradigma di computazione che mediante adattamenti automatici permette di gestire la complessità del sistema).

Questi sistemi si adattano rispetto ai fallimenti, al carico di lavoro, alle variazioni di dispositivo utilizzato dall'utente.

**Sistema autonomico:** sistema in grado di ridurre al minimo l'interazione con un amministratore, poiché richiede un intervento umano molto limitato. Può adattarsi rispetto a cambiamenti che occorrono nel sistema stesso o nell'ambiente che lo circonda. L'adattamento può essere reattivo (il sistema si accorge dell'occorrenza di un evento e reagisce) o proattivo (maggior grado di complessità del sistema perché ha un comportamento predittivo: il sistema di controllo cerca di predire le azioni necessarie ad adattare il sistema).

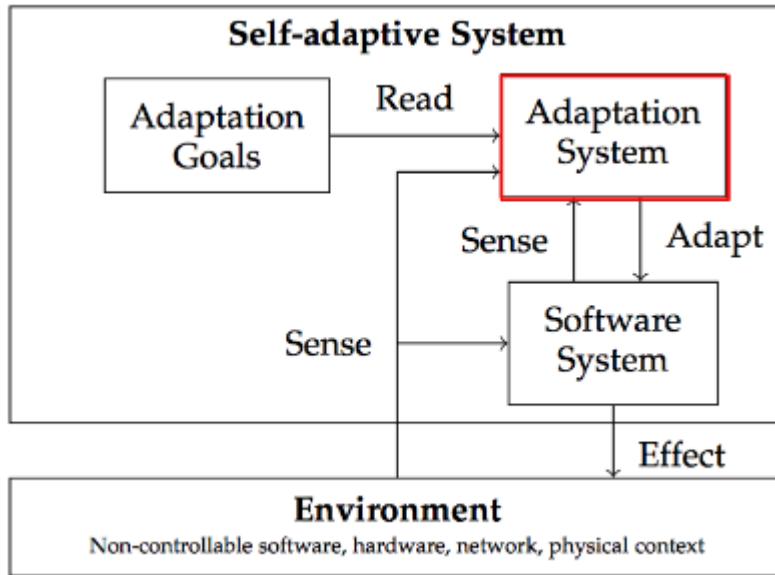
**Obiettivi per cui un sistema si adatta:**

- Auto configurarsi - il sistema è in grado di fare un tuning automatico dei parametri più adatti
- Auto guarirsi - capacità di scoprire, diagnosticare e reagire a guasti
- Auto ottimizzarsi - capacità di ottimizzare le proprie prestazioni
- Auto proteggersi - Capacità di proteggersi da attacchi esterni

**Cosa è necessario affinchè un sistema si automatizzi**

Il sistema deve conoscere il suo stato interno (self-awareness) e le condizioni operative in cui deve operare (self-situation). Ciò è possibile tramite un'attività di monitoraggio (self-monitoring), così che di conseguenza possa adattarsi (self-adjustment).

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.



### 3.5.1. Pattern MAPE (Monitor, analyze, plan, execute)

Si tratta di un'architettura di riferimento (pattern architettonico per sistemi software auto adattativi). Il tutto si basa su un ciclo che viene periodicamente eseguito (non è detto che da una fase si passi per forza ad un'altra, se per esempio in analisi non si evince nulla non si fa un plan).

**Monitor:** Raccoglie i dati di monitoraggio dalle risorse attraverso una rete di sensori; aggrega, filtra e correla questi dati affinchè possano essere analizzati.

**Analyze:** Osserva e analizza le situazioni per determinare se è necessario apportare qualche cambiamento rispetto alle politiche e agli obiettivi. Se sono necessarie modifiche, passa la richiesta di modifica a Plan.

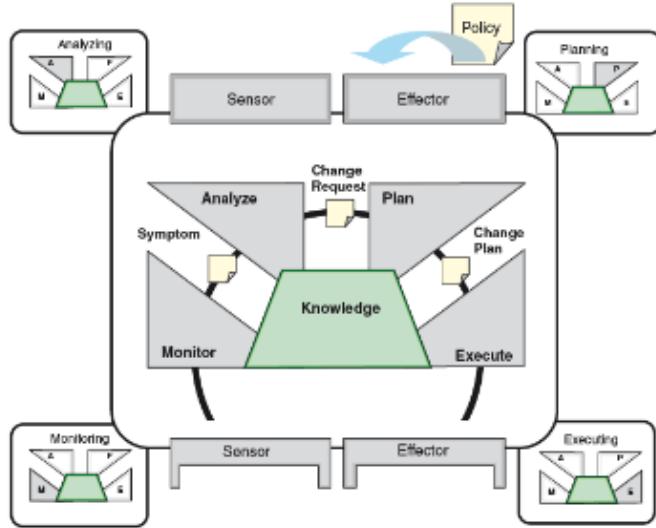
**Plan:** Determina quali azioni correttive devono essere eseguite in modo da attuare un'alterazione desiderata nelle risorse gestite. Per quanto riguarda la fase di planning ci sono diverse metodologie e tecniche che possono essere usate per capire l'adattamento più consono (teoria dell'ottimizzazione, euristiche, machine learning, teoria del controllo, teoria delle code).

**Execute:** Effettua le modifiche necessarie alle risorse gestite eseguendo le azioni determinate dal Plan tramite gli effettori.

I dati utilizzati dal sistema autonomo sono infine archiviati come conoscenza condivisa (Knowledge).

MAPE viene anche chiamato CADA (collect, analyze, decide, act) o OODA (observe, orient, decide, act), che sono altri acronimi per descrivere la stessa tecnica.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.



### 3.5.2. Esempi di sistemi auto-adattativi

Lo scopo dell'adattamento è quello di soddisfare requisiti applicativi riguardanti la qualità del servizio. Questi stessi requisiti possono essere stabiliti da Service Level Agreement, dove in particolare quelli non funzionali sono specificati nelle SLO (service level objective). Tra questi requisiti ci sono tempo di risposta dell'applicazione, disponibilità, costo pagato dall'utente per usare determinati servizi. I diversi esempi differiscono nella strategia di planning adottata, tra cui la formalizzazione di problemi di ottimizzazione, utilizzo di metodi derivanti dalla teoria del controllo, teorie euristiche e tecniche di machine learning.

#### 3.5.2.1. Amazon EC2 auto-scaling

Il servizio auto scaling fornisce una capacità autonoma di diminuire il numero di istanze EC2 utilizzate da un'applicazione; questo lo fa in base allo stato di salute delle istanze stesse e in base a come l'utente utilizza il sistema. "HeartBit monitoring" sono i messaggi che vengono inviati periodicamente per verificare lo stato di salute delle istanze. In questo caso si tratta di "auto-scaling dinamico" poiché dipende anche da metriche specificate dall'utente.

Un gruppo di autoscaling è composto da massimo quattro istanze, minimo 2. Questo numero è aumentato in modo reattivo, nonostante Amazon mette a disposizione oltre questa anche una politica proattiva dagli ultimi anni.

- POLITICA AUTO SCALING EC2 (reattiva - di riferimento per tutti i sistemi elastici):
 

Politica euristica in cui l'utente specifica una metrica che si traduce in una soglia, al superamento della quale vengono aggiunte istanze. La politica quindi risulta semplice ed intuitiva, ma le soglie pre specificate non sono relative all'applicazione quando invece dipenderebbero da essa (risulta non robusta contro variazioni del carico). Lo scale out è basato sul principio di aggiungere un'istanza ogni volta che l'applicazione utilizza una CPU maggiore del 70% della soglia per 1 minuto.
- PREDICTIVE SCALING EC2

La politica predittiva invece prova a predire il carico atteso dall'applicazione e l'utilizzazione delle

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

risorse EC2. Viene seguito un modello di machine learning di cui viene fatto il training (processo dove la macchina acquisisce informazioni storiche su cui basarsi per determinare i suoi comportamenti futuri). In questo caso l'attivazione della politica di scaling non viene fatta quindi sulla base di misure osservate. Come vantaggi è proattiva, ma come svantaggio richiede il training della rete neurale dove impara i dati storici che dovrà utilizzare.

### 3.5.2.2. MAPE decentralizzato

Per le soluzioni viste finora, tutti i componenti del sistema MAPE sono eseguiti dallo stesso nodo, il che costituisce una soluzione centralizzata. Quest'ultimo ha un evidente limite di scalabilità rispetto ad un sistema distribuito. Nel MAPE decentralizzato ci sono più modelli a disposizione, e si sceglie quello migliore in base alle caratteristiche e ai requisiti del sistema e dell'applicazione.

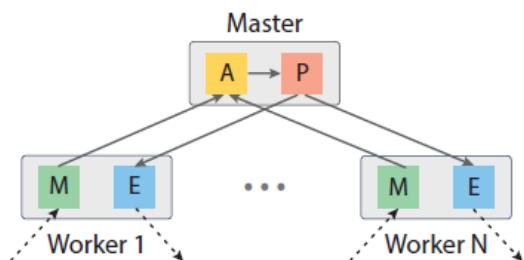
Si distinguono due tipi di architettura per il MAPE decentralizzato:

- **Architettura gerarchica:** le diverse fasi del ciclo mape sono distribuite fra i diversi nodi.
  - **Svantaggio:** chi si trova più in alto nella gerarchia può costituire un collo di bottiglia
  - **Vantaggio:** chi coordina vede più facilmente lo stato del sistema da controllare; la coordinazione risulta più semplice.
- **Architettura flat:** non c'è un coordinatore, ovvero non è presente una gerarchia (dove chi è al vertice può costituire collo di bottiglia e single point of failure). Il ciclo di lavoro è basato sulla politica che ogni nodo possiede tutte le fasi del ciclo MAPE. I nodi cooperano come peers eseguendo cicli.
  - **Vantaggio:** scala meglio rispetto alla gerarchica
  - **Svantaggio:** risulta più difficile coordinare il lavoro fra i nodi visto che tutti fanno tutto

#### 3.5.2.2.1. Esempi di architetture gerarchiche

**MAPE GERARCHICO MASTER WORKER:** Si identifica un nodo master e dei nodi worker; Il master si occupa delle fasi più delicate, effettua fase di analisi e planning (richiede conoscenza sull'ambiente in cui opera). I worker effettuano la fase di monitoraggio ed esecuzione.

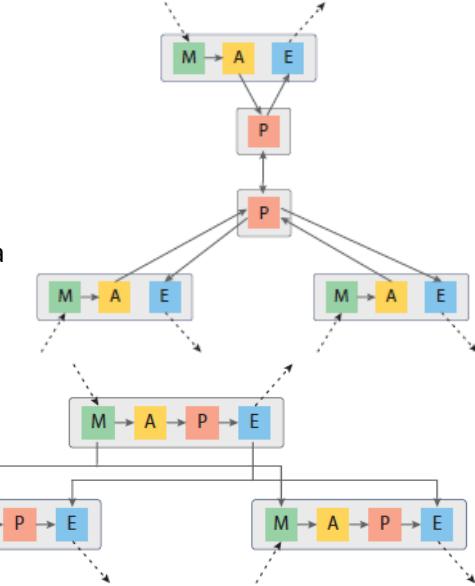
- Vantaggi: il master ha una visione globale in grado di raggiungere obiettivi di adattamento globali.
- Svantaggi: sovraccarico di comunicazione e rischio di collo di bottiglia da parte del master



Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

**PATTERN GERARCHICO REGIONALE:** Si hanno regioni multiple e indipendenti (basso accoppiamento), ognuna avendo una struttura gerarchica propria. Esistono dei nodi “planner” per ogni regione che si coordinano fra loro per prendere una decisione.

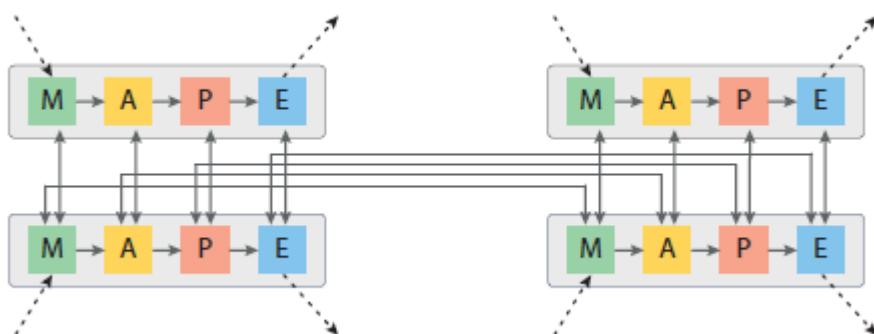
- Vantaggi: maggiore flessibilità in termini amministrativi, poiché le regioni possono essere di proprietà o amministrazioni diverse.
- Svantaggi: risulta più difficile trovare una strategia di planning per la coordinazione dei diversi planner regionali (più difficile ottenere obiettivi globali).



### 3.5.2.2.2. Esempi di architetture flat

**PATTERN DI CONTROLLO COORDINATO:** Si basa su molteplici cicli di controllo, ciascuno incaricato di controllare una parte del sistema. I cicli sono coordinati attraverso l’interazione reciproca.

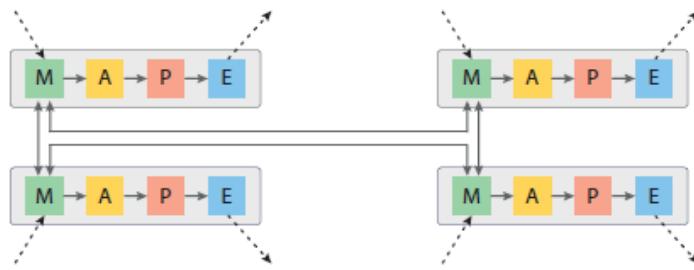
- Vantaggi: migliore scalabilità
- Svantaggi: risulta più difficile prendere decisioni adattative che tengano conto dell’intero sistema.



**PATTERN DI INFORMATION SHARING:** Il coordinamento avviene solo fra le fasi di monitoraggio.

- Vantaggi: maggior scalabilità
- Svantaggi: mancanza di coordinazione fra le fasi di planning. Le azioni di adattamento possono essere conflittuali.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.



## 4.COMUNICAZIONE NEI SISTEMI DISTRIBUITI

Comunicazione nei sistemi distribuiti è prettamente basata sullo scambio di messaggi. La soluzione largamente utilizzata per la comunicazione è quella di suddividere il problema in livelli, in modo tale che a livello logico ciascun livello del sistema comunica con lo stesso livello nell'altro sistema, quindi il livello fisico con il livello fisico e livello di rete con il livello di rete (e via dicendo quindi).

Il modello di riferimento ISO/OSI si può adattare ai sistemi distribuiti, andando a introdurre un livello legato al middleware, che è il collante dei sistemi distribuiti, posto appunto in mezzo tra i sistemi sottostanti e le applicazioni.

Quindi a livello del middleware vengono forniti dei servizi comuni e dei protocolli di tipo general-purpose, tipicamente indipendenti dalle specifiche applicazioni che verranno realizzate al di sopra del middleware, con l'obiettivo di nascondere con diversi gradi l'eterogeneità dei sistemi sottostanti.

Nell'ambito dei protocolli realizzabili a livello middleware per i sistemi distribuiti abbiamo:

- **protocolli di comunicazione**: per invocare chiamate a procedura remota, invocazione di metodi remoti, accodare messaggi, supportare streaming e multi casting
- **protocolli di naming**: permettono di condividere risorse tra applicazioni
- **protocolli di sicurezza**: per consentire alle applicazioni di comunicare in modo sicuro
- **protocolli per il consenso distribuito** quindi andremo a studiare nell'ambito del corso diversi algoritmi per raggiungere il consenso in modo distribuito: per far sì che venga raggiunto l'accordo tra i diversi componenti del sistema distribuito
- **protocolli di locking distribuito**
- **protocolli di consistenza dei dati**, quindi come mantenere diversi livelli di consistenza

### **4.1. Tipi di comunicazione**

In un sistema / applicazione distribuita possiamo avere diverse tipologie di comunicazione, in particolare distinguiamo la comunicazione in base a:

- persistenza
- sincronizzazione
- dipendenza dal tempo

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

### 4.1.1. Comunicazione persistente o transiente

#### **Comunicazione persistente:**

Il messaggio viene memorizzato dal middleware di comunicazione per tutto il tempo necessario alla consegna, quindi il mittente non deve necessariamente essere in esecuzione una volta che hai inviato il messaggio (perché una volta che hai inviato il messaggio questo sarà memorizzato dal middleware di comunicazione) e inoltre non è necessario che nel momento in cui il mittente invia il messaggio il destinatario sia in esecuzione (perché appunto il messaggio viene memorizzato dal middleware di comunicazione).

Quindi, se abbiamo una comunicazione persistente possiamo avere il [disaccoppiamento temporale](#).

#### **Comunicazione transiente:**

Il messaggio è memorizzato dal middleware di comunicazione soltanto finché il mittente e il destinatario sono in esecuzione.

Se la consegna non è disponibile, il messaggio viene cancellato.

### 4.1.2. Comunicazione sincrona

Una volta che viene sottoposto il messaggio al middleware di comunicazione, il mittente si blocca fintanto che l'operazione non è completata.

L'invio e la ricezione sono quindi operazioni bloccanti.

#### **4.1.2.1. Tipologie di comunicazione sincrona**

Esistono diverse tipologie di comunicazione sincrona che possiamo andare a differenziare tra di loro in base al momento fino a cui il mittente rimane bloccato.

In particolare, supponiamo che il mittente sia il client e il destinatario sia il server, una volta effettuato il servizio il server diventa il mittente e il client il destinatario.

Esaminando l'invio della richiesta di servizio da parte del client, abbiamo diverse situazioni possibili nella comunicazione sincrona:

1. Il mittente (client) rimane bloccato finché il middleware non prende il controllo della trasmissione.
2. Il mittente rimane bloccato finché il messaggio non viene ricevuto dal destinatario
3. Il mittente rimane bloccato finché il messaggio non viene elaborato dal destinatario.
  - La sincronizzazione in questo caso avviene successivamente al processamento effettuato dal server

### 4.1.3. Comunicazione asincrona

Una volta che il mittente ha inviato il messaggio al middleware di comunicazione, esso continua l'elaborazione e il messaggio viene memorizzato temporaneamente dal middleware fino all'avvenuta trasmissione.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

L'invio è un'operazione non bloccante, infatti una volta che il mittente ha inviato il messaggio continua l'elaborazione, mentre la ricezione può essere bloccante o non bloccante a seconda della specifica implementazione.

#### 4.1.4. Comunicazione discreta o streaming

##### Comunicazione discreta

Ogni messaggio inviato è un'unità di informazione completa a sé stante, indipendente dagli altri messaggi.

##### Comunicazione a stream (streaming)

La comunicazione prevede l'invio di molti messaggi, in relazione temporale tra di loro o in base all'ordine di invio, che servono a ricostruire l'informazione completa.

### 4.2. Combinazioni dei tipi di comunicazione

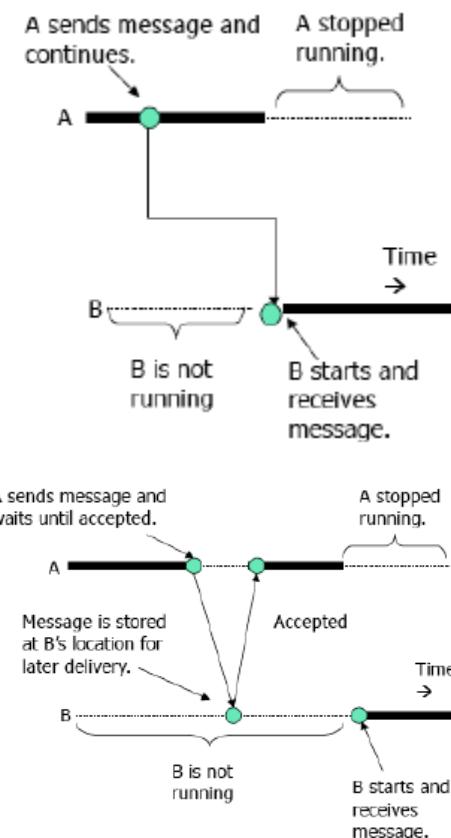
#### 4.2.1. Combinazioni tra persistenza e sincronizzazione

##### 4.2.1.1. Comunicazione persistente e asincrona

Il mittente invia il messaggio, e nel momento in cui il mittente invia il messaggio il destinatario del messaggio può non essere attivo (ad esempio può non essere in quel momento collegato a Teams, quindi Teams memorizza il messaggio e nel momento in cui il destinatario del messaggio accede a Teams in quel caso riceve la notifica del messaggio, e nel momento in cui riceve la notifica del messaggio può accadere che il mittente del messaggio non sia più attivo/presente all'interno del sistema di comunicazione). Ci offre persistenza e ci permette di disaccoppiare temporalmente mittente e destinatario

##### 4.2.1.2. Comunicazione persistente e sincrona

Il middleware di comunicazione ci permette di memorizzare l'informazione, ma il mittente rimane bloccato finché il destinatario non accetta la comunicazione. Il mittente quindi invia il messaggio, il quale, poiché la comunicazione è di tipo persistente, viene memorizzato mentre il destinatario del messaggio non è attivo, e il middleware di comunicazione si occupa di inviare una notifica relativa all'accettazione del messaggio. Quindi potrà accadere che nel momento in cui il destinatario riceve il messaggio, il mittente non sia presente all'interno del sistema.

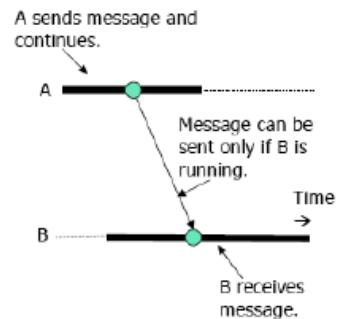


Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

## 4.2.2. Combinazioni tra transienza e sincronizzazione

### 4.2.2.1. Comunicazione transiente e asincrona

Il mittente invia il messaggio e poi continua nella sua elaborazione/attività mentre il destinatario riceve il messaggio. Poiché la comunicazione è transiente, mittente e destinatario devono essere co-presenti temporalmente, quindi il mittente non deve attendere la ricezione del messaggio ma il messaggio viene perso se il destinatario non è raggiungibile.

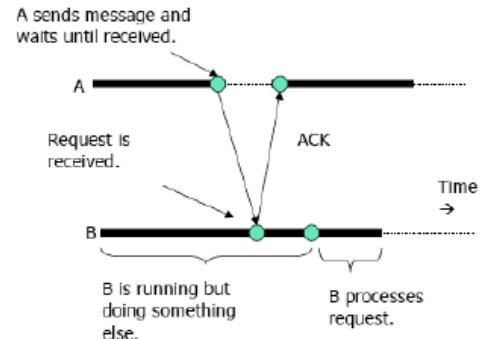


### 4.2.2.2. Comunicazione transiente e sincrona

In base ai diversi momenti rispetto a cui il mittente rimane bloccato, si avrà:

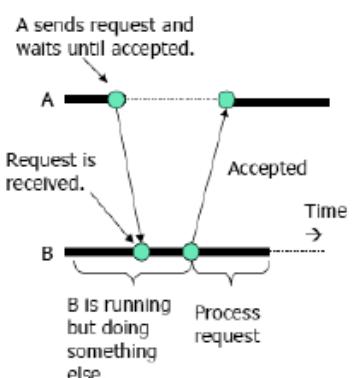
- Comunicazione sincrona basata sulla ricezione di un ACK:** il mittente invia il messaggio, il destinatario lo riceve, non lo elabora subito però nel momento in cui il destinatario lo riceve viene inviato un ACK al mittente. In questo caso quindi il mittente sa che il suo messaggio è stato ricevuto, quello che non sa nel momento in cui riceve questo ACK è se il suo messaggio verrà elaborato.

Il mittente è quindi bloccato fino alla copia (non garantita) del messaggio nello spazio del destinatario.

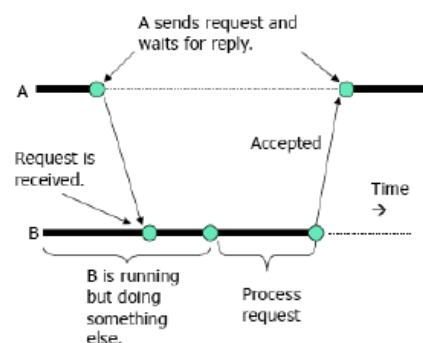


- Comunicazione sincrona basata su delivery:** il destinatario del messaggio invia un ACK nel momento in cui inizia il processamento della richiesta. A differenza dei precedenti in cui l'ACK viene inviato alla ricezione del messaggio ma non è detto che quel messaggio sia poi consegnato dall'applicazione, l'ACK di ricezione viene inviato dopo la consegna del messaggio. Non sappiamo quindi se il messaggio è preso in carico dall'applicazione, ma sappiamo che il messaggio è stato consegnato all'applicazione.

Il mittente è quindi bloccato fino alla consegna del messaggio al destinatario.



- Architettura client server di tipo sincrono:** il mittente rimane bloccato fino a quando non riceve la risposta dal destinatario.



Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

## 4.3. Semantica di comunicazione in un sistema client-server

Problemi più frequenti nella comunicazione client- server:

- Il messaggio di richiesta e/o risposta può essere perso o ritardato, oppure la connessione resettata.
- Il server può subire un crash prima o dopo essere riuscito ad eseguire il servizio.
- Il client può subire un crash

Nello scenario di processamento dei servizi distinguiamo:

1. Semantica *may-be*
2. Semantica *at-least-once*
3. Semantica *at-most-once*
4. Semantica *exactly-once*

### 4.3.1. Meccanismi di base per la realizzazione di semantiche di comunicazione

- Lato client: *riprova - Request Retry* (RR1)
  - Se il client non riceve risposta dal server, continuerà a mandare la richiesta fino a quando non ottiene risposta. Se non la ottiene per diverse volte è sicuro che il server ha crashato.
- Lato server: *filtra i duplicati - Duplicate Filtering* (DF)
  - Se il server riceve più richieste provenienti dallo stesso client da parte dello stesso servizio, il server è in grado di scartare i duplicati. Questo meccanismo è utile per eliminare le richieste di servizio non idempotenti (ovvero che non ha effetti collaterali; eseguire un'operazione di questo tipo più volte non altera lo stato di servizio).
- Lato server: *ritrasmetti le risposte - Result Retransmit* (RR2)
  - Il server memorizza il risultato della computazione effettuata in modo tale da poterla ritrasmettere successivamente senza doverla ricalcolare in caso riceva una richiesta duplicata. È un meccanismo necessario se l'operazione eseguita dal server non è idempotente.

### 4.3.2. Tipi di semantiche

#### 4.3.2.1. Semantica may-be

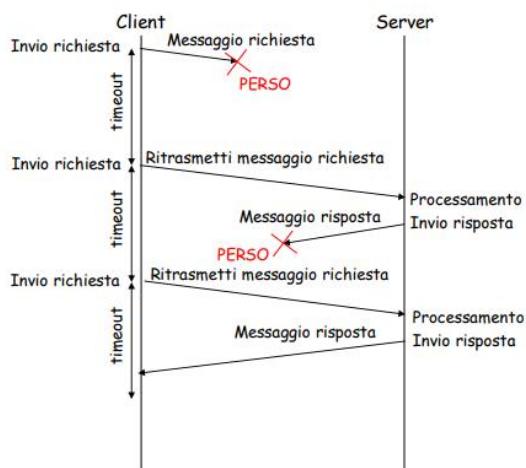
- Il servizio lato server può avvenire o meno. Dato che il server può subire un crash durante il processamento della richiesta, non si attua nessun meccanismo per garantire l'affidabilità della comunicazione. Il client invia la richiesta, se riceve la risposta bene altrimenti fa niente.
- Ciò che si ha nel protocollo best effort UDP.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.



#### 4.3.2.2. Semantica at-least-once

- Servizio eseguito almeno una volta (magari più volte a cause dei duplicati dovuti alle ritrasmissioni). Per cui viene impiegato il primo meccanismo (ritrasmissione della richiesta).
- Quando arriva la risposta il client non sa quante volte sia stata processata dal server, non ne conosce lo stato.
- Adatta a servizi idempotenti, server stateless (non deve ricordarsi nulla, se avesse uno stato servirebbe replicarlo e renderlo consistente). Il server stateless è più scalabile e di più facile gestione in caso di tolleranza ai guasti (non bisogna memorizzare lo stato in sistemi di memorizzazione persistenti).

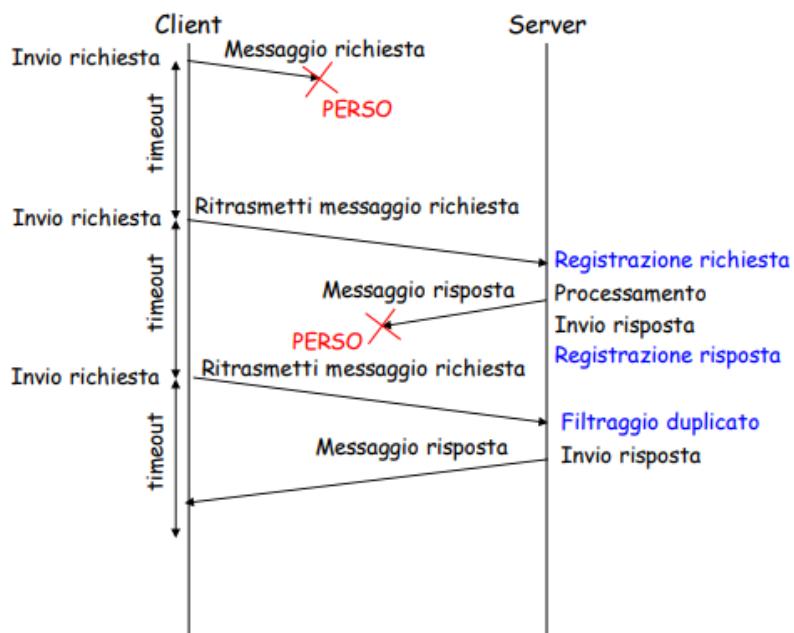


#### 4.3.2.3. Semantica at-most-once

- Dal lato server, per implementare tale semantica servono tutti e tre i meccanismi di base DF, RR1 e RR2.
- Il server ha bisogno di identificare richieste duplicate e restituire tramite RR2 una risposta calcolata in precedenza.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

- La richiesta duplicata viene identificata con un meccanismo per cui il client inserisce nella sua richiesta un ID univoco. Quando invierà di nuovo la richiesta duplicata, lo farà con lo stesso ID.
- L'id deve essere univoco.
- Per assicurare l'ID univoco si può usare un digest che tenga conto di informazioni in modo da evitare che 2 client differenti generino una richiesta avente lo stesso ID.



Problemi:

- Il server ha una richiesta duplicata ma non ha la risposta precedentemente computata da inviare nei seguenti due scenari:
  - Il server riceve una richiesta ma è sovraccarico.
  - Lo stesso vale se il client ha un time-out di ritrasmissione troppo breve rispetto al tempo di processamento del server.

#### 4.3.2.4. Semantica exactly once

- Servizio eseguito esattamente una volta. Per cui offre maggiori garanzie.
- Difficile da attuare in sistemi distribuiti e si complica soprattutto su larga scala in cui è presente asincronia.
- Richiede un accordo completo sull'interazione tra client e server.
- Il servizio è eseguito una sola volta oppure non è eseguito: semantica tutto o niente
- Se va tutto bene: il servizio viene eseguito una sola volta, riconoscendo i duplicati
- Se qualcosa va male: client o server sanno se il servizio è stato eseguito (una sola volta - tutto) o se non è stato eseguito (nessuna volta - niente)
- Semantica con conoscenza concorde dello stato dell'altro e senza ipotesi sulla durata massima del protocollo di interazione tra client e server.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

- Mancanza di vincoli sulla durata massima: poco praticabile in un sistema reale!

Le semantiche più usate sono at-least e at-most, perché più semplici da implementare ma che offrono più garanzie di una semantica di tipo may-be.

#### 4.3.2.4.1. Meccanismi aggiuntivi exactly-once

- I meccanismi base server-side non sono sufficienti (RR1, DF, RR2).
- C'è bisogno di meccanismi addizionali per tollerare errori lato server:
  - **Transparent server replication:** replicazione trasparente rispetto al client.  
*Esempio:* si vogliono contare le occorrenze di tweet con un certo hashtag per fare una classifica dei tweet più popolari. Se il numero di tweet è basso basta una replica del servizio che implementa il contatore. Ma se il numero è alto c'è bisogno di più repliche del servizio contatore (che è un servizio *con stato*). Per replicare il servizio bisogna partizionare lo stato tra le diverse repliche. Quindi, se ho 10 repliche e voglio tener traccia di 100 hashtag, distribuisco uniformemente 10 hashtag su ciascuna replica. C'è poi bisogno di un sofisticato meccanismo di distribuzione richieste che, in base all'hashtag, distribuisca i vari tweet sulle repliche. Tutto ciò deve avvenire in modo trasparente rispetto al client che non deve accorgersi della replicazione.
  - **Write-Ahead Logging (WAL):** i cambiamenti nel sistema devono essere resi effettivi solo dopo che sono stati registrati.  
Prima scrivo (write ahead) ed il log deve essere registrato su un dispositivo di storage persistente; in questo caso se il server crasha il log può essere recuperato dalla persistenza.
  - **Recovery:** meccanismo di ripresa da qualunque stato di fallimento precedente.  
Per far sì che il server che ha subito fallimento, possa recuperare correttamente il proprio stato e ricominciare da un punto sicuro. Due framework che li implementano sono distributed snapshot e state checkpointing.

### 4.4. Programmazione di applicazioni di rete

#### 4.4.1. Programmazione di rete esplicita

- Usata nella maggior parte delle applicazioni di rete.
- Vincolo: distribuzione dei componenti non trasparente.
- Gran parte del peso dello sviluppo è sulle spalle del programmatore.
- Come innalzare livello di astrazione della programmazione distribuita: viene introdotto un middleware di comunicazione tra sistema operativo e applicazioni, che permette di svincolare parte del peso dalle spalle del programmatore (che può così occuparsi di aspetti funzionali), e di nascondere la complessità degli strati sottostanti

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

#### 4.4.2. Programmazione di rete implicita

Il middleware offre strumenti di comunicazione e si occupa della gestione di quest'ultima (si occupa di errori che possono avvenire durante la comunicazione con chiamata a procedura remota/invocazione di metodo remoto). Gestisce anche l'eterogeneità dei dati scambiati.

- **Marshaling/Unmarshaling** dei parametri, cioè dati assemblati/disassemblati in modo opportuno per trasmissione di messaggi
- Java RMI usata serializzazione dei parametri: dati strutturati come flusso di byte, vengono serializzati

Viene usata la **chiamata a procedura remota RPC**:

- La sua trasposizione in un linguaggio di programmazione a oggetti è invocazione del metodo remoto (Java RMI)
- Entrambi ci offrono diversi gradi di trasparenza distribuzione, e anche supporto di diverse tipologie di comunicazione --> comunicazione supportata è di tipo sincrono e transiente (nel caso di Go supportata anche asincrona transiente).
- Nel caso di RPC e RMI comunicazione sempre transiente

#### 4.4.3. Gestione dell'eterogeneità nella rappresentazione dei dati

Client e server possono talvolta utilizzare una diversa codifica dei dati (in termini di codifica dei caratteri, rappresentazione numeri interi e virgola mobile, ...).

Metodi di gestione:

- Inserire codifica all'interno del messaggio stesso dentro l'header in un campo apposito (li sarà riportata la codifica utilizzata). Il mittente che invia il messaggio converte i dati nel formato che il destinatario si aspetta (conversione di formato).
  - **Vantaggio:** La conversione avviene subito, e ciò garantisce delle prestazioni di conversione elevate
  - **Svantaggio:** Il mittente deve avere conoscenza completa di tutte le funzioni di conversione, quindi ogni utente deve conoscere N(N-1) funzioni di conversione.
- Esiste un formato di dati concordato tra chi invia e chi riceve, quindi chi invia trasforma la codifica da proprietaria a comune, chi riceve invece effettua la decodifica da comune a proprietaria. Questo è il metodo utilizzato in RPC.
  - **Vantaggio:** basta conoscere un basso numero di funzioni di conversione, ovvero 2N (conversioni da proprietario a comune e viceversa).
  - **Svantaggio:** prestazioni più basse nella conversione, in quanto c'è una doppia conversione.
- Si utilizza un pattern

##### 4.4.3.1. Pattern di gestione dell'eterogeneità

- **Proxy:** è un componente che viene aggiunto sia dal lato client che server, che supporta trasparenza all'accesso e alla locazione. Il Proxy è locale nello spazio di indirizzamento, e crea una sorta di replica lato

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

client o lato server dell'altro endpoint, quindi sul client ci sarà un proxy che simula le funzioni del server, e viceversa sul server.

- **Broker:** si tratta di un intermediario che si interpone tra mittente e destinatario; esso ha lo scopo di separare ed incapsulare dettagli relativi alla comunicazione poiché è a conoscenza della tecnica di codifica di entrambi. Permette di indentificare verso chi deve essere inoltrata la richiesta.

#### 4.4.3.2. Binding

Effettuare binding tra client e server significa agganciare client e server fornendo un servizio di procedura o metodo remoto. Il collegamento fra client e server può essere:

- **Statico:** più semplice da realizzare in quanto l'indirizzo del server sul quale viene effettuata la procedura remota è cablata all'interno del codice del client; è quindi semplice da realizzare e non aggiunge overhead visto che sa subito quale server contattare. Manca trasparenza rispetto alla migrazione: se viene spostato un server da un nodo ad un altro, l'indirizzo indicato all'interno del codice non è più valido. Inoltre sgode di Mancanza di flessibilità.
- **Dinamico:** il collegamento effettivo tra client e componente che effettua il servizio avviene solo al momento dell'esecuzione. Il collegamento avviene con un'entità intermedia, il che garantisce maggiori trasparenza e flessibilità. Di contro, comporta un alto overhead. Ogni chiamata richiede un collegamento dinamico per cui spesso, dopo aver ottenuto il primo binding, lo si riusa come se fosse statico. Il binding può così avvenire meno frequentemente delle chiamate. In genere si usa lo stesso binding per più chiamate allo stesso server. Si distinguono due fasi:
  - **Naming:** Fase statica che precede l'esecuzione. Il client specifica a chi vuole essere connesso con un nome unico che identifichi il servizio; successivamente si associano nomi unici alle operazioni o alle interfacce astratte e si attua il binding con l'interfaccia specifica di servizio.
  - **Addressing:** Fase dinamica che avviene durante l'esecuzione. Il client deve essere realmente collegato al server che fornisce il servizio al momento dell'invocazione. Si cercano eventuali server pronti per il servizio tramite addressing implicito o esplicito.
    - **Implicito:** C'è un name server che registra i servizi e agisce su opportune tabelle di binding, fornendo servizi di ricerca, registrazione, aggiornamento, rimozione
    - **Esplicito:** Broadcast (invio a tutti) o multicast (invio a un sottoinsieme di repliche) da parte del client, attendendo solo la prima risposta.

Problemi:

1. Come viene gestita la rappresentazione dei dati in modo da cercare di avere trasparenza rispetto a eterogeneità
2. Come vengono gestiti gli errori che possono avvenire durante chiamata a procedura remota
3. Come viene gestita il collegamento al server che esegue la procedura remota

#### 4.4.4. Remote Procedure Call (RPC)

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

È il primo meccanismo che ha permesso di realizzare middleware di comunicazione che astrasse da programmazione di base di rete, permettendo la definizione di applicazioni distribuite secondo i pattern architetturali esaminati.

**Idea** (proposta da Birrel e Nelson nel 1984): utilizzare il modello di interazione di tipo client/server (client invoca il server, e server fornisce servizio e una risposta) con la stessa semantica di una chiamata di procedura.

1. Un processo in esecuzione sulla macchina A (nodo del sistema, svolge ruolo di client) invoca una procedura che viene eseguita sulla macchina B (svolge ruolo di server)
2. Il processo (comunicazione sincrona) chiamante su A viene sospeso
3. Viene eseguita la procedura chiamata su B
4. Parametri di input e di output della procedura chiamata vengono inviati tramite messaggi scambiati tra client e server e viceversa
5. Lo scambio di messaggi è trasparente al programmatore, poiché gestito da 2 proxy lato client e lato server chiamati **stub**.

#### 4.4.4.1. Chiamata di procedura locale (per esempio in un main)

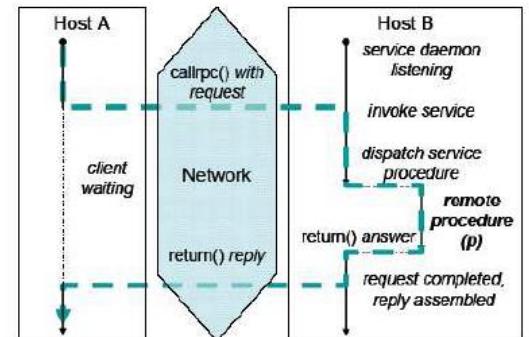
Sospesa esecuzione della procedura chiamante



La procedura chiamante inserisce nello stack i parametri di input e l'indirizzo di ritorno (viene quindi avviata la procedura chiamata)



Al termine, la procedura chiamata restituisce il controllo alla procedura chiamante

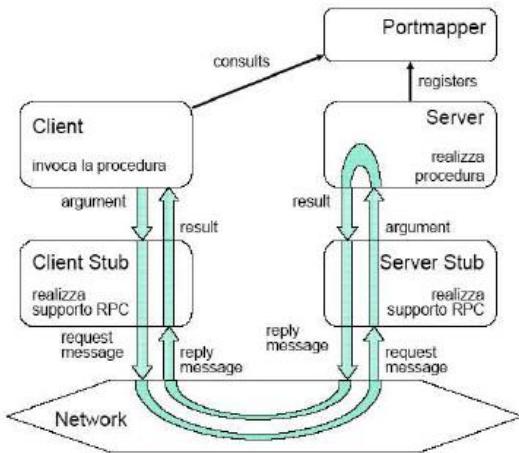


#### 4.4.4.2. Chiamata di procedura remota

Problema nel realizzare procedura remota è come realizzare tutto questo nel caso in cui il chiamante (procedura chiamante) e chiamata siano in esecuzione su 2 macchine differenti, poiché questo provocherebbe diversi problemi:

1. Diversi ambienti di esecuzione
2. Eterogeneità e passaggio dei parametri non può avvenire tramite lo stack, poiché ci sono 2 diversi spazi di indirizzamento (quello del nodo su cui avviene chiamata procedura e quello su cui avviene esecuzione effettiva della procedura)

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.



Nel momento in cui lato client viene effettuata chiamata a procedura remota, essa viene presa in gestione da **client stub**, il quale si occupa di:

- identificare il server
- gestire parametri
- richiedere supporto run-time
  - inviare richiesta

↓

C'è un **demone** in attesa, che va a ricevere la richiesta di esecuzione della procedura; tramite procedura di *dispatching* va a determinare l'esatta procedura remota di cui è stata chiesta invocazione, e attiverà la procedura localmente

↓

Il proxy lato server (**server stub**) ottiene la richiesta, la assembla in un messaggio di risposta gestendo i parametri, il quale verrà poi inviato al nodo chiamante (**client stub**).

**Come trasformare meccanismo di chiamata a procedura locale in uno a chiamata a procedura remota?**  
Questa trasformazione viene attuata nel meccanismo di RPC tramite introduzione (usando il pattern **proxy**) di 2 componenti che realizzano il middleware di comunicazione.

Si avranno un proxy lato client (**client stub**) e un proxy lato server (**server stub**).

- Client stub → svolge sulla macchina lato client il ruolo del server
- Server stub → svolge sulla macchina del server il ruolo di client

#### 4.4.4.3. Passi di RPC

1. Lato client, la procedura chiamata invoca il client stub tramite una normale chiamata di procedura locale, al quale verrà passata la procedura chiamata con i parametri di input. A seconda della specifica implementazione c'è differenza nella trasparenza all'accesso.
2. Il **client stub** gestisce tutti gli aspetti che il middleware di comunicazione vuole nascondere rispetto al client, quindi identifica qual è il server (**binding** con il server), gestisce eterogeneità nella rappresentazione dei dati (**gestione parametri** input e output) e gestisce il messaggio di richiesta inviato al lato server e messaggio di risposta ricevuto da esso.
- 2.1. Il client stub identifica la procedura richiesta e costruisce un messaggio all'interno del quale identifica procedura richiesta (mette identificativo richiesta) e inserisce i parametri di input, effettuando il marshaling dei parametri di input.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

- **Marshaling dei parametri:** operazione di impacchettare in un messaggio i parametri, convertendoli da formato locale a formato comune.
3. Il client stub invia il messaggio di richiesta che verrà ricevuto dal **server stub**, che si occupa di gestire i parametri di input in essa contenuti, e di invocare la procedura. Il server stub spacchetta quindi il messaggio, prelevando i parametri e convertendoli in formato locale (unmarshaling), e invoca il server come una procedura locale
    - **Unmarshaling dei parametri:** spacchettamento del messaggio ricevuto, e conversione dei parametri di input della procedura contenuti nel messaggio da formato comune a formato locale.
  4. Il server esegue il lavoro e restituisce il risultato al server stub
  5. Il server stub lo impacchetta in un messaggio (marshaling dei parametri), il quale verrà ricevuto dal client stub
  6. Il client stub spacchetta il messaggio di risposta (unmarshaling dei parametri), gestisce eterogeneità nella rappresentazione dei dati, e infine restituisce il risultato al client (procedura chiamante).

Tutto questo meccanismo (scambio messaggi) più le conversioni di dati che avvengono al di sotto del client e server, sono trasparenti rispetto a client e al server, e gli stub sono prodotti in modo automatico  
➔ il client e il server sono gli unici componenti dello schema in figura che devono essere progettati dallo sviluppatore dell'applicazione distribuita.

Un ulteriore componente, il **portmapper**, ha il ruolo di *service registry*, cioè effettua il binding tra client e server.

#### 4.4.4.4. Problemi da risolvere per realizzare meccanismo di chiamata a procedura remota:

1. Gestire eterogeneità nella rappresentazione dei dati
2. Come realizzare passaggio parametri per riferimento
  - Procedura chiamante e chiamata sono in esecuzione su 2 nodi del sistema che hanno un diverso spazio di indirizzamento, quindi non posso passare indirizzo di memoria, che non avrebbe significato rispetto al nodo che riceve chiamata a procedura
3. Quale semantica di comunicazione adottare in caso di errori
4. Come viene realizzato il binding al server che esegue procedura chiamata

##### 4.4.4.4.1. Rappresentazione dei dati

Il middleware RPC fornisce un supporto automatizzato:

Il codice per il marshaling/unmarshaling viene generato automaticamente dal middleware di comunicazione e diviene parte degli stub, tramite:

- **Interface Definition Language (IDL):** Ci permette di descrivere e rappresentare la procedura indipendente dal linguaggio e dalla specifica piattaforma
- **Un formato comune di rappresentazione dei dati** usato per la comunicazione

##### IDL per RPC

- Linguaggio per la definizione delle interfacce (Interface Definition Language - IDL)

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

- Permette la descrizione di operazioni remote, la specifica del servizio (detta **signature**, cioè specificare identificativo del servizio) e la generazione automatica degli stub
- Deve consentire
  - Identificazione non ambigua del servizio (procedura remota chiamata)
    - Uso di nome astratto del servizio, spesso prevendendo versioni diverse del servizio) la procedura remota chiamata (il servizio)
  - Definizione astratta dei dati da trasmettere in input e output
    - Uso di un linguaggio astratto di definizione dei dati (operazioni e parametri dell'interfaccia)
- Supporta lo sviluppo dell'applicazione
  - L'interfaccia specificata dal programmatore in IDL può essere usata per generare automaticamente gli stub

#### 4.4.4.2. Tipi di passaggio dei parametri

- **Passaggio per valore (call by value)**
  - I dati sono copiati in uno stack locale del chiamante
  - Il chiamato agisce su tali dati e le modifiche non influenzano il chiamante
- **Passaggio per riferimento (call by reference)**
  - L'indirizzo (puntatore) dei dati è copiato in uno stack locale del chiamante
  - Il chiamato legge l'indirizzo e agisce direttamente sui dati del chiamante, sapendo dove trovarli
- **Passaggio per copia/ripristino (call by copy/restore)**
  - Caso speciale di passaggio per riferimento: i dati sono copiati in uno stack del chiamante; il chiamato agisce su quello stack, e quando la procedura chiamata ritorna, i dati vengono ricoperti dallo stack alla memoria del chiamante. Si tratta di un caso speciale rispetto al precedente in quanto i dati non vengono toccati direttamente dal chiamato che agisce sulla copia e non su un'indirizzo; è il chiamante che alla fine se li aggiorna da solo con quelli che trova elaborati sullo stack che aveva messo a disposizione.

#### Problema per il passaggio dei parametri

Il riferimento è un indirizzo di memoria, il quale è valido solo nel contesto locale in cui è usato, cioè in quello specifico nodo / SO.

Per risolvere il problema di passare un parametro per riferimento in RPC si simula il passaggio dei parametri per riferimento usando il meccanismo copia-ripristino del passaggio dei parametri:

- Il client stub copia la struttura dati puntata all'interno del messaggio e invia il messaggio al server stub.
- Il server stub riceve il messaggio con una copia dell'area di memoria, quindi agisce sulla copia, e usa lo spazio di indirizzi del nodo ricevente.
- Se viene effettuata una modifica sui dati passati, la modifica verrà poi inserita dal server stub nel messaggio di risposta, e lato client verrà riportata dal client stub nella struttura dati originale utilizzata da client.
- Occorre conoscere la dimensione dei dati da copiare.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

#### 4.4.4.4.3. Comunicazione

##### RPC sincrona

In modalità **sincrona**, la chiamata RPC provoca il blocco del client (client rimane in attesa finché non riceve risultato della procedura chiamata), quindi client effettua chiamata a procedura remota, che viene presa in carico da client stub che invia richiesta lato server. La richiesta viene ricevuta da server stub che fa unmarshaling dei parametri, chiama la procedura locale che fornisce il risultato che viene impacchettato nel messaggio di risposta e inviato a client stub. Il client stub spacchetta il messaggio, fa unmarshaling e lo inoltra al client. **Per tutto questo tempo il client rimane bloccato in attesa del risultato.**

- Tutto questo non è necessario se il server non deve restituire nessun risultato

Per il suddetto motivo, alcuni middleware RPC supportano RPC **asincrona**.

##### RPC asincrona

Una volta che il client ha inviato il messaggio di richiesta, rimane in attesa solo di un ACK, da parte del middleware di comunicazione, che la sua richiesta di chiamata a procedura è stata presa in carico, poi riprende l'esecuzione senza restare bloccato in attesa del messaggio. **Il client quindi può dedicarsi ad altri task una volta effettuata la chiamata ed aver ricevuto dal server un ACK che la chiamata di procedura è stata ricevuta ed avviata.**

- Se il client riprende l'esecuzione senza aspettare l'ACK, la RPC asincrona è detta **one-way**
- Se la RPC produce un risultato, si può spezzare l'operazione in due (**RPC sincrona differita**):
  - Una prima RPC da client a server per avviare l'operazione di richiesta a procedura remota
  - Una seconda RPC da server a client per restituire il risultato

#### 4.4.4.4.4. RPC e TRASPARENZA

Domande: Il meccanismo di chiamata a procedura remota è veramente trasparente? Rispetto ai diversi gradi di trasparenza, quali di questi supporta?

Risposta: Il meccanismo di chiamata a procedura remota non è completamente trasparente. Nel caso di *Sun RPC*, non rispetta neanche la trasparenza all'accesso, poiché lo sviluppatore deve essere consapevole del fatto che sta invocando chiamata a procedura remota anziché locale, e deve aggiungere un ulteriore parametro di input rispetto alla procedura locale che altrimenti avrebbe chiamato; tale parametro sarà il gestore del protocollo di trasporto lato client.

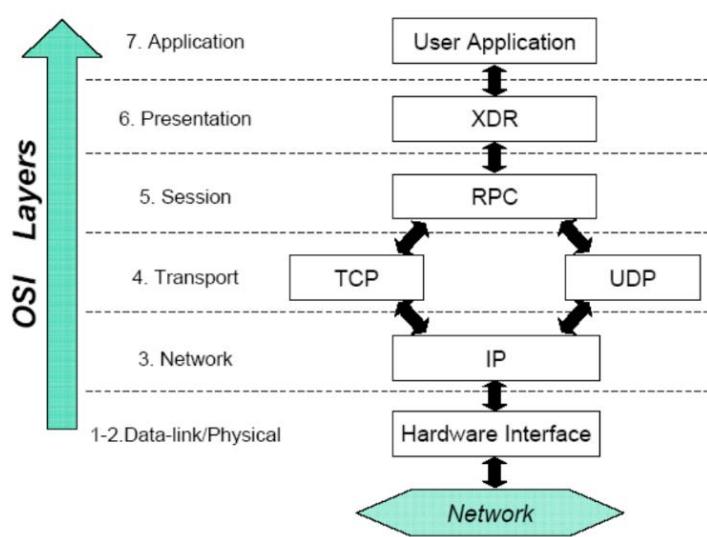
Oltre a non essere completamente trasparente, ha anche un impatto in termini di prestazioni:

- Chiamata a procedura locale → in termini di tempi di prestazione, costa sull'ordine di 10 cicli, quindi qualche nanosecondo
- Chiamata a procedura remota → anche se non esegue operazioni, ha un costo di 5 ordini di grandezza (100 K) superiori rispetto a quella locale → impatto non trascurabile sulle prestazioni dell'applicazione.
  - Questo perché anche nel caso in cui eseguo chiamata a procedura remota in una LAN, ci richiede tra un decimo di millisecondo e 1 secondo, poiché devo considerare tutto overhead隐含的 nel meccanismo di chiamata a procedura remota:

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

- lato client e server si ha context switch per passare da user space a kernel space
- copie che vengono effettuate
- comunicazione tra i processi (ha un suo overhead relativo a rete sottostante)

#### 4.4.4.5. IMPLEMENTAZIONE DI RPC: Sun RPC



- È un esempio di RPC di prima generazione
- Un' implementazione e middleware RPC forniti da Sun Microsystems: **Open Network Computing (ONC)** RPC, noto anche come **Sun RPC**
- Implementazione di base e ampiamente utilizzata
- ONC è una suite di prodotti che include:
- **eXternal Data Representation (XDR)**: IDL utilizzato da Sun RPC per gestire l'eterogeneità nella rappresentazione dei dati
- **Remote Procedure Call**

**GENerator (RPCGEN)**: programma per generare automaticamente lo stub del client e lo stub del server

- **Port mapper**: servizio di binding per collegare il client al server
- **Network File System (NFS)**: file system distribuito, ossia realizzazione in chiave distribuita di un file system con API che conosciamo per l'utilizzo di un file system

#### 4.4.4.6. Sun RPC , modello OSI e esempi.

Lo stack RPC si pone fra il livello 5/6 dell'ISO OSI model.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

## Definizione del programma RPC (Da punto di vista sviluppatore applicazione distribuita):

1. Scrivere file con estensione .x utilizzando il linguaggio XDR; tale file conterrà 2 parti descrittive:
  - Definizioni di programmi RPC: specifiche del protocollo RPC per le procedure (servizi) offerte, ovvero l'identificazione delle procedure ed il tipo di parametri
  - Definizioni XDR: definizioni dei tipi di dati dei parametri
    - Presenti solo se il tipo di dato usato per parametri di input e output non è già noto in XDR

### Esempio di chiamata a procedura remota:

Calcolare quadrato di un numero intero passato come input.

```
struct square_in {          /* input (argument) */  
    long arg1;  
};  
struct square_out {         /* output (result) */  
    long res1;  
};  
program SQUARE_PROG {  
    version SQUARE_VERS {  
        square_out  SQUAREPROC(square_in) = 1; /* procedure number = 1 */  
        } = 1;                      /* version number */  
    } = 0x31230000;                /* program number */
```

**Esempio: square.x**

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

La tripla numero procedura, numero versione, numero programma permette di indentificare la procedura.

Definizione della procedura remota **SQUAREPROC**

- Ogni procedura ha un solo parametro d'ingresso e un solo parametro d'uscita
- Gli identificatori (i nomi) delle procedure usano lettere maiuscole
- Ogni procedura è associata ad un numero di procedura unico all'interno di un programma (nell'esempio = 1)

## Implementazione del programma RPC

- Il programmatore deve sviluppare:
  - il **programma client**: implementazione del *main()* e della logica necessaria per reperimento e binding del servizio/i remoto/i (esempio: *square\_client.c*)
  - il **programma server**: implementazione di tutte le procedure (servizi) (esempio: *square\_server.c*)
- **Attenzione:** il programmatore **non** realizza il *main()* lato server...
  - Chi invoca la procedura remota (lato server)?

Esempio *square.x*: effettua il quadrato di un numero intero.

**square: procedura convenzionale/locale**

**square: procedura remota**

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

```
#include <stdio.h>
#include <stdlib.h>

struct square_in { /* input (argument) */
    long arg;
};

struct square_out { /* output (result) */
    long res;
};

typedef struct square_in square_in;
typedef struct square_out square_out;

square_out *squareproc(square_in *inp) {
    static square_out out;

    out.res = inp->arg * inp->arg;
    return(&out);
}

int main(int argc, char **argv) {
    square_in in;
    square_out *outp;

    if (argc != 2) {
        printf("usage: %s <integer-value>\n", argv[0]);
        exit(1);
    }
    in.arg = atol(argv[1]);

    outp = squareproc(&in);
    printf("result: %ld\n", outp->res);
    exit(0);
}
```

### Esempio: square\_local.c

### Lato server:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "square.h" /* generated by rpcgen */
square_out *squareproc_1_svc(square_in *inp, struct svc_req *rqstp) {

    static square_out out;

    out.res1 = inp->arg1 * inp->arg1;
    return(&out);
}
```

### Esempio: server.c

- Parametri di ingresso e uscita passati per riferimento
- Il parametro di uscita punta ad una variabile statica (allocazione globale), per essere presente anche oltre la chiamata della procedura
- Il nome della procedura cambia leggermente (si aggiunge \_ seguito dal numero di versione), tutto in caratteri minuscoli

### Lato client:

- Il client viene lanciato passando hostname remoto e il valore intero e richiede il servizio di square remoto

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "square.h" /* generated by rpcgen */
int main(int argc, char **argv) {
    CLIENT *clnt;
    char *host;
    square_in in;
    square_out *result;

    if (argc != 3) {
        printf("usage: client <hostname> <integer-value>\n");
        exit(1);
    }
    host = argv[1];
    clnt = clnt_create(host, SQUARE_PROG, SQUARE_VERS, "tcp");
    if (clnt == NULL) {
        clnt_pcreateerror(host);
        exit(1);
    }
    in.arg1 = atol(argv[2]);
    if ((result = squareproc_1(&in, clnt)) == NULL) {
        printf("%s", clnt_sperror(clnt, argv[1]));
        exit(1);
    }
    printf("result: %ld\n", result->res1);
    exit(0);
}
```

### Esempio: client.c

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

- `cInt_create()`: crea il **gestore di trasporto client** che gestisce la comunicazione col server (TPC o UDP)
- Il client deve conoscere:
  - nome dell'host su cui è in esecuzione il servizio
  - informazioni per invocare il servizio (programma, versione e nome della procedura)
- Per la chiamata della procedura remota:
  - Cambia il nome della procedura: si aggiunge il carattere “\_” seguito dal numero di versione (nome in caratteri minuscoli)
  - Due parametri di ingresso della procedura chiamata:
    - parametro di ingresso vero e proprio
    - gestore di trasporto del client
- Gestione degli errori che si possono presentare durante la chiamata remota
  - `cInt_pcreateerror()` e `cInt_perror()`

#### 4.4.4.6.1. Passi base sviluppo RPC

1. definire i servizi e i tipi di dato
2. Generare tramite rpcgen gli stub del client e del server e le eventuali funzioni di conversione **xdr**.
3. Il programmatore crea server e client, compila i file sorgente e fa il linking dei file oggetto
4. Vengono pubblicati i servizi

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

5. Tramite il port mapper viene reperito l'endpoint del server per il lato client e viene creato il gestore di trasporto per l'interazione col server

## Caratteristiche di Sun RPC

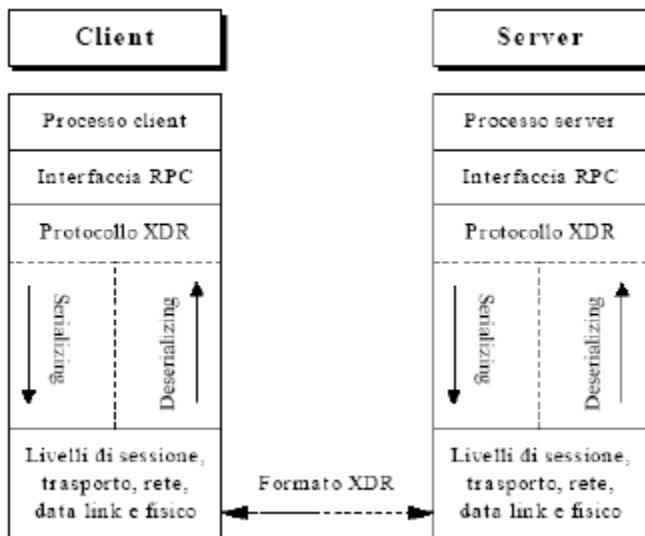
---

- Un programma tipicamente contiene più procedure remote
  - Possibili versioni multiple di ogni procedura
  - Un unico argomento in ingresso ed in uscita per ogni invocazione (passaggio per copia-ripristino)
- Mutua esclusione garantita dal programma (e server): di default no concorrenza lato server
  - Server sequenziale ed una sola invocazione eseguita per volta
  - Per server multithreaded (non su Linux): rpcgen con opzioni -M e -A
- Fino al ritorno della procedura, il client è in attesa sincrona bloccante della risposta
- Semantica at-least-once della comunicazione
  - Ritrasmissione allo scadere di un intervallo di time-out
  - UDP come protocollo di trasporto di default

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

#### 4.4.4.6.2. XDR (External Data Representation)

XDR gestisce tramite un unico formato di rappresentazione l'eterogeneità dei dati. Le funzioni di XDR di conversione sono built in e sono relative a tipi atomici o strutturali predefiniti. Ogni funzione XDR creata da rpcgen puo essere usato per serializzare o deserializzare.



### Definizione del file.x

- Prima parte del file
  - Definizioni XDR delle **costanti**
  - Definizioni XDR dei **tipi di dato** dei parametri di ingresso e uscita per tutti i tipi di dato per i quali non esiste una corrispondente funzione XDR built-in
- Seconda parte del file
  - Definizioni XDR delle **procedure**
- Esempio in square.x: **SQUAREPROC** è la procedura numero 1 della versione 1 del programma numero 0x31230000
- In base alle specifiche di RPC:
  - Il numero di procedura zero è riservato a **NULPROC**
  - Un solo parametro d'ingresso e d'uscita per ogni procedura
  - Gli identificatori di programma, versione e procedura usano tutte lettere maiuscole

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

In sun RPC la procedura deve essere registrata prima di essere invocata (va assegnato l'identificatore RPC: numero di programma, versione, procedura, protocollo di trasporto). Il client deve conoscere il numero di porta su cui il server risponde. Il server RPC registra il programma nella portmap, che è:

- Tabella dinamica dei servizi RPC dell'host
- Ogni riga della portmap contiene la tripla (num programma, versione, protocollo) e la porta

La **tavella di portmap** è gestita da un solo processo per ogni host, chiamato **portmapper**.

**Portmapper (RPC bind):** Il portmapper è in ascolto sulla porta 111; All'avvio il server stub registra sul portmapper le informazioni sui servizi RPC offerti (num programma, versione, protocollo).

Il client stub contatta il portmapper prima di invocare la procedura remota per conoscere la porta corrispondente. Il portmapper registra i servizi offerti e supporta inserimento di un servizio, eliminazione, ricerca porta, lista servizi registrati (rpcinfo –p namehost).

```
>$ rpcinfo -p
   program      vers     proto   port
      100000        4       tcp     111      rpcbind
      100000        4       udp     111      rpcbind
      824377344     1       udp     59528
      824377344     1       tcp     49311
```

## Processo di sviluppo in SUN RPC

Data una specifica di partenza scritta in XDR, per esempio square.x , rpcgen è il comando che produce square.h (header), square\_client.c (client stub), square\_svc.c (server stub), squareXDR.c (routine XDR). Lo sviluppatore scrive client e server stub.

### 4.4.4.7. Seconda generazione di RPC

Con la diffusione dei linguaggi di programmazione ad oggetti arriva la seconda generazione, viene introdotto il supporto per gli oggetti distribuiti. Nello specifico esaminiamo il supporto per Java RMI

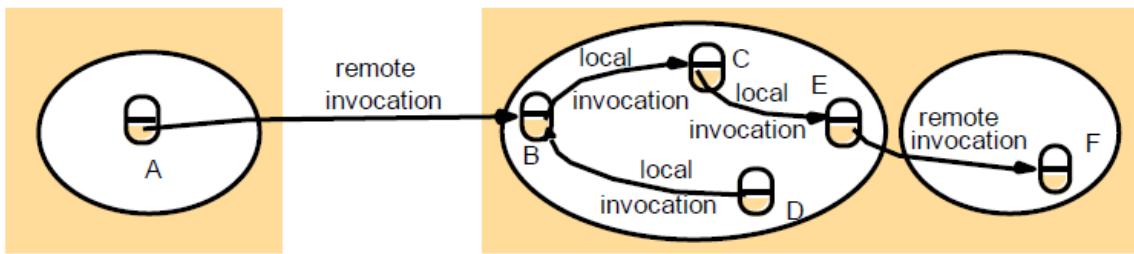
#### 4.4.4.7.1. Java RMI (Remote Method Invocation)

RMI permette di invocare un metodo di un'interfaccia remota su un oggetto remoto. Fornisce strumenti, politiche e meccanismi che permettono a java di invocare un metodo remoto in esecuzione su un host differente. Java RMI mantiene trasparenza all'accesso facendo sì che l'invocazione locale e la remota siano il più possibile simili. Anche in Java RMI non si ha completa trasparenza all'accesso ma risulta migliore di RPC comunque. La trasparenza della distribuzione non

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

è completa neanche per java RMI (niente trasparenza alla replicazione), In Java RMI abbiamo trasparenza alla concorrenza. In nessuna delle due abbiamo trasparenza alla migrazione.

Java RMI ci consente di invocare un metodo remoto. Localmente viene creato un riferimento all'oggetto remoto (attivo su un altro host) che espone il metodo che stiamo invocando. L'oggetto remoto in questione potrà invocare altri oggetti. Le differenze rispetto ad un metodo locale sono affidabilità, semantica della comunicazione, durata.



Il principio di RMI è quello di separare interfaccia (definizione comportamento) dall'oggetto (implementazione comportamento). L'interfaccia remota permette di specificare quali sono i metodi dell'oggetto che possono essere invocati da remoto. Lo stato interno di un oggetto non è distribuito, ma sarà sull'oggetto remoto. Nel caso di RMI utilizziamo il proxy pattern che abbiamo visto che garantisce un certo grado di trasparenza nella gestione di chiamate a procedura remota. Lato client abbiamo lo stub e lato server lo skeleton. Entrambi svolgono ruolo analogo a client e server stub di sun rpc.

#### 4.4.4.7.2. Stub e Skeleton

Quando lato client invochiamo metodo remoto viene effettuato il binding del client con il server remoto, la copia dell'interfaccia del server viene caricata nello spazio del client. La richiesta in arrivo all'oggetto remoto viene trattata dallo skeleton (locale per il server). Abbiamo un unico ambiente di lavoro per via dell'utilizzo di Java; usando serializzazione e deserializzazione (trasformazione di oggetti complessi in seq di byte e decodifica per viceversa) messe a disposizione da java si affronta il problema dell'eterogeneità dei dati.

Stub e Skeleton quindi svolgono ruolo di client e server stub, nascondono a livello applicativo la natura dell'applicazione. Lo stub è un proxy locale che espone in entrambe le parti le interfacce dell'altro.

#### Serializzazione e Deserializzazione vs Marshaling e Unmarshaling

Lo stub lato client e lo skeleton andranno a utilizzare serializzazione e deserializzazione per gestire correttamente i parametri di input e di output per il metodo remoto.

Nel caso di sun rpc usiamo marshaling dei parametri che converte i dati dell'oggetto da formato locale a comune usando linguaggio xdr. Nel caso di java rmi usiamo serializzazione che converte l'oggetto in un flusso di byte, basandosi quindi sul codice di base.

Importante dal punto di vista della comunicazione è che il proxy usi schemi di compattazione dell'informazione ottimizzati in modo da ridurre banda occupata, latenza di comunicazione e occupazione di memoria.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

#### 4.4.4.7.2.1. Interazione tra stub e skeleton

1 il client ottiene un'istanza dello stub, ottenendo una copia dell'interfaccia remota tramite un componente intermedio (in sun rpc era il port mapper, in rmi lo ottiene tramite rmi registry)

2 il client invoca i metodi sullo stub (sintassi invocaz remota identica alla locale)

3 lo stub effettua la serializzazione delle informazioni per l'invocazione del metodo remoto (id e parametri) e le invia allo skeleton incapsulate in un messaggio.

4 skeleton riceve messaggio, deserializza dati, invoca chiamata sull'oggetto che implementa il server (dispatching), ottiene valore ritorno lo serializza e lo invia allo stub in un messaggio

5 Lo stub effettua la deserializzazione del valore di ritorno e restituisce il risultato al client

RMI registry quindi consente al server di registrare metodi remoti che vengono esposti e al client consente di recuperarne lo stub. URL RMI inizia con rmi e contiene hostname opzionale e numero porta opzionale e nome oggetto remoto. Non c'è trasparenza all'ubicazione (il client deve conoscere l'hostname per sapere dove andare) e no gestione sicurezza

## Java RMI: passi essenziali

---

- Realizzare i componenti remoti lato server
  - Definizione del comportamento: interfaccia che
    - E' dichiarata **public** per poter essere utilizzata da altre JVM
    - Estende **java.rmi.Remote**
    - Ogni metodo remoto deve sollevare l'eccezione **java.rmi.RemoteException** per gestire anomalie derivanti da errori di rete o indisponibilità del server (può anche sollevare eccezioni specifiche dell'applicazione)
  - Implementazione del comportamento: classe che
    - Implementa l'interfaccia definita
    - Estende **java.rmi.UnicastRemoteObject**
    - unicast perché viene definito il riferimento ad un solo oggetto remoto (singolo indirizzo IP e porta: no replicazione dell'oggetto remoto)
  - Codice del server:
    - Istanzia l'oggetto remoto
    - Registra l'oggetto presso l'RMI registry, invocando **bind/rebind** (in **java.rmi.Naming**); **rebind** sostituisce un'eventuale associazione già esistente
- Realizzare i componenti locali lato client
  - Ottiene il riferimento all'oggetto remoto, invocando il metodo **lookup** sul registry
  - Lo assegna ad una variabile che ha l'interfaccia remota come tipo

Dopo aver sviluppato il codice, occorre:

1. Compilare le classi

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

2. Attivare l'RMI registry (comando rmiregistry), che viene lanciato in un processo separato (rispetto a quello in cui è eseguito il server RMI) ed ha struttura e comportamento standard
  - o In alternativa, si può creare all'interno del codice server un proprio registry locale tramite il metodo createRegistry (in java.rmi.registry)
  - o Registry locale al server per motivi di sicurezza
3. Avviare il server
4. Avviare il client

#### 4.4.4.7.3. Interfaccia

Bisogna realizzare l'interfaccia in modo tale che estenda l'interfaccia Remote. I metodi così realizzati devono quindi gestire le eccezioni lanciando RemoteException per fallimenti di comunicazioni; L'invocazione dei metodi remoti non è completamente trasparente. Per quanto riguarda il passaggio dei parametri non abbiamo la limitazione di javaRPC (1 parametro di ingresso). Il passaggio dei parametri avviene per valore (tipi primitivi o oggetti che implementano l'interfaccia java.io.Serializable, o anche per riferimento se sono oggetti Remote).

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface EchoInterface
    extends Remote{
    String getEcho(String echo)
        throws RemoteException;
}
```

##### Esempio ECHO: server

Esiste una classe che implementa il server, la quale deve estendere UnicastRemoteObject trattandosi di un oggetto remoto. Abbiamo il costruttore che invoca super() per inizializzare il tutto. Il metodo getEcho() restituisce la stringa ricevuta in ingresso (si noti che essendo metodo remoto lancia eccezioni remote). Abbiamo poi il main, che crea l'istanza dell'oggetto server. Il servizio viene esposto tramite l'RMI registry locale al server. Verrà a quel punto usato il metodo bind/rebind affinché il server registri i metodi. Il client utilizza un nome logico per reperire l'oggetto remoto.

##### Esempio ECHO: client

I servizi sono acceduti tramite l'interfaccia ottenuta tramite lookup all'RMI registry. Viene reperito un riferimento remoto, ovvero un'istanza di stub dell'oggetto remoto. Quando il metodo remoto verrà invocato sarà una chiamata sincrona bloccante con parametri specificati dall'interfaccia.

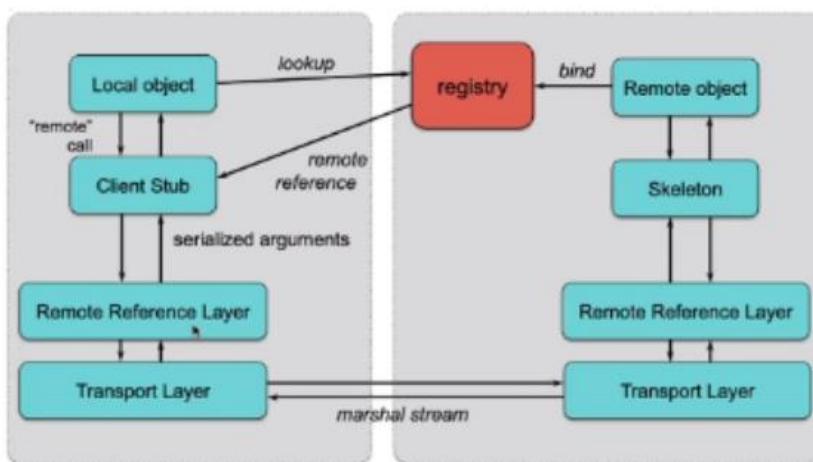
[Il codice di ECHO e SUN RPC sono sul sito del corso]

Una volta compilati client e server si può eseguire il server avviando prima l'RMI registry.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

#### 4.4.4.7.4. Java RMI: passaggio dei parametri

Nel caso di invocazione a metodo locale i tipi primitivi vengono passati per valore e tutti gli oggetti vengono passati per riferimento. In RMI il passaggio di parametri avviene per valore solo se si tratta di dati primitivi o oggetti serializable (in generale, per oggetti la cui locazione non è rilevante per lo stato). Viene serializzata l'istanza dell'oggetto e deserializzata all'arrivo così da crearne una copia locale. Il passaggio dei parametri può anche avvenire per riferimento qualora si tratta di un oggetto remoto (caso in cui la sua funzione è strettamente collegata alla sua località). Lo stab identifica l'oggetto remoto tramite un ID univoco rispetto a dove esso si trova in esecuzione. Quindi l'architettura java RMI può essere descritta così:



#### Concorrenza su oggetti remoti in Java RMI

Nel momento in cui il server mette a disposizione dei metodi sull'oggetto remoto questi possono essere invocati da molteplici client in maniera concorrente. La specifica di java RMI ci dice che l'implementazione di un metodo remoto deve essere thread-safe. Per rendere quindi i metodi sincronizzati si utilizza definirlo come synchronized.

#### Distributed garbage collection in Java RMI

È inutile tenere memoria allocata per oggetti che non vengono più referenziati da client. Risulta quindi utile eseguire azioni di garbage collection, il quale deve quindi sapere quanti client stub stanno utilizzando l'oggetto in questione.

Come funziona la garbage collection in una JVM: Java mantiene in locale un contatore di riferimenti all'oggetto e ne schedula la cancellazione nel momento in cui questo contatore arriva a 0 (oggetto non più utilizzato). In Java RMI l'idea è simile ma deve essere più tollerante a guasti che si possono verificare (per esempio di connessione). L'idea infatti è di far basare su lease, il server delega l'operazione di mantenere i riferimenti attivi al client in modo da decentralizzare il carico sul server. RMI mette a disposizione le operazioni di dirty e clean per il garbage collection. Periodicamente la JVM del client che sta usando oggetto remoto manda call dirty al server nel momento in cui l'oggetto è in utilizzo sul client. Questa dirty call viene rinfrescata entro un timeout (lease assegnato da client al server). Compito relegato ai client di dire che l'oggetto è ancora in uso per loro.

Nel momento in cui la JVM locale al client non usa più oggetto remoto usa operazione di cleean per dire che non ci sono più riferimenti locali attivi per l'oggetto remoto. Come avviene la cancellazione dell'oggetto: se la JVM non riceve messaggi di dirty o clean prima che scada il timer del lease allora l'oggetto viene cancellato in quanto non ci sono più client stub che lo usano.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

### Esempio: Compute Engine

Si vuole realizzare applicazione distribuita che permetta ad un client di invocare un task su un server remoto. Si vuole che i task siano quindi definiti da client ed eseguiti su server remoto. Il compute engine deve consentire al server di ricevere un qualunque task (che deve implementare l'interfaccia serializable). Il server esegue così localmente il task ricevuto dal client e restituisce il risultato al client. Esempio interessante di offloading di un task computazionalmente oneroso.

### Riflessioni:

- Quindi come è possibile fornire trasparenza alla replicazione non fornita dal middleware in Java RMI o Sun RPC? Basta avere un intermediario (load balancer) fra i server replicati ed il client. Lo sviluppo di questo intermediario è lasciato allo sviluppatore ovviamente. Il client si collega al proxy, il proxy sceglie (secondo politiche variabili) uno fra i molteplici server a cui inoltrare la richiesta del client.
- La chiamata all'RMI registry è sincrona e bloccante. Se si volesse realizzare una chiamata asincrona, bisogna realizzare un meccanismo di callback: i client interessati ad elaborazione asincrona si registrano sul server; lato client servirebbe un oggetto remoto tramite il quale il server possa comunicare il risultato della computazione che ha effettuato.

### 4.4.4.8. Confronto tra SUN RPC e Java RMI

- SUN RPC
  - Basato su meccanismo process oriented, offre una **trasparenza all'accesso incompleta** (la proc remota può avere 1 solo parametro di input, il **client nell'invocare la proc remota deve specificare un secondo parametro** ovvero puntatore a gestore di trasporto del client, bisogna specificare il numero di versione nella procedura chiamata, nell'implementazione della procedura chiamata lato server va messo \_svc e nome procedura e versione mentre lato client basta solo nome versione). Allo sviluppatore quindi non è completamente trasparente che sta chiamando procedura remota. **Niente trasparenza alla locazione** perché il **client deve conoscere l'hostname sul quale viene offerto il servizio remoto**, Abbiamo visto che il portmapper si occupa del bind tra client e server e reperisce il numero della porta su cui è invocato il server e deve essere in esecuzione su stessa macchina dove è eseguito il server remoto.
  - Si possono **richiedere operazioni o funzioni**
  - La **comunicazione** supportata dalla versione base di Sun RPC è **sincrona bloccante**. Esistono varianti su cui sono supportate anche chiamate asincrone
  - Nel caso di sun rpc si opta per una semantica di comunicazione non esatta (non implementano semantica exactly once perché troppo impegnativa per discorsi di overhead; **usa at least once** invece). Lato server bisogna avere accortezza di realizzare servizi idempotenti tali che la loro invocazione molteplice non alteri lo stato.
  - **Timeout** per ritrasmissione e **gestione errori**
  - Binding a server eseguito tramite **portmapper** in ascolto sulla porta 111 (il client stub sa quale porta andare a contattare)
  - Per gestire l'eterogeneità della rappresentazione dei dati si usa un IDL specifico (**XDR**) e abbiamo visto come client e server stub vengono generati automaticamente tramite rpcgen esplicitamente utilizzato da programmatori
  - Passaggio parametri avviene tramite meccanismo passaggio parametri **copia-ripristino**

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

- Java RMI
  - È un meccanismo basato sugli oggetti, che offre **trasparenza all'accesso, non offre trasparenza alla locazione, non c'è trasparenza alla replicazione** (distribuzione non trasparente).
  - Si possono richiedere metodi di oggetti tramite interfacce
  - **Comunicazione di tipo sincrona**
  - Semantica di comunicazione di tipo **at-most-once**
  - **Gestione degli errori**
  - Binding a server ottenuto tramite **RMI registry**
  - Si usa **Java come IDL** e la generazione automatica di stub e skeleton
  - **Passaggio parametri avviene per valore** (tipi primitivi e oggetti serializable) e **per riferimento** (oggetti con interfaccia remota)

## 4.5. LEZIONI SU GO

Go è un linguaggio che permette di costruire software semplice affidabile ed efficiente. GO è un linguaggio tipato in modo statico. Risulta snello ed efficiente sia in fase di compilazione che di esecuzione. Ha alcuni aspetti che costringono lo sviluppatore a scrivere del codice pulito. GO eredita dal linguaggio C la sintassi delle espressioni per l'assegnazione delle variabili, gli statement per il controllo del flusso. GO eredita dal C anche i tipi di dati di base più un utilizzo più agevole degli array introducendo il concetto di slice (ovvero porzioni di array); Come in C anche qui abbiamo passaggio parametri per valore, uso dei puntatori.

Rispetto ad altri linguaggi di programmazione GO introduce facilmente per la concorrenza nuove ed efficienza (uso di canali di comunicazione per le GO routine), gestione della memoria automatico (nelle funzioni possiamo anche restituire una variabile locale, che in C darebbe un segmentation fault). Oltre tutto questo GO ha un approccio flessibile all'astrazione dei dati e la programmazione ad oggetti. Go ci permette di concentrarci sui problemi dei sistemi distribuiti, è buono per supportare la concorrenza, supporta RPC, è di tipo type safe ed è garbage collected , oltre essere semplice da imparare. GO è un linguaggio per applicazioni cloud native, ed ha l'obiettivo di permettere agli sviluppatori di sviluppare applicazioni cloud su ogni combinazione di cloud provider.

Go è un linguaggio di programmazione compilato e ha diversi tool quali go run e go build per far partire programmi, compilarlo e memorizzare codice binario.

Il codice in GO si divide in package e supporta l'import statement per le librerie utilizzate.

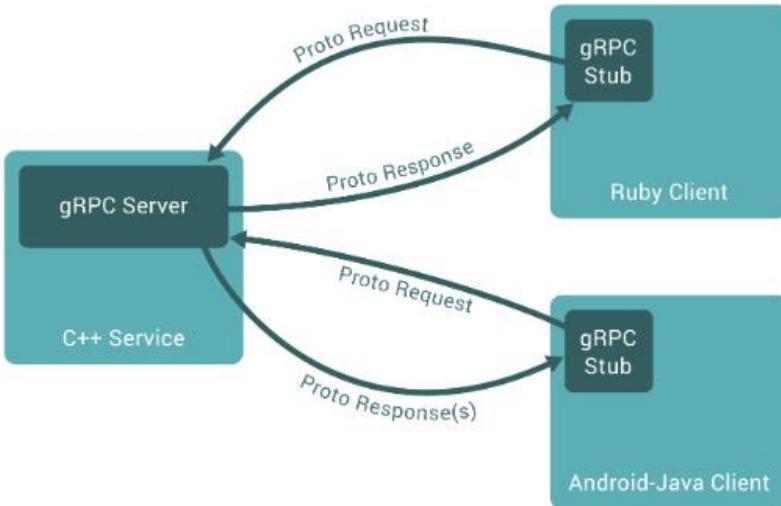
---

### gRPC

gRPC è un framework molto utilizzato per realizzare applicazioni con stile architetturale a micro-servizi. L'idea è quella di avere l'applicazione composta da tanti piccoli servizi che si invocano fra di loro (con sistemi a code di messaggi). RPC è un supporto inizialmente sviluppato da Google, che poi l'ha reso open source e messo a disposizione della comunità. La caratteristica principale è che non è orientato ad un linguaggio di programmazione, ma ne supporta molteplici. gRPC consente di realizzare i servizi poliglotti, ovvero applicazioni i cui servizi componenti sono scritti in linguaggi di programmazione differenti. gRPC utilizza il proxy pattern tramite stub che interagiscono fra di loro; nello specifico utilizza protocol buffer come interface definition language. Protocol buffer è un IDL sviluppato pochi anni fa da google che risulta molto più efficiente di JSON e XML. gRPC è strettamente connesso con HTTP 2 e ne sfrutta tutti i vantaggi, fra cui il multiplexing che sfrutta più canali di connessione.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

Per utilizzare protocol buffer si utilizza un file esterno .proto per definire i dati da serializzare.



## 4.6. Comunicazione message oriented

Il supporto RPC fornisce un certo grado di trasparenza alla distribuzione che è maggiore rispetto a quello della programmazione di rete fornito dalle socket, ed ha una sincronia fra gli agenti (gli agenti di rpc devono essere contemporaneamente presenti durante lo scambio dei messaggi). C'è anche una condivisione di dati fra i due endpoint che comunicano. Nell'applicazione distribuita, gli aspetti legati alla sincronizzazione e alla comunicazione sono strettamente correlati fra loro. Bisogna introdurre modelli di comunicazione che migliorano disaccoppiamento fra app e sistema distribuito. Il **middleware orientato ai messaggi** è un tipo di middleware di comunicazione che supporta invio e ricezione messaggi in maniera persistente. È importante perché permette di realizzare disaccoppiamento spaziale e temporale in quanto i componenti non devono essere presenti e attivi nello stesso momento durante la comunicazione, i componenti non devono necessariamente conoscersi fra di loro, può essere supportato anche il disaccoppiamento di sincronia ovvero il client che invia il messaggio di richiesta non deve rimanere bloccato in attesa del messaggio di risposta.

### Queue message pattern (producer e consumer)

Abbiamo a disposizione una coda che permette di memorizzare in modo persistente i messaggi. Una volta che vengono inviati alla coda vengono memorizzati li finchè non vengono prelevati dai destinatari dei messaggi ed eventualmente da loro cancellati. Quindi molteplici consumatori possono prelevare i messaggi è vero, ma il messaggio può essere consegnato 1 singola volta ad un solo consumatore (1:1); La coda di messaggi mette la comunicazione tra un consumer ed un producer disaccoppiandoli temporalmente (il disaccoppiamento di sincronia dipende dal supporto offerto dalla coda, stessa cosa per il disaccoppiamento spaziale). Il sender ed il receiver in generale usano due code differenti.

La API fornisce:

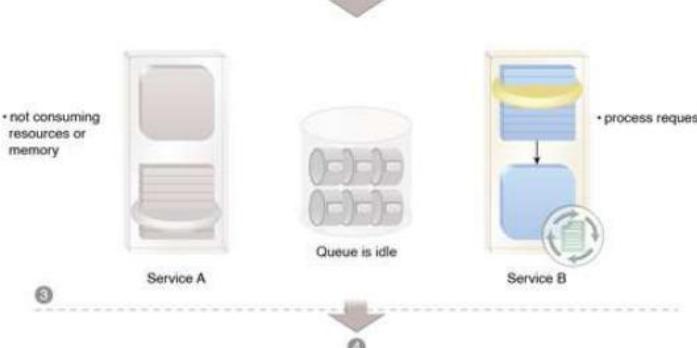
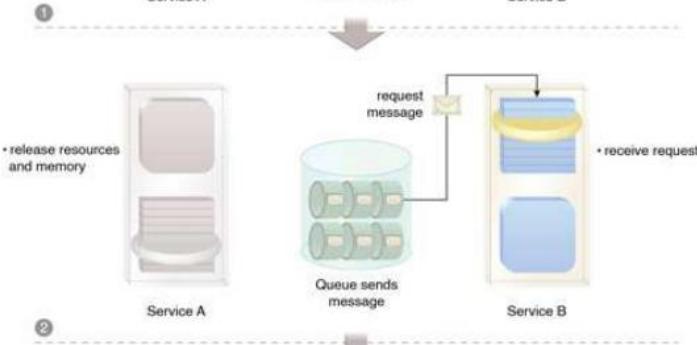
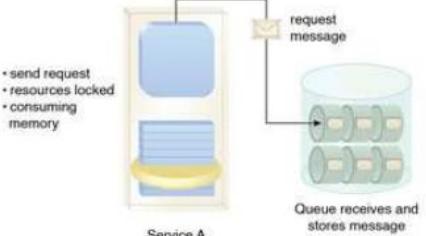
- put (metti mess in coda)
- get (bloccati finche la coda è non vuota e rimuovi il primo messaggio)
- poll (il consumer non rimane bloccato se la coda non è vuota, controlla se c'è messaggio in coda e lo rimuove)

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

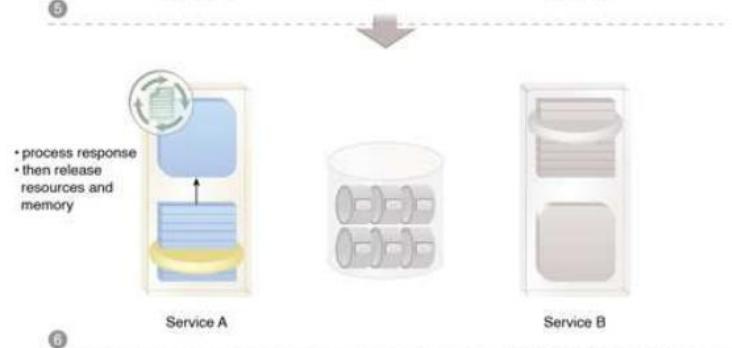
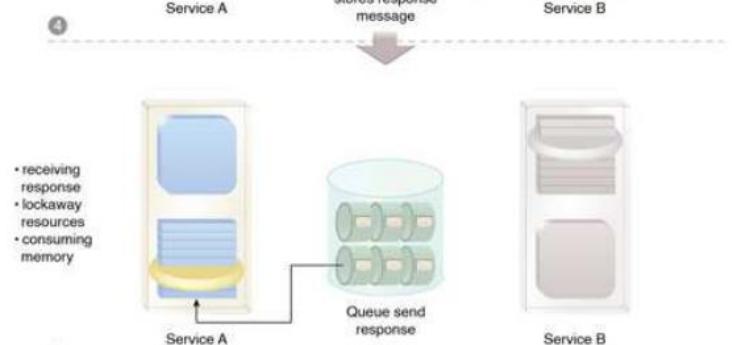
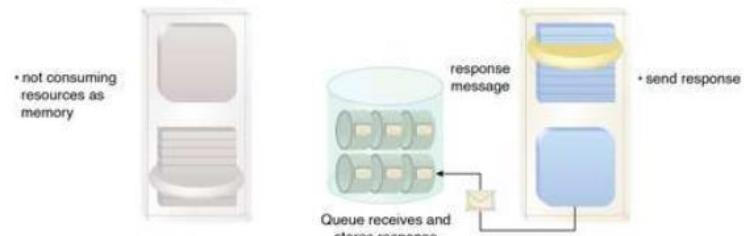
- notify (c'è un handler e una funzione di callback che viene chiamato appena viene messo in coda messaggio che corrisponde alle specifiche indicate dal consumer; Il consumer viene chiamato appena viene messo un messaggio nella coda specificata dal consumer).

La put è un send non bloccante, la get è un receive bloccante. Poll e notify sono invece receive non bloccanti.

## A sends a message to B



## B issues a response message back to A



## Il pattern publish subscribe (publisher e subscribers)

La comunicazione diventa di tipo one to many. Il publisher può consegnare un messaggio a molti subscriber. Permette di comunicare in modo asincrono quindi (notifiche di eventi avvenuti). Di solito questi sistemi sono infatti topic based, i subscriber sottoscrivono a topic e appena diventa disponibile un messaggio relativo a un topic esso viene consegnato a tutti i subscriber sottoscritti a quel topic.

I subscriber possono usare filtri per filtrare eventi a cui sono interessati in uno specifico topic.

Il **dispatcher degli eventi** è responsabile di effettuare l'instradamento degli eventi verso tutti i subscriber interessati ad un determinato topic (implementazione distribuita che può differire). Questo pattern garantisce un elevato grado di disaccoppiamento e rende facile la gestione di un sistema dinamico di componenti del sistema che vanno e vengono che possono essere rimossi a runtime (i subscriber). I suddetti

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

due pattern quindi sono simili ma non la stessa cosa, in quanto il secondo è una generalizzazione del primo.

La API offerta include

- publish(event): pubblica un evento che puo essere espresso in diversi tipi e formati di dati a seconda dell'implementazione (puo contenere anche metadati)
- Subscribe(filter,notify,expry): per sottoscriversi ad un evento, si puo specificare anche un tempo di scadenza alla sottoscrizione
- Unsubscribe
- Notify (): invia notifiche per i subscriber che soddisfano determinati filtri specifici di uno stesso topic

## 4.6.1. MOM

Gestisce la complessità di routing, addressing e availability dei sistemi di comunicazione.

Le semantiche di comunicazione usate in questi sistemi MOM sono:

- **At least once delivery**

Il sender manda un messaggio e se entro un timer non riceve l'ack lo rimanda. Si puo perdere l'ack e puo succedere che il receiver riceve piu volte lo stesso messaggio. Bisogna fare in modo quindi che il sistema sia in grado di scartare i duplicati.

- **Exactly once**

Permette di fare in modo che il messaggio sia consegnato esattamente una volta. Per riuscirci esiste un filter (middleware) che grazie al nick message ID (univoco) riesce a filtrare messaggi ed eliminare duplicati. I messaggi devono anche resistere ai fallimenti dei componenti del sistema MOM.

### 4.6.1.1. Consegnata basata su transazione

Il sistema MOM assicura che i messaggi siano cancellati dalla coda solo se sono stati ricevuti con successo dal destinatario.

Dopo che il messaggio è stato consegnato e processato il receiver invia un ack alla coda di messaggi. Solo alla ricezione di quell'ack verrà cancellato dalla coda. Più forte dell'at least once perché in questa il messaggio viene cancellato solo quando è stato ricevuto e processato dal destinatario, nell'at least once viene inviato ack già senza che il destinatario l'abbia processato.

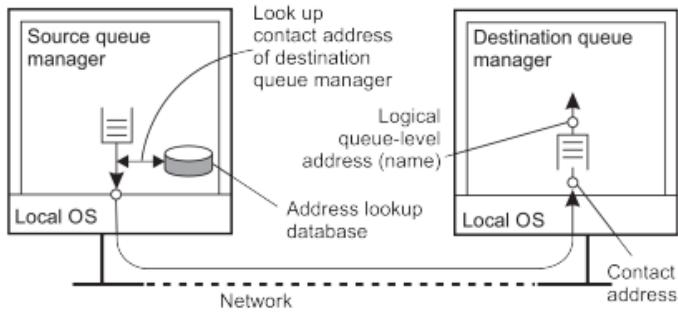
### 4.6.1.2. Consegnata basata su timeout

L'obiettivo di questa semantica è garantire che il messaggio venga cancellato dalla coda messaggi quando è stato ricevuto con successo almeno una volta. La transazione è ACID e quindi atomica (tutto o niente), questa qua è "basta 1 volta che viene ricevuto e il messaggio si cancella". Appena il messaggio è visibile nella coda di messaggi e può essere consumato da un receiver, nel momento in cui il receiver lo preleva, il messaggio non viene cancellato dalla coda ma viene impostato come invisibile (presente nella coda ma non può essere prelevato da altri consumer). Una volta che il consumer ha processato il messaggio invia un ack. Quando viene ricevuto l'ack, se non è scattato il timeout relativo alla visibilità del messaggio (è ancora invisibile) allora viene cancellato. Se ormai ci ha messo troppo e il messaggio era tornato visibile scaduto il timeout, allora potrà essere ricevuto ancora da un altro consumer (maggiore tolleranza ai guasti grazie a questa semantica! Il messaggio sarà consegnato con successo garantito almeno 1 volta).

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

#### 4.6.1.3. Modello di Message Routing

Le code vengono gestite dai Queue Managers (QMs); è possibile apporre messaggi solo all'interno di code locali, come anche il prelevamento. I QMs devono indirizzare i messaggi; per fare ciò utilizzano delle overlay network.



L'overlay network serve quindi a indirizzare i messaggi utilizzando routing tables (le quali sono salvate e gestite dai QMs).

Le routing tables vengono create e impostate manualmente. Le overlay network dinamiche devono dinamicamente gestire il mapping tra i nomi delle code e le loro locazioni.

#### Message Broker

Si tratta di un componente che si occupa di gestire l'eterogeneità dei dati in un sistema MOM.

Ciò che fa è:

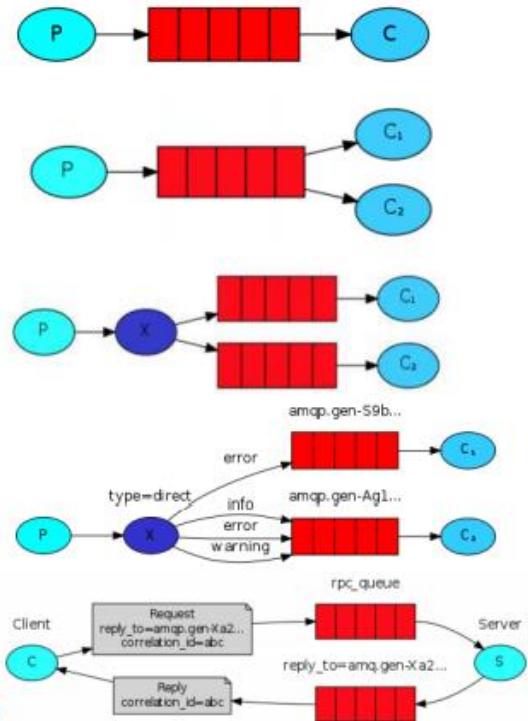
- converte i messaggi in arrivo nel formato di destinazione garantendo trasparenza all'accesso.
- Si comporta come un application gateway.
- Gestisce un repository di regole di conversione e programmi per trasformare i messaggi nei diversi tipi.
- Può fornire routing subject-based.
- Affinchè sia scalabile e affidabile può essere implementato in maniera distribuita.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

## Using message queues: use cases

1. Store and forward messages which are sent by a producer and received by a consumer (**message queue pattern**)
2. Distribute tasks among multiple workers (**competing consumers pattern**)
3. Deliver messages to many consumers at once (**pub/sub pattern**)
4. Receive messages selectively: producer send messages to an **exchange**, that selects the queue
5. Run a function on a remote node and wait for the result (**request/reply pattern**)

Source: RabbitMQ tutorial <http://bit.ly/2zPPMJO>



### 4.6.1.4. Vediamo 3 tipi di MOM frameworks:

#### 4.6.1.4.1. IBM

Tecnologia di messaggistica; i messaggi inviati dalle applicazioni vengono posti all'interno di code e rimossi dalle stesse. Le code sono gestite dal queue manager, e le applicazioni possono inserire i loro messaggi in code locali (sulla stessa macchina) o in code remote (gestite da una macchina diversa) tramite RPC.

MCA sono agenti responsabili dell'instaurazione del canale e dell'invio e ricezione dei messaggi sugli stessi. Oltre ciò si occupano anche di crittografare e decrittografare i messaggi. Si tratta di canali di comunicazioni direzionali, quindi per avere un dialogo fra due entità servono due canali, uno per direzione. IBM MQ mette a disposizione la possibilità di creare una overlay network di queue manager. Si può creare una rete logica (overlay network) che connette fra loro più manager. Il routing può avvenire utilizzando nomi logici assegnati alle code (maggiore flessibilità).

#### 4.6.1.4.2. SQS

È un sistema a code di messaggi basato su un servizio cloud-based. I componenti dell'applicazione che usano SQS possono funzionare indipendentemente e in modo asincrono e possono essere sviluppati con differenti tecnologie. La semantica offerta da SQS può essere configurata per avere una semantica di tipo timeout based. Se non arriva l'ack entro il timeout di visibilità si assume che il processamento da parte del consumer sia fallito e quindi il messaggio diventa visibile da altro

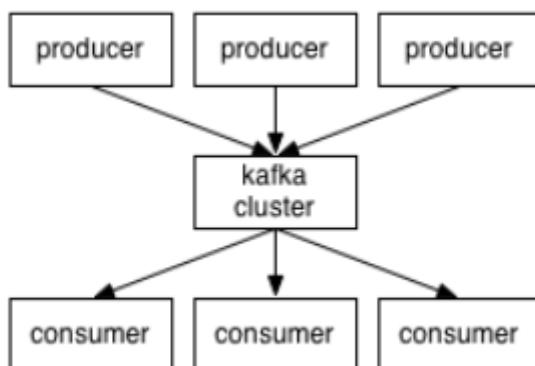
Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

consumer. Può essere integrato con il servizio SNS che permette di effettuare pushing di un messaggio per avere code parallele

#### 4.6.1.4.3. Kafka

Caratterizzato da:

- broker che mettono a disposizione un log distribuito per la gestione dei dati.
- Cluster che permettono ai producer di mettere a disposizione dei topic a cui i consumer possono interessarsi.



I **topic** sono categorie gestite dai broker in cui i messaggi vengono pubblicati, che **mantengono i log** (data structure di tipo append only - totally ordered sequenza di record ordinati temporalmente).

Kafka è un sistema altamente scalabile, il **topic** è infatti accessibile da più producer e consumer contemporaneamente; a tal scopo il suddetto viene **suddiviso in un certo numero di partizioni** che definiscono la quantità di parallelismo sul topic stesso. Ciascuna partizione, per migliorare la scalabilità e tolleranza a guasti può essere replicata per tutti i broker. Il fattore di replicazione è definito in fase di configurazione.

Il log è composto da record e memorizzato in partizioni distribuite e replicate in numerosi broker di kafka. Il **topic su cui fare la pubblicazione viene scelto secondo modalità round robin o partizionamento basato su chiave** (con un hash basato su contenuto del messaggio). I consumer possono leggere i record da un determinato topic. Ciascuna partizione è un log, quindi sequenza di record ordinata ed immutabile alla quale avvengono solo operazioni di append. **Ciascun record è numerato**, importante all'atto del recupero delle informazioni dal topic. Il numero di sequenza a cui ogni record è associato si chiama offset.

Le operazioni di scrittura avvengono su leader della partizione. I **follower** sono copie di backup della partizione che replicano la partizione del leader e possono subentrare tramite **meccanismo di elezione** nel caso in cui il server che svolge ruolo di leader per quella partizione subisca un fallimento.

I producer in kafka sono coloro che pubblicano i dati mandandoli direttamente al leader della partizione. Inoltre i producer sono anche responsabili di scegliere a quale partizione assegnare un determinato record. Per scegliere a quale record assegnare quale partizione di topic si sceglie in base alla chiave o randomicamente.

##### Riassunto punti salienti kafka:

Le caratteristiche di kafka sono : scalabile e tolleranza ai guasti

Si crea un kafka cluster al quale si agganciano diversi broker. Permette di creare dei topic all'interno dei quali vengono pubblicati e prelevati i messaggi. Il cluster viene gestito da kafka come un log

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

partizionato (struttura dati distribuita sulla quale si possono effettuare solo operazioni di scrittura in coda). I record in un topic sono ordinati in base al tempo.

Cosa fa kafka per avere alte prestazioni è dividere ogni topic in un determinato numero di partizioni deciso dall'utente. La partizione rappresenta l'unità di parallelismo del topic. Quindi produttori e consumatori diversi potranno accedere in modo concorrente a partizioni differenti. Il nome del topic deve essere unico tramite un id.

Kafka ha un modello che puo essere basato su **push**, quindi sono i broker di kafka che progressivamente spingono i messaggi verso i consumer. Da un punto di vista di realizzazione questo complica le cose, perché se i broker devono pushare i messaggi devono sapere gestire i diversi tipi di consumer che possono avere diverse capacità computazionali che gli permettono di ricevere i dati con cadenze differenti. I broker poi possono essere settati per accumulare o mandare subito i messaggi.

In un **modello pull** è il consumer che si prende la responsabilità di prendere i messaggi dal broker, quindi svolge un ruolo attivo nel reperimento dei messaggi. Il consumer deve mantenere traccia del punto del log fino al quale ha scaricato i messaggi, mantenendo memorizzato l'offset.

Avere meno carico sui broker garantisce una maggiore flessibilità, così che il broker non deve adattarsi ai diversi consumer in quanto sono questi ultimi a prendersi i messaggi.

Basare kafka sul push si esclude quindi, e si decide di basarlo su pull.

I consumer sono divisi in consumer group, gruppi contraddistinti da id che sono come un unico subscriber logico. Questo sistema migliora scalabilità e tolleranza ai guasti. Questi consumer possono leggere contemporaneamente su partizioni differenti sfruttando il parallelismo. L'**offset** ha un ruolo fondamentale nell'identificazione del punto del log a partire dal quale il consumer deve partire a leggere.

Per semplificare il design si sceglie di ordinare i messaggi solo all'interno della partizione, non è detto che in un'altra partizione dello stesso topic i messaggi siano ordinati allo stesso modo. Quindi due consumer che leggono due partizioni dello stesso topic non è detto che vedano i messaggi nello stesso ordine. Garantire ordinamento tra partizioni introdurrebbe un overhead notevole.

Kafka supporta sia la semantica at least once defalut, sia quella exactly once per le consegne dei messaggi. Potrebbe implementare at most once disabilitando il prelievo dei consumer. Il motivo per cui di default c'è l'at least once è perché negli altri casi aumenterebbe troppo l'overhead.

**Zookeeper:** Insieme distribuito gerarchico di coppie key-value; zookeeper è caratterizzato da servizi di coordinazione e sincronizzazione per grandi sistemi distribuiti, inoltre usa l'algoritmo di elezione dei leader. Kafka usa zookeeper per coordinare broker, producer e consumer. Zookeeper mantiene i metadati di kafka salvati, quindi una lista delle tre entità sudette.

Kafka mette a disposizione 4 API, quella per i producer (permette alle app di pubblicare i record sui topic), per i consumer (permette alle app di leggere record dai topic), di connessione (permette l'implementazione dei connettori che connettono i topic ad applicazioni esistenti e sistemi di dati, così da muovere grandi dati dentro e fuori kafka) e stream processor (abilita la trasformazione degli stream di dati da un input topic ad un output topic).

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

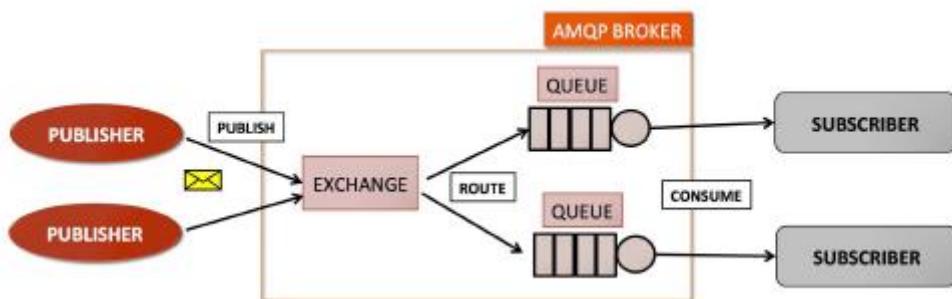
Kafka non è solo un pub/sub system, ma anche un real time streaming platform.

#### 4.6.1.5. *Messaging protocols and IoT*

Spesso si utilizza un protocollo di coda di messaggi per inviare dati dai sensori ai device che li processeranno. Questi protocolli di messaggistica in ambito di IoT ci permettono di caricare i messaggi in sistemi MOM garantendo disaccoppiamento e resistenza a guasti, oltre una notevole gestione dei picchi di carico.

##### AMQP

È un MOM protocollo basato su uno standard aperto largamente supportato dalle aziende nel settore, protocollo di tipo binario di livello applicativo che si basa su TCP per il trasporto. AMQP supporta tutte e tre le semantiche di consegna e si caratterizza per essere un protocollo programmabile. Risulta formato da numerosi componenti che sono programmabili. È un protocollo supportato da diversi framework di messaggistica. Nell'architettura di AMQP abbiamo i publisher, i subscriber ed il broker. Nel broker ci sono diverse entità, fra cui le code, gli exchange (punti di scambio) ed i binding (collegamenti fra i punti di exchange che svolgono il ruolo di collegare le exchange alle code a cui i subscriber si collegano, quindi come instradano i messaggi verso le code).

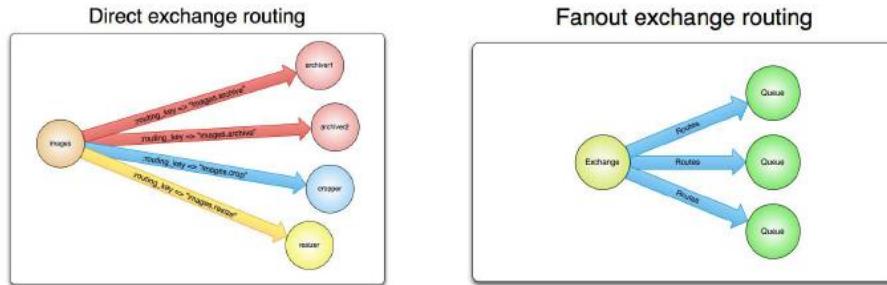


I binding, ovvero le regole con cui l'exchange si collega nelle diverse code nel sistema è programmabile nel sistema AMQP, secondo

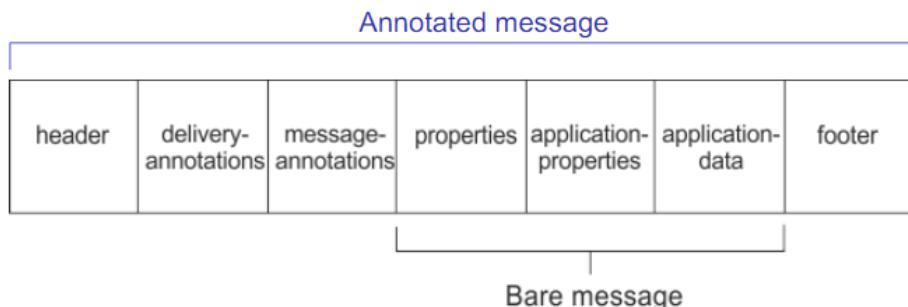
- direct exchange (routing nelle code di messaggi basato su una chiave di routing).
- fanout exchange (l'exchange replica il messaggio in uscita su tutte le code).
- topic exchange (il messaggio viene consegnato ad una o più code sulla base del matching del topic).

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

- headers exchange (la consegna dei messaggi si basa su attributi espressi all'interno dell'header).



AMQP supporta due tipi di messaggi: **Bare messages** (mandati dai sender) e **Annotated messages** (visti dal receiver e aggiunti da intermediari durante il transito).



## 4.6.2. Multicast applicativo

Principio basato sulla replicazione dei pacchetti con routing gestiti dagli end host. L'idea di base è quella di organizzare i nodi in una overlay network che servirà a diffondere le informazioni di multicast.

Il multicast applicativo può essere di tipo strutturato (percorsi di comunicazione espliciti nell'overlay network) o non strutturato (sfrutta flooding e gossiping).

### 4.6.2.1. MULTICAST APPLICATIVO STRUTTURATO

Come costruire in modo strutturato la rete overlay? Si può sfruttare una struttura ad albero (unico percorso fra ogni coppia di nodi) o a mesh (molti percorsi fra ogni coppia di nodi).

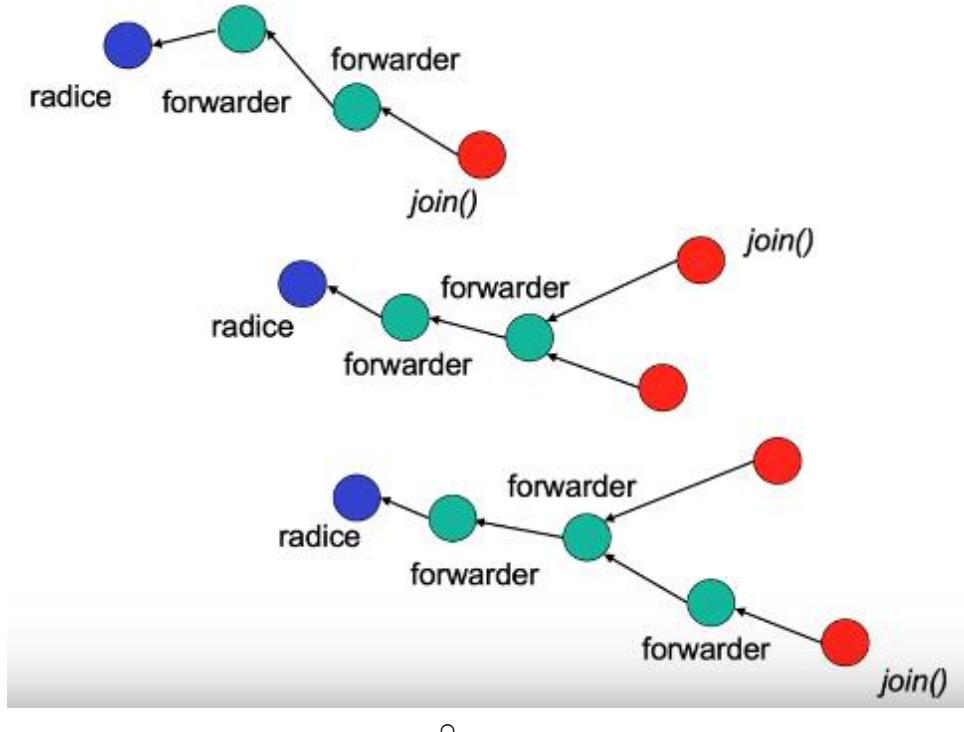
#### Esempio di multicast applicativo strutturato ad albero: Scribe

Si tratta di un sistema pub/sub con architettura decentralizzata basato su pastry. L'albero qui si costruisce sfruttando un algoritmo così strutturato;

- Il nodo che inizia la sessione identifica il gruppo di multicast (mid) tramite un identificatore.
- Questo nodo utilizza pastry per trovare il nodo responsabile per il valore identificativo in questione.
- Quel nodo diventa la radice dell'albero del multicast.
- Se il nodo P vuole unirsi all'albero di multicast identificato da mid invia una richiesta di join.
- Quando la richiesta di join arriva al nodo Q, allora se Q non ha mai ricevuto una richiesta di join per mid, Q diventa il forwarder mentre P diventa figlio di Q; Q procederà con l'inoltrare la richiesta di join

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

verso la radice. Se Q era già un forwarder per mid, allora P diventa figlio di Q; non occorrerà inoltrare la richiesta di join alla radice.



○

#### 4.6.2.2. MULTICAST APPLICATIVO NON STRUTTURATO

Sfrutta il flooding (nodo P invia messaggio flooding a tutti i suoi vicini, e ognuno al suo vicino ogni volta eliminando i messaggi già visti).

Il gossiping invece a differenza del flooding sfrutta un approccio probabilistico. Un nodo che riceve il messaggio potrà inoltrare il messaggio solo ad un sottoinsieme dei nodi che conosce. Questo sottoinsieme può essere scelto in modo casuale. Ogni nodo che lo riceve ne reinvia una copia ad un altro sottoinsieme, anch'esso scelto casualmente, e così via.

#### 4.6.2.3. Gossiping

Questi protocolli di gossiping sono tutt'oggi utilizzati in molti sistemi distribuiti. Uno per esempio è usato da amazon nel servizio di storage S3. Furono inventati negli anni 80 da Demers, e all'inizio l'idea di base era:

- Operazioni di aggiornamento sono eseguite inizialmente su una o alcune repliche.
- Una replica comunica il suo stato aggiornato ad un numero di vicini
- La propagazione dell'aggiornamento non è immediato
- Al termine l'aggiornamento raggiunge tutte le repliche

Il gossiping è caratterizzato da assenza di nodi centralizzati, quindi risulta essere un protocollo robusto (grazie alla ripetizione dei messaggi) e scalabile (ogni nodo manda solo un numero limitato di messaggi), oltre essere molto semplice a livello algoritmico.

Vediamo ora i **modelli di propagazione del gossiping**:

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

- **Gossiping puro:** un peer che ha appena ricevuto un'informazione contatta un altro peer scelto casualmente inviandogli la propria informazione. Se tuttavia Q ha già ricevuto l'aggiornamento, perde interesse a diffondere il gossip e con probabilità pari a  $1/k$  smette di contattare altri peer; per garantire che un alto numero di peer sia aggiornato occorre combinare il gossip puro con l'anti-entropia.
- **Anti-entropia:** periodicamente ciascun peer sceglie casualmente un altro peer e i due si scambiano gli aggiornamenti per raggiungere uno stato simile su entrambi. L'obiettivo è minimizzare l'entropia nel sistema, ovvero la differenza di informazioni fra i diversi nodi deve essere minima. Si cerca di aumentare la similarità fra i peer aumentando così l'ordine. Un peer P sceglie casualmente un altro peer Q nel sistema, e lo aggiorna secondo push(P invia solo i suoi aggiornamenti a Q), pull (P prende soltanto gli aggiornamenti da Q), push-pull (P e Q si scambiano aggiornamenti in modo reciproco). È la strategia più veloce, impiega  $O(\log N)$  round per propagare un aggiornamento agli N peer del sistema. Il round di gossip è un ciclo, ovvero intervallo di tempo in cui ogni peer ha preso almeno una volta l'iniziativa di scambiare aggiornamenti).

- Due peer P e Q, con P che ha scelto Q per lo scambio di dati; P è eseguito una volta ad ogni round (ogni  $\Delta$  unità di tempo)

Active thread (peer P):		Passive thread (peer Q):
(1) <b>selectPeer</b> (&Q);		(1)
(2) <b>selectToSend</b> (&bufs);	----->	(2)
(3) sendTo(Q, bufs);		(3) receiveFromAny(&P, &bufr);
(4)		(4) <b>selectToSend</b> (&bufs);
(5) receiveFrom(Q, &bufr);	<-----	(5) sendTo(P, bufs);
(6) <b>selectToKeep</b> (cache, bufr);		(6) <b>selectToKeep</b> (cache, bufr);
(7) processData(cache);		(7) processData(cache)

Per implementare un protocollo di gossiping bisogna gestire membership (come i peer possono conoscersi fra loro e quanti conoscenti avere), avere consapevolezza della rete (come fare in modo che i collegamenti fra i peer riflettano la topologia della rete per prestazioni soddisfacenti), bisogna gestire i buffer (ovvero le memorie dei peer, considerando quali informazioni scartare e quali inserire in cache in caso di memoria piena), gestire filtraggio dei messaggi (ridurre la probabilità di ricezione di notizie a cui non sono interessati i peer).

#### 4.6.2.4. Gossip vs Flooding

Il gossiping è un protocollo di tipo probabilistico dove ciascun nodo prende decisioni locali che hanno un impatto a livello globale sulla diffusione informazioni nella rete; è più leggero del flooding in quanto sono meno messaggi scambiati nella rete e è protocollo resistente ai guasti per la ripetizione dei messaggi. Il flooding ha dei vantaggi perché si riduce la quantità di informazione scambiata e si copre tutta la rete con 1 messaggio a fronte però dell'eccessivo numero di messaggi scambiati.

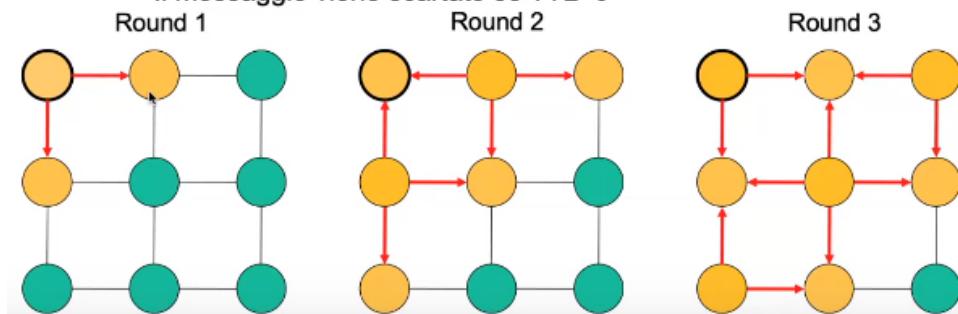
Il gossiping quindi riduce il numero dei messaggi ridondanti che troviamo nel flooding e cerca di mantenerne i vantaggi.

Oltre alla diffusione delle informazioni, il gossiping si può usare per peer sampling (fornire a ciascun peer una lista di peer da contattare), monitoraggio di risorse in sistemi distribuiti a larga scala, computazioni

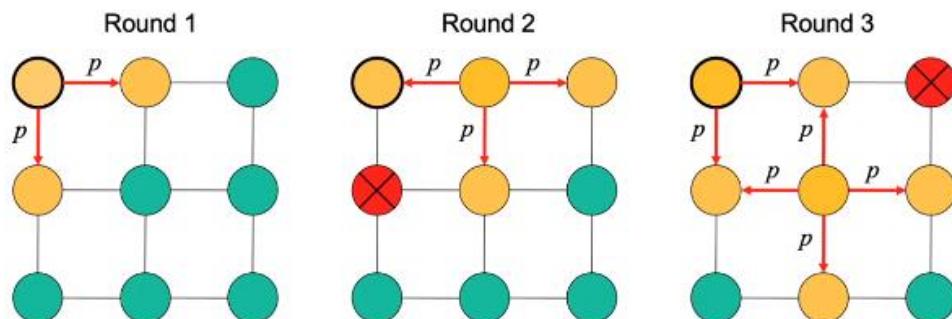
Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

distribuite per l'aggregazione di dati (tipico utilizzo nelle reti di sensori dove serve computazione di valori aggregati come per esempio la media).

- La diffusione dell'informazione è l'applicazione classica e più popolare del gossiping nei SD
  - Valida alternativa rispetto al flooding
- Nel caso di flooding
  - Ogni peer che riceve il messaggio lo invia a tutti i suoi vicini (è una degenerazione del gossiping)
  - Il messaggio viene scartato se TTL=0



- Nel caso di gossiping semplice
  - Il messaggio viene inviato con una probabilità di gossiping  $p$
  - for each msg  $m$   
 $\text{if random}(0,1) < p \text{ then send } m$



### Esempi odierni di protocolli di gossiping

- Blind counter rumor mongering: (rumor mongering = diffusione del pettegolezzo, blind perché non c'è interesse sul chi sia che riceve il pettegolezzo, counter perché perde interesse dopo un tot di volte che viene contattato). Un nodo inizia la diffusione dell'informazione in un sottoinsieme randomico fra i suoi vicini con cardinalità  $B$ . Quando nodo  $P$  riceve messaggio  $M$  da nodo  $Q$ , se  $P$  ha ricevuto  $M$  meno di  $F$  volte allora  $P$  manda  $M$  a  $B$  vicini che conosce che non hanno visto  $M$ . Epico.
- Bimodal multicast (o pbcast): Algoritmo semplice ed elegante con l'obiettivo di effettuare un invio in multicast di un messaggio nella rete in modo tale che l'invio risulti essere affidabile. Caratterizzato da due fasi

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

- **Message distribution** - un processo manda in multicast un messaggio senza particolari garanzie di affidabilità.
- **Gossip repair** – dopo che un processo riceve un messaggio, inizia a fare gossip del messaggio su un set di peers, ovvero interroga i nodi vicini sul cosa hanno ricevuto facendo un confronto del proprio stato con i loro. Si chiama così proprio perché tramite gossip corregge l'affidabilità in quanto se scopre così di non sapere qualcosa, chiede agli altri nodi di mandargli cosa gli manca.

L'algoritmo si chiama multicast perché invia messaggio in multicast, **bimodale per il comportamento bimodale dell'algoritmo in quanto quest'ultimo ci permette di consegnare il messaggio a tutti/quasi tutti o solo a pochi processi, ma non accade che il messaggio raggiunga la metà dei nodi** (rilassamento del concetto di atomicità del tutti o nessuno). Altro motivo per cui si chiama bimodale è per via delle latenze sperimentate nella consegna dei messaggi (se messaggio arriva subito o all'ultimo round di gossip repair ovviamente la latenza può essere molto diversa).

## 5. VIRTUALIZZAZIONE

Può essere realizzata a diversi livelli e su diverse risorse, noi ci concentreremo sul contesto operativo a livello di sistema e a livello di sistema operativo.

La virtualizzazione è una delle realizzazioni in informatica del concetto di indirezione in comunicazione; essa ci offre un livello alto di astrazione che permette di nascondere i dettagli sottostanti garantendo un'astrazione delle risorse computazionali, così da presentare all'utilizzatore una vista logica diversa da quella fisica. Per ottenere ciò è necessario il disaccoppiamento dell'architettura ed il comportamento delle risorse hardware e software percepiti dall'utente dalla loro realizzazione fisica. Gli obiettivi della virtualizzazione sono molti, fra cui affidabilità, prestazioni e sicurezza.

La virtualizzazione può riguardare, oltre le risorse hardware e software di sistema, anche altre parti di sistemi informatici (storage, rete o interi data center).

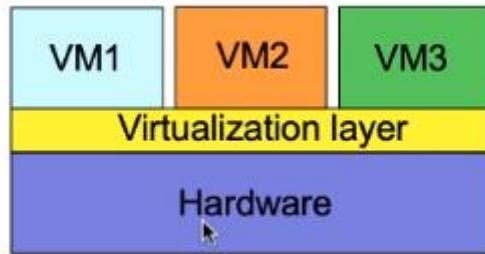
Nella virtualizzazione sono fondamentali 3 componenti: Host, Guest, Layer di visualizzazione.

- **Guest**: componente che interagisce con il layer di virtualizzazione piuttosto che con l'host.
- **Layer di virtualizzazione**: livello di indirezione che permette di astrarre le risorse fisiche e software che l'host mette a disposizione.
- **Host**: ambiente originale dove vengono gestiti i guest.

### 5.1. Macchina virtuale

Permette di rappresentare le risorse hardware e software di una macchina reale diversamente da quelle che sono in realtà. Usando una VM si può avere una vista logica quindi differente da quella fisica delle effettive risorse. Una singola macchina fisica può essere usata come differenti ambienti di elaborazione

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.



Con la virtualizzazione può essere raggiunta una densità di container di un ordine di grandezza o due maggiori rispetto a quelle istanziabili sulla macchina fisica.

Lo sviluppo di un supporto hardware nell'architettura dei calcolatori con l'evoluzione nel tempo della tecnologia ha permesso lo sviluppo delle macchine virtuali fino a quelle che abbiamo oggi, che sono capaci di rendere quasi impercettibile l'utilizzo delle stesse. La virtualizzazione è particolarmente importante nel cloud computing perché rende possibile agli utenti utilizzare le stesse risorse nei server.

## 5.2. Vantaggi della virtualizzazione

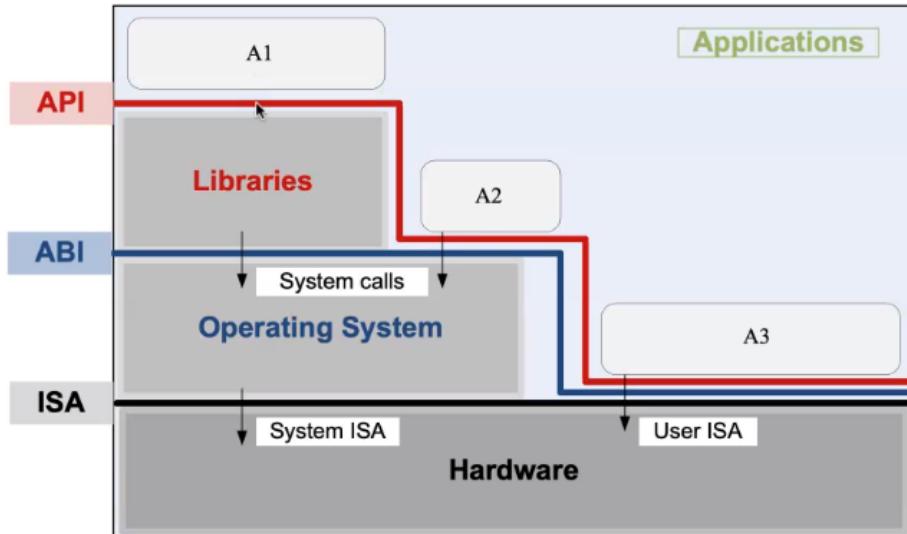
La virtualizzazione permette la **portabilità** in quanto rende le applicazioni usate indipendenti dall'hardware e software sottostante. Permette inoltre sia a livello di macchine virtuali che di container di spostare le macchine virtuali dall'ambiente di lavoro a quello di esecuzione per una migliore gestione del carico di lavoro al fine di **miglior efficienza**. È possibile anche la migrazione dei container.

La virtualizzazione permette inoltre il consolidamento dei server in un data center, offrendo **vantaggi economici, gestionali ed energetici**; Il multiplexing di molteplici VM su uno stesso server con l'obiettivo di ridurre il numero totale dei server usati permette maggiore efficienza, riduzione costi e semplificazione della gestione. La virtualizzazione rende le applicazioni più sicure in quanto permettono di isolare i componenti malfunzionanti o soggetti ad attacchi di sicurezza (macchine virtuali di differenti applicazioni non possono avere accesso alle rispettive risorse, quindi bug o crash di una VM non va a danneggiare le altre VM in esecuzione sulla stessa macchina fisica), incrementando **affidabilità e sicurezza**, permette di isolare le prestazioni di diverse VM (scheduling di risorse fisiche divise fra le molteplici VM in esecuzione sulla stessa macchina) e bilanciare il carico sui server (tramite migrazione della VM da un server ad un altro).

### Utilità degli ambienti di esecuzione virtualizzati:

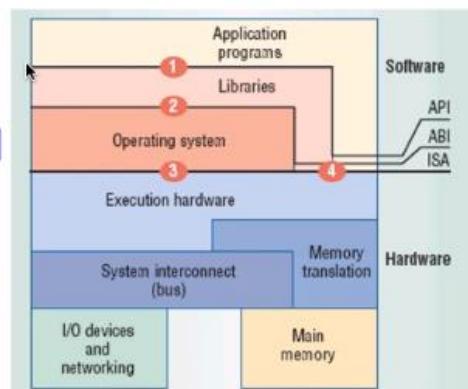
- Eseguire simultaneamente diversi SO sulla stessa macchina
- Semplificare l'installazione di software
- Debugging, testing e sviluppo di applicazioni
- Consolidare l'infrastruttura del data center
- Garantire business continuity, incapsulando interi sistemi in singoli file (system image) che possono essere migrati, replicati o installati su altri server.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.



## A che livello realizzare la virtualizzazione?

- Dipende fortemente dalle interfacce offerte dai vari componenti del sistema
  - Interfaccia tra hw e sw (**user ISA**: istruzioni macchina non privilegiate, invocabili da ogni programma) [interfaccia 4]
  - Interfaccia tra hw e sw (**system ISA**: istruzioni macchina invocabili solo da programmi privilegiati) [interfaccia 3]
  - Chiamate di sistema [interfaccia 2]
    - **ABI** (Application Binary Interface): interfaccia 2 + interfaccia 4
  - Chiamate di libreria (**API**) [interfaccia 1]
- Obiettivo della virtualizzazione
  - Imitare il comportamento di queste interfacce



Ci sono 5 livelli di virtualizzazione:

- **Livello ISA**: l'emulazione del livello ISA (instruction set architecture) può essere effettuata tramite interpretazione delle istruzioni o traduzione binaria dinamica.
- **Livello Hardware** (livello di sistema VM): basato sul virtual machine monitor (VMM), chiamato hypervisor.
- **Livello del sistema operativo** (containers): permette di ottenere le migliori soluzioni di virtualizzazione.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

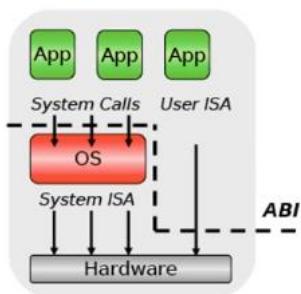
- Livello delle librerie
- Livello delle applicazioni utente

### Process Virtual Machine

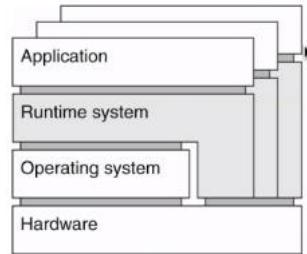
Si tratta di un calcolatore astratto (virtuale) di un processo.

- Piattaforma virtuale che esegue un processo individuale
- Fornisce un ambiente virtuale ABI o API per le applicazioni dell'utente
- L'applicazione è compilata tramite un intermediario, trattando un codice portabile (per esempio il bytecode di Java)

- Examples: JVM, .NET CLR



Multiple instances of combinations  
<application, runtime system>

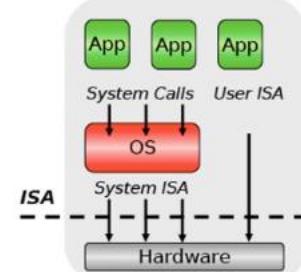


### System Virtual Machine (quella che studiamo noi)

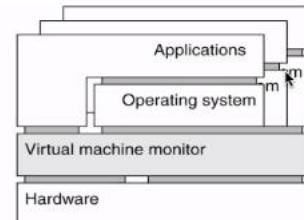
Fornisce un ambiente completo dove il sistema e i processi possono coesistere;

- Il VMM gestisce il set delle risorse hardware e le suddivide fra le diverse macchine virtuali garantendo isolamento e protezione delle stesse.
- Quando una macchina virtuale esegue un'istruzione privilegiata o un'operazione che interagisce direttamente con l'hardware condiviso, il VMM intercetta l'operazione, controlla la correttezza e la esegue (per esempio VirtualBox).

- Examples: VMware, KVM, Xen, Parallels, VirtualBox



Multiple instances of combinations  
<application, operating system>

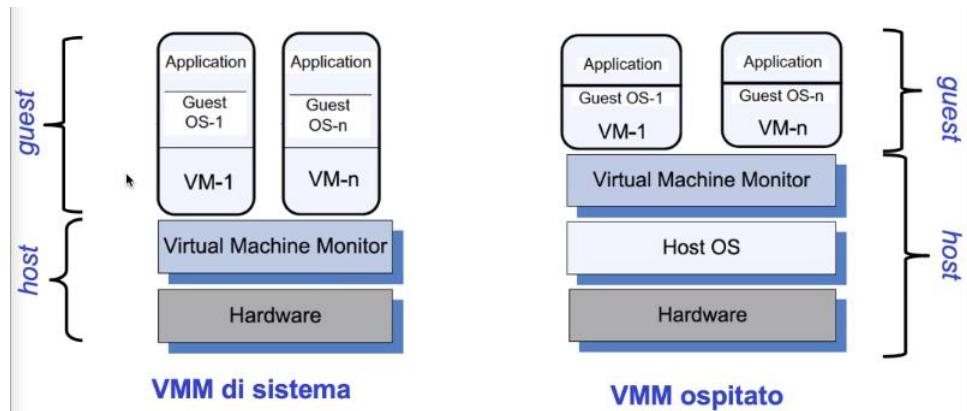


## 5.3. VIRTUALIZZAZIONE A LIVELLO DI SISTEMA (ottenuta grazie ad un VMM)

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

L'host include la macchina fisica, il possibile sistema operativo ed il monitor della macchina virtuale. Con il termine guest invece intendiamo tutto ciò che possiamo trovare nella singola macchina virtuale.

La macchina virtuale può essere messa su diversi system level: o direttamente sull'hardware (**VMM di sistema**) o sopra il SO host (**VMM ospitato**).



Fra le due, ovviamente le prestazioni migliori si ottengono con il VMM di sistema, essendo il layer di indirezione collocato direttamente sopra l'hardware (avendo meno astrazione). Risulta però più semplice realizzare un VMM ospitato in quanto si può sfruttare l'API del sistema operativo host sottostante.

VMM di sistema: eseguito direttamente sull'hardware offre funzionalità di virtualizzazione integrate in un sistema operativo semplificato. L'hypervisor può avere un'architettura a micro-kernel o monolitica.

VMM ospitato: eseguito sul SO host, accede alle risorse hardware tramite le chiamate di sistema del SO host. Interagisce con quest'ultimo tramite l'ABI ed emula l'ISA di hardware virtuale per i SO guest. I vantaggi sono che può usare il SO host per gestire le periferiche ed utilizzare servizi di basso livello (cheduling delle risorse) e che SO guest non va modificato. Uno svantaggio è il degrado delle prestazioni rispetto al VMM di sistema.

### 5.3.1. Virtualizzazione completa o paravirtualizzazione

Sono le due modalità di dialogo tra la VM ed il VMM per l'accesso alle risorse fisiche (come quindi vengono gestite le istruzioni privilegiate).

#### Virtualizzazione completa:

- Il monitor della macchina virtuale presenta ad ogni macchina virtuale interfacce hardware simulate funzionalmente identiche a quelle della sottostante macchina fisica.
- Il VMM intercetta le richieste di accesso privilegiato all'hardware e ne emula il comportamento atteso.

Vantaggi: non occorre modificare il SO guest e viene garantito un isolamento completo fra le diverse macchine virtuali

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

Svantaggi: il layer di virtualizzazione è più complesso e per avere delle implementazioni più efficienti abbiamo bisogno di assistenza da parte del processore

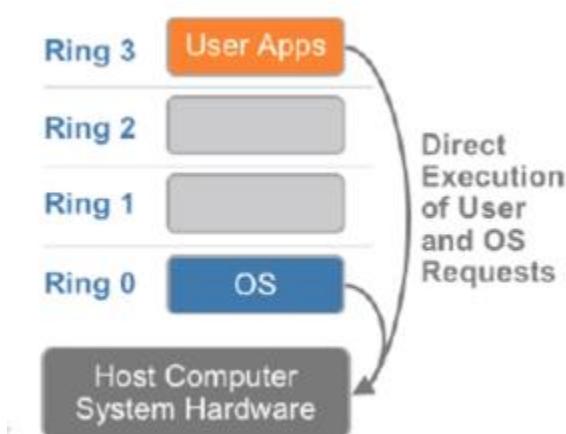
#### Paravirtualizzazione:

- Il monitor della macchina virtuale espone alle VM un interfaccia hardware simulata funzionalmente simile ma non più identica a quella della sottostante macchina fisica.
- Non viene emulato l'hardware, ma viene creato uno strato minimale di software (Virtual Hardware API) per assicurare la gestione delle VM ed il loro isolamento.

### 5.3.2. Problemi per realizzare la virtualizzazione di sistema

L'architettura del processore opera secondo almeno 2 livelli (ring) di protezione: supervisor e user

- Ring 0: privilegi massimi
- Ring 3: privilegi minimi



#### Con la virtualizzazione:

- il VMM opera in supervisor mode (ring 0)
- Il SO guest si trova ad operare in un ring che non sarebbe a lui proprio, perché opererebbe nel ring 1 (le user apps restano nel ring 3).
- **Problema del ring deprivilegging**, ovvero il SO guest opera in un ring che non è proprio e non può eseguire istruzioni privilegiate
- **Problema del ring compression**, ovvero le applicazioni e SO guest eseguono allo stesso livello ed occorre proteggere lo spazio del SO.

### 5.3.3. Virtualizzazione completa: soluzioni

Soluzioni al ring deprivilegging:

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

- **Istruzioni privilegiate:** Meccanismo trap and emulate:

Quando il SO guest tenta di eseguire un'istruzione privilegiata occorre notificare un'eccezione (trap) al VMM e trasferirgli il controllo. A questo punto lui controlla la correttezza dell'operazione e ne emula il comportamento.

- **Istruzioni non privilegiate** eseguite dal SO sono invece eseguite direttamente.

### 5.3.3.1. Trap and emulate

Il meccanismo consiste nell'invio all'hypervisor di un'eccezione nel momento in cui il SO tenta di eseguire un'azione privilegiata, in modo che esso (l'hypervisor) prenda in gestione istruzione privilegiata, emulando il comportamento dell'istruzione.

Tale meccanismo può essere realizzato sia a livello hardware che software, con differenze di prestazioni.

- A livello hardware → la macchina fisica deve fornire un supporto per la virtualizzazione. Ci aspettiamo che sia la soluzione che permette di avere prestazioni migliori, poiché introduce un overhead minore per la gestione del meccanismo.
  - se il processore fornisce supporto alla virtualizzazione → **hardware-assisted CPU virtualization**
- A livello software → unica soluzione possibile nel caso in cui ho a disposizione un processore che non fornisce supporto hardware alla virtualizzazione.
  - La soluzione si basa su un meccanismo di traduzione delle istruzioni, per cui a livello software viene introdotta nel codice macchina una chiamata a hypervisor nel momento in cui il SO guest deve eseguire istruzione privilegiata → **fast binary translation**.

### **Hardware-assisted CPU virtualization**

Soluzione di virtualizzazione completa che richiede per la realizzazione del meccanismo di trap and emulate il supporto hardware da parte del processore dell'host fisico su cui sto installando il sistema di virtualizzazione.

Processori che supportano questa soluzione hanno una V nella sigla per indicare virtualization.

Si introducono 2 forme di modi operativi della CPU, detti **root-mode** e **non-root-mode**, ognuno dei quali supporta tutti e 4 gli anelli di protezione introdotti nell'architettura x86.

- Il monitor della macchina virtuale opera in *root mode* con ring 0
- Il SO guest della macchina virtuale opera in *non root mode* nel ring 0.

Così SO guest è in esecuzione nel ring che gli è proprio (lo 0), ed è in esecuzione con i privilegi di *non root mode*, così nel momento in cui il SO guest esegue l'istruzione privilegiata, tramite il supporto hardware per la virtualizzazione viene inviata notifica al monitor della macchina virtuale, che si fa carico di gestire istruzione privilegiata.

Poiché il SO guest opera nel ring 0, **risolviamo i problemi** secondo i quali il SO guest opera nel ring che gli è proprio (lo 0) e di conseguenza abbiamo risolto il problema di compressione del ring, poiché il SO guest non si troverà più a operare nello stesso ring delle applicazioni utente.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

Questo meccanismo mette a disposizione dell'hypervisor delle strutture dati memorizzate nella memoria RAM.

### **Fast binary translation**

Come virtualizzare un'architettura che non supporta il meccanismo di trap a livello hardware?  
Con architettura Intel a 32 bit.

Si usa una soluzione realizzata a livello software detta **fast binary translation**:

Questa soluzione introduce un meccanismo di eccezione all'interno del codice eseguito dal SO guest.

Il monitor della macchina virtuale VMM scansiona il codice che deve essere seguito prima dell'esecuzione del codice, e sostituisce blocchi di codice contenenti istruzioni privilegiate con blocchi funzionalmente equivalenti che contengono istruzioni per la notifica di eccezioni al VMM.

Nel momento in cui il SO guest deve gestire istruzione privilegiata, il monitor della macchina virtuale se ne fa carico.

Operazioni che non richiedono esecuzioni di istruzioni non privilegiate vengono eseguite direttamente.

Problemi:

L'overhead per scansionare il codice introduce un degrado delle prestazioni, legato all'esigenza di dover scansionare il codice prima della sua esecuzione, per effettuare il meccanismo di traduzione.

Per ridurre il degrado delle prestazioni, i blocchi tradotti vengono conservati in una cache per eventuali riusi futuri, per evitare che vengano tradotti di nuovo.

Questo provoca una maggiore complessità dell'hypervisor e un aumento del numero di righe di codice per hypervisor.

Il layer diventa più spesso rispetto a quello di cui abbiamo bisogno per hypervisor che usa supporto hardware alla virtualizzazione.

### **5.3.4. Paravirtualizzazione**

È un altro meccanismo per realizzare la virtualizzazione di sistema.

Il layer di virtualizzazione espone ai SO guest un'interfaccia che non è più funzionalmente identica a quella del livello hardware sottostante, ma diventa funzionalmente simile.

Così la soluzione di virtualizzazione realizzata con paravirtualizzazione non sia più trasparente rispetto ai SO guest, in quanto il SO guest per poter colloquiare con il layer di virtualizzazione, deve essere modificato per andare a invocare API di virtualizzazione che viene esposta da hypervisor.

Usando questa API le istruzioni eseguite dal SO guest che non sono virtualizzabili vengono sostituite da chiamate a questa API virtualizzata esposta da hypervisor. Chiamate dette **hypercalls**.

Hypercalls:

Con hypercalls viene realizzato il meccanismo di trap and emulate per richiedere il supporto da parte del layer di virtualizzazione.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

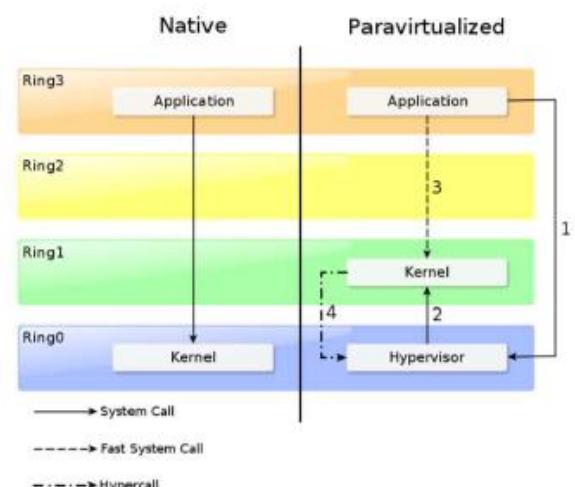
Sono una specie di chiamata di sistema effettuata dal SO guest verso l'hypervisor: il SO guest invece di eseguire l'istruzione privilegiata, invoca con hyper calls il layer di virtualizzazione.

Le hyper calls sostituiscono istruzioni eseguite dal SO guest non virtualizzabili.

Esecuzione di hypercall:

Native: abbiamo direttamente la chiamata di sistema da applicazione che opera nel ring 3 al kernel che opera nel ring 0.

Paravirtualizzazione: nel momento in cui applicazione in esecuzione nella macchina virtuale richiede una chiamata di sistema al SO guest che richiede istruzione privilegiata, il flusso di controllo viene passato a hypervisor, che si fa carico della gestione e passa poi il controllo al kernel del SO guest.



### 5.3.4.1. Vantaggi e svantaggi paravirtualizzazione

**Vantaggi** (rispetto a virtualizzazione completa):

Con i 2 meccanismi trap and emulate realizzati a livello hardware/software:

1. Con **paravirtualizzazione** la chiamata **hypercall** a hypervisor si realizza più facilmente rispetto al **binary translation**, e introduce un **overhead ridotto**, poiché non serve tradurre blocchi di codice per introdurre meccanismo di trap a hypervisor, che è il problema che fa sì che traduzione sia lenta.
2. Rispetto alla virtualizzazione con supporto hardware: la paravirtualizzazione non richiede il supporto da hardware, ma rappresenta un meccanismo per realizzare un hypervisor che non richiede un hardware di ultima generazione o un hardware che supporti la virtualizzazione (non tutti processori supportano in hardware la virtualizzazione).

**Svantaggi:**

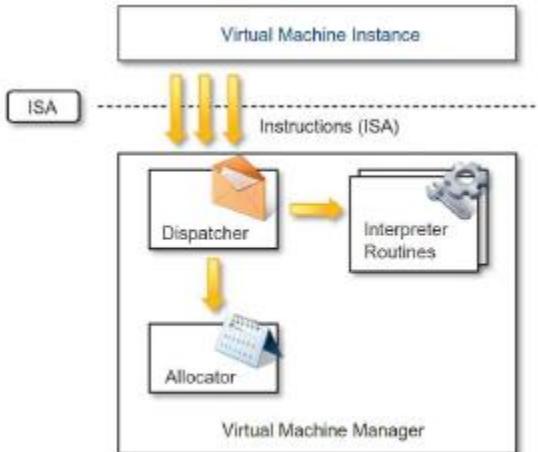
1. Per realizzare il meccanismo hypercall dobbiamo modificare il SO guest. Si richiede quindi che il **codice sorgente dei SO** che vengono paravirtualizzati, ossia resi in grado di operare su un layer che usa la paravirtualizzazione, **deve essere disponibile per poterlo modificare**.
  - I sistemi operativi che non possono essere portati (ad esempio Windows) possono utilizzare driver di dispositivo ad-hoc che rimappano l'esecuzione di istruzioni critiche all'API virtuale esposta dal VMM.
2. Modificando il codice sorgente dei SO guest per introdurre il meccanismo di hypercall c'è anche un **costo di manutenzione**. I sistemi paravirtualizzati non sono più in grado di essere eseguiti direttamente su hardware (bare metal).

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

## 5.4. Architettura di riferimento VMM

Tre componenti hypervisor

- **Dispatcher:** punto di ingresso VMM che reindirizza l'istruzione privilegiata emessa dalla VM a uno degli altri due moduli. Si interfaccia con istanze di macchine virtuali in esecuzione e distribuisce le istruzioni di richieste, ossia è un'interfaccia per gestione istruzioni privilegiate. Ha il compito di effettuare dispatching di istruzioni privilegiate verso le altre due componenti del sistema (routine e allocator).
- **Allocatore** (o scheduler): decide le risorse di sistema da fornire alla VM. Decide come le risorse del sistema fisico (risorse hardware) vengano assegnate alle macchine virtuali in esecuzione su hypervisor.
- **Interprete:** esegue una routine appropriata quando la VM esegue un'istruzione privilegiata



Lo scheduler realizzato da allocator introduce un layer addizionale di scheduling, in quanto nell'ambiente tradizionale non virtualizzato siamo abituati a scheduler della CPU che schedula l'assegnazione CPU tra diversi processi in esecuzione.

Con virtualizzazione aggiungiamo un layer di scheduling, poiché dobbiamo assegnare CPU virtuali viste dai SO guest sulle CPU fisiche che abbiamo a disposizione nella macchina fisica.

Lo scheduler all'interno del SO guest schedula processi eseguiti nel SO guest sulle CPU virtuali che vengono viste da macchina virtuale, mentre scheduler dell'hypervisor schedula le CPU virtuali sulle CPU fisiche a disposizione → indirezione ci permette di risolvere qualunque problema in informatica, ma (in questo caso) introduce un overhead legato alla presenza di un ulteriore layer di scheduling per gestione di risorse hardware.

## 5.5. Virtualizzazione della memoria

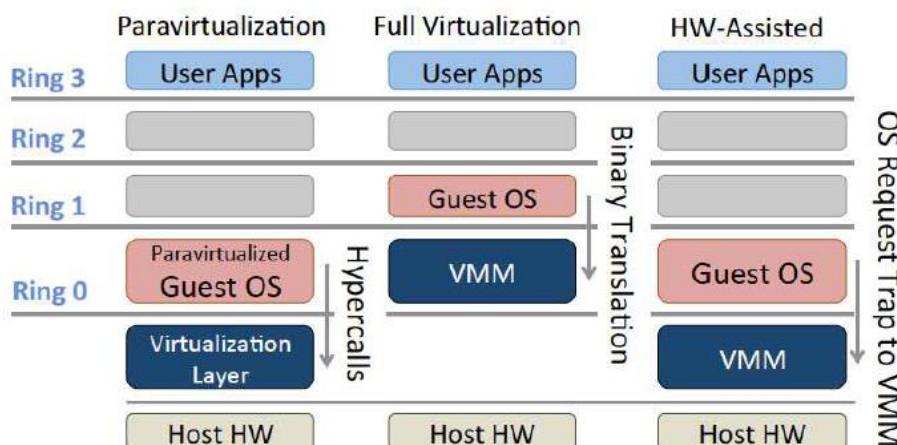
- In un ambiente non virtualizzato: Distinzione memoria fisica- memoria virtuale.
  - I processi vedono la memoria virtuale, e con il meccanismo di traduzione da memoria virtuale a fisica fornito da meccanismo di paginazione abbiamo traduzione/mapping da memoria virtuale a memoria fisica.
  - Nell'architettura dell'allocatore abbiamo 2 componenti hardware dedicati a ottimizzare le prestazioni del meccanismo di gestione della memoria virtuale, ossia sono **Memory Management Unit MMU** e meccanismo della **Translation Lookaside Buffer TLB** (traduzione al livello hardware) per ridurre l'aggravio delle prestazioni legato a introduzione del concetto di memoria virtuale.
    - Mappatura a un livello: dalla memoria virtuale alla memoria fisica fornita dalle tabelle delle pagine

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

- Componenti hardware MMU e TLB per ottimizzare le prestazioni della memoria virtuale
- In un ambiente virtualizzato
  - La situazione si complica, poiché abbiamo in esecuzione sulla stessa macchina fisica molteplici macchine virtuali, che vanno a condividere tra loro la stessa memoria fisica che la macchina ha a disposizione. **Tutte le VM condividono la stessa memoria della macchina e VMM deve partizionarla tra le VM.**
  - Hypervisor deve partizionare memoria fisica a disposizione tra le diverse macchine virtuali, e c'è la necessità di introdurre un ulteriore livello di mapping, in modo che vi sia al interno della macchina virtuale una traduzione da memoria virtuale del SO guest a memoria fisica del SO guest a memoria fisica dell'host. **Mappatura a due livelli: dalla memoria virtuale guest alla memoria fisica guest alla memoria fisica host.**
  - Terminologia:
    - Memoria virtuale guest: memoria che è visibile alle applicazioni in esecuzione all'interno della macchina virtuale. Le applicazioni vedono lo spazio di memorizzazione virtuale che viene presentato dal SO guest alle applicazioni.
    - Memoria fisica del guest: memoria fisica visibile al SO guest.
    - Memoria fisica dell'host: memoria che abbiamo effettivamente a disposizione sulla macchina fisica.

### 5.5.1. Differenze tra approcci pre realizzare esecuzione istruzioni privilegiate:

#### Summing up the different approaches



Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

### 5.5.1.1. Shadow page table

**Tabella pagine del SO:** traduce indirizzi virtuali in fisici. Meccanismo supportato a livello hardware dal TLB (cache usata per rendere meccanismo di traduzione più veloce).

**Tabella pagine dell'hypervisor:** Stessa idea della tabella delle pagine sfruttata all'interno dell'hypervisor, che mantiene una Shadow page table. Per evitare un calo insopportabile delle prestazioni dovuto alla mappatura della memoria aggiuntiva, VMM mantiene le tabelle delle pagine shadow (SPT) e le utilizza per accelerare la mappatura.

Tale tabella ci permette di evitare 1 livello di mapping, poiché contiene mapping tra indirizzo virtuale usato da guest a indirizzo fisico host → **mappatura diretta dell'indirizzo da virtuale guest a fisico host.**

Tale tabella mappa indirizzo virtuale del guest in indirizzo fisico dell'host.

Per ogni fame di memoria fisica del SO guest, il monitor della macchina virtuale lo deve mappare in un corrispondente frame di memoria fisica, e mantiene mapping nella tabella delle pagine shadow.

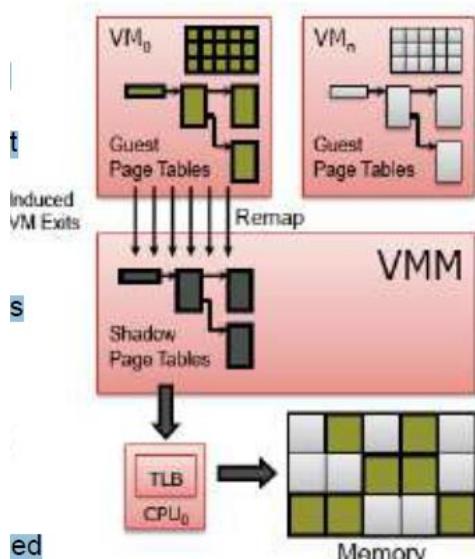
**Problema:**

Mantenere consistente informazione memorizzata all'interno della tabella delle pagine shadow nel momento in cui il SO guest va a fare operazioni che cambiano tabella pagine.

L'hypervisor deve aggiornare tabella pagine shadow per far sì che il link continui ad essere valido. Questo è overhead legato a gestione delle tabelle pagine shadow.

**Come si realizza meccanismo tabella pagine shadow?**

Richiede gestione meccanismo traduzione degli indirizzi → bisogna dare al SO guest l'illusione di avere a disposizione spazio di memoria fisica che inizia da 0 (questo è ciò a cui è abituato SO guest), però nell'ambiente virtualizzato lo spazio di indirizzamento fisico effettivo nella memoria dell'host può essere discontinuo, quindi da una parte il sistema di virtualizzazione deve fare questo mapping. Mantenere



**Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.**

illusione che spazio per macchina verde e bianca inizi da 0, quando in realtà questo spazio può essere allocato in modo discontinuo.

Esigenza di mantenere aggiornate tabelle shadow rispetto alle page table nei SO guest. Monitor della VM deve intercettare operazioni di paginazione richieste dai SO guest e deve via via costruirsi e mantenere aggiornata contenuto delle tabelle pagine shadow.

Meccanismi gestione memoria introduce overhead non trascurabili ,poiché c'è passaggio da SO guest a hypervisor che introduce overhead. Le tabelle delle pagine shadow sono in memoria, quindi occupano spazio di memoria sulla memoria totale a disposizione, e tali tabelle possono occupare porzioni di memoria fisica non trascurabile, e bisogna mantenere consistente l'informazione sul mapping nella tabella pagine shadow.

### **Supporto hardware per la virtualizzazione della memoria**

Per ridurre parte di overhead si considera implementazione che richiede supporti hardware: realizzata introducendo un meccanismo di SLAT.

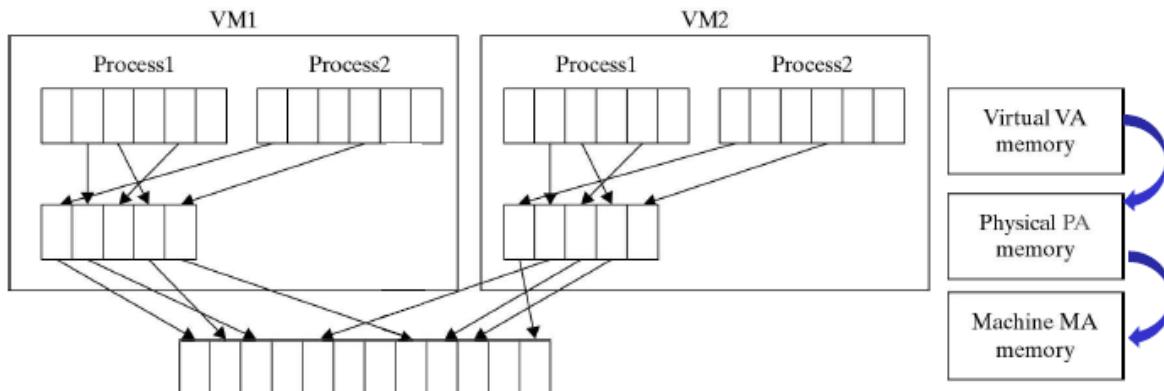
Soluzione hardware per virtualizzazione memoria , che viene introdotta ad hoc nelle architetture hardware che supportano virtualizzazione.

Lo scopo del meccanismo è realizzare in hardware le shadow page table, ossia il meccanismo di traduzione da indirizzo virtuale visto dal SO guest a indirizzo fisico host, bypassando traduzione intermedia da indirizzo virtuale guest a indirizzo fisico guest a indirizzo fisico host.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

## 5.5.2. Mappatura a due livelli

Rispetto all'unico livello di mapping da indirizzi virtuali a fisici nell'ambiente non virtualizzato, con virtualizzazione della memoria introduciamo un ulteriore livello di traduzione rappresentato:



- Il passaggio dalla memoria virtuale guest alla memoria fisica host richiede una mappatura della memoria a due livelli:  
Guest VA (indirizzo virtuale) → guest PA (indirizzo fisico) → host MA (indirizzo macchina)

- Indirizzo fisico ospite ≠ indirizzo macchina host

I processi in esecuzione nelle singole macchine virtuali usano indirizzi di memoria virtuali nel SO guest che dovranno essere tradotti dal meccanismo di gestione della memoria virtuale presente nel SO guest in indirizzi fisici visti dal SO guest.

Indirizzi fisici visti dal SO guest dovranno essere poi tradotti da hypervisor in indirizzi fisici della memoria del SO host (negli effettivi indirizzi fisici dell'host). Gli indirizzi fisici visti dai SO guest sono diversi dall'indirizzo fisico dell'host.

Se devo implementare questo meccanismo, per ogni traduzione di indirizzo ho in realtà 2 diverse traduzioni.

Per evitare un degrado di prestazioni a causa del secondo livello di mapping degli indirizzi introdotto con la virtualizzazione della memoria, l'hypervisor mantiene una sorta di sua tabella delle pagine.

## 5.6. Caso di studio: Xen

È l'esempio più notevole di paravirtualizzazione, sviluppato presso l'Università di Cambridge.

- Hypervisor open source di tipo 1 (sistema VMM) con design microkernel
- Offre al SO guest un'interfaccia virtuale (API hypercall) a cui il SO guest deve fare riferimento per accedere alle risorse fisiche della macchina
- Supporta sia la paravirtualizzazione (PV) che la virtualizzazione assistita da hardware (HVM)

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

### 5.6.1. Pro e contro

#### Pro

1. Rispetto alle altre soluzioni di virtualizzazione di sistema ha un hypervisor sottile
  - 300.000 righe di codice su architetture x86, 65.000 su Arm
  - Occupa poca quantità di memoria → ingombro e interfaccia ridotti (circa 1 MB di dimensione)
  - Scalabile: fino a 4.095 CPU host con 16 TB di RAM. Avendo a disposizione un hardware non alla portata di tutti con ad esempio 16 TB di RAM si riescono ad ospitare circa 4K macchine virtuali.
  - Più robusto e sicuro di altri hypervisor, poiché espone superficie di attacco minore, avendo un numero minore di linee di codice.
  - Ma ancora vulnerabile agli attacchi
2. Miglioramento continuo
3. Flessibilità nella gestione
  - Ottimizzazione delle prestazioni
4. Basso overhead (entro il 2%) rispetto alla macchina bare metal senza virtualizzazione
5. Supporta la migrazione live delle VM

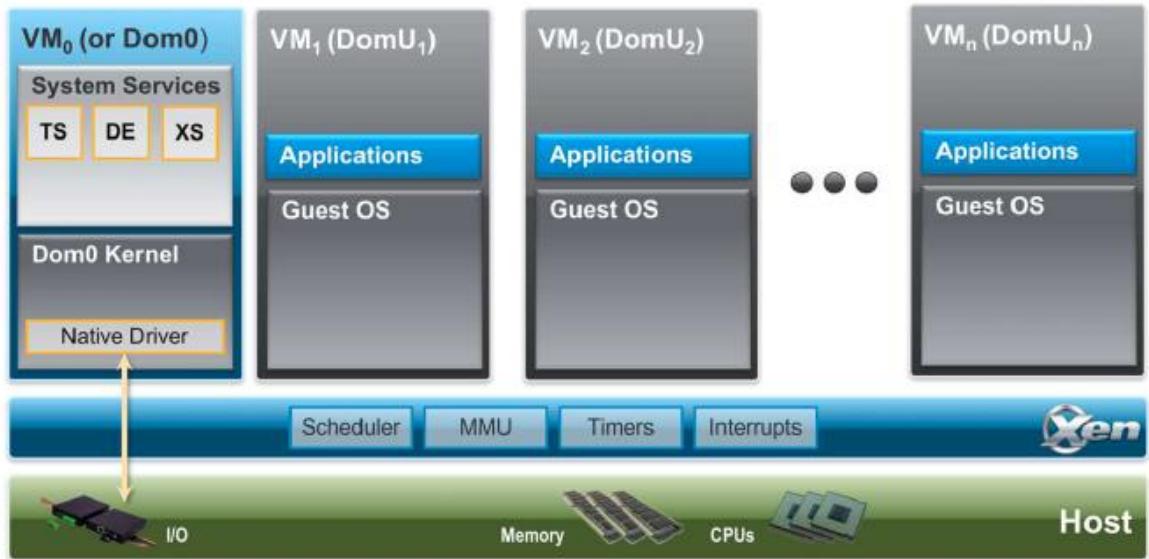
#### Contro

Le prestazioni di I / O rimangono ancora impegnative

### 5.6.2. Architettura Xen

- Obiettivo del gruppo Cambridge (fine anni '90):
  - Progettare un VMM in grado di scalare a circa 100 VM che eseguono applicazioni e servizi senza alcuna modifica all'ABI
- Design microkernel
- Cosa può essere paravirtualizzato con Xen?
  - Disco e dispositivi di rete
  - Interrupts e timer hardware
  - Scheda madre emulata e avvio (boot) legacy
  - Istruzioni privilegiate e tabelle delle pagine (accesso alla memoria)
    - Le istruzioni privilegiate emesse da un SO guest vengono sostituite da hypercall

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.



#### Architettura Xen: hypervisor

- Responsabile della pianificazione, gestione della memoria, interrupt e controllo del dispositivo. Schedula le CPU fisiche dell'host tra le diverse VM in esecuzione, gestisce la memoria, controlla i dispositivi e gestisce gli interrupt. Mantiene informazioni per ogni dominio che viene ospitato al di sopra di XEN, e per le diverse CPU virtuali viste dai domini.
- Gestione delle informazioni per dominio e per CPU virtuale

I **domini** sono istanze di macchine virtuali. Ogni dominio ha il proprio SO guest con relative applicazioni. Ogni dominio è eseguito su CPU virtuali; lo scheduling dalle CPU virtuali alle CPU fisiche nell'host è gestito da un componente di scheduling all'interno di Xen.

I **domini U** (unprivileged) sono i domini all'interno dei quali sono eseguiti i SO guest. Questi domini sono completamente isolati dall' hardware, non hanno privilegi per accedere all'hardware o usare funzionalità di I/O.

Dominio 0 --> (dominio di controllo): dominio speciale dedicato all'esecuzione di funzioni di controllo Xen e istruzioni privilegiate.

- Obbligatorio-> viene avviato da Xen all'avvio
- Contiene i driver per tutti i dispositivi e alcuni servizi (DE, XS, TS)
- Privilegi speciali: può accedere direttamente all'hw e alle funzioni di I / O del sistema, può interagire con altre VM

Ha i privilegi --> può accedere direttamente a hw (freccia gialla immagine), e può interagire anche con le altre VM.

- Implementato come archiviazione gerarchica di valori-chiave (xen memorizza info in forma di chiave valore), e se cambia un valore nello spazio di storage, c'è una funzione di watch che notifica i driver in ascolto sullo spazio di storage i ambienti della chiave.
- Quando i valori vengono modificati in store, una funzione di orologio notifica agli ascoltatori (ad esempio, i conducenti) le modifiche della chiave a cui si sono abbonati
- Comunica con le VM guest tramite la memoria condivisa utilizzando i privilegi Dom0

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

- **Toolstack:** consente al dom0 di gestire il ciclo di vita della VM (creazione, spegnimento, pausa, migrazione)
- Per creare una nuova VM, un utente fornisce un file di configurazione che descrive le allocazioni di memoria e CPU (quantità di mem e numero di CPU virtuali allocate alla VM) e le configurazioni del dispositivo.
- Toolstack analizza questo file (parsing) e scrive queste informazioni in XenStore, che tiene emmorate le info relative alla gestione delle varie macchine virtuali in esecuzione.
- Sfrutta i privilegi di Dom0 per mappare la memoria del guest, caricare il kernel e il BIOS virtuale e impostare i canali di comunicazione iniziali con XenStore e con la console virtuale quando viene creata una nuova VM.

### Architettura Xen e gestione del sistema operativo guest

L'hypervisor Xen viene eseguito nella modalità più privilegiata e controlla l'accesso dei SO guest all'hw sottostante

- I domini vengono eseguiti nell'anello 1
- Applicazioni nel ring 3

Foto slide 49.

### CPU scheduler in Xen

Ha il compito di decidere le CPU virtuali viste dalle VM, quali sono quelle che verranno eseguite sulla macchina fisica.

Introduce ulteriore livello di scheduling. Xen ci permette di scegliere tra diverse politiche di scheduling delle CPU.

Scheduler di default in xen: **credit scheduler**. Alle CPU virtuali assegnati crediti che si possono spendere nell'utilizzare CPU fisiche, e a mano a mano il credito diminuisce.

L'obiettivo dello scheduler è assegnare quote eque di CPU alle diverse macchie guest e fare in modo che la CPU fisica sia sempre utilizzata.

A ogni VM associato un peso e un parametro cap configurabili:

- Il peso rappresenta una quantità di allocazione della CPU per dominio. Di default è 256.
- Cap: quantità max di CPU che un dominio può usare. Se cap=0 --> CPU virtuale, se altre CPU non svolgono lavoro, può ricevere più CPU rispetto a quella di cui ha diritto. Se il cap viene modificato --> limito/metto upper bound sulla quantità di CPU che una CPU virtuale può ricevere. Se cap = 100 --> può ricevere CPU fisica, se cap = 50 riceve al max mezza CPU fisica.

Per ogni CPU fisica, lo scheduler mantiene code per diverse CPU virtuali, e da queste code sceglie quella da schedulare che ha il valore del credito più basso. Le code vengono mantenute ordinate in base al valore del credito. Il carico è inoltre bilanciato tra le molteplici CPU fisiche.

## 5.7. MIGRAZIONE E RESIZING

Le tecniche di virtualizzazione si sono sviluppate, riducendo i costi della virtualizzazione a livello di sistema. Ci sono tuttavia ancora degli overhead non trascurabili per esempio quando ci sono più VM sulla stessa macchina. A creare overhead maggiori sono le VM con alto traffico di rete.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

Analizziamo la portabilità delle macchine virtuali.

**VM image:** copia della macchina virtuale, che contiene sistema operativo, file di dati ed applicazioni. È l'immagine di un'istanza di esecuzione della macchina virtuale.

Il problema che si pone è come fare a facilitare la portabilità di VM tra diversi ambienti di virtualizzazione, quindi fra diversi hypervisor (come evitare il vendor lock in).

Per affrontare il problema è stato definito un formato aperto di virtualizzazione chiamato OVF (Open Virtualization Format). Ciò che cercano di fare è definire un formato XML nel file di configurazione della VM che sia supportato da diverse macchine virtuali.

Due strumenti utili per le macchine virtuali:

### 5.7.1. Resizing

Cambiare a runtime la tipologia e la quantità di risorse hardware viste dalla macchina virtuale.

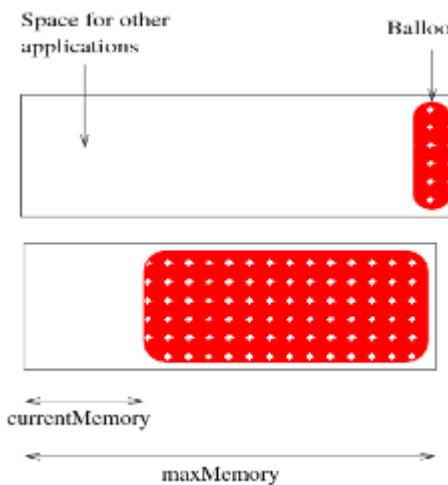
È un meccanismo di controllo a grana fine delle caratteristiche di una VM rispetto alla migrazione.

Un caso d'uso è quello in cui la VM ha finito la memoria RAM a disposizione e vogliamo allocarne di più a runtime.

Vantaggi sono che è una soluzione a basso costo, ed è più veloce di effettuare un effettivo reboot della macchina dove viene modificato il file di configurazione (per allocare più memoria). Lo svantaggio è che non è supportata da tutti i prodotti di virtualizzazione.

Senza spegnere la macchina è possibile effettuare il resizing solo nei due modi seguenti:

- Aggiungere o rimuovere CPU virtuali
- Aumentare o diminuire memoria a disposizione: meccanismo del **memory ballooning**, che permette di gonfiare o sgonfiare un "palloncino" che toglie più o meno spazio alla memoria virtuale facendo il swap delle pagine dalla memoria sul disco (ovviamente quindi il palloncino avrà una dimensione predefinita che per cambiare bisogna riavviare la macchina).



Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

## 5.7.2. Migrazione

Possibilità di spostare una macchina virtuale in esecuzione su diverse macchine fisiche in esecuzione (per esempio tra due data center in esecuzione in luoghi geograficamente diversi senza impatto sull'utente che non se ne deve accorgere). I vantaggi della migrazione è che è molto utile nei cluster e data center virtuali per consolidare l'infrastruttura, avere flessibilità nel failover, bilanciare il carico. Di contro gli svantaggi sono il poco supporto da parte del VMM, un overhead di migrazione non trascurabile ed una migrazione in ambito WAN non facile da gestire. Approcci per migrare le istanze di VM tra macchine fisiche sono:

- **Stop and copy**, ovvero si spegne la VM sorgente e si trasferisce l'immagine della VM sull'host di destinazione. Visto che l'immagine della VM può essere grande e la banda di rete limitata, il downtime potrebbe essere troppo lungo.
- **Live migration**, ovvero la VM rimane attiva durante la migrazione

### Live Migration:

Prima di avviare la migrazione live c'è una fase di setup in cui si seleziona quale VM bisogna migrare e su quale host bisogna migrarla. Questa scelta può essere effettuata da soluzione adattativa o da una euristica, tenendo conto di diversi fattori come riduzione del consumo energetico, massimizzazione della condivisione del carico di lavoro, tolleranza ai guasti

- Cosa migrare: la memoria, lo storage o le connessioni di rete.
- Come? In modo trasparente alle applicazioni in esecuzione sulla VM (in modo che applicazioni in esecuzione sulla VM non se ne accorgano)
  - Costo della migrazione live: vi è comunque un downtime dell'applicazione (applicazioni in esecuzione sulla VM non saranno disponibili per un downtime)

### Migrazione dello storage

1. Si può utilizzare uno storage condiviso tra sorgente e destinazione.

Potremmo avere a disposizione:

- SAN (Storage area network) → permette di avere uno storage di rete condiviso dai diversi server, basta che l'immagine della macchina virtuale è nel SAN e si può condividere tra più host
- Un più economico NAS (Network attached server)
- Un file system distribuito → il classico file system dove i file e le directory non sono su un'unica macchina ma distribuite su molteplici macchine sulle quali viene installato il file system distribuito

2. In assenza di uno storage condiviso invece il VMM procede con il salvare tutti i dati della VM sorgente in un file immagine che viene trasferito sull'host di destinazione.

### Migrazione delle connessioni di rete

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

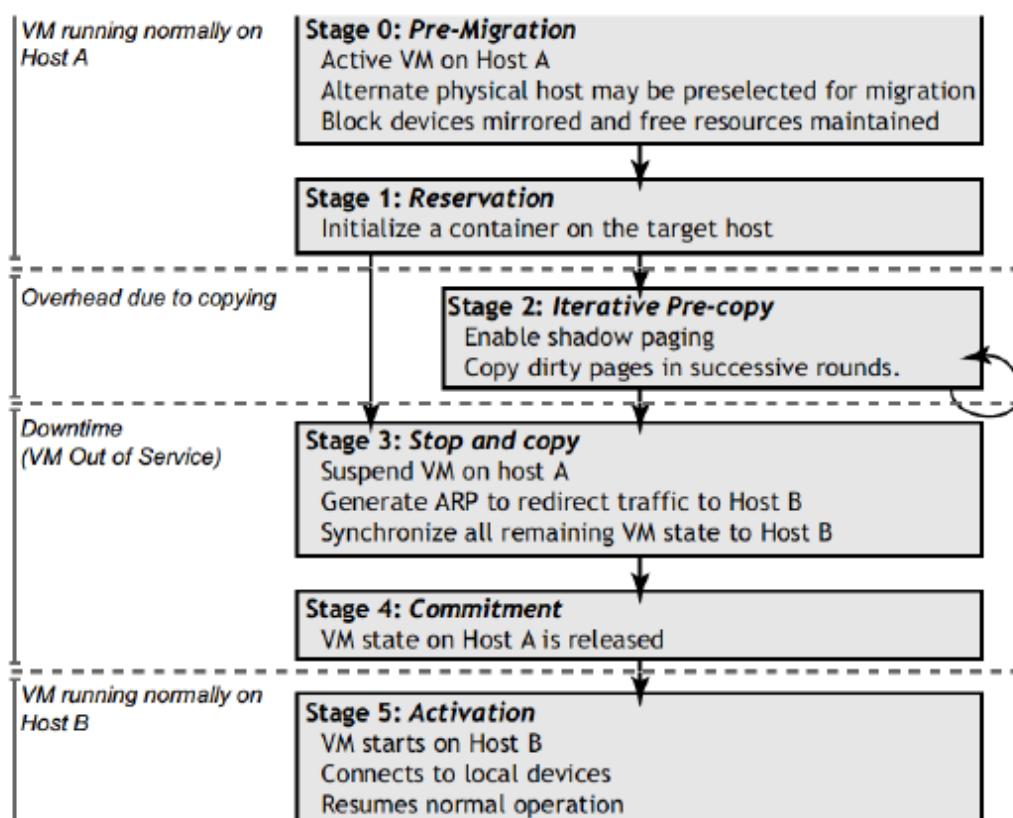
La VM sorgente ha un IP virtuale, e se l'host è sulla stessa rete IP basta cambiare le tabelle di routing. Se sorgente e destinazione sono sulla stessa sottorete IP non occorre fare forwarding sulla sorgente. L'invio di risposta ARP non è richiesto da parte della destinazione per avvisare che l'indirizzo IP è stato spostato in una nuova locazione ed aggiornare quindi le tabelle ARP.

## Migrazione della memoria (inclusi registri della CPU e stato dei device driver)

Richiede il maggior effort per l'hypervisor. Si parla di cache, RAM o qualunque tipo di memoria utilizzata dalla VM in quel momento. Ci sono diversi approcci per effettuare la migrazione della memoria;

### 1)Approccio pre-copy (memoria copiata prima che la VM sull'host venga avviata):

1. 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.
2. Fase di stop-and-copy: la VM sorgente viene fermata e vengono copiate le pagine dirty (stato della CPU e dei device). Il tempo di downtime dipende dalla dimensione della memoria, dal tipo di app e banda di rete.
3. Fasi di commitment e reactivation: la VM di destinazione carica lo stato e riprende l'esecuzione; la VM sorgente viene rimossa.



Source: C. Clark et al., "Live Migration of Virtual Machines", NSDI'05.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

È noto come approccio pre-copy, poiché la memoria è copiata prima che l'esecuzione della VM su host di destinazione sia avviata. Migrazione memoria prima di avviare VM su host di destinazione.

Svantaggi pre-copy

Se abbiamo CPU o memory intensive carichi di lavoro, perché porterebbe ad un downtime troppo lungo, rendendo la migrazione live non più trasparente alle applicazioni.

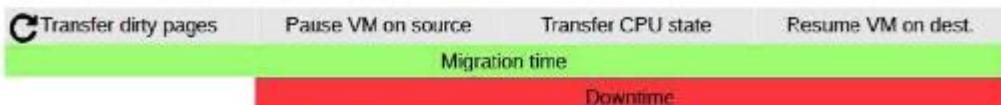
**2)Approccio post-copy:** sposta l'esecuzione sull'host di destinazione all'inizio del processo di migrazione, così che le applicazioni inizino da subito ad andare in esecuzione sull'host di destinazione. Le pagine di memoria vengono trasferite on demand quando vengono richieste dalla VM di destinazione.

Vantaggio: riduce tempo di downtime, poiché in background mente VM e attiva su host di destinazione, trasferisco ciò che mi serve.

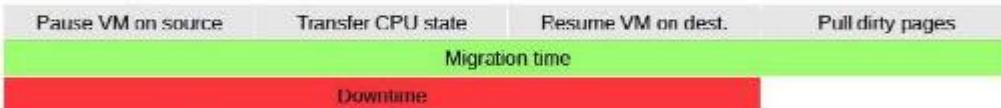
Svantaggio: è una soluzione meno tollerante ai guasti perché la VM primaria deve restare in esecuzione finchè non sono state trasferite tutte le pagine. Se subisse un crash nel tempo di trasferimento sarebbe un problema.

**3)Approccio ibrido:** Caso speciale di post copy migration, in quanto è un post copy preceduto da un precopy stage. L'idea è quella di trasferire le pagine di memoria più frequentemente usate prima che l'esecuzione della VM sia trasferita sull'host, così da ridurre il degrado delle performance quando viene ripresa l'esecuzione.

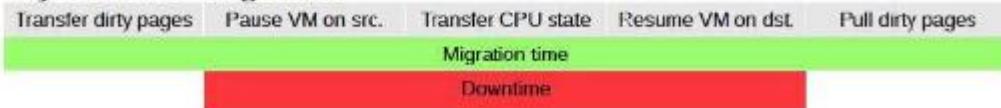
- Pre-copy Live Migration



- Post-copy Live Migration



- Hybrid Live Migration



Courtesy of C.Vojtech, <http://bit.ly/2h7wSWB>

## Migrazione delle VM in ambienti WAN

Come ottenere la migrazione in tempo reale delle VM su più data center geo-distribuiti?

- Key challenge: mantenere la connettività di rete e preservare le connessioni aperte durante e dopo la migrazione
- Supporto limitato in hypervisor open-source e commerciali

## Migrazione dello storage

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

Diverse alternative:

- **Shared storage**: condividere lo storage fra source e host.
  - Svantaggio: il tempo di accesso allo storage può essere troppo lento
- **Fetching on-demand**: vengono trasferiti solo alcuni blocchi sulla destinazione e poi fetchati i rimanenti blocchi dalla sorgente quando richiesto.
  - Svantaggio: tutto non funziona se crasha la sorgente.
- **Pre-copy/write throttling**: viene precopiata l'immagine disco della VM sulla destinazione mentre la VM source continua a runnare, tenendo traccia delle operazioni di write sulla source, per poi applicarle sulla destinazione.
  - Se la velocità di scrittura all'origine è troppo elevata, utilizzare la limitazione di scrittura per rallentare la VM in modo che la migrazione possa procedere

## Rete

Approcci per migrare le connessioni network in WAN sono:

- **IP tunneling**: Viene definito un tunnel fra il vecchio indirizzo ip (host sorgente) ed il nuovo indirizzo ip (host di destinazione). Quando la migrazione dei pacchetti tra sorgente e destinazione è completata e la VM può rispondere nella sua nuova locazione, viene aggiornato il DNS con il nuovo IP. Il tunnel viene rimosso quando non restano più connessioni che usano il vecchio IP. Il contro è che se la source crasha non funziona.
- **Rete privata virtuale (VPN)**
  - Utilizzare la VPN basata su MPLS per creare l'astrazione di una rete privata e uno spazio di indirizzi condiviso da più data center
- **Rete definita dal software**
  - Modificare il piano di controllo, non è necessario modificare l'indirizzo IP!

## 5.8. Virtualizzazione a livello di SO

Permette di eseguire molteplici ambienti di esecuzione fra di loro all'interno di un singolo SO. Tali ambienti sono chiamati container, jail, zone, virtual execution environment (VE).

Ogni container ha il proprio insieme di processi, file system, utenti, interfacce di rete, tabelle di routing ecc; I container condividono il kernel dello stesso SO.

### Meccanismi

Per realizzare i container nei sistemi Unix ci sono i comandi:

- **chroot** (change root directory) che permette di cambiare la directory di riferimento per i processi in esecuzione, quindi di avere diverse root directory per i diversi container.
  - Questo comando da solo non basta per implementare i container all'interno di un SO, ma servono meccanismi di isolamento che permettono di isolare le risorse hardware viste dall'insieme dei processi in esecuzione all'interno dei container (**cgroups**), e

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

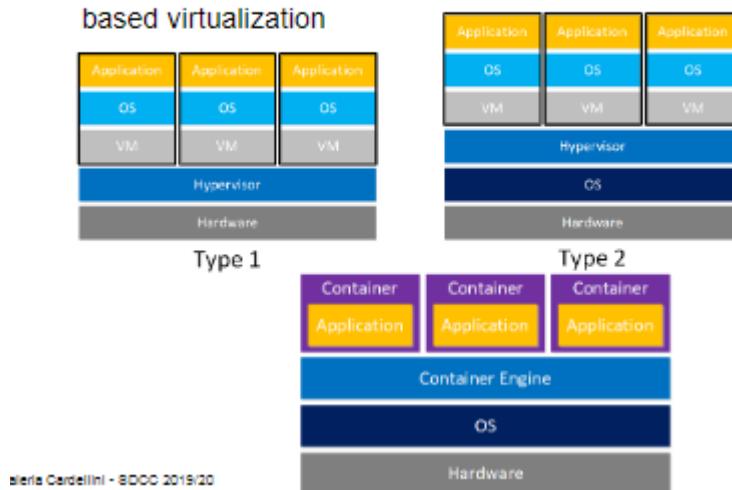
meccanismi che consentono di isolare risorse software viste dai processi in esecuzione all'interno dei container ([namespaces](#)).

- [cgroups](#) (control groups), un meccanismo che limita, misura ed isola l'utilizzo delle risorse hardware viste da un insieme di processi
- [namespaces](#) è un meccanismo che permette di isolare ciò che un insieme di processi può vedere dell'ambiente operativo (PID, NETWORK, USER, MOUNT, IPC e UTS).

## Vantaggi dalla virtualizzazione basata su container (quindi a livello del SO)

### Virtualizzazione a livello di SO: vantaggi

- VMM-based (type 1 and type-2) vs container-based virtualization



Da immagine:

soluzione 1 e 2: stack software e hardware nel caso in cui ho (le 2 in alto) una soluzione di virtualizzazione a livello di sistema usando hypervisor di tipo 1 (bare metal) o di tipo 2, in cui hypervisor in esecuzione su SO host sfrutta SO host per la gestione di risorse hardware.

Soluzione 3: In basso c'è la soluzione di virtualizzazione a livello di SO basata su container, in cui ho stesso kernel del SO host condiviso tra tutti container → layer di virtualizzazione detto **Engine dei container**, si pone al di sopra del SO, e usando i meccanismi precedenti ci dà la possibilità di creare dei container.

**Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.**

Nei singoli container abbiamo applicazioni in esecuzione e librerie e framework di cui le applicazioni necessitano, a differenza dell'immagine delle VM dove nel tipo 1 e 2 avremo il SO, le applicazioni, e tutto ciò di cui le applicazioni hanno bisogno (librerie, framework...).

Dove mettiamo meno tempo a avviare ambiente di virtualizzazione? Nella soluzione 3!

Nella soluzione 3 abbiamo meno overhead, perché per l'esecuzione delle istruzioni privilegiate non c'è bisogno di parlare con l'hypervisor in quanto il kernel del SO è sempre lo stesso, quindi meno overhead.

#### **Vantaggi e svantaggi della virtualizzazione basata su container:**

Le applicazioni possono quindi invocare le chiamate di sistema direttamente del kernel (al di sopra del quale c'è l'engine dei container), cosa che toglie overhead rispetto all'uso di hypervisor. Si può installare un maggior numero di container rispetto alle classiche macchine virtuali, essendo più leggeri. Rispetto alla soluzione classica di hypervisor però i container soffrono di minore flessibilità perché non possiamo eseguire kernel di SO differenti. Possiamo però eseguire distribuzioni diverse di uno stesso SO. I container inoltre soffrono di minore isolamento (tramite bug si poteva accedere a pagine e risorse sulla stessa macchina fisica) e maggior vulnerabilità, perché una singola vulnerabilità nel kernel del SO comprometterebbe l'intero sistema.

Tra i prodotti principali per la virtualizzazione basata su container c'è Docker, FreeBSD, LXC, Virtuozzo. La virtualizzazione basata su container da qualche anno è supportata anche da Windows, oltre che da Linux.

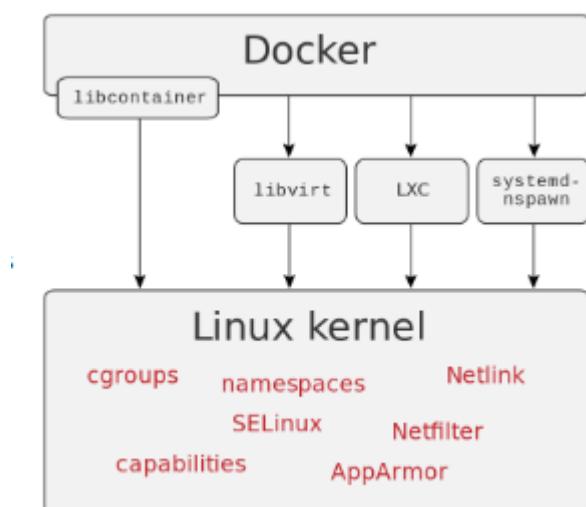
I container forniscono una nuova modalità per andare a costruire, impacchettare, condividere e fare il deployment delle applicazioni. **DevOps**: Development and operation; è una metodologia consistente in una serie di pratiche mirate a ridurre il gap tra sviluppo ed operazioni, garantendo comunicazione e collaborazione, integrazione continua, consegna con sviluppo automatizzato e assicurazione sulla qualità.

I container permettono di realizzare in modo agevole la continua integrazione tra sviluppo e fase operativa; Il ciclo di vita di un container è build – run – share. Docker è uno dei tool fondamentali del DevOps, un altro tool è per esempio Kubernetes.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

### 5.8.1. Docker

Si tratta di un ambiente di virtualizzazione a livello del sistema operativo, un framework open source. Permette di creare dei container che introducono un livello minimale di overhead nelle prestazioni delle applicazioni eseguite nei container rispetto a quelle eseguite bare metal. Molteplici container condividono tra loro il sistema operativo e non sono legati a una specifica infrastruttura. L'aspetto fondamentale è quello che ci sia il layer di istanziazione. Docker è stato sviluppato in Go, indice di un software efficiente. Docker sfrutta i meccanismi offerti dal kernel di linux per la realizzazione dei container (meccanismo dei **cgroups** e dei **namespaces**) e si basa sulla sua libcontainer, una “cross-system abstraction layer aimed to support a wide range of isolation technologies”.



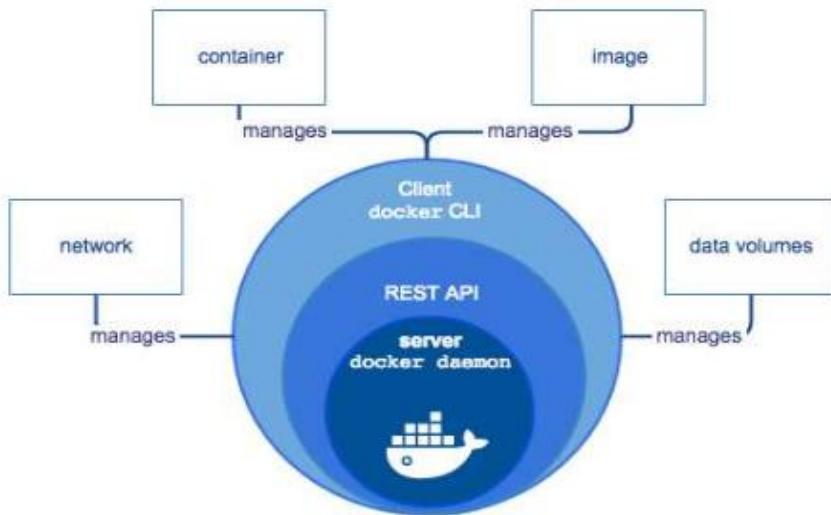
Dal punto di vista di Docker sappiamo che si basa su un'architettura client server; il client a riga di comando invia richieste al demone di docker. Altri aspetti fondamentali di Docker sono la possibilità di definire il contenuto dell'immagine del container tramite un Docker file. I container potranno essere creati da riga di comando specificando una serie di istruzioni nel caso semplice, oppure nel caso complesso (realizzare container a partire da un'immagine) definire l'immagine del container utilizzando un docker file (sequenza di comandi eseguiti in un ordine specificato) per poi chiedere all'engine di docker l'istanziazione vera e propria.

#### Docker Engine

Client server application composta da un server chiamato coker demon, una API REST che specifica le interfacce che i programmi possono usare per controllare ed interagire con i demoni, ed una linea di

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

comando di interfaccia client (CLI). Il server sarà il docker demon, il client sarà l'interfaccia da riga di comando.



Docker utilizza un'architettura client server, dove quindi il client si interfaccia col docker demon permettendoci di costruire dei container. Dalla linea di comando client si potrà eseguire il comando docker build per costruire container eventualmente tramite immagini da parte del docker demon; altri comandi sono docker pull o docker run.

**Docker image:** template di tipo read only utilizzato per creare un'istanza in esecuzione di un container docker. Quindi si tratta dello strumento che ci consente di distribuire applicazioni in tutto il suo ambiente di istanziazione. L'unico requisito è quello di avere la macchina su cui andare ad instanziare l'immagine che possiede il docker engine (deve essere abilitato). Le immagini possono essere pulled e pushed verso un registro che può essere pubblico o privato.

Le docker image vengono create da un Dockerfile (istruzioni per assemblare l'immagine) e un contesto (il contesto è un set di file e librerie che vengono collegate all'immagine); spesso l'immagine è basata su un'altra immagine.

Le istruzioni in un Dockerfile vengono eseguite in ordine.

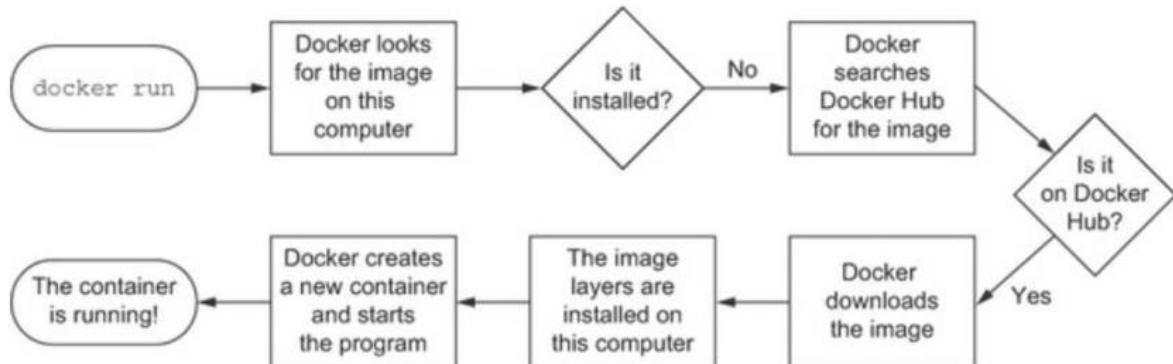
L'immagine di un container in docker consiste in una serie di layer sovrapposti uno all'altro; in particolare docker utilizza i **filesystem union** per combinare i diversi layer che compongono l'immagine del container e forniscono un'unica immagine (grazie allo Union file system). Ciascun layer è identificato da un id e può avere dimensioni differenti (immagini che vanno da kilobyte a megabyte).

I vantaggi del layering nell'immagine di docker è che possiamo condividere e riutilizzare i layer fra le differenti immagini risparmiando spazio di memorizzazione in locale sulla macchina, e si può anche evitare così di scaricare da repository i layer facenti parte di una nuova immagine che stiamo definendo nel caso in cui quei layer siano già istanziati.

**I container devono essere stateless**, idealmente, poiché pochi dati sono scritti sul layer di scrittura del container. I dati devono infatti essere scritti su volumi Docker per essere mantenuti. Pochi carichi di lavoro richiedono di scrivere dati sul layer di scrittura. Lo storage driver controlla come le immagini ed i container sono salvati e gestiti sull'host docker.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

Un docker container è un'istanza runnable di una docker image. L'immagine è la parte build di docker, il container è la parte run. I container sono stateless, quindi quando rimuoviamo il container tutti i dati scritti in lui vengono persi.

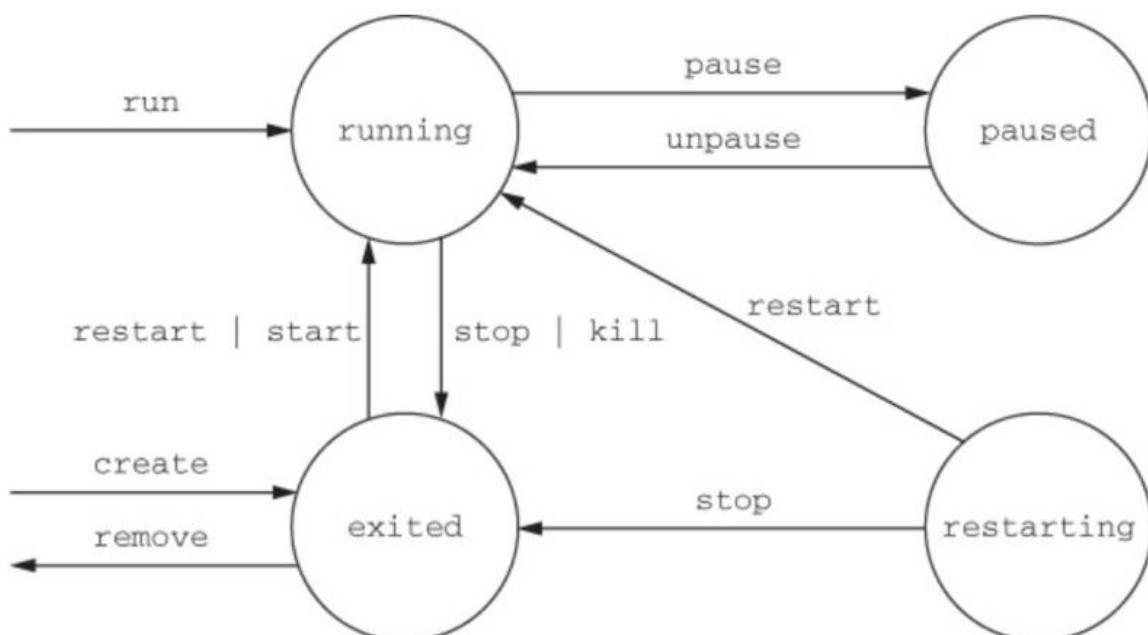


Il docker registry è un'applicazione server side che ci permette di distribuire le immagini.

Docker run command: docker cerca localmente l'immagine nei depositi locali del computer e qualora non sia presente la cerca nel Docker hub.

Su Docker è possibile eseguire molteplici container all'interno della stessa macchina senza dover configurare la rete creandola e collegandoci i container manualmente. Si utilizza anche **Docker Compose** per creare container multipli all'interno di uno stesso host. Per passare a container in esecuzione su host differenti si può fare con l'engine di orchestrazione dei container (Docker swarm e kubernetes).

Docker compose permette di creare più container su uno stesso host. Per specificare la composizione, la si scrive in un file .yml che di default si chiama docker-compose. Per iniziare la composizione e per fermarla si usa il comando docker-compose up e docker-compose down.



Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

I container permettono di raggruppare le applicazioni e le loro dipendenze dentro un singolo package che può essere eseguito ovunque. Vengono usate meno risorse di una VM tradizionale.

Come le macchine virtuali, anche i container possono essere migrati in modo live. Il resizing per uno scale up si può fare andando ad agire sui limiti dei container, mentre la live migration richiede di salvare lo stato, trasferirlo e recuperarlo dall'altra parte.

In docker ci sono 2 tool per la migrazione live, che sono CRIU e P.Haul. Tale supporto risulta assente per le VM, ecco perché risulta più complicata la migrazione (che crea un downtime non trascurabile tra l'altro, non potendo sfruttare approcci di pre copy e post copy).

I container ed il loro sviluppo vengono inclusi nel CaaS (Container as a service) nel cloud computing.

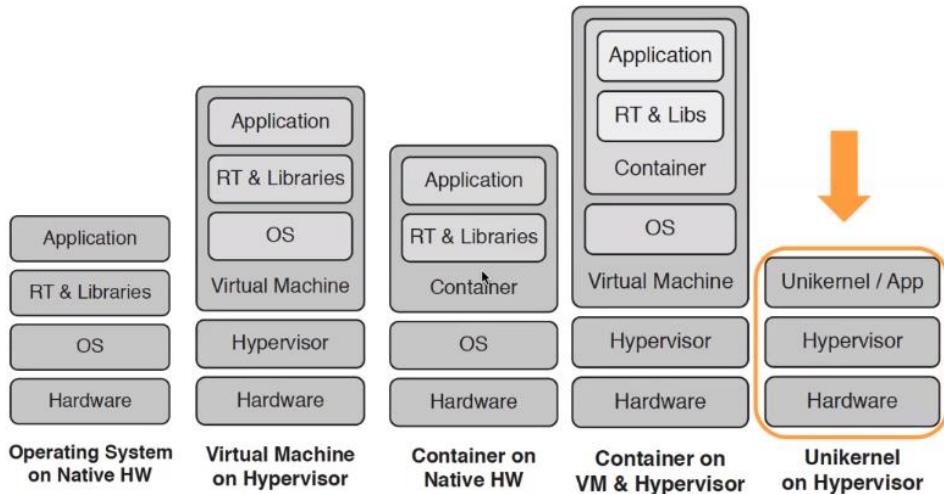
## 5.9. Lightweight operating systems e Unikernel

Abbiamo bisogno di soluzioni di containerizzazioni leggere, che risolvono i problemi di sicurezza e di non completo isolamento dei container. Un esempio è FireCracker che fornisce ad Amazon i container di Lambda per il servizio di serverless computing. Vediamo ora una soluzione più leggera dei container, gli unikernel.

Soluzioni di virtualizzazione:

- Assenza di virtualizzazione: eseguiamo su bare metal
- Virtualizzazione di tipo 1 dove l'hypervisor è direttamente sopra l'hardware e all'interno di ciascuna macchina virtuale possiamo avere un differente OS
- Soluzione dei container dove il container engine è posto al di sopra dell'OS. Nei container abbiamo l'applicazione, le dipendenze e le librerie dell'applicazione.
- Abbiamo visto la soluzione dove un container è istanziato dentro una macchina virtuale, quindi abbiamo di nuovo un hypervisor di tipo 1 con installato l'ambiente di virtualizzazione basato su container all'interno del SO della macchina virtuale. Con questa soluzione abbiamo una maggiore flessibilità in termini di istanziazione di container che utilizzano diversi SO, perché così possiamo differenziare i SO. Abbiamo maggior flessibilità, più sicurezza, pagando un costo in termini di overhead per via dei 2 gradi di indirezione (engine dei container e hypervisor).
- Nell'ultima soluzione, quindi quella dell'unikernel abbiamo uno stack piuttosto ridotto, risultando più leggera rispetto alle classiche VM. Questa ci dà meno limitazioni sulle tipologie di app istanziate sull'unikernel.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.



Il motivo per il quale si cerca di scindere applicazioni monolitiche in microservizi che comunicano tramite sistemi a code di messaggi, o si cerca di realizzare sistemi operativi più leggeri e unikernels è perché in entrambi i casi **si vuole evitare l'overhead del sistema operativo e ridurre la superficie di attacco**. Oltre ciò è anche vero che le soluzioni di virtualizzazioni sono decisamente troppo in certe situazioni, si cercano soluzioni più minimali a livello di tempo di startup e shutdown, di risorse di archiviazione e calcolo (ovviamente con caratteristiche di sicurezza che mancano ai container).

## SO leggeri

Si tratta di sistemi operativi minimali container based, con una architettura kernel monolitica. Esempi sono CoreOS Container Linux. Mentre gli unikernel necessitano l'hypervisor, questi invece sono eseguiti su bare metal.

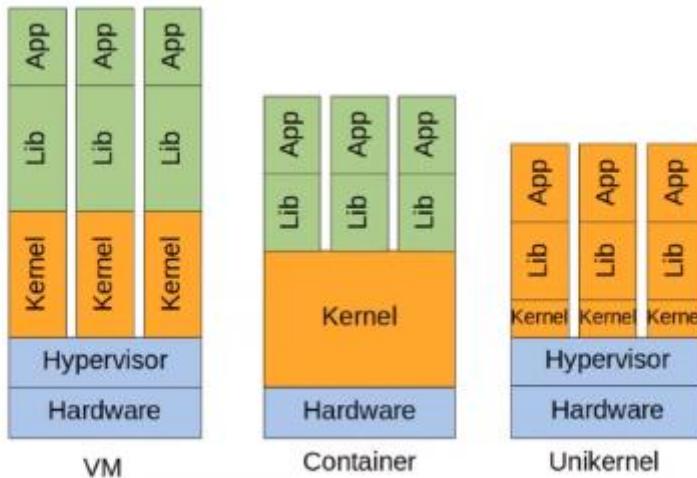
- **CoreOS Container Linux** <https://coreos.com/os/docs/latest/>
  - Smaller, more compact Linux distribution
    - Only minimal functionalities required for deploying apps inside containers, together with built-in mechanisms for service discovery, container management and process management
  - Designed for large-scale deployments, mostly targeting enterprises, with focus on automation, ease of application deployment, security, and scalability
  - Runs also on bare metal servers
  - Merged with Atomic Host (Fedora CoreOS)

## Unikernel (o "sistemi operativi di libreria")

L'idea è quella di avere una macchina virtuale minimale che sia in grado di eseguire una singola applicazione scritta in un unico linguaggio di programmazione. All'interno dell'unikernel si vuole un sistema operativo minimale, un set minimale di libreria che offrono i costrutti di cui necessita l'applicazione

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

eseguita, un unico spazio di indirizzamento. Esempi di Unikernel sono LinuxKit, ClickOS, OSv, IncludeOS, Xen MirageOS.



Vantaggi dell'unikernel:

- Sono ambienti di virtualizzazione con una footprint ridotta al minimo (leggieri e piccoli)
- Veloci (non c'è context switch avendo unico spazio di indirizzamento)
- Sicuri (superficie d'attacco ridotta essendo piccoli)
- Avvio veloce in termine di microsecondi (a differenza dei container che si parla di minuti)

Svantaggi:

- Lavorano solo in presenza di un hypervisor sottostante
- Pochi strumenti di debugging a disposizione
- Singolo linguaggio di programmazione a runtime
  - **OSV** <http://osv.io>
    - Unikernel designed for the Cloud
    - Run only on top of hypervisor (e.g., KVM, Xen, Firecracker)
    - Goals: isolation benefits of hypervisors without overhead of guest OS
    - Requires to build an image by fusing OSv kernel and application files together

Studi sulle performance delle soluzioni di virtualizzazione fra quelle basate su hypervisor e quelle più leggere hanno messo in evidenza come a livello qualitativo che l'overhead prodotto dai container è piuttosto ridotto (veloce istanziazione, basso impatto sulla memoria, alta densità), pagando in termini di sicurezza però.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

### 5.9.1. Orchestrazione dei container

Docker compose ci permetteva di realizzare un'applicazione multi container i cui componenti della stessa sono eseguiti su diversi container. Con Docker compose istanziamo quei container su una stessa macchina fisica. **Se volessimo avere applicazioni multi-container istanziabili all'interno di un cluster geograficamente distribuito c'è il supporto delle piattaforme di orchestrazione dei container.** Queste piattaforme:

- permettono di eseguire su diversi nodi le container packaged application
- permettono di configurare l'ambiente di deployment
- di suddividere i container fra i nodi del sistema distribuiti in località differenti
- di monitorare a runtime i componenti delle nostre applicazioni
- garantiscono flessibilità (caso kubernetes implementa la soluzione di elasticità con politica basata su threshold di utilizzazione di risorse per la replicazione).

Gli engine di orchestrazione dei container ci danno strumenti per lavorare con applicazioni messe su diversi container che vengono sviluppate su diversi nodi dell'architettura che stiamo sviluppando.

Le due più importanti piattaforme per ambienti non cloud sono Docker Swarm e Kubernetes.

#### Docker swarm

Docker include lo swarm mode, quindi in una situazione con molteplici nodi con docker installato ci permette di istanziare container docker su diversi nodi. Nel caso di swarm i container che fanno parte di uno stesso servizio vengono chiamati task.

Cosa ci mette a disposizione swarm:

- Scaling: ci permette di avere molteplici repliche per lo stesso container
- State reconciliation: ci permette di gestire il fallimento dei nodi su cui sono ospitati i container andandoli a reinstanziare su altri host
- Ci permette di istanziare overlay network fra diversi servizi
- Load balancing: permette di specificare come distribuire i diversi container fra i nodi

Uno swarm è un insieme di host che hanno docker installato, eseguiti appunto in modalità swarm.

Ha la classica architettura master worker, dove un nodo master coordina il lavoro dei nodi worker. Gli worker hanno dei container assegnati e ne monitorano il funzionamento; è il master che gestisce le assegnazioni. Lo svantaggio dell'architettura di questo tipo è come sempre il single point of failure che potrebbe costituire il master. Serve quindi salvarlo in una persistenza per essere recuperato da un altro nodo eletto come nuovo master. Il master potrebbe inoltre costituire un collo di bottiglia, e quindi vengono introdotte delle tecniche di ottimizzazione.

Docker swarm quindi permette quindi un bilanciamento del carico fra i diversi nodi.

#### 5.9.2. Kubernetes

È stato sviluppato da Google, e si tratta del primo prodotto opensource che è stato reso disponibile da google. Questa piattaforma permette l'orchestrazione di applicazioni multicontainer hostate su più nodi. Ha un algoritmo per decidere su quali nodi ospitare le singole applicazioni. Permette elasticità per le repliche dei container e supporta tolleranza ai guasti. Tutto ciò che offre lo fa in modo trasparente per lo sviluppatore.

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

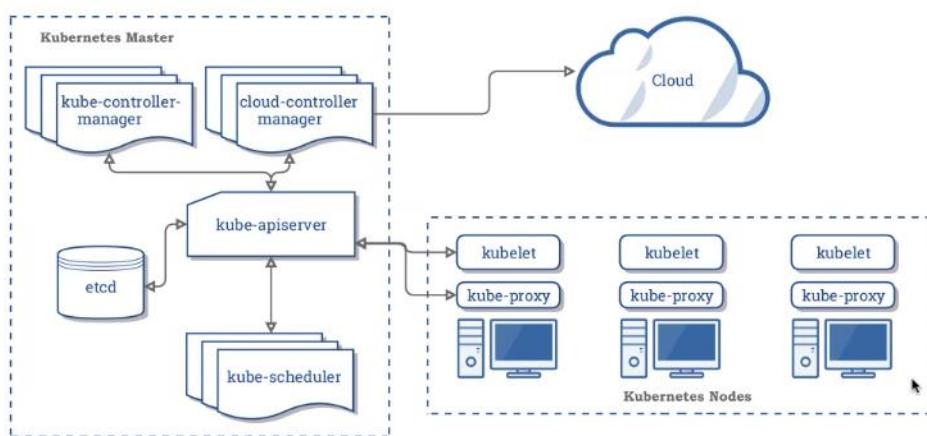
Caratteristiche:

- Portabilità (può essere eseguito su cluster privato, in ambito cloud o multi-cloud).
- Estendibile (i componenti sono stati progettati con l'ottica di essere sostituibili ed estendibili).
- Auto-healing (controlla il placement dei container sui nodi, è in grado di avviare i container e modificarne a runtime il grado di replicazione).

Kubernetes può essere eseguito anche al di sopra delle applicazioni per la gestione di cluster, oppure può essere utilizzato all'interno di open stack (sistema per realizzare cloud privata).

**Pod**: unità minima di scheduling in Kubernetes. Si tratta di un insieme (altamente accoppiati) di container che condividono storage e rete, insieme a delle istruzioni sul come eseguirli. I container in un pod sono visti come tutt'uno, quindi vengono schedulati insieme ed eseguiti in un contesto condiviso. Kubernetes assegna ai pod il loro IP ed un singolo DNS per un set di pod.

L'architettura di Kubernetes è distribuita di tipo **master-worker**:



**MASTER**: prende decisioni globali per i cluster e ne gestisce il piano di controllo. Potenzialmente per andare incontro ad esigenze di failover ed availability si possono realizzare più worker.

- **Kube scheduler**: decide come assegnare i pod ai nodi
- **Kube-apiserver**: server che espone l'interfaccia di controllo per controllare da riga di comando kubernetes.
- **Etcdb**: un sistema di storage chiave valore altamente disponibile utilizzato da kubernetes per mantenere i metadati relativi alla gestione del cluster
- **Horizontal pod autoscaler**: componente aggiuntivo che gestisce le repliche a disposizione di uno specifico container monitorando l'utilizzo dei pod

**NODI**: macchine worker che possono essere fisiche o virtuali.

- Il **kubelet** è un agente su ogni nodo, che tramite meccanismo di heartbit monitor controlla se i pod in esecuzione sul nodo sono funzionanti. Tramite questi componenti si possono ottenere statistiche sull'utilizzo dei nodi di kubernetes.

**Come condividere le risorse concorrenti nei cluster fra multipli e non omogenei framework?**

Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 1 to the text that you want to appear here. - Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here..Error! Use the Home tab to apply Titolo 2 to the text that you want to appear here.

**Static partitioning** - Ho diversi framework applicativi, partiziono staticamente le macchine che ho a disposizione per ognuno di loro. Funziona, ma altamente inefficiente in quanto l'utilizzazione dei diversi framework da parte degli utenti può variare notevolmente nel tempo. Questo partizionamento statico potrebbe rivelarsi sbagliata.

**Apache Mesos** viene infatti qui in supporto, trattandosi di un cluster manager (gestore delle risorse all'interno di un cluster) che fornisce un layer di controllo per le risorse in un cluster al di sopra del quale possiamo eseguire i molteplici framework applicativi. L'obiettivo di Mesos è fornire un'astrazione delle risorse computazionali a disposizione nel cluster, permettendo alle applicazioni di condividere in modo efficiente le risorse nel cluster partizionando fra i diversi framework applicativi CPU e memoria. In base alla richiesta di utilizzo da parte dei framework applicativi, le suddette partizioni possono variare nel tempo. Anche questo funziona con l'architettura master-worker, dove gli worker sono chiamati agenti. Mesos gestisce dei master aggiuntivi "standby" che vengono eletti come principali e possono recuperare lo stato del master precedente da uno ZooKeeper quorum qualora fallisse il master precedente.

### Distribuzioni di Kubernetes

Si può scaricare dal codice sorgente, ma qualora si volesse installare su diversi nodi non è la più facile attuazione. In alternativa ci sono delle scorciatoie fornite da altre distribuzioni.

- **Distribuzioni pure:** pre-built Kubernetes (esempio Charmed Kubernetes)
- **Distribuzioni plus:** installare delle piattaforme che non soltanto offrono kubernetes, ma lo integrano con altre tecnologie (aggiunte di container, sistemi operativi o pannelli di controllo). Esempio Red Hat OpenShift
- **Kubernetes-as-a-service:** Kubernetes offerto come servizio cloud, quindi completamente gestito dal provider cloud dello specifico servizio; In questo caso l'utente non deve preoccuparsi dell'installazione e la gestione di Kubernetes. Esempio è Azure AKS.
- **Distribuzioni limited-purpose:** caso in cui vogliamo utilizzare kubernetes per uno specifico scopo, quindi delle installazioni su specifiche architetture, singoli nodi, edge & IoT. Esempio è minikube.

# SISTEMI DISTRIBUITI E CLOUD COMPUTING

## PARTE 2

## Sommario

<b>1. Micro-servizi.....</b>	5
<b>1.1. Architetture orientate ai servizi SOA.....</b>	6
1.1.1. Implementazione SOA: Web Services.....	7
<b>1.2. SOA vs micro-servizi.....</b>	7
<b>1.3. Microservizi e Container .....</b>	7
1.3.1. Come decomporre l'applicazione in microservizi.....	8
<b>1.4. Microservizi e scalabilità .....</b>	8
1.4.1. Servizi Stateless vs. stateful .....	8
1.4.2. Comunicazione sincrona / asincrona per i micro-servizi.....	9
<b>1.5. Pattern: orchestrazione e coreografia .....</b>	10
1.5.1. Vantaggi e svantaggi di orchestrazione vs coreografia.....	10
<b>1.6. Pattern per applicazioni a micro-servizi.....</b>	11
1.6.1. Circuit breaker.....	11
1.6.2. Pattern SAGA .....	11
1.6.3. Log aggregation.....	13
1.6.4. Distributed tracing.....	13
<b>1.7. Generazioni di architetture a micro-servizi.....</b>	13
1.7.1. Serverless and Faas: caratteristiche .....	14
<b>2. Sincronizzazione.....</b>	15
<b>2.1. Modello di computazione .....</b>	15
2.1.1. Timestamping .....	15
2.1.2. Sincronizzazione dei clock.....	16
<b>2.2. Clock fisico .....</b>	16
2.2.1. Caratteristiche di un clock fisico nei sistemi distribuiti.....	17
2.2.2. UTC (Universal Coordinated Time) .....	17
2.2.3. Sincronizzazione di clock fisici.....	17
<b>2.3. Algoritmo di sincronizzazione interna tra due processi di un SD sincrono .....</b>	18
<b>2.4. Sincronizzazione mediante time service.....</b>	18
2.4.1. Algoritmo di Cristian.....	19
2.4.2. Algoritmo di Berkley .....	19
2.4.3. Network time protocol NTP .....	20
<b>2.5. Tempo nei SD asincroni.....</b>	23
<b>2.6. Clock logico scalare .....</b>	24
<b>2.7. Algoritmo introdotto da Leslie Lamport per implementare il clock logico scalare .....</b>	24
<b>2.8. Relazione di ordine totale .....</b>	25
<b>2.9. Clock logico vettoriale .....</b>	26
2.9.1. Significato e confronto di clock .....	26

<b>2.10.</b>	<b>Multicast causalmente e totalmente ordinato</b>	28
2.10.1.	Multicast totalmente ordinato (clock logico scalare)	28
2.10.2.	State machine replication	29
2.10.3.	Multicast causalmente ordinato (clock logico vettoriale)	30
<b>2.11.</b>	<b>Mutua esclusione e sistemi concorrenti</b>	31
2.11.1.	Algoritmo del panificio di Lamport	31
<b>2.12.</b>	<b>Mutua esclusione distribuita</b>	34
2.12.1.	Mutua esclusione distribuita: modello del sistema	34
2.12.2.	Adattamento dell'algoritmo del panificio	34
2.12.3.	Paronamica sugli algoritmi per ME distribuita	35
<b>2.13.</b>	<b>Algoritmi basati su autorizzazioni</b>	35
2.13.1.	Algoritmo centralizzato	35
2.13.2.	Algoritmo decentralizzato: algoritmo di Lamport distribuito	36
2.13.3.	Algoritmo decentralizzato: algoritmo di Ricart e Agrawala	37
<b>2.14.</b>	<b>Algoritmi basati su token</b>	38
2.14.1.	Algoritmo basato su token centralizzato	38
2.14.2.	Algoritmo basato su token decentralizzato	41
<b>2.15.</b>	<b>Algoritmi basati su quorum</b>	42
2.15.1.	Algoritmo di Maekawa	42
<b>2.16.</b>	<b>Algoritmi di elezione distribuita</b>	45
2.16.1.	Algoritmo bully	46
2.16.2.	Algoritmo di Fredrickson & Lynch	47
2.16.3.	Algoritmo di Chang & Robert	47
<b>2.17.</b>	<b>Proprietà degli algoritmi di elezione:</b>	48
<b>3.</b>	<b>Consistenza e replicazione</b>	49
<b>3.1.</b>	<b>Problemi di consistenza</b>	49
<b>3.2.</b>	<b>Modelli di consistenza data centrici</b>	50
3.2.1.	Consistenza stretta: il modello ideale	51
3.2.2.	Consistenza sequenziale: indebolimento di quella linearizzabile	51
3.2.3.	Consistenza linearizzabile	53
3.2.4.	Consistenza causale	54
3.2.5.	Teorema CAP	55
3.2.6.	Consistenza finale	56
<b>3.3.</b>	<b>ACID vs BASE</b>	57
<b>3.4.</b>	<b>Modelli di consistenza client-centrici</b>	58
3.4.1.	Read after read (monotonic read)	58
3.4.2.	Write after write (monotonic write)	58
3.4.3.	Read after write (read your writes)	59

3.4.4.	Consistenza writes-follow-reads .....	59
3.4.5.	Esempi di consistenza client-centrica.....	59
<b>3.5.</b>	<b>Protocolli di consistenza.....</b>	<b>60</b>
3.5.1.	Protocolli primary based.....	61
3.5.2.	Protocolli replicated write .....	63
3.5.3.	Protocolli per la consistenza client-centrica .....	64
<b>3.6.</b>	<b>Replicazione.....</b>	<b>64</b>
<b>4.</b>	<b>Tolleranza ai guasti.....</b>	<b>65</b>
<b>4.1.</b>	<b>Failure, error, fault: differenze .....</b>	<b>65</b>
4.1.1.	Tipi di failure .....	66
4.1.2.	Modelli di failure .....	66
<b>4.2.</b>	<b>Rilevamento dei fallimenti .....</b>	<b>66</b>
<b>4.3.</b>	<b>Ridondanza .....</b>	<b>67</b>
4.3.1.	Esempio di ridondanza fisica: Ridondanza tripla modulare .....	67
<b>4.4.</b>	<b>Resilienza dei processi.....</b>	<b>67</b>
<b>4.5.</b>	<b>Modelli di replicazione: passiva vs. Attiva .....</b>	<b>68</b>
<b>4.6.</b>	<b>Gruppi e mascheramento dei guasti.....</b>	<b>68</b>
<b>4.7.</b>	<b>Consenso nei sistemi distribuiti.....</b>	<b>68</b>
4.7.1.	Condizioni necessarie al consenso: .....	69
<b>4.8.</b>	<b>Teorema dell'impossibilità di FLP (Fischer, Lynch and Patterson).....</b>	<b>69</b>
<b>4.9.</b>	<b>Algoritmo di Paxos .....</b>	<b>70</b>
4.9.1.	Fase 1: prepare.....	71
4.9.2.	Fase 2: accept.....	71
4.9.3.	Proprietà di Paxos .....	71
4.9.4.	Esempio di Paxos.....	72
<b>4.10.</b>	<b>RAFT (Replicated And Fault Tolerant) .....</b>	<b>74</b>
4.10.1.	LEADER ELECTION: .....	75
4.10.2.	LOG REPLICATION:.....	75
<b>4.11.</b>	<b>Problema dell'accordo bizantino.....</b>	<b>76</b>
4.11.1.	<b>Algoritmo di Lamport per l'accordo Bizantino (Oral Message OM) .....</b>	<b>78</b>
4.11.2.	La tolleranza ai guasti bizantini a livello pratico.....	79
<b>5.</b>	<b>Problema delle commit distribuite .....</b>	<b>79</b>

# 1. Micro-servizi

Sono un nuovo stile architetturale per la realizzazione di applicazioni distribuite, derivante dallo stile architetturale dei **web services**.

Le applicazioni sono suddivise in un numero anche elevato di servizi (**micro-servizi**), poco accoppiati tra loro.

Si passa da applicazioni a strutturazione monolitica, a suddividere l'applicazione in molte componenti.

Obiettivo: definire, costruire, gestire applicazioni distribuite, composte da piccole unità (micro-servizi) auto contenute.

L'applicazione è suddivisa in un insieme di servizi che possono essere deployati nell'ambiente a runtime in modo indipendente, sono poco accoppiati tra loro, cooperano e comunicano tramite RPC o code di messaggi e sistemi publish/subscribe. I singoli micro-servizi potranno essere facilmente istanziati e facilmente scalati (il numero di repliche di ogni micro-servizio potrà essere variato a runtime applicando le idee chiave dell'elasticità → possiamo avere scalabilità orizzontale in cui si sceglie di variare il numero di repliche aumentando o riducendo lo scale-out o lo scale-in, o usando scalabilità verticale in cui variamo la quantità risorse allocate per ogni micro-servizio).

Proprietà dei micro-servizi:

1. Istanziabili e scalabili rapidamente: tra le tecnologie di virtualizzazione usiamo quella dei container a livello del sistema operativo. Ciò lascia evincere lo stretto collegamento tra micro-servizi e container.
2. Stateless: quello che istanziamo all'interno dei micro-servizi non ha stato, poiché se avesse lo stato diventerebbe più complesso gestirne l'elasticità. Lo stato verrà salvato in strumenti e tool/framework dedicati ad offrire persistenza della memoria.

All'interno delle architetture a micro-servizi troveremo utilizzati non solo database relazionali, ma anche datastore di tipo no sequel (NoSQL, offrono migliore scalabilità e diversi modelli di consistenza rispetto al classico modello di consistenza sequenziale/linearizzabile dei DB relazionali tradizionali). L'ambiente di sviluppo diventa quindi molto più variegato di DB relazionali.

Le parole chiave sono **disaccoppiamento** e **modularità** (suddivisione dell'applicazione in un set di servizi sviluppabili indipendentemente, poco accoppiati che cooperano e che possono essere rapidamente sviluppati e facilmente se ne può fare lo scaling). I servizi sono dotati di strumenti per la persistente memoria dedicata. Vedremo che ci sarà uno stretto collegamento tra micro-servizi e container.

## 1.1. Architetture orientate ai servizi SOA

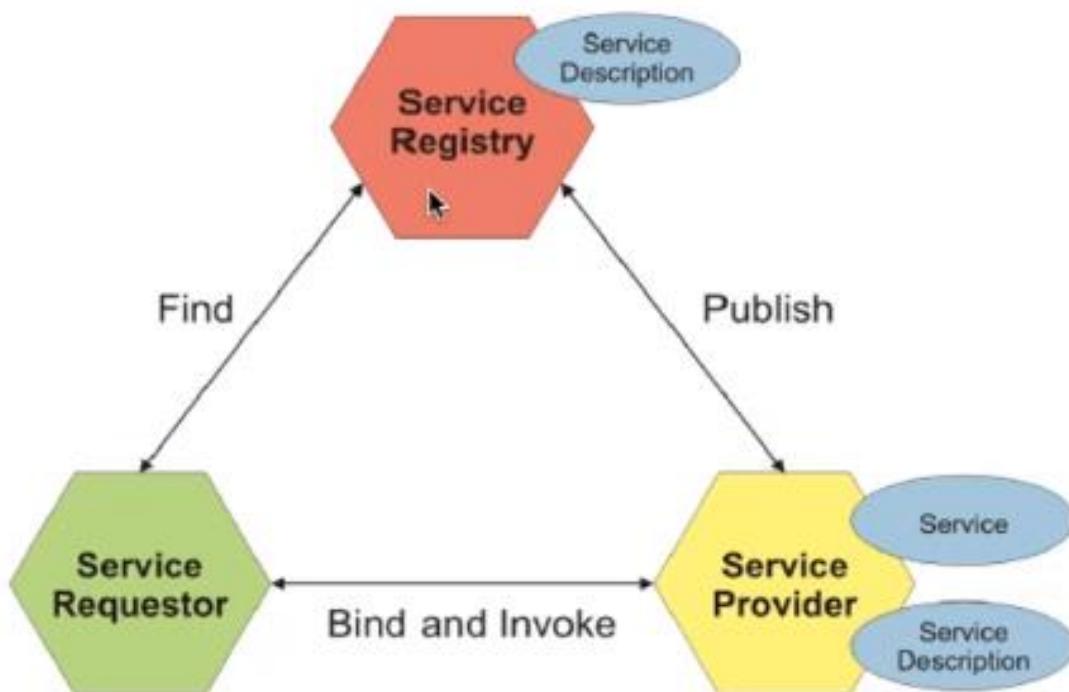
**Service Oriented Architecture (SOA):** paradigma per realizzare applicazioni distribuite, in cui servizi componenti sono poco accoppiati tra loro (punto in comune con architetture a micro-servizi).

- **Definizione OASIS:** paradigma per l'**organizzazione e l'utilizzazione di risorse distribuite** che possono essere sotto il controllo di domini di proprietà differenti (si hanno più amministratori). Fornisce un mezzo uniforme per offrire, scoprire (discovery), interagire ed usare le capacità di produrre gli effetti voluti in modo consistente con presupposti e aspettative misurabili
- Proprietà:
  - Vista logica dell'applicazione
  - Orientamento ai messaggi e alla descrizione: i componenti dell'applicazione comunicano tra loro tramite scambio di messaggi
  - Granularità dei servizi, orientamento alla rete, neutralità della piattaforma: per ogni servizio viene fornita descrizione di tipo semantico (cosa fa servizio) e sintattico (metodi che servizio espone, tipologia dei parametri di input e output).

Distinguiamo tre entità che interagiscono tra loro:

- **Service requestor** o consumer: chi richiede l'esecuzione di un servizio. Cerca il servizio che gli interessa in un registro di servizi (service registry)
- **Service provider**: fornitore del servizio, implementa il servizio e lo rende disponibile. Pubblica la descrizione del servizio offerto sul service registry, così che il service requestor, una volta identificato il service provider, si collegherà ad esso per invocarne i metodi del servizio esposti.
- **Service registry**: offre un servizio di pubblicazione e di ricerca di servizi disponibili. Espone una descrizione dei servizi, ed il service requestor ricerca tra i servizi esposti quello più adatto alle sue esigenze (discovery servizi).

Si tratta di tre entità istanziate e gestite da diversi amministratori e su diversi domini.



### 1.1.1. Implementazione SOA: Web Services

Un web service è un sistema software sviluppato per fornire interazione machine to machine su una rete, che ha un'interfaccia descritta in un formato elaborabile dalla macchina (in particolare WSDL).

Altri sistemi interagiscono con il servizio web in un modo prescritto dalla sua descrizione utilizzando messaggi SOAP (protocollo applicativo che usa protocolli sottostanti come protocolli di trasporto e si basa su protocollo HTTP). I messaggi SOAP che vengono realizzati come lettere (ossia hanno un header contenente il destinatario del messaggio e un body contenente l'oggetto del messaggio) vengono incapsulati in richieste http.

Esistono in realtà più di 60 standard e specifiche; i protocolli più utilizzati sono:

- Per descrivere i servizi: **WSDL** (Web Service Description Language).
  - a. Quando il service provider pubblica descrizione servizio, questa descrizione sarà in formato WSDL.
- Per comunicare: **SOAP** (Simple Object Access Protocol).
  - b. Utilizzato tra service requestor e provider per espletare richiesta e fornitura servizio.
- Per registrarsi: **UDDI** (Universal Description, Discovery and Integration)
  - c. Per scoprire i servizi, usato nell'interazione tra service requestor e registry.
- Per definire i processi aziendali: **BPEL** (Business Process Execution Language), **BPMN** (Business Process Model and Notation).
  - d. Definire il workflow dell'applicazione a servizi (stesso concetto dell'applicazione a microservizi, ossia abbiamo tanti microservizi integrati tra loro per definire una composizione di servizi).
- Per definire SLA: **WSLA**

Esistono poi molte tecnologie inerenti, tra cui **ESB** (Enterprise Service Bus), piattaforma di integrazione che fornisce servizi fondamentali di interazione e comunicazione per applicazioni complesse.

### 1.2. SOA vs micro-servizi

- **Heavyweight vs lightweight technologies:** SOA tende ad affidarsi fortemente su middleware pesanti, mentre i micro-servizi su tecnologie leggere.
- **Protocol families:** SOA è spesso associato con siti web e protocolli, i micro-servizi tipicamente si affidano a REST e HTTP
- **Viste:** SOA è visto come una soluzione integrativa, i micro-servizi sono applicati per costruire applicazioni software individuali.

### 1.3. Microservizi e Container

I micro-servizi sono visti come il complemento ideale della virtualizzazione basata su container, poiché vengono utilizzati i container per incapsulare i micro-servizi. Ogni servizio è impacchettato come se fosse l'immagine del contenitore ed ogni istanza del servizio viene distribuita come un container. Ogni container viene gestito in fase di esecuzione, e può essere scalato (ridimensionato) e migrato, ovvero spostato da un host all'altro.

## Pro:

- Elasticità → Per la scalabilità orizzontale/verticale dell'istanza del servizio basta cambiare il numero di istanze del contenitore.
- Isolamento dell'istanza di servizio → I diversi micro-servizi sono tra loro isolati.
- Si costruisce e si inizializza rapidamente

## Contro:

È necessaria l'orchestrazione dei container per gestire applicazioni multi-container

### 1.3.1. Come decomporre l'applicazione in microservizi

- Seguendo la **business capability**, considerando lo scopo e le capacità dei microservizi.
- Identificando i sottodomini utilizzando la soluzione di decomposizione usata dal **dominio domain driven design DDD**.
- Decomporre a seconda dei **casi d'uso** dell'applicazione definendo servizi responsabili per determinati casi d'uso.
- Decomporre a seconda di **nomi o risorse** e definire un servizio responsabile di tutte le operazioni su una determinata entità o risorsa.

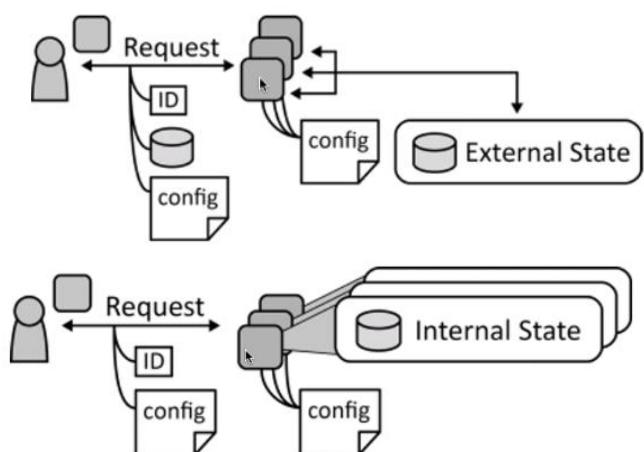
## 1.4. Microservizi e scalabilità

Per garantire scalabilità del servizio si usano multiple istanze dello stesso servizio e load balance fra le molteplici istanze. Per migliorare la scalabilità del servizio si cercano di usare servizi stateless, in quanto più facili da gestire e scalare rispetto a quelli stateful. Abbiamo bisogno anche di service discovery, perché le istanze dei servizi ricevono locazioni di rete assegnate dinamicamente, e questo set cambia dinamicamente proprio per l'autoscaling, fallimenti e gli upgrade. Serve un service discovery (ovvero scoprire a runtime a quale servizio andare a collegarsi).

### 1.4.1. Servizi Stateless vs. stateful

**Servizi stateless:** lo stato non è presente in un servizio stateless, non è presente nel servizio stesso ma è salvato esternamente. Sarà più facile replicarlo.

**Servizi stateful:** Lo stato è incorporato nel servizio stesso, e qualora lo si voglia scalare ad esempio aumentando le repliche a fronte di un aumento del numero di richieste, quando lo si replica va replicato anche lo stato interno e va anche mantenuto sincronizzato; Ecco perché risulta più difficile da gestire. Gli utenti fanno richieste, ed esiste una funzione di hashing basata sul contenuto della richiesta per indirizzarli al micro-servizio corretto.



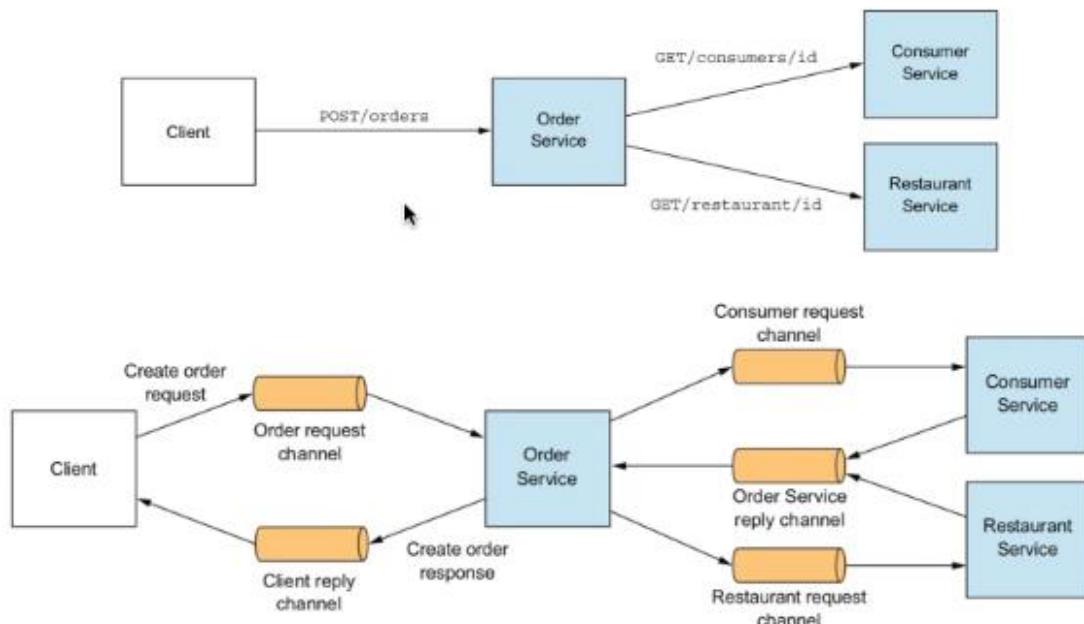
### 1.4.2. Comunicazione sincrona / asincrona per i micro-servizi

Per realizzare un'applicazione a micro-servizi, quindi un'applicazione non composta da un solo micro-servizio, i pattern di comunicazione più usati sono RPC o code di messaggi.

Con RPC abbiamo uno stile di comunicazione sincrono, con le code di messaggi asincrono.

- Nel **caso sincrono** andremo a usare tipicamente richieste HTTP per interfacce REST o gRPC.
- Nel **caso asincrono** incorporeremo nella app a micro-servizi dei middleware che supportano la comunicazione asincrona come Kafka, AMQP.

## • Synchronous communication reduces availability



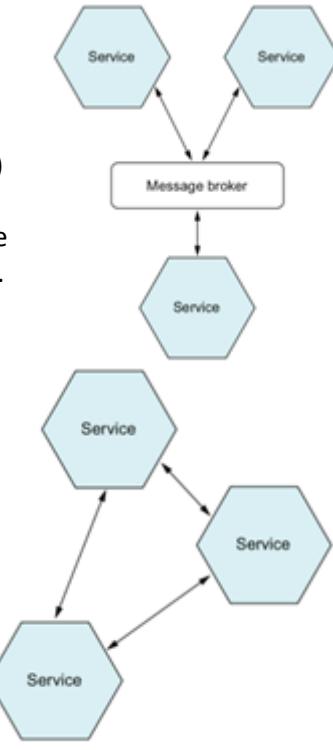
## 1.5. Pattern: orchestrazione e coreografia

I micro-servizi possono interagire tra loro seguendo due modelli:

- Orchestrione
- Coreografia

**Orchestrione:** approccio centralizzato alla composizione

- Un unico processo centralizzato (orchestrator o message broker) coordina l'interazione tra i diversi servizi
- L'orchestratore è responsabile del richiamo e della combinazione dei servizi, che può non essere a conoscenza della composizione. Permette ai micro-servizi di interagire tra loro.



**Coreografia:** approccio decentralizzato alla composizione

- Una descrizione globale dei servizi che fanno parte dell'applicazione, definita dallo scambio di messaggi, regole di interazione e accordi tra due o più endpoint
- I micro-servizi possono scambiare messaggi direttamente (in orchestrione lo scambio di msg è mediato da orchestratore, qui è diretto)

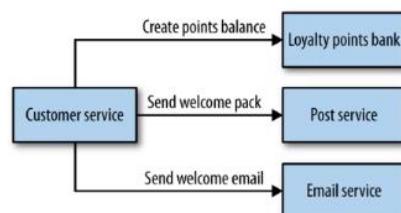
### 1.5.1. Vantaggi e svantaggi di orchestrione vs coreografia

**L'orchestrione:**

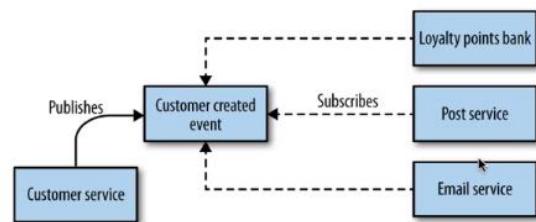
- Risulta più semplice e popolare rispetto alla coreografia ma può costituire un single point of failure per il direttore d'orchestra.
- Ha un accoppiamento più stretto fra i servizi e ha più latenza.

### Example: orchestration and choreography

#### Orchestration



#### Choreography



**La coreografia:**

- Garantisce un minore accoppiamento e minore latenza
- Garantisce più flessibilità, ma risulta più complesso monitorare le prestazioni ed effettuare il tracking dei servizi per via della coreografia.

## 1.6. Pattern per applicazioni a micro-servizi

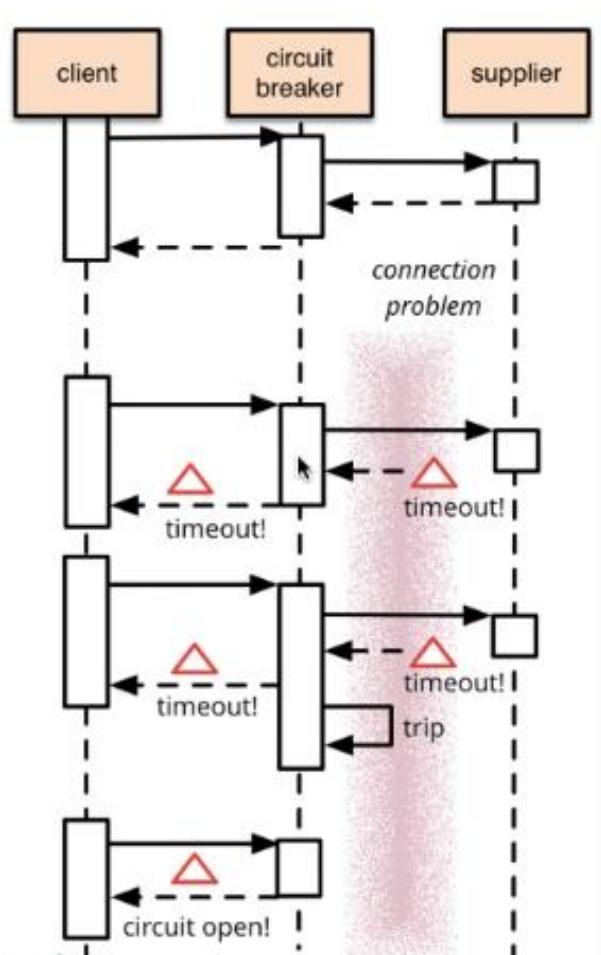
Esaminiamo dei pattern per realizzare le applicazioni a micro-servizi.

### 1.6.1. Circuit breaker

È un design pattern che fornisce uno schema risolutivo per affrontare una situazione che si verifica in modo frequente nello sviluppo di un'applicazione. L'obiettivo è quello di prevenire che un fallimento nella rete o nel servizio invocato si ripercuota a cascata sugli altri servizi. La soluzione è quella di interrompere il circuito fra il servizio che invoca ed il servizio che dovrebbe fornire la funzionalità giusta.

Viene introdotto una sorta di micro-servizio **proxy intermedio chiamato circuit breaker** che riceve le richieste del client e le inoltra al servizio "protetto" dall'interruttore. Questo circuit breaker subentra quando si verifica un errore e si interpone comportandosi da proxy. Il circuit breaker tiene traccia del numero di invocazioni al servizio; Quando nota che queste invocazioni consecutive senza successo supera una soglia preconfigurata per lui, apre l'interruttore e per un certo periodo di tempo si comporterà nel suddetto modo.

Quando il problema viene risolto, il circuit breaker chiude l'interruttore e viene ripreso il funzionamento normale della comunicazione tra il micro-servizio client e quello invocato. Se a quel punto il servizio invocato ha ancora un fallimento, viene riattivato subito il circuit breaker.



### 1.6.2. Pattern SAGA

**Database per service:** questa soluzione fornisce un **layer di persistenza** privato per singolo servizio accessibile utilizzando l'API registrata apposta per quest'ultimo.

Pro:

- I servizi sono poco accoppiati
- Ogni servizio sfrutta il tipo di database che meglio soddisfi i suoi bisogni

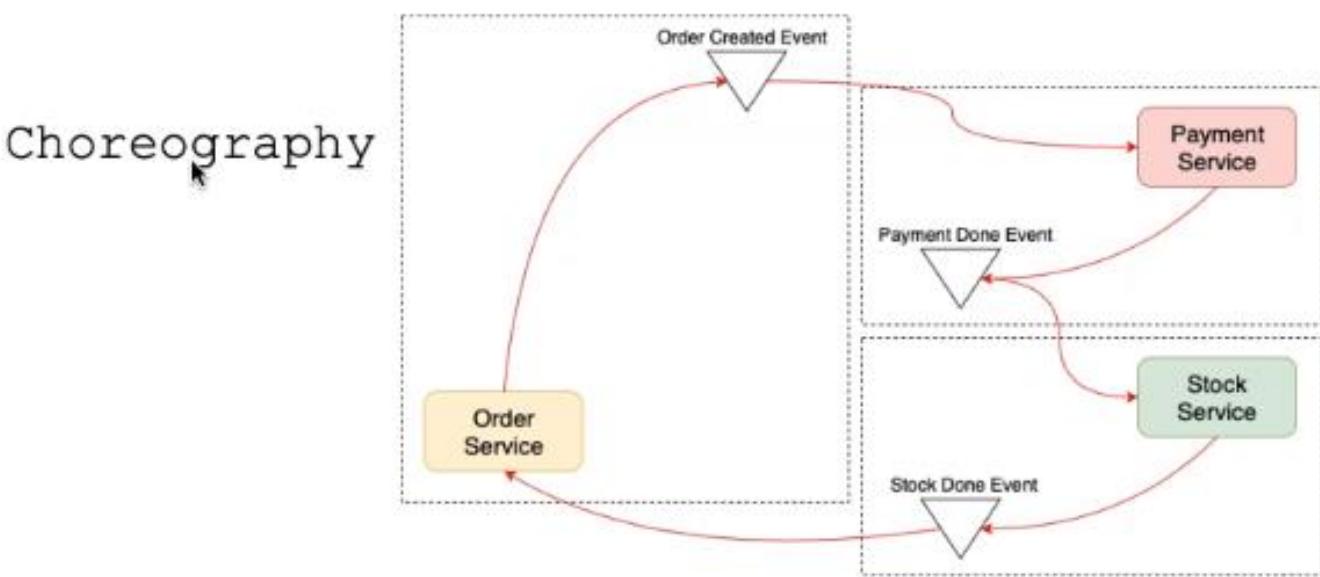
Contro:

- Implementazione delle transazioni più complesse per gestire la diversità dei servizi
- Complessità per la gestione di database multipli (vedi pattern SAGA)

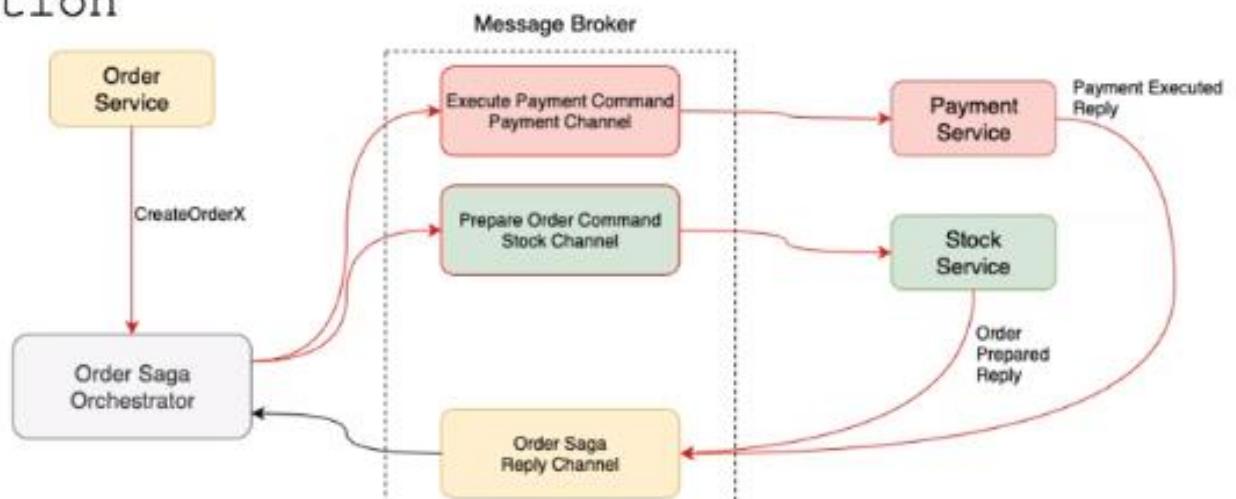
In caso di transazione che coinvolge molteplici servizi, come assicurare la consistenza dei dati fra i servizi dato che ognuno di loro ha il suo database?

Il pattern SAGA implementa ciascuna transazione come una "SAGA" per risolvere il suddetto problema; **una saga è una sequenza di transazioni locali**. Ogni transazione locale aggiorna il database e pubblica un messaggio o un evento per mandare un trigger alla successiva transazione. Se una transazione locale fallisce, la saga esegue una serie di transazioni di compensazione per annullare le transazioni precedenti. Infatti si evince come questo pattern si basi sul principio del tutto o niente, o riescono tutti i servizi o niente (**principio dell'atomicità**). Come riuscire a coordinare le transazioni?

- **Coreografia:** ogni transazione locale pubblica eventi che attivano le transazioni locali successive quando va a buon fine.
- **Orchestrazione:** un direttore d'orchestra dice a ciascuna transazione locale quale è la prossima che va eseguita.



## Orchestration



### 1.6.3. Log aggregation

Viene introdotto all'interno di micro-servizi un servizio di logging centralizzato con l'obiettivo di legare i log delle singole istanze dei micro-servizi che compongono l'applicazione. Questa soluzione permette di comprendere il comportamento dell'applicazione e risolvere problemi. I contro di questo pattern sono che risulta una soluzione centralizzata e che gestire grandi unità di log può essere dispendioso.

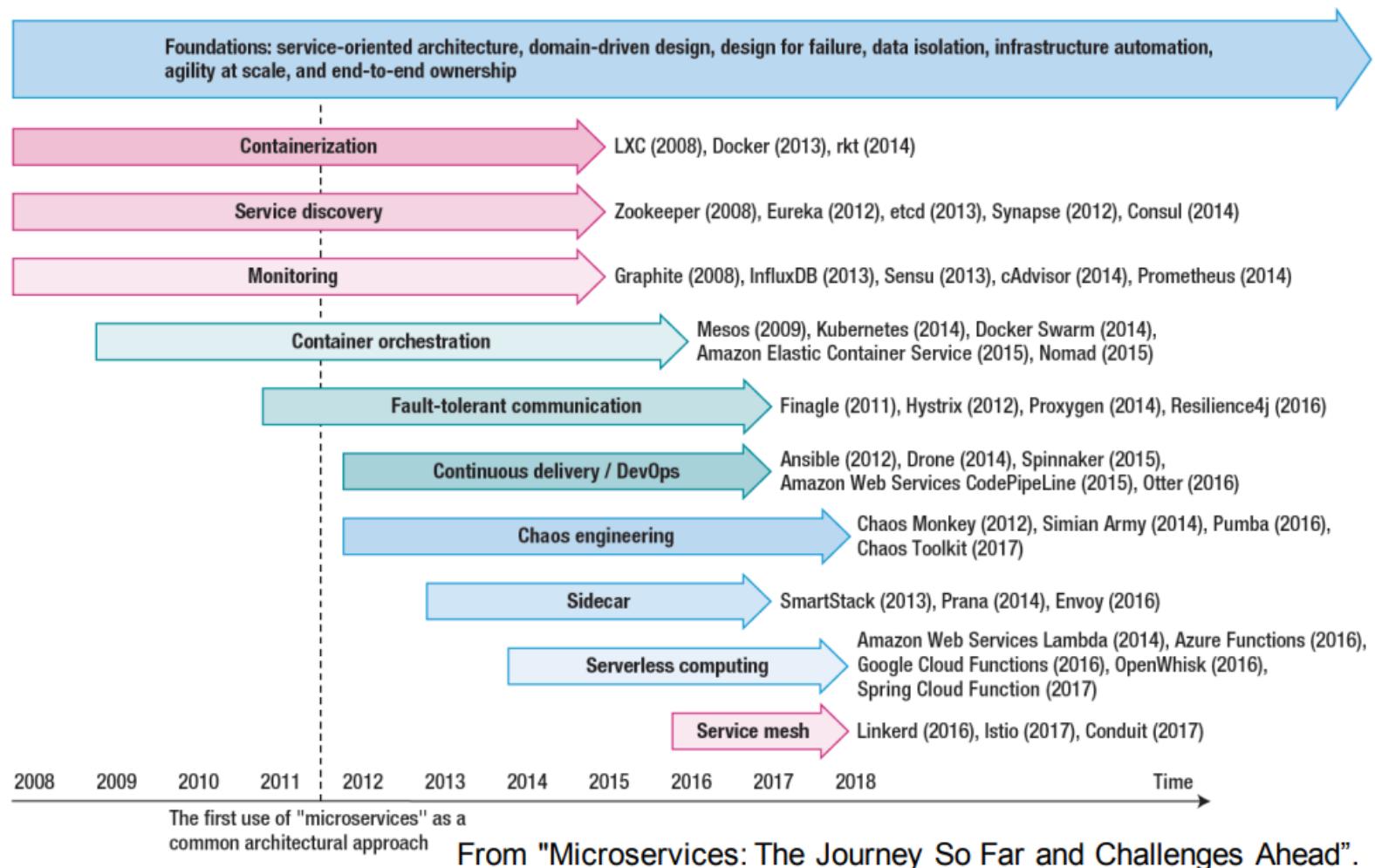
### 1.6.4. Distributed tracing

Si tiene traccia di quello che è il flusso di richiesta transitata tra i micro-servizi; nello specifico registra le informazioni riguardanti le richieste e le operazioni eseguite quando viene gestita una richiesta esterna in un servizio centralizzato.

Si assegna ad ogni richiesta esterna un ID, il quale verrà passato a tutti i servizi coinvolti nella richiesta. L'ID verrà incluso in tutti i messaggi di log che lo riguardano.

I contro di utilizzare il distributed tracing risiede nel fatto che aggregare e salvare le tracce potrebbe richiedere una infrastruttura significativa.

## 1.7. Generazioni di architetture a micro-servizi



Ci sono 4 diverse generazioni di architetture a microservizi:

- La **prima generazione** utilizza come soluzioni la **virtualizzazione basata su container**, **framework per server discovery** (permettendo ai server di scoprirsi da soli senza conoscere la locazione geografica, maggior flessibilità). Inoltre prevedeva il **monitoring a runtime dei micro-servizi** e la tecnica di orchestrazione dei container.
- La **seconda generazione** usa anche **servizi di discovery e librerie di comunicazione** con lo scopo di estrarre dai micro-servizi parte della logica utilizzata per il discovery o per attività di comunicazione: questo rendeva più snelli i micro-servizi.
- La **terza generazione** si basa sulla **tecnologia sidecar** cercando di snellire ancora di più i micro-servizi; ogni sidecar implementa alcune features utili al micro-servizio, come il servizio per il discovery, la gestione del traffico, ed altre utilità.
- La **quarta generazione** si basa sul concetto di **FaaS (function as a service)** e sul concetto di **serverless computing** per semplificare lo sviluppo dei microservizi. Serverless computing significa che i server sono gestiti completamente da qualcun altro, è un modello di cloud computing che si concentra solo sul codice applicativo astraendo la gestione del server e le decisioni di basso livello lontane dall'utente.

### 1.7.1. Serverless and Faas: caratteristiche

- Risorse di computazione minime
- Elasticità automatizzata
- Pay per use
- Event-driven
- Può semplificare lo sviluppo del codice in quanto lo scaling, il capacity planning, la manutenzione possono essere nascosti dallo sviluppatore

L'elasticità è gestita completamente dal cloud provider e non è compito dello sviluppatore preoccuparsene. L'utente può specificare soltanto requisiti sulle quantità di risorse da allocare alla funzione. La maggior parte dei cloud provider permette solo di specificare la ram associata, caratteristica vista dagli utenti come limitazione; tuttavia questa soluzione semplifica notevolmente il provisioning delle risorse. Con il serverless computing abbiamo il pay per use, quindi paghiamo a seconda dell'utilizzo e non per unità pre-determinate.

Il FaaS presenta ancora delle limitazioni studiate in ambito di ricerca; ad esempio, le funzioni nel momento in cui vengono invocate, se non esiste già il container o la micro-macchina virtuale che deve eseguire quella funzione, la funzione sperimenta un tempo di startup alto, dell'ordine dei secondi. Una altra limitazione dipende dalla scelta del linguaggio. I maggiori cloud provider supportano funzioni scritte in Java, Phyton e Go. La scelta del linguaggio ha un impatto sulle prestazioni, e nello specifico java risulta il più carente a livello prestazionale. Altro problema è quello del vendor lock in (l'API esposta dai singoli provider è differente).

Altro aspetto rilevante è la possibilità di comporre funzioni serverless (realizzare applicazioni definite da un workflow, una sequenza di diverse invocazioni).

#### Esempio: OpenWhisk

Framework il cui sviluppo è stato avviato da IBM. Piattaforma serverless che esegue funzioni in risposta ad eventi. I container Docker possono essere orchestrati su un singolo nodo usando Docker compose, oppure può usare Kubernetes per orchestrazione su diversi nodi. I linguaggi supportati da OpenWhisk sono diversi e le funzioni verranno schedulate in modo dinamico da OpenWhisk e verranno attivate in corrispondenza a determinati trigger esterni o richieste http.

## 2. Sincronizzazione

All'interno di un sistema distribuito i componenti del sistema devono cooperare per portare a compimento una determinata applicazione. La computazione nel sistema avviene tramite messaggi tra i nodi connessi tramite una rete. Questi nodi richiedono una sincronizzazione fra i diversi processi in esecuzione; bisogna garantire una nozione comune di tempo per coordinarli.

Il tempo rappresenta un fattore critico nei sistemi distribuiti. In un sistema centralizzato il problema si risolve banalmente grazie al clock fisico globale e la memoria comune.

In un sistema distribuito è impossibile avere un unico clock fisico comune ai processi in esecuzione. La soluzione sta nello stabilire l'ordine degli eventi come tempo. Come?

- **Sincronizzazione degli orologi fisici:** sincronizziamo fisicamente fra loro i diversi nodi del sistema distribuito, o rispetto ad un clock di riferimento.
- **Sincronizzazione degli orologi logici:** nozione rilassata di tempo, nel quale sincronizziamo fra loro gli orologi logici; si evince l'importanza dell'ordinamento degli eventi.

Leslie Lamport dimostrò che avere un meccanismo di tempo logico e sincronizzazione di orologi logici è più importante della sincronizzazione dei clock fisici.

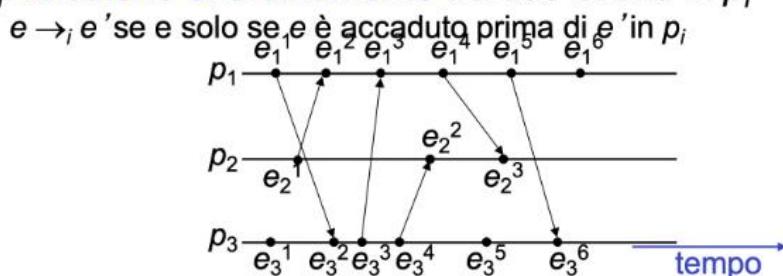
### 2.1. Modello di computazione

Ogni processo genera una sequenza di eventi che possono essere interni (che cambiano lo stato del processo) o esterni (il processo comunica tramite scambio di messaggi con gli altri processi). La notazione per i singoli eventi prevede uso di un pedice e di un apice vicino la e (pedice è il processo che ha generato l'evento e apice è l'ordinamento dell'evento che ha generato il processo).

–  $e_i^k$ : k-esimo evento generato da  $p_i$

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

$\rightarrow_i$ : relazione di ordinamento tra due eventi in  $p_i$



#### 2.1.1. Timestamping

Ogni processo può determinare l'ordinamento dei propri eventi associando ad essi un timestamp basato sul valore del clock fisico dell'orologio fisico del nodo.

**SD sincrono:** caratterizzato da 3 proprietà

- Esistono vincoli sulla velocità di esecuzione di ciascun processo, nello specifico ogni passo del processo in esecuzione è limitato sia con lower che upper bound temporali.

- Ciascun messaggio trasmesso sui canali di comunicazione del sistema distribuito sarà sempre ricevuto in un tempo limitato
- Ciascun processo ha un clock fisico che ha un tasso di scostamento dovuto all'accelerazione o decelerazione che ha rispetto agli altri (**drift rate del clock**) che è conosciuto e limitato.

**SD asincrono:** Non ci sono vincoli su velocità di esecuzione di un processo, sul tempo di trasmissione dei messaggi o sul tasso di scostamento.

È più facile gestire un sistema distribuito asincrono rispetto ad uno sincrono.

### 2.1.2. Sincronizzazione dei clock

È possibile tramite algoritmi ottenere la sincronizzazione fisica.

**Prima soluzione:** tentare di sincronizzare con una certa approssimazione (per via del clock drift rate che ora vedremo) i clock fisici dei processi attraverso opportuni algoritmi. Ogni processo può quindi etichettare gli eventi con il valore del suo clock fisico (che risulta approssimativamente sincronizzato con gli altri clock). Si evince come il timestamping sia basato sulla nozione di tempo fisico (**clock fisico**).

Non risulta possibile mantenere limitata l'approssimazione dei clock fisici in un sistema distribuito asincrono. Infatti in questi sistemi il timestamping non può basarsi sul concetto di tempo fisico; a tale scopo viene introdotto il tempo logico (**clock logico**).

## 2.2. Clock fisico

A un certo istante di tempo  $t$  il sistema operativo può leggere il clock hardware dal computer, producendo un clock software che approssimativamente misura il tempo fisico del processo eseguito su quel nodo. In generale il clock non è completamente accurato e può essere diverso da  $t$  (tempo reale).

- All'istante di tempo reale  $t$ , il sistema operativo legge il tempo dal clock hardware  $H_i(t)$  del computer
- Allora produce il clock software  $C_i(t)$  che misura l'istante di tempo fisico  $t$  per il processo  $i$ .

$$C_i(t) = aH_i(t) + b$$

- La risoluzione del clock è il periodo che intercorre tra due aggiornamenti di valore del clock per poter distinguere due eventi

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

### 2.2.1. Caratteristiche di un clock fisico nei sistemi distribuiti

**Skew:** differenza istantanea fra il valore di due qualsiasi clock

**Drift:** i clock contano il tempo con frequenze differenti, fenomeno dovuto a variazioni fisiche dell'orologio per via del quarzo con cui sono realizzati. Quindi divergono rispetto al tempo reale

**Drift rate:** differenza per unità di tempo di un clock rispetto ad un orologio ideale

Fondamentale il concetto di riallineare i clock per sincronizzarli con i fisici (**processo di sincronizzazione periodica**); anche se differenti clock di un SD vengono sincronizzati in un certo istante, a causa del drift rate, dopo un certo intervallo di tempo, saranno di nuovo disallineati.

### 2.2.2. UTC (Universal Coordinated Time)

È lo standard internazionale per mantenere il tempo, basato sul tempo atomico ma occasionalmente è corretto utilizzando il tempo astronomico. Tempo atomico significa che un secondo equivale al tempo impiegato dall'atomo di cesio 133 per compiere 9192631770 transizioni di stato. I clock fisici che usando oscillatori atomici sono i più accurati. L'output dell'orologio atomico è inviato in broadcast da stazioni radio su terra e da satelliti. I nodi con ricevitori possono sincronizzare i loro clock con questi segnali:

- Segnali da stazioni radio su terra: accuratezza nell'intervallo 1-10 msec.
- Segnali da satellite: accuratezza da 0.5 msec fino a 50msec.

### 2.2.3. Sincronizzazione di clock fisici

**Sincronizzazione esterna:** i clock fisici vengono sincronizzati rispetto ad un clock esterno. È il caso in cui si hanno a disposizione n clock fisici sincronizzati rispetto ad una sorgente UTC per esempio; tanto migliore è la sincronizzazione tanto migliore sarà il valore di alfa. Alfa è una caratteristica che riguarda la sincronizzazione esterna, e rappresenta la **accuratezza**.

$|S(t) - C_i(t)| \leq \alpha$  per  $1 \leq i \leq N$  e per tutti gli istanti in I  
I clock  $C_i$  hanno **accuratezza**  $\alpha$ , con  $\alpha > 0$

**Sincronizzazione interna:** i clock fisici vengono sincronizzati fra di loro. In particolare se andiamo a considerare il clock i-esimo ed il j-esimo, se la loro differenza è minore di un certo valore pi greco diciamo che i clock hanno una precisione pari a pi greco. Pi greco rappresenta la **precisione**, una caratteristica che riguarda il concetto di sincronizzazione interna.

$|C_i(t) - C_j(t)| \leq \pi$  per  $1 \leq i, j \leq N$  nell'intervallo I  
I clock  $C_i$  e  $C_j$  hanno **precisione**  $\pi$ , con  $\pi > 0$

**La sincronizzazione interna ed esterna non si implicano a vicenda, ma in particolare se i clock sono sincronizzati esternamente lo sono anche internamente, non vale il viceversa.** Questo perché tutti i clock sincronizzati internamente potrebbero deviare rispetto ad una sorgente esterna. Se l'insieme dei processi p sincronizzato esternamente con accuratezza alfa allora p sincronizzato anche internamente con precisione  $2\alpha$ .

Un clock hardware è **corretto** se il suo drift rate si mantiene entro un certo intervallo limitato, meno rho e più rho. Allora l'errore che commette per misurare un intervallo di istanti reali sarà limitato.

Quando consideriamo la sincronizzazione fisica i diversi clock fisici possono essere fatti in materiali differenti che ne costituiscono i drift rate e dobbiamo sincronizzarli periodicamente. Quando sincronizzarli?

Presi 2 clock con due tassi di scostamento massimo da UTC uguali pari a p, ipotizziamo che dopo la sincronizzazione i 2 clock si scostino da UTC in senso opposto, quindi dopo delta t si sono scostati di  $2p \Delta t$ . Per garantire che i 2 clock non differiscano mai più di un valore omega occorre sincronizzarli almeno ogni omega fratto  $2p$  secondi.

## 2.3. Algoritmo di sincronizzazione interna tra due processi di un SD sincrono

Siamo nell'ipotesi quindi di un SD sincrono; i passi sono:

- Un processo  $p_1$  manda il suo clock locale  $t$  ad un processo  $p_2$  tramite un messaggio  $m$ .
- $p_2$  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 SD sincrono,  $T_{min} \leq T_{trasm} \leq T_{max}$ . Definiamo  $u = (T_{max} - T_{min})$  l'incertezza sul tempo di trasmissione (ovvero l'ampiezza dell'intervallo).
- Se  $p_2$  imposta il suo clock a  $t + (T_{max} + T_{min})/2$ , il lower bound ottimo sullo skew tra i due clock è pari a  $U/2$ .

Questo algoritmo può essere generalizzato per sincronizzare N processi, con lower bound ottimo sullo skew pari a  $U(1-1/N)$ . Per un SD asincrono  $T_{trasm} = T_{min} + x$ , con  $x \geq 0$  e non noto.

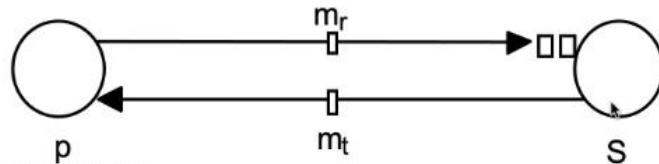
## 2.4. Sincronizzazione mediante time service

Un time service può fornire l'ora con precisione (dotato di un ricevitore UTC o di un clock accurato). Il gruppo di processi che deve sincronizzarsi usa un time service, implementato in modo centralizzato da un solo processo (time server) oppure in modo distribuito da più processi.

- **Time service centralizzati:** *algoritmo di Cristian* (sincronizzazione esterna) e *Berkeley Unix* (sincronizzazione interna).
- **Time service distribuiti:** *Network Time Protocol* (sincronizzazione esterna).

### 2.4.1. Algoritmo di Cristian

Permette la sincronizzazione del clock di un nodo rispetto ad un time server centralizzato garantendo sincronizzazione esterna. Il processo p che vuole sincronizzarsi rispetto al time server S richiede ad S il suo tempo inviandogli un messaggio di richiesta. Il time server risponde con il valore del proprio clock. In questo modo p imposta il suo clock ad s.



- **Osservazioni**

- Un singolo time server potrebbe guastarsi
  - Soluzione: usare un gruppo di time server sincronizzati
- Non prevede time server maliziosi
- Permette la sincronizzazione solo se  $T_{round}$  è breve → adatto per reti locali con bassa latenza

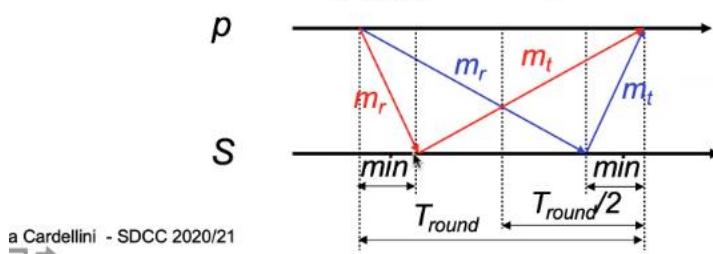
**Accuratezza nell'algoritmo di Cristian:** per valutarla dobbiamo riflettere su due casi estremi:

**Caso 1:** S non può mettere  $t$  in  $m_t$  prima che sia trascorso  $min$  dall'istante in cui  $p$  ha inviato  $m_r$ ,

- $min$  è il minimo tempo di trasmissione tra  $p$  e S

**Caso 2:** S non può mettere  $t$  in  $m_t$  dopo il momento in cui  $m_t$  arriva a  $p$  meno  $min$

- Il tempo su S quando  $m_t$  arriva a  $p$  è compreso in  $[t + min, t + T_{round} - min]$ 
  - L'ampiezza di tale intervallo è  $T_{round} - 2 min$
- Accuratezza  $\leq \pm (T_{round}/2 - min)$



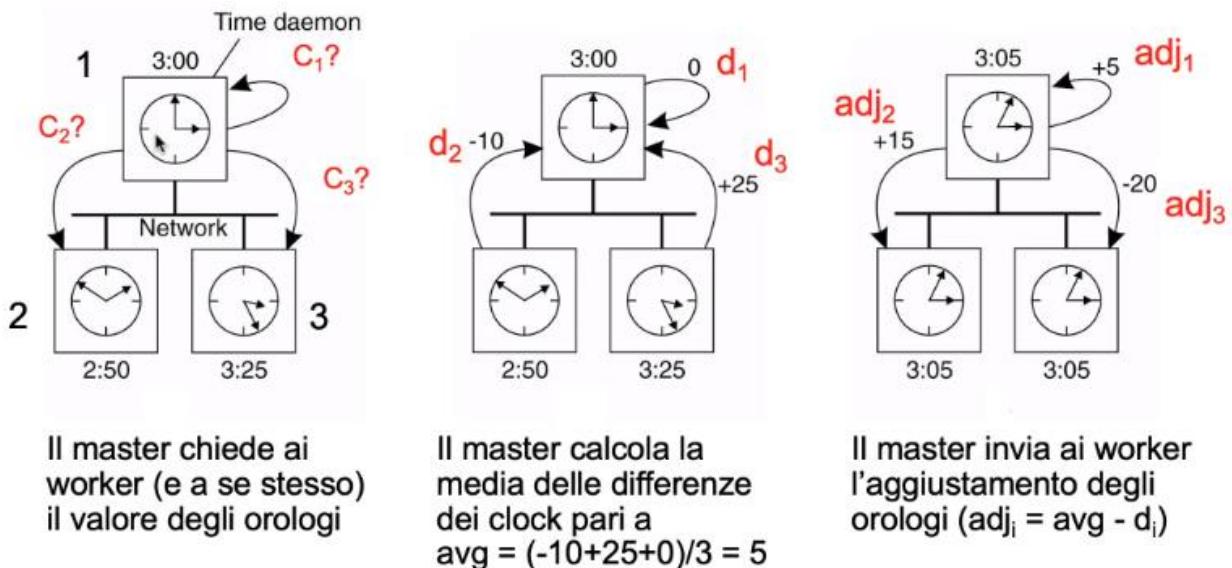
### 2.4.2. Algoritmo di Berkley

Usato per la sincronizzazione interna, che punta a sincronizzare internamente un gruppo di server (un server svolgerà il ruolo di master che farà da time server attivo che coordinerà la sincronizzazione fra gli altri worker).

È un algoritmo per la sincronizzazione interna di un gruppo di macchine. Il master (time server attivo) richiede in broadcast il valore dei clock delle altre macchine(worker). Il master usa i RTT per stimare i valori dei clock dei diversi slave;

-Differenza  $d_i$  tra clock del master M e quello del worker i (in modo simile all'algoritmo di Cristian):  $d_i = (C_M(t_1) + C_M(t_3))/2 - C_i(t_2)$  dove  $C_M(t_1)$  e  $C_M(t_3)$  sono i valori del clock sul master,  $C_i(t_2)$  è il valore del clock sul worker i.

Calcola la media delle differenze dei clock ed invia un valore correttivo opportuno ai worker



Caratteristiche dell'algoritmo di Berkley sono

- L'accuratezza dell'algoritmo dipende da un RTT massimo tra i nodi: il master non considera i valori di clock associati a RTT superiori al massimo (si scartano gli outlier)
- Tolleranza ai guasti (molto più tollerante ai guasti rispetto all'algoritmo di Cristian, perché li avevamo un solo time server e qui se il master fallisce tramite elezione possiamo eleggere un altro master).
- Robustezza Questo algoritmo è più robusto in quanto tollerante rispetto a comportamenti arbitrari di worker che inviano valori errati dei clock. Il master sa quali valori considerare scartando gli outlier qui infatti.

### 2.4.3. Network time protocol NTP

È l'algoritmo più utilizzato in pratica in quanto è il servizio di time usato in internet standardizzato all'interno di un documento (RFC 5905). Questo protocollo assicura una sincronizzazione esterna più accurata rispetto a UTC. Si tratta di un servizio configurabile su diversi SO.

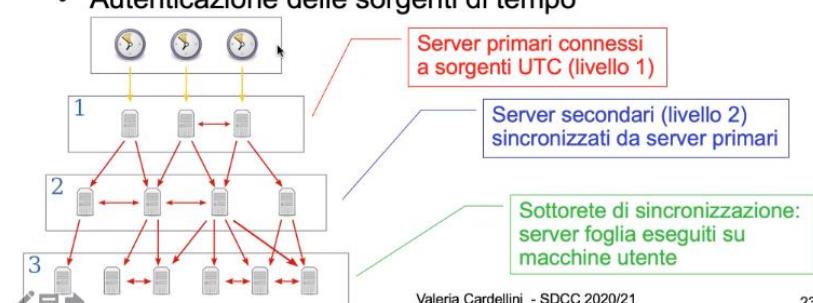
- Time service per Internet (standard in RFC 5905)

- Sincronizzazione esterna accurata rispetto a UTC
- Servizio configurabile su diversi SO. In GNU/Linux: demone `ntpd`, comando `ntpdate` per sincronizzare manualmente

- Architettura di time service disponibile e scalabile

- Time server e path ridondanti

- Autenticazione delle sorgenti di tempo



Fondamentale è l'**utilizzo del protocollo UDP**, considerando che in questo tipo di sistemi si vogliono evitare le latenze il più possibile.

La sottorete di sincronizzazione si riconfigura in caso di guasti

- Server primario che perde la connessione alla sorgente UTC diventa server secondario
- Server secondario che perde la connessione al suo primario (ad es. crash del primario) può usare un altro primario.

**Modi di sincronizzazione:**

- **Multicast:** server all'interno di LAN ad alta velocità invia in multicast il suo tempo agli altri, che impostano il tempo ricevuto assumendo un certo ritardo di trasmissione. Accuratezza relativamente bassa
- **Procedure call:** un server accetta richieste da altri computer (con algoritmo di Cristian). Accuratezza maggiore rispetto a multicast, è utile se non è disponibile multicast
- **Simmetrico:** coppie di server scambiano messaggi contenenti informazioni sul timing. Accuratezza molto alta (per livelli alti della gerarchia)

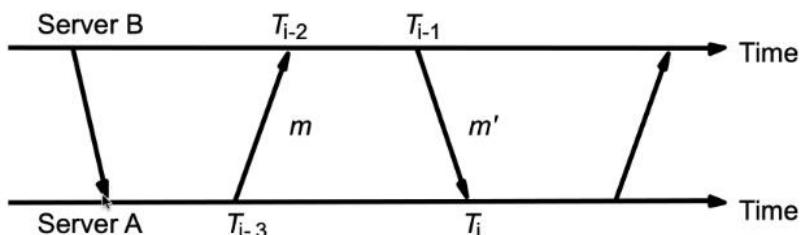
Tutti i modi di sincronizzazione usano UDP.

#### 2.4.3.1. NTP in modo simmetrico

I server si scambiano coppie di messaggi ( $m, m'$ ) per migliorare l'accuratezza della loro sincronizzazione. Ogni messaggio NTP conterrà il timestamp degli eventi recentemente accaduti:

- **Tempo locale di send ( $T_{i-1}$ ) del messaggio corrente  $m'$**
- **Tempo locale di send ( $T_{i-2}$ ) e di receive ( $T_{i-3}$ ) del messaggio precedente  $m$**

Il ricevente di  $m'$  registra il tempo locale  $T_i$ , ed il tempo tra l'arrivo di  $m$  e l'invio di  $m'$   $T_{i-1} - T_{i-2}$  [37] può essere non trascurabile



## NTP: accuratezza

- Per ogni coppia di messaggi  $m$  ed  $m'$  scambiati tra 2 server, NTP **stima l'offset  $o_i$**  tra i 2 clock e il **ritardo  $d_i$**  (pari al tempo totale di trasmissione per  $m$  ed  $m'$ )
- Assumendo che:
  - $o$ : offset reale del clock di B rispetto ad A (differenza tra clock di B e clock di A)
  - $t$  e  $t'$ : tempi di trasmissione di  $m$  ed  $m'$  rispettivamente
$$T_{i-2} = T_{i-3} + t + o \quad e \quad T_i = T_{i-1} + t' - o$$
- $d_i$ , il tempo totale di trasmissione di  $m$  e  $m'$ , è:
$$d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}$$
- Sottraendo le equazioni si ottiene:
$$o = o_i + (t' - t)/2 \quad \text{con } o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2$$
- Poiché  $t, t' > 0$  si può dimostrare che
$$o_i - d_i/2 \leq o \leq o_i + d_i/2$$
- Quindi,  $o_i$  è la **stima dell'offset** e  $d_i/2$  l'**accuratezza di questa stima**
- I server NTP applicano un algoritmo di filtraggio statistico sulle 8 coppie  $\langle o_j, d_j \rangle$  più recenti, scegliendo come stima di  $o$  il valore di  $o_j$  corrispondente al minimo  $d_j$
- Applicano poi un algoritmo di selezione dei peer per modificare eventualmente il peer da usare per sincronizzarsi
  - Vedi <http://www.eecis.udel.edu/~mills/ntp/html/warp.html>
- Accuratezza di NTP:
  - 10 ms su Internet
  - 1 ms su LAN

Perfect synchronization over networks  
is actually impossible

^

Come si calcola l'accuratezza della sincronizzazione simmetrica del protocollo NTP:

**True time (TT)** – meccanismo di sincronizzazione fisica definito da google. TT si basa su una sincronizzazione molto accurata che richiede la presenza di numerosi clock gps, sia molti clock atomici. Richiede dell'hardware specializzato ed un protocollo di sincronizzazione particolare. Google usa questo meccanismo per raggiungere la sincronizzazione fra i server nei datacenter distribuiti geograficamente connessi da fibra ottica. La soluzione è proprietaria e specifica di google e solo lui può permettersela, proprio per l'hardware specializzato.

Finora abbiamo visto la sincronizzazione nei sistemi sincroni, vediamo ora

## 2.5. Tempo nei SD asincroni

Non sono soddisfatti i 3 vincoli che sono soddisfatti nei sincroni (tempo di esecuzione dei passi - conoscenza di upper bound di t trasmissione - tempi di drift rate limitati). I tempi di trasmissione non sono sempre predicibili in questo tipo di sistemi, non risulta quindi sempre possibile calcolare con il tempo fisico l'ordinamento degli eventi. L'unica cosa su cui concordano i processi è l'ordinamento logico dei processi avvenuti, e non il tempo.

Tempo logico: ordinamento degli eventi gestito sulla base di due osservazioni intuitive:

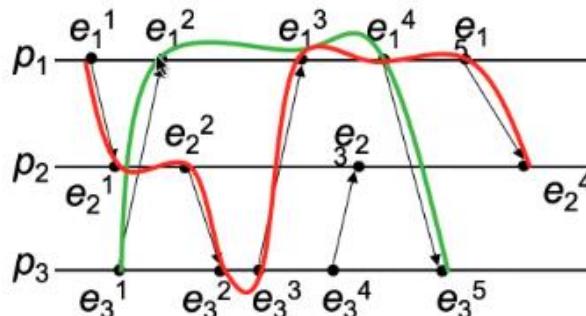
1. Due tempi avvenuti sullo stesso processo si verificano nell'ordine in cui lo stesso processo li ha osservati.
2. Due eventi su due processi differenti ( $p_i$  invia messaggio a  $p_j$ ) sono caratterizzati sempre da un ordine logico in quanto il send sappiamo che deve prevenire il receive.

Leslie Lamport introdusse il concetto di happened-before (nozione di precedenza o ordinamento causale): possiamo ordinare gli eventi per relazione di causa effetto fra loro.

- Indichiamo con  $\rightarrow_i$  la relazione di ordinamento su un processo  $p_i$
- Indichiamo con  $\rightarrow$  la relazione di happened-before tra due eventi qualsiasi

La relazione di happened before rappresenta un ordinamento parziale in quanto valgono le 3 proprietà dell'ordinamento parziale (transitiva, non riflessiva, anti-simmetrica); infatti due eventi si dicono in relazione di happened before se è vera una delle 3 proprietà. Rispettando queste tre condizioni è possibile costruire sequenze di eventi causalmente ordinati.

- La relazione happened-before rappresenta un ordinamento parziale (proprietà: non riflessivo, antisimmetrico, transitivo).
- Non è detto che la sequenza  $e_1, e_2, \dots, e_n$  sia unica.
- Data una coppia di eventi, questa non è sempre legata da una relazione happened-before; in questo caso si dice che gli eventi sono **concorrenti** (indicato da  $\parallel$ ).



- Sequenza  $s_1 = e_1^1, e_2^1, e_2^2, e_3^2, e_3^3, e_1^3, e_1^4, e_1^5, e_2^4$
- Sequenza  $s_2 = e_3^1, e_1^2, e_1^3, e_1^4, e_3^5$
- Gli eventi  $e_3^1$  ed  $e_2^1$  sono concorrenti  
 $e_3^1 \not\rightarrow e_2^1$  ed  $e_2^1 \not\rightarrow e_3^1$

Per comprendere le sequenze vedremo degli algoritmi che sfrutteranno il concetto di tempo logico.

## 2.6. Clock logico scalare

Contatore software che cresce in modo monotono il cui valore non ha alcuna relazione con il clock fisico. Ogni processo che interviene nella computazione distribuita avrà il clock logico  $L$  che userà per applicare i timestamp agli eventi. 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  $L(e) < L(e')$
- **Osservazione:**
  - Se  $L(e) < L(e')$  non è detto che  $e \rightarrow e'$ ; tuttavia,  $L(e) \prec L(e')$  implica che  $e \not\rightarrow e'$

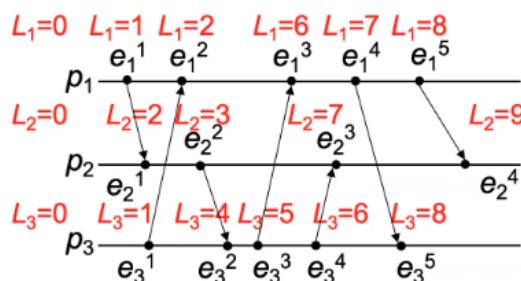
La relazione **happened-before** → introduce un ordinamento parziale degli eventi: nel caso di eventi concorrenti non è possibile stabilire quale evento avviene effettivamente prima.

## 2.7. Algoritmo introdotto da Leslie Lamport per implementare il clock logico scalare

Definisce le regole in base a le quali i processi incrementano il loro clock logico scalare.

- Ogni processo  $p_i$  inizializza il proprio clock logico  $L_i$  a 0 (per ogni  $i=1, \dots, N$ )
- $L_i$  è incrementato di 1 prima che il processo  $p_i$  esegua l'evento (interno oppure esterno di send o receive),  $L_i = L_i + 1$
- Quando  $p_i$  invia il messaggio  $m$  a  $p_j$ 
  - Incrementa il valore di  $L_i$
  - Allega al messaggio  $m$  il timestamp  $t = L_i$
  - Esegue l'evento  $\text{send}(m)$
- Quando  $p_j$  riceve il messaggio  $m$  con timestamp  $t$ 
  - Aggiorna il proprio clock logico  $L_j = \max(t, L_j)$
  - Incrementa il valore di  $L_j$
  - Esegue l'evento  $\text{receive}(m)$

### Clock logico scalare: esempio



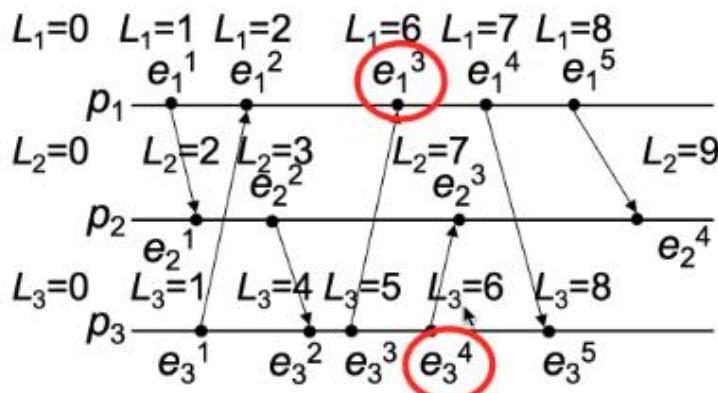
- **Osservazioni**
  - $e_1^1 \rightarrow e_2^1$  ed i relativi timestamp riflettono questa proprietà ( $L_1=1$  e  $L_2=2$ ); infatti if  $e_1^1 \rightarrow e_2^1$  then  $L(e_1^1) < L(e_2^1)$
  - $e_1^1 \parallel e_3^1$  ed i relativi timestamp sono uguali ( $L_1=1$  e  $L_3=1$ ); infatti if  $L(e_1^1) \geq L(e_3^1)$  then  $e_1^1 \not\rightarrow e_3^1$
  - $e_2^1 \parallel e_3^1$  ed i relativi timestamp sono diversi ( $L_2=2$  e  $L_3=1$ ); infatti if  $L(e_2^1) \geq L(e_3^1)$  then  $e_2^1 \not\rightarrow e_3^1$

## 2.8. Relazione di ordine totale

Usando il clock logico scalare, due o più eventi possono avere lo stesso timestamp. Per realizzare un ordinamento totale tra eventi, evitando che due eventi accadano nello stesso tempo logico, si utilizza (oltre il clock logico) il numero del processo in cui è avvenuto l'evento. Si stabilisce quindi un ordinamento totale  $\prec$  tra i processi:

- **Relazione di ordine totale** tra eventi (indicata con  $e \Rightarrow e'$ ): 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:
  1.  $L_i(e) < L_j(e')$  or
  2.  $L_i(e) = L_j(e')$  and  $p_i \prec p_j$
- Applicata negli algoritmi di Lamport distribuito e di Ricart-Agrawala per la mutua esclusione distribuita

ESEMPIO:



- $e_1^3$  e  $e_3^4$  hanno lo stesso valore di clock logico: come ordinarli?
- $e_1^3 \Rightarrow e_3^4$  poiché  $L_1(e_1^3) = L_3(e_3^4)$  e  $p_1 \prec p_3$

## Problema del clock logico scalare

- Il clock logico scalare ha la seguente proprietà
  - Se  $e \rightarrow e'$  allora  $L(e) < L(e')$
- Ma non è possibile assicurare che
  - Se  $L(e) < L(e')$  allora  $e \rightarrow e'$   
Vedi lucido 34:  $L(e_3^1) < L(e_2^1)$  ma  $e_3^1 \parallel e_2^1$

Conseguenza: non è possibile stabilire, solo guardando i clock logici scalari, se due eventi sono concorrenti o meno. Per superare questa limitazione si introducono i clock logici vettoriali.

## 2.9. Clock logico vettoriale

- Il clock logico vettoriale per un sistema con N processi è un vettore di N interi
- Ciascun processo  $p_i$  mantiene il proprio clock vettoriale  $V_i$
- Per il processo  $p_i$ ,  $V_i[i]$  è il clock logico locale.
- Ciascun processo usa il suo clock vettoriale per assegnare il timestamp agli eventi
- Analogamente al clock scalare di Lamport, il clock vettoriale viene allegato al messaggio m ed il timestamp diviene vettoriale
- Con il clock vettoriale si catturano completamente le caratteristiche della relazione happened-before

$$e \rightarrow e' \text{ se e solo se } V(e) < V(e')$$

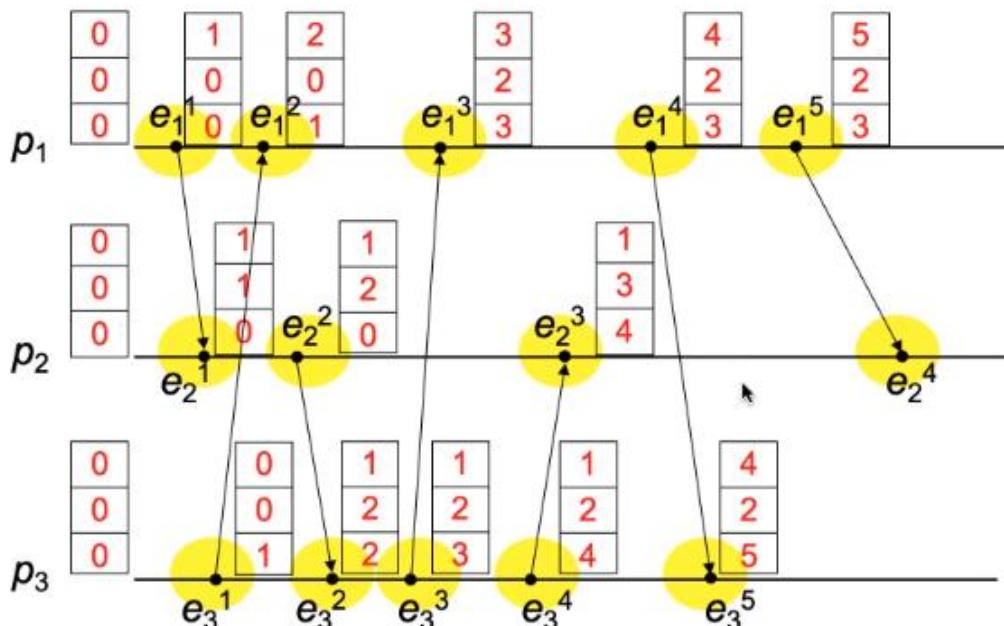
### 2.9.1. Significato e confronto di clock

- Dato il clock vettoriale  $V_i$ 
  - $-V_i[i]$  è il numero di eventi generati da  $p_i$
  - $-V[j]$  con  $i \neq j$  è il numero di eventi occorsi a  $p_j$  di cui  $p_i$  ha conoscenza
    - Confronto di clock vettoriali
  - $-V = V'$  se e solo se per ogni  $j$ :  $V[j] = V'[j]$
  - $-V \leq V'$  se e solo se per ogni  $j$ :  $V[j] \leq V'[j]$
  - $-V < V'$  (e quindi l'evento associato a  $V$  precede quello associato a  $V'$ ) se e solo se
    - Per ogni  $i$  appartenente  $[1, \dots, N]$ :  $V[i] \leq V'[i]$   
**AND**
    - Esiste  $j$  appartenente  $[1, \dots, N]$ :  $V[j] < V'[j]$
  - $-V \parallel V'$  (e quindi l'evento associato a  $V$  è concorrente a quello associato a  $V'$ ) se e solo se
    - $Not(V < V')$  and  $not(V' < V)$

## Clock logico vettoriale: implementazione

- Ogni processo  $p_i$  inizializza il proprio clock vettoriale  $V_i$   
 $V_i[k]=0 \quad \forall k = 1, 2, \dots, N$
- Prima di eseguire un evento (interno o esterno),  $p_i$  incrementa la sua componente del clock vettoriale  
 $V_i[i] = V_i[i] + 1$
- Quando  $p_i$  invia il messaggio  $m$  a  $p_j$ 
  - Incrementa di 1 la componente  $V_i[j]$
  - Allega al messaggio  $m$  il timestamp vettoriale  $t = V_i$
  - Esegue l'evento  $send(m)$
- Quando  $p_j$  riceve il messaggio  $m$  con timestamp  $t$ 
  - Aggiorna il proprio clock logico  $V_j[k] = \max(t[k], V_j[k]) \quad \forall k = 1, 2, \dots, N$
  - Incrementa di 1 la componente  $V_j[j]$
  - Esegue l'evento  $receive(m)$

## Clock logico vettoriale: esempio



# Confronto di clock vettoriali

- Confrontando i timestamp basati su clock vettoriale si può capire se due eventi sono concorrenti o se sono in relazione happened-before

1
2
0

V

1
2
2

V'

$V(e) < V'(e)$  e quindi  $e \rightarrow e'$

1
2
0

V

1
0
2

V'

$V(e) \neq V'(e)$  e quindi  $e \parallel e'$

Il clock vettoriale ci permette di catturare le relazioni di causa effetto, mentre il clock scalare tiene solo conto degli avvenimenti senza riuscire a carpire le relazioni.

Gli eventi sia interni che esterni aumenteranno il contatore del clock, vettoriale o scalare che sia.

La conoscenza dei clock dei processi varia a seconda di quanto essi sono coinvolti nello scambio dei messaggi.

Diremo che una violazione della relazione di causa effetto tra due eventi avviene quando viene ricevuto prima l'effetto poi la causa, quindi ad esempio arriva p causato da c e poi arriva c.

Solo vedendo il clock scalare non si può capire se due eventi sono concorrenti a differenza del clock vettoriale. (?)

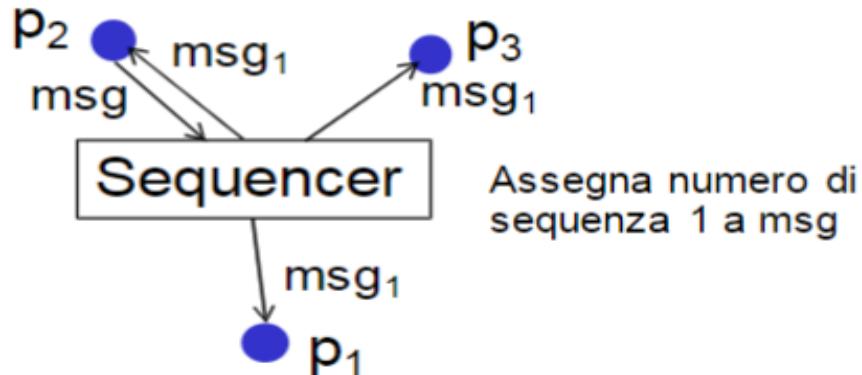
## 2.10. Multicast causalmente e totalmente ordinato

Esaminiamo ora due applicazioni di clock logico scalare e vettoriale:

### 2.10.1. Multicast totalmente ordinato (clock logico scalare)

Il multicast totalmente ordinato si occupa di garantire che aggiornamenti concorrenti su un database replicato siano visti nello stesso ordine da ogni replica. Nello specifico **si tratta di un tipo di multicast dove tutti i messaggi sono consegnati nello stesso ordine ad ogni destinatario**. Affinchè le suddette proprietà siano possibili, serve una comunicazione affidabile FIFO ordered (no perdita messaggi e messaggi consegnati in ordine di invio).

La versione centralizzata di multicast totalmente ordinato prevede un elemento coordinatore chiamato sequencer. Ogni processo si occuperà di inviare il proprio messaggio di update al sequencer. Quest'ultimo assegna ad ogni messaggio di update un numero di sequenza univoco e poi invia il messaggio in multicast a tutti i processi, che eseguono gli aggiornamenti in ordine in base al numero di sequenza. Questa soluzione porta a problemi di scalabilità e single point of failure.



Per risolvere il problema del multicast totalmente ordinato si può adottare una soluzione completamente distribuita tramite l'ausilio di clock logici scalari; posto che ogni messaggio ha come timestamp il clock logico scalare del processo che lo invia:

1.  $P_i$  invia in multicast incluso se stesso il messaggio di update  $msg_i$
2.  $msg_i$  viene posto da ogni processo ricevente  $p_j$  in una coda locale  $q_j$  ordinata in base al valore del timestamp
3.  $P_j$  consegna  $msg_i$  all'applicazione solo quando
  - a.  $msg_i$  è in testa a  $q_j$  e tutti gli ack relativi a  $msg_i$  sono stati ricevuti da  $p_j$
  - b. Per ogni processo  $p_k$  c'è un messaggio  $msg_k$  in  $q_j$  con timestamp maggiore di  $msg_i$

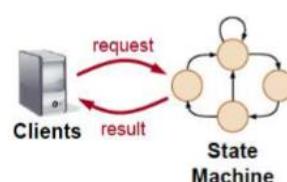
Ovvero, più semplicemente,  $msg_i$  viene consegnato solo quando  $p_j$  sa che nessun altro processo può inviare in multicast un messaggio con timestamp minore o uguale a quello di  $msg_i$

Il costo del multicast totalmente ordinato è  $O(n^2)$ , avendo  $n$  processi che inviano un ack in multicast per ogni messaggio ricevuto ( $n^2$  ack). Si tratta di un meccanismo che è alla base della

### 2.10.2. State machine replication

Si tratta di una tecnica di replicazione software applicata ai servizi che possono essere implementati tramite macchine a stati finiti deterministici, ovvero la transazione di stato dipende solo dallo stato corrente e dal comando che viene eseguito.

Next state of the service =  
 $f(\text{current state}, \text{command executed})$



Per garantire tolleranza ai guasti, il servizio è replicato su diversi nodi, ognuno dei quali mantiene in esecuzione una SMR middleware (SMR=state machine replication).

I diversi set di repliche si comportano come server centralizzati; ogni server mantiene lo stato del servizio, che deve essere lo stesso servizio replicato per ogni server. Ogni singola replica deve eseguire i comandi nello stesso ordine: se ciò avviene, e se effettivamente gli stati sono deterministici, allora tutte le repliche finiranno nello stesso stato. Il servizio progredisce finché una maggioranza delle repliche restano in esecuzione.

### 2.10.3. Multicast causalmente ordinato (clock logico vettoriale)

Un messaggio viene consegnato solo se tutti i messaggi che lo precedono (causalmente – relazione di causa effetto) sono stati già consegnati. Per relazione di causa effetto in un sistema distribuito non si intende solo come relazione effettiva, ma anche come relazione potenziale, ovvero il secondo evento (effetto) è potenzialmente influenzato dal primo evento che è la causa. Questo multicast causalmente ordinato è più debole di quello totalmente ordinato, perché in quest'ultimo dobbiamo consegnare tutti i messaggi nello stesso ordine a tutti i processi. In questo vogliamo consegnare in ordine solo quelli che sono in relazione di causa effetto. Dei messaggi non correlati da causa effetto, quindi concorrenti, non ci interessa l'ordine di consegna ai processi, potrebbero perciò arrivare in ordine differente.

- Esempio:

- $p_1$  invia i messaggi  $m_A$  ed  $m_B$
- $p_2$  invia i messaggi  $m_C$  ed  $m_D$
- $m_A$  causa  $m_C$
- Alcune sequenze di consegna compatibili con l'ordinamento causale (e FIFO ordered) sono:

$m_A \ m_B \ m_C \ m_D \quad m_A \ m_C \ m_B \ m_D \quad m_A \ m_C \ m_D \ m_B$

ma NON  $m_C \ m_A \ m_B \ m_D$

Assumiamo che la comunicazione sia affidabile (nessun messaggio perso, duplicato o spurio - ovvero che qualcuno lo riceve ma nessuno lo ha inviato) e FIFO ordered. Applicando i clock logici vettoriali per risolvere il problema del multicast ordinato in modo completamente distribuito, otteniamo il seguente algoritmo del multicast causalmente ordinato.

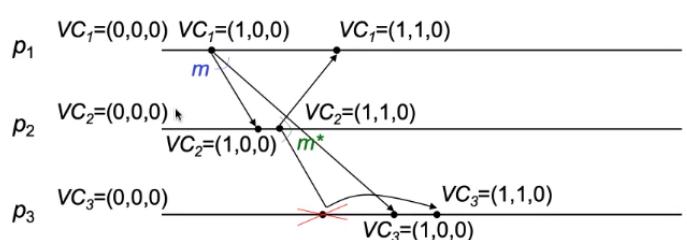
- $p_i$  invia il messaggio  $m$  con timestamp  $t(m)$  basato sul clock logico vettoriale  $V_i$ 
  - $V_i[i]$  conta il numero di messaggi da  $p_i$  a  $p_j$
- $p_j$  riceve  $m$  da  $p_i$  e ne ritarda la consegna al livello applicativo (ponendo  $m$  in una coda di attesa) finché non si verificano entrambe le condizioni
  1.  $t(m)[i] = V_i[i] + 1$   
 $m$  è il messaggio successivo che  $p_j$  si aspetta da  $p_i$
  2.  $t(m)[k] \leq V_i[k]$  per ogni  $k \neq i$   
per ogni processo  $p_k$ ,  $p_j$  ha visto almeno gli stessi messaggi visti da  $p_i$ 
    - $p_1$  invia  $m$  a  $p_2$  e  $p_3$
    - $p_2$ , dopo aver ricevuto  $m$ , invia  $m^*$  a  $p_1$  e  $p_3$
    - Supponiamo che  $p_3$  riceva  $m^*$  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  $m$

#### Esempio di applicazione multicast totalmente ordinato:

Supponiamo di avere  $p_1$ ,  $p_2$  e  $p_3$  processi.

Aggiornamento clock

$p_i$  invia msg:  $VC_i[i] = VC_i[i] + 1$   
 $p_i$  riceve msg con  $t(msg)$ :  $VC_i[k] = \max\{VC_i[k], t(msg)[k]\}$



## 2.11. Mutua esclusione e sistemi concorrenti

La mutua esclusione nasce nei sistemi concorrenti, dove diversi processi vogliono accedere ad una risorsa condivisa e dove ogni processo vuole acquisire la risorsa ed utilizzarla in modo esclusivo senza avere interferenze con gli altri processi. Ogni algoritmo di mutua esclusione comprende una sequenza di istruzioni chiamata **sezione critica (CS)** che consiste nell'accesso alla risorsa condivisa. La sequenza di istruzioni che precede la sezione critica si chiama **trying protocol (TP)**, quella che la segue invece si chiama **exit protocol (EP)**. Un algoritmo di mutua esclusione è caratterizzato da:

1. **Mutua esclusione (ME):** detta anche proprietà di **safety**, al più un solo processo alla volta può eseguire la CS
2. **No deadlock (ND):** se un processo rimane bloccato nella sua trying section, esistono uno o più processi che riescono ad entrare ed uscire dalla CS
3. **No starvation (NS):** detta anche assenza di posticipazione indefinita, nessun processo può rimanere bloccato per sempre nella trying section, ovvero le richieste di ingresso e uscita dalla sezione critica sono prima o poi soddisfatte.

Importante notare:

- NS implica ND
- NS è una proprietà di **liveness** (no ritardi indefiniti)
- L'assenza di starvation è una condizione di imparzialità (**fairness**) nei confronti dei processi
- L'ordering è un ulteriore requisito di imparzialità; le richieste di accesso in CS sono servite nell'ordine di arrivo.

La sezione critica è relativa ad una porzione di codice; esistono delle soluzioni basate su variabili condivise che servono a realizzare la mutua esclusione tra N processi: si tratta di:

**Algoritmo di Dijkstra (1965):** Ideato per sistemi a singolo processore, serve a garantire la mutua esclusione ed assenza di deadlock; non garantisce l'assenza di starvation tuttavia.

**Algoritmo del panificio di Lamport (1974):** Ideato per sistemi multiprocessore a memoria condivisa.

### 2.11.1. Algoritmo del panificio di Lamport

È una soluzione semplice ispirata ad una situazione reale (attesa di essere serviti in un panificio). Il modello di sistema concorrente per Lamport è costituito come segue:

- I processi comunicano leggendo/scrivendo variabili condivise
- Letture e scrittura di una variabile non sono operazioni atomiche: un processo può scrivere mentre un altro processo sta leggendo.
- Ogni variabile condivisa è di proprietà di un processo. Tutti possono leggere la sua variabile ma solo il processo proprietario potrà scriverla.
- Nessun processo può eseguire due scritture contemporaneamente
- Le velocità di esecuzione dei processi non sono correlate tra loro

Processo entra in panificio e prende biglietto, mette a true componente i esima la componente dell'array choosing, e sceglie il biglietto andando a vedere il massimo dei valori dell'array num incrementandolo di 1.

- Ciclo ripetuto all'infinito dal processo  $p_i$

```

// sezione non critica
// prende un biglietto
choosing[i] = true; // inizio della selezione del biglietto
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
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

```

- Ciclo ripetuto all'infinito dal processo  $p_i$

```

// sezione non critica
// prende un biglietto

```

```

choosing[i] = true; // inizio della selezione del biglietto
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

```

```
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
```

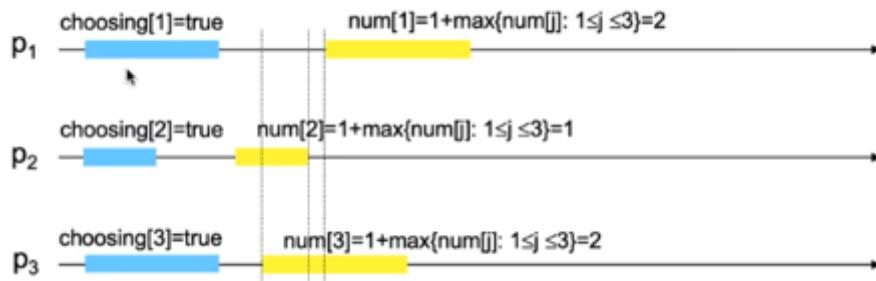
doorway

bakery


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

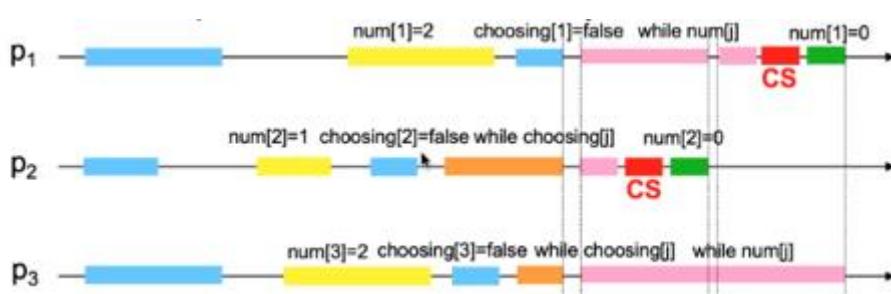
### 2.11.1.1. Doorway:

1. Ogni processo  $p_i$  che entra lo segnala agli altri tramite il choosing[i]
2. Prende un numero di biglietto pari al massimo dei numeri scelti dai processi precedenti più 1.
3. Altri processi possono accedere concorrentemente alla doorway
4. Esempio di esecuzione della doorway



### 2.11.1.2. Bakery:

- Ogni processo  $p_i$  deve controllare che tra i processi in attesa sia lui il prossimo a poter accedere alla CS
- Il primo ciclo permette a tutti i processi nella doorway di terminare la scelta del biglietto
- Il secondo ciclo lascia  $p_i$  in attesa finché:
  - Il suo numero di biglietto non diventa il più piccolo
  - Tutti i processi che hanno scelto un numero di biglietto uguale al suo non hanno identificativo maggiore
  - Osservazione:
- I casi in cui viene scelto lo stesso numero di biglietto sono risolti basandosi sull'identificativo del processo
  - Esempio di esecuzione del bakery



### 2.11.1.3. Le proprietà dell'algoritmo del panificio di Lamport

- Proprietà di mutua esclusione

Deriva da: se  $p_i$  è nella doorway e  $p_j$  è nel bakery allora  $\{num[j], j\} < \{num[i], i\}$

- Proprietà di no starvation

Nessun processo attende per sempre, poiché prima o poi avrà il numero di biglietto più piccolo

- L'algoritmo gode anche della proprietà FCFS (First Come First Served)

Se  $p_i$  entra nel bakery prima che  $p_j$  entri nella doorway, allora  $p_i$  entrerà in CS prima di  $p_j$

## 2.12. Mutua esclusione distribuita

Rispetto al modello di Lamport aggiungiamo il vincolo di comunicazione tramite scambio di messaggi:

- Un processo non può leggere direttamente il valore di una variabile di proprietà di un altro processo, ma deve inviare un messaggio di richiesta ed attendere un messaggio di risposta contenente il valore

La comunicazione tra processi avviene in base alle seguenti assunzioni:

- I processi non hanno variabili condivise ma comunicano tramite scambio di messaggi
- Il ritardo di trasmissione di un messaggio è impredicibile ma finito
- I canali di comunicazione tra i processi sono affidabili

### 2.12.1. Mutua esclusione distribuita: modello del sistema

- Consideriamo un sistema di  $N$  processi  $p_i, i = 1, \dots, N$
- Il sistema è asincrono
- I processi non sono soggetti a fallimenti
- La comunicazione è affidabile e FIFO ordered
- I processi trascorrono un tempo finito nella CS

### 2.12.2. Adattamento dell'algoritmo del panificio

- Proviamo ad adattare ai SD l'algoritmo del panificio di Lamport ideato per i sistemi concorrenti
- Costo totale pari a  $6N$  messaggi
  - Per accedere alla CS servono 3 scambi di messaggi (1 per doorway, 2 per bakery)
  - Per ogni "lettura" vengono scambiati  $2N$  messaggi.
- La latenza per un singolo scambio di messaggi è uguale al tempo impiegato dalla combinazione canale di comunicazione-processo più lenta
- Quindi: scarsa efficienza e scalabilità
- I problemi derivano dalla mancanza di cooperazione tra i processi

Ogni processo  $p_i$  si comporta da server rispetto alle proprie variabili  $num[i]$  e  $choosing[i]$

Comunicazione basata non su memoria condivisa ma su scambio di messaggi

- Ogni processo legge i valori locali agli altri processi tramite messaggi di richiesta-risposta

### 2.12.3. Paronamica sugli algoritmi per ME distribuita

I principali criteri usati per valutare gli algoritmi di ME distribuita sono il numero di messaggi scambiati per l'ingresso e l'uscita dalla CS (misura indiretta della banda di rete consumata e del tempo di attesa). Distinguiamo:

- **Algoritmi basati su autorizzazioni**

- Un processo che vuole accedere ad una risorsa condivisa chiede l'autorizzazione
- Autorizzazione gestita:

In modo centralizzato (unico coordinatore)

In modo completamente distribuito (algoritmo di Lamport distribuito, algoritmo di Ricart e Agrawala)

- **Algoritmi basati su token**

-Tra i processi circola un messaggio speciale, detto token

Il token è unico in ogni istante di tempo

Solo chi detiene il token può accedere alla risorsa condivisa

-Algoritmo centralizzato e decentralizzato

- **Algoritmi basati su quorum (o votazione)**

- Si richiede il permesso di accedere ad una risorsa condivisa solo ad un sottoinsieme di processi
- Algoritmo di Maekawa

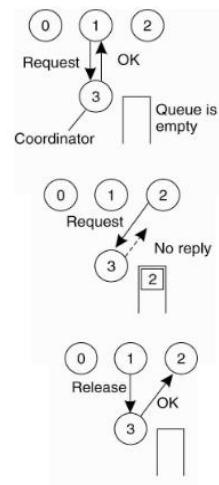
### 2.13. Algoritmi basati su autorizzazioni

Distinguiamo soluzioni centralizzate e decentralizzate

#### 2.13.1. Algoritmo centralizzato

garantisce mutua esclusione, assenza di starvation e fairness in quanto le code sono gestite secondo politica FIFO; si tratta di un algoritmo a basso costo in termine di comunicazione perché per ogni ingresso e uscita in sezione critica vengono scambiati solo 3 messaggi, ma soffre di tutti i svantaggi di centralizzazione, ovvero il coordinatore può essere single point of failure o collo di bottiglia, oltre un problema legato alla perdita del messaggio di rilascio nel caso in cui il processo fallisca mentre si trova in CS. I passi dell'algoritmo sono:

- La richiesta di accesso (ENTER) ad una risorsa in mutua esclusione viene inviata al coordinatore centrale
- Se la risorsa è libera, il coordinatore informa il mittente che l'accesso è consentito (GRANTED)
- Altrimenti, il coordinatore accoda la richiesta con politica FIFO e informa il mittente che l'accesso non è consentito (DENIED); nel caso di un SD sincrono invece non risponde
- Il processo che rilascia la risorsa ne informa il coordinatore (RELEASED)
- Il coordinatore preleva dalla coda la prima richiesta in attesa e invia GRANTED al suo mittente



## 2.13.2. Algoritmo decentralizzato: algoritmo di Lamport distribuito

L'algoritmo di Lamport distribuito ha un costo di comunicazione pari a  $3N$  dove  $N$  è il numero di processi che partecipano, Ricart e Agrawala ha invece un costo pari a  $2n$ .

L'obiettivo di questo algoritmo è quello di ottenere un ordinamento totale di accesso alla sezione critica in modo distribuito. Il processo che vuole entrare in sezione critica invia a tutti gli altri processi un messaggio di richiesta dove manda il suo timestamp in valore di clock scalare. Ogni processo ha la sua coda, il richiedente aggiunge la propria richiesta alla propria coda locale. Chi riceve la richiesta memorizza nella sua coda ed invia al processo che ha inviato messaggio il messaggio di ACK. Chi ha inviato la richiesta potrà accedere alla sezione critica (prima condizione) se e solo se la sua richiesta precede tutti gli altri messaggi presenti all'interno della coda, ovvero il valore del timestamp della sua richiesta (coppia fatta da id del processo e clock logico scalare) è il minimo. Seconda condizione, il processo  $i$ -esimo deve ricevere da ogni altro processo un messaggio di nuova richiesta con timestamp maggiore di quando ha mandato il suo.

- Ogni processo mantiene un **clock logico scalare** ed una **coda locale** (per memorizzare le richieste di accesso alla CS)  
-Si applica anche la **relazione di ordine totale** →
- Regole dell'algoritmo:
  - $p_i$  richiede l'accesso in CS:  $p_i$  invia a tutti gli altri processi un **messaggio di richiesta** avente come timestamp il suo clock scalare ed aggiunge la coppia <richiesta, timestamp> alla sua coda
  - $p_j$  riceve una richiesta da  $p_i$ :  $p_j$  memorizza la richiesta (con il timestamp) nella sua coda ed invia a  $p_i$  un **messaggio di ack**
  - $p_i$  accede alla CS se e solo se:
    - La richiesta di  $p_i$  con timestamp  $t$  precede tutti gli altri messaggi di richiesta in coda (ossia  $t$  è il minimo applicando →)
    - $p_j$  ha ricevuto da ogni altro processo un messaggio (di ack o nuova richiesta) con timestamp maggiore di  $t$  (applicando →)
  - $p_i$  rilascia la CS:  $p_j$  elimina la richiesta dalla sua coda ed invia un **messaggio di release** a tutti gli altri processi
  - $p_j$  riceve un messaggio di release: elimina la richiesta corrispondente dalla sua coda

Un processo che desidera entrare in sezione critica prima o poi entrerà. I processi non si bloccano a vicenda, c'è fairness perché le richieste vengono eseguite nell'ordine del timestamp. In termini di costo di comunicazione sono necessari  $n-1$  messaggi di richiesta (per il processo che vuole entrare in sezione critica),  $n-1$  messaggi di ack e  $n-1$  messaggi di release. Costo di comunicazione  $3n$ .

### 2.13.3. Algoritmo decentralizzato: algoritmo di Ricart e Agrawala

#### Algoritmo di Ricart e Agrawala

- Estensione ed ottimizzazione dell'algoritmo di Lamport distribuito
  - Basato su **clock logico scalare** e **relazione d'ordine totale**
- Un processo che vuole accedere alla CS manda un **messaggio di REQUEST** a tutti gli altri contenente:
  - proprio identificatore
  - *timestamp* basato su clock logico scalare
- Si pone in attesa della risposta da tutti gli altri
- Ottenuti tutti i **messaggi di REPLY** entra nella CS
- All'uscita dalla CS manda REPLY a *tutti* i processi nella coda locale
- Un processo che riceve il messaggio di REQUEST può
  - non essere nella CS e non volervi accedere → manda REPLY al mittente
  - essere nella CS → non risponde e mette il messaggio in coda locale
  - 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

Hanno cercato di migliorare l'algoritmo di Lamport, riuscendoci portando il costo a  $2n$ . Si tratta di un'estensione ed un'ottimizzazione del suddetto.

Rispetto a Lamport infatti viene eliminato un messaggio facendo scendere il costo. Anche Agrawala usa ordine totale e clock logico scalare.

#### Algoritmo di Ricart e Agrawala

- Variabili locali per ciascun processo
  - #replies: numero di risposte ricevute (inizializzata a 0)
  - State  $\in \{\text{Requesting}, \text{CS}, \text{NCS}\}$  (inizializzata a NCS)
  - Q: coda di richieste pendenti (inizialmente vuota)
  - Last\_Req: timestamp del messaggio di richiesta (inizializzata a 0)
  - Num (inizializzata a 0): clock logico scalare
- Ogni processo  $p_i$  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;

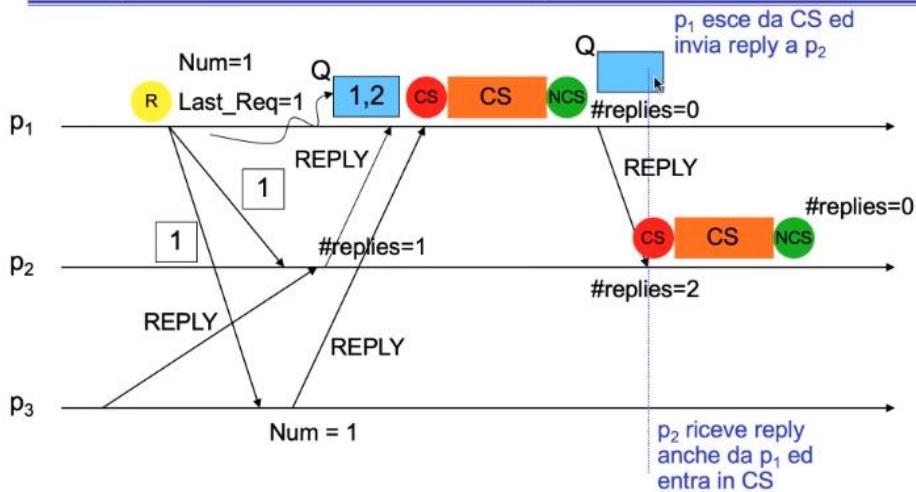
##### Upon receipt REQUEST( $t$ ) from $p_j$

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 REPLY from $p_j$

1. #replies = #replies+1

## Algoritmo di Ricart e Agrawala: esempio



### Vantaggi di Ricart e Agrawala:

- È un algoritmo completamente distribuito, non ci sono elementi di centralizzazione potenziali single point of failure
- rispetto a Lamport è meno costoso in termini di comunicazione in quanto ci sono 3 messaggi (richiesta – ack – release). Qui invece non abbiamo il messaggio di release, ed il messaggio di ack è differente all'uscita della sezione critica. Abbiamo quindi n-1 messaggi di richiesta che sarà seguito da n-1 messaggi di reply (2(N-1) messaggi).

### Svantaggi di Ricart e Agrawala:

- Se un processo fallisce, nessun altro potrà entrare nella CS
- Tutti i processi possono essere collo di bottiglia in quanto ogni processo partecipa ad ogni decisione.

## 2.14. Algoritmi basati su token

In questi algoritmi c'è un token nel sistema, che permette al processo lo detiene di accedere alla sezione critica. Possiamo dividere gli algoritmi basati su token in due classi: centralizzati o decentralizzati.

- **Centralizzati:** approccio di **token-asking** dove la gestione del token è centralizzata, offerta in particolare da un coordinatore eletto tramite algoritmo di elezione. Nel centralizzato token asking c'è un coordinatore eletto tramite algoritmo di elezione che gestisce il token concedendolo ai processi che ne fanno richiesta.
- **Decentralizzati:** approccio **perpetuum mobile** perché in questo caso il token si muove nel sistema arrivando a tutti quanti i processi prima o poi, così che quello che desidera accedere alla sezione critica lo trattenga, per poi rimetterlo in circolo quando esce.

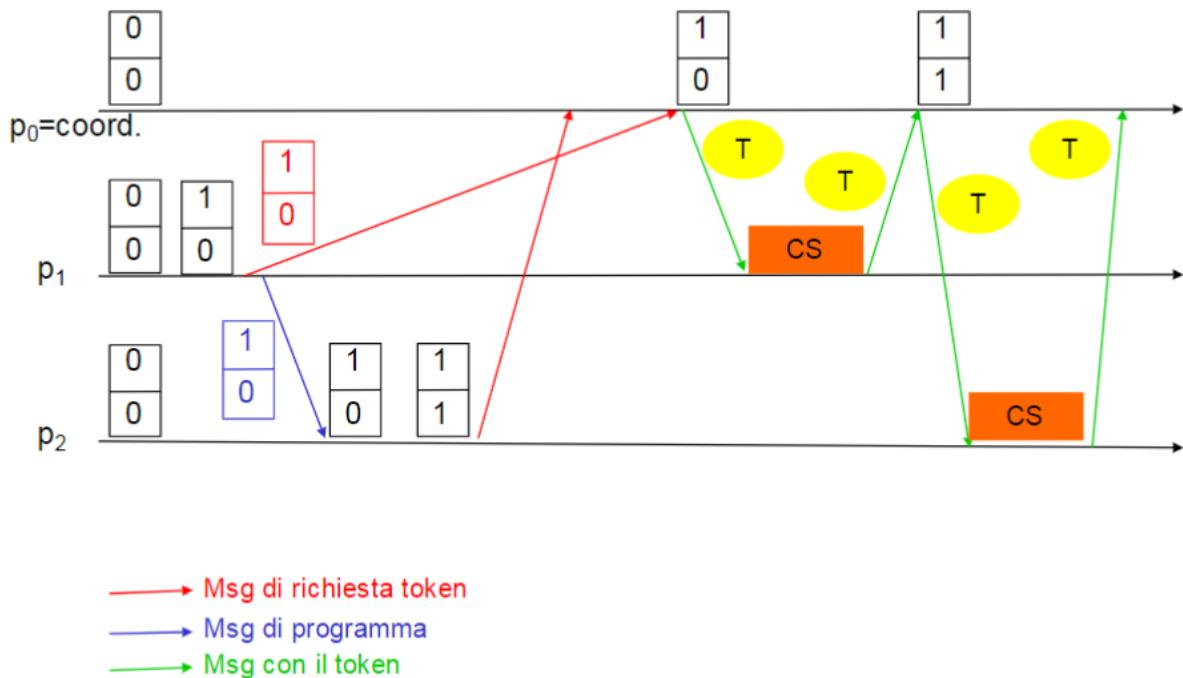
### 2.14.1. Algoritmo basato su token centralizzato

Esiste un processo coordinatore che tiene traccia delle richieste degli altri processi per richiedere il token. I processi utilizzano il clock vettoriale, così da poter avere un ordinamento totale grazie alle relazioni di causa effetto (relazione di happened before). Chi vuole entrare in sezione critica invia la richiesta al coordinatore allegando il proprio clock vettoriale come timestamp. Il coordinatore infila questa richiesta in una coda di richieste pendenti. Quando la richiesta diventerà eleggibile il coordinatore gli manderà il token. All'uscita dalla sezione critica restituirà il token al coordinatore.

- Strutture dati del coordinatore:
  - Reqlist
    - Lista delle richieste pendenti, ricevute dal coordinatore ma non ancora servite
  - V
    - Array di dimensione pari al numero di processi
    - $V[i]$ : numero di richieste di  $p_i$  già servite
- Regole:
  - All'arrivo della richiesta di token da parte di  $p_i$  con timestamp  $T_{pi}$   
 $\text{Reqlist} = \text{Reqlist} \cup \{p_i, T_{pi}\}$
  - Quando la richiesta di  $p_i$  diventa eleggibile ( $T_{pi} < V$ ) e il coordinatore ha il token  
 Invia token a  $p_i$

- Strutture dati di ogni processo:
  - VC
    - Clock vettoriale per il timestamp dei messaggi
- Regole:
  - All'invio della richiesta di token:  
 $VC[i] = VC[i] + 1$   
 Invia msg al coordinatore (con timestamp VC)
  - Alla ricezione del token  
 Entra nella CS
  - All'uscita dalla CS  
 Invia token al coordinatore
  - All'invio del messaggio di programma  
 Invia msg (con timestamp VC)
  - Alla ricezione del messaggio di programma  
 $VC = \max(VC, \text{msg}.VC)$

## Algoritmo basato su token centralizzato: esempio



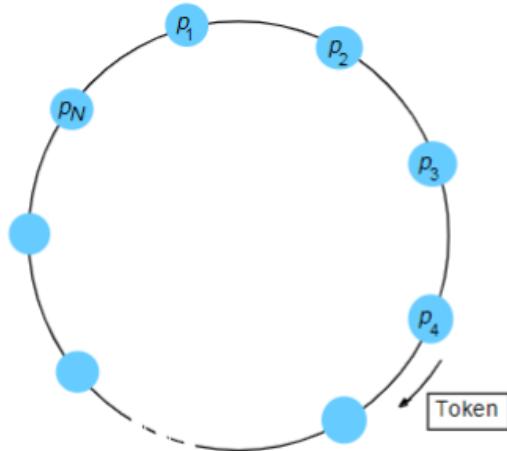
- **Vantaggi** dell'algoritmo basato su token centralizzato:
  - Efficiente in termini di numero di MSG scambiati con il coordinatore ( e se considerassimo anche i MSG di programma?)
  - Garantita anche **fairness** e **ordering**
- **Svantaggi** dell'algoritmo basato su token centralizzato:
  - Coordinatore: singolo point of failure e collo di bottiglia per scalabilità
  - In caso di crash del coordinatore occorre un algoritmo di elezione per determinare il nuovo coordinatore

Ricard e Agrawala a differenza dell'algoritmo basato su token centralizzato non soddisfa la proprietà di ordering, in quanto può accadere che una richiesta a parità di clock scalare vince su un'altra per ID minore.

## 2.14.2. Algoritmo basato su token decentralizzato

Ipotizziamo che i processi che partecipano a mutua esclusione siano organizzati in una overlay network ad anello. L'anello è percorso dal token in modo unidirezionale. Di solito questa topologia non c'entra niente con i collegamenti fisici veri e propri dei nodi. Il processo che ha il token accede alla sezione critica e quando ha fatto lo rimette in circolo dandolo al suo successivo dell'anello.

Se l'anello è percorso in maniera unidirezionale l'algoritmo garantisce la fairness; Qualora fosse bidirezionale la circolazione del token, la fairness non ci sarebbe poiché potrebbe succedere che un processo non riceva mai il token (starvation di quel processo). A differenza del centralizzato c'è un miglior gestione del carico. Svantaggi dell'algoritmo sono che il token deve circolare in modo continuo anche nel caso in cui nessun processo vuole entrare in sezione critica. Nei precedenti non c'era spreco di banda di rete in quanto non c'era scambio di messaggi se nessuno voleva entrare in sezione critica. Questo invece spreca banda per mantenere sempre in circolo token. Poi il token va rigenerato se viene perso, se un processo fallisce o crasha bisogna riconfigurare l'anello.



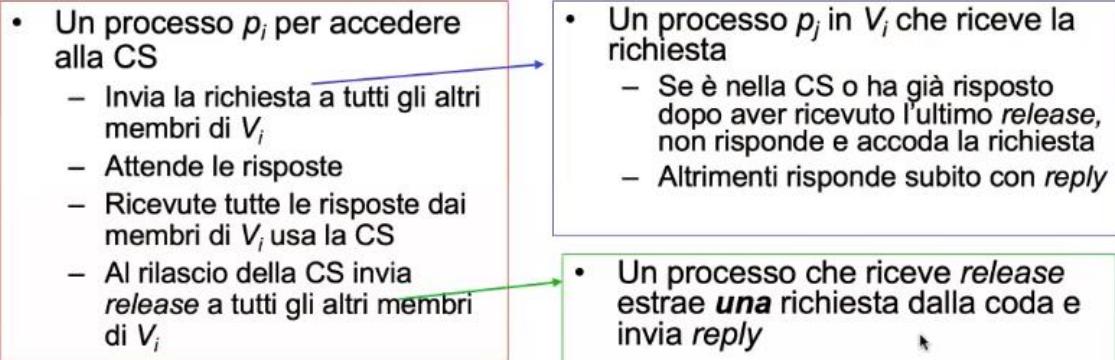
1. Vantaggi dell'algoritmo basato su token decentralizzato:
  - a. Se l'anello è unidirezionale, viene garantita anche la fairness
  - b. Rispetto al centralizzato, migliora il bilanciamento del carico
2. Svantaggi dell'algoritmo basato su token decentralizzato:
  - a. Consumo di banda di rete per trasmettere il token anche quando nessuno chiede l'accesso alla CS
  - b. In caso di perdita del token occorre rigenerarlo: perdita anche per malfunzionamenti hw/sw
  - c. Crash dei singoli processi, poiché se un processo fallisce, occorre riconfigurare l'anello; se fallisce il processo che possiede il token, occorre eleggere il prossimo processo che avrà il token
  - d. Guasti temporanei possono portare alla creazione di token multipli

## 2.15. Algoritmi basati su quorum

Frequenti nei sistemi distribuiti, si basano sull'idea che per entrare in sezione critica bisogna raccogliere i voti non da tutti i processi nel sistema, ma solo da un sottoinsieme di processi chiamato quorum. La votazione avviene all'interno di questo sottoinsieme di processi.

### Algoritmi basati su quorum

- Idea: per entrare in CS occorre raccogliere voti solo da un sottoinsieme di processi (**quorum**), non da tutti
- Votazione all'interno del sottoinsieme
  - I processi votano per stabilire chi è autorizzato ad entrare in CS
  - Un processo può votare per un solo processo
- **Insieme di votazione**  $V_i$ : sottoinsieme di  $\{p_1, \dots, p_N\}$ , associato ad ogni processo  $p_i$



### 2.15.1. Algoritmo di Maekawa

È un esempio di algoritmo basato su quorum: sottoinsieme di votazioni la cui cardinalità è inferiore rispetto alla totalità dei processi che devono sincronizzarsi tra loro.

All'inizio ogni processo usa 2 variabili:

1. **State**: stato in cui si trova il processo
  - a. Se non è in sezione critica e non ha espresso voto: *RELEASED*
  - b. Se vuole entrare in sezione critica: *WANTED*
  - c. Se è in sezione critica: *HELD*
2. Variabile booleana **voted**

Ogni processo  $p_i$  esegue l'algoritmo:

## Inizializzazione

```
state = RELEASED  
voted = FALSE
```

## Processo $p_i$ che vuole entrare in sezione critica

```
state = WANTED
```

invia in multicast a tutti gli altri processi appartenenti a  $V_i$  un messaggio di request, e aspetta fino a quando non riceve risposte da parte di tutti i processi, ossia finché il numero di riposte ricevute = K, con K = cardinalità  $V_i$ .

```
state = HELD
```

## Alla ricezione di una richiesta da $p_j$ ( $i \neq j$ )

```
if (state = HELD or voted = TRUE) then  
    accoda request da pj senza rispondere;  
else  
    invia reply a pj; // vota a favore di pj  
    voted = TRUE;  
end if
```

## Protocollo di uscita dalla sezione critica per $p_i$

- Il processo  $i$  che esce dalla sezione critica mette stato = RELEASED
- invia in multicast il msg di release a tutti i processi appartenenti a  $V_i$ , per denotare il fatto che sta terminando il round di votazione, e i processi saranno abilitati a poter esprimere un nuovo voto.
- Se la coda di richiesta è non vuota:
  - il processo estrae la prima richiesta in coda
  - invia il voto a favore del processo che ha mandato la richiesta (msg reply)
  - voted = TRUE
- altrimenti voted = FALSE e può ancora esprimere il suo voto se gli arriverà una richiesta di votazione.

## Alla ricezione di un messaggio di release da $p_j$ ( $i \neq j$ )

- Se il processo  $i$  riceve msg release da parte di  $j$ , il processo  $i$  valuta lo stato della coda di richieste:
- se ci sono richieste in coda estrae la prima richiesta in coda e invia voto a favore del proc  $j$  e imposta voted = TRUE
- altrimenti voted = FALSE

### 2.15.1.1. Definizione dell'insieme di votazione

## Il punto critico di un algoritmo basato su quorum è la costruzione dell'insieme di votazione.

Insieme votazione  $V_i$  viene definito in modo da garantire che :

- Venga assicurata safety algoritmo, ossia, poiché è un algoritmo di mutua esclusione distribuita, la proprietà safety vuol dire **1 solo processo alla volta è nella sezione critica**.

Questa proprietà si ottiene così:

Poiché 1 processo in un round può votare per 1 solo processo alla volta, quindi 1 solo voto, se abbiamo insieme di votazione a intersezione non nulla, un processo che ha già votato e appartiene a un altro insieme di votazione non può votare per un altro insieme di votazione a cui appartiene, quindi:

Esiste sempre un' intersezione tra due coppie di insiemi di votazione in cui sono suddivisi i processi appartenenti al sistema.

- Far sì che la decisione relativa all'algoritmo sia equamente distribuita tra tutti i processi, ossia **ogni processo faccia lo stesso sforzo**.

Tutti gli insiemi di votazione hanno la stessa cardinalità pari a K (k variabile dell'algoritmo di Maekawa)

- Far sì che ogni processo sia equamente responsabile delle decisioni prese all'interno del sistema.

Ogni processo è contenuto in K insiemi di votazione .

- Cercare di ridurre al minimo il quantitativo di messaggi trasmessi.

Ogni processo appartiene al suo insieme di votazione, in maniera tale che i messaggi di request e release che il processo invia all'insieme di votazione saranno ridotti di 1 in termini di traffico di rete.

Si dimostra che la soluzione ottima che minimizza K è  $K \approx \sqrt{N}$

Essendo  $N = K(K-1) + 1$

**Esempio:**

**Esempio: N=3**

$V_1 = \{1, 2\}$
$V_3 = \{1, 3\}$
$V_2 = \{2, 3\}$

$N = 3 \rightarrow$  abbiamo 3 insiemi di votazione poiché 3 sono i processi.

- Processo 1 appartiene a  $V_1$ , proc 2 a  $V_2$  e proc 3 a  $V_3$ .
- Per garantire mutua esclusione, l'intersezione tra qualunque coppia di insiemi di votazione sarà non nulla.
- Tutti i processi hanno stessa cardinalità = 2 ( $\sqrt{3} = 1.7$  approssimo a 2). Ogni processo appartiene a  $k=2$  insiemi di votazione.

Quindi, la costruzione degli insiemi di votazione garantisce:

- **Proprietà di safety:** dal momento che l'intersezione tra 2 insiemi di votazione è non nulla, un processo che ha votato a uno dei due quorum non può votare a un altro nello stesso round di votazione per un altro insieme di votazione a cui appartiene.
- **Equa distribuzione del carico di lavoro** tra tutti i processi

### 2.15.1.2. Algoritmo di Maekawa: esempio di deadlock

$N = 3 \rightarrow 3$  processi, definisco  $V1, V2, V3$ .

Si può verificare un deadlock se tutti e 3 i processi richiedono contemporaneamente accesso alla sezione critica.

Ogni processo vota per un processo, però non può votare per nessun altro.

Dal grafico di dipendenza che rappresenta voti esposti:

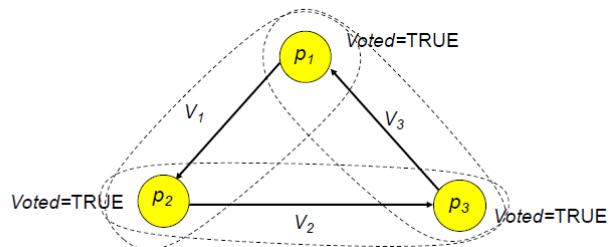
3. **p1** ha votato per p2,
  4. **p2** per p3
  5. **p3** per p1
- poiché appartengono ai loro insiemi di votazione, ma né p1 né p2 né p3 possono esprimere un altro voto.

$$P = \{p_1, p_2, p_3\}$$

$$V_1 = \{p_1, p_2\}, V_2 = \{p_2, p_3\}, V_3 = \{p_3, p_1\}$$

#### Situazione di deadlock

- $p_1, p_2, p_3$  richiedono contemporaneamente l'accesso in CS
- $p_1, p_2, p_3$  impostano ognuno `voted=TRUE` ed aspettano la risposta dell'altro



Nel grafico c'è un ciclo, che denota situazione di deadlock.

Per evitare situazione di deadlock, che si verifica in casi particolari quando tutti i processi richiedono contemporaneamente l'accesso alla sezione critica, bisogna modificare l'algoritmo di Maekawa aggiungendo maggior numero di msg scambiati dai processi.

#### Analisi algoritmi mutua esclusione distribuita

Ne abbiamo analizzati diversi, quello di token centralizzato e di autorizzazione centralizzato che hanno un numero ridotto di messaggi scambiati, ma soffrono del problema del crash del coordinatore, che se fallisce nessuno è in grado di prendere una decisione. Tramite algoritmo di elezione bisogna eleggere un nuovo coordinatore, cosa presente in Ricart & Agrawala. Ricart & Agrawala ha un costo di  $2N$  che viene migliorato da Maekawa che ha  $3$  radice di  $n$ . Maekawa è più tollerante ai guasti rispetto a Ricart e Agrawala, perchè qui se qualunque processo impatta la possibilità di entrare in sezione critica, in Maekawa invece il crash di un processo influenza solo i processi che fanno parte di quello stesso insieme di votazione.

## 2.16. Algoritmi di elezione distribuita

In caso di crash del coordinatore, negli algoritmi di elezione distribuita per i centralizzati (token centralizzato o autorizzazione centralizzato) bisogna eleggere un processo che svolga le mansioni di coordinatore o leader. Un altro esempio di algoritmo distribuito dove dobbiamo eleggere un coordinatore è quello di multicast totalmente ordinato già visto. L'elezione deve essere dinamica, ovvero in grado di essere eseguita mentre il sistema è in esecuzione. Nel caso di elezione distribuita bisogna mettersi d'accordo su chi eleggere come coordinatore.

**Proprietà di safety:** sta nel "eleggere un solo coordinatore alla volta"

**Proprietà di liveness:** "prima o poi viene eletto un nuovo coordinatore".

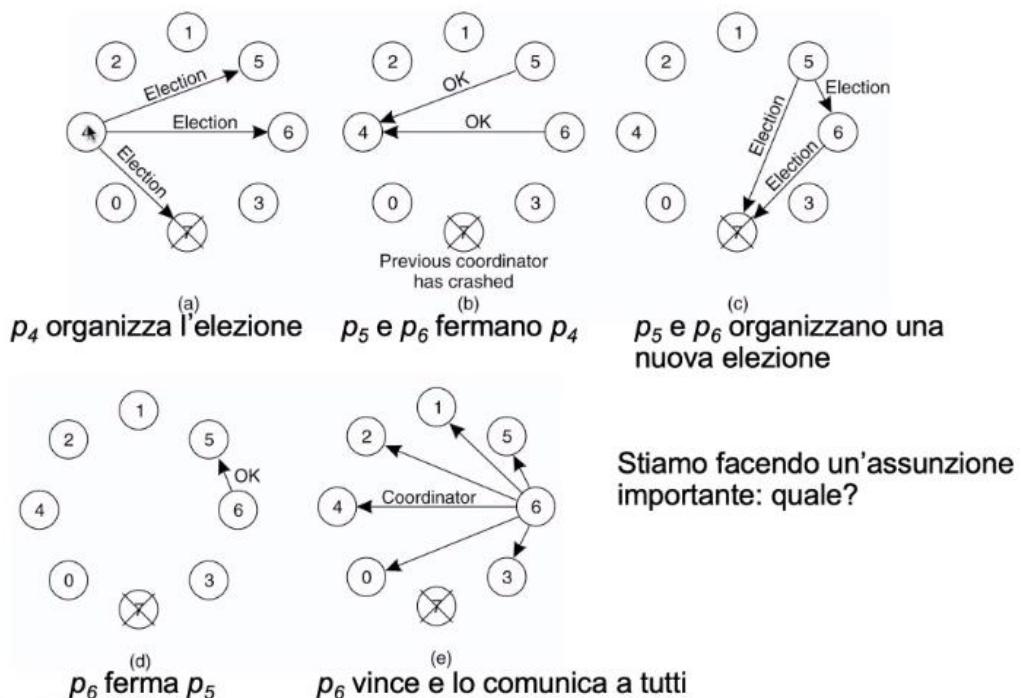
Negli esempi che vedremo:

Supponiamo di avere un insieme composto da  $n$  processi che possono essere soggetti a fallimenti di tipo crash (il processo smette immediatamente di funzionare, ma fino a quel momento aveva funzionato bene), supponiamo la comunicazione sia affidabile (no messaggi persi o duplicati), un processo non sostiene più di un'elezione alla volta, ogni processo ha un ID univoco e viene eletto il processo non guasto con l'ID più elevato.

Esistono:

### 2.16.1. Algoritmo bully

Il nodo che ha l'id più alto fa il bullo e cerca di ottenere la leadership, il ruolo di coordinatore del sistema. Come funziona, supponiamo che pi rileva che il coordinatore attuale non risponde più e quindi ha subito un crash. Pi manda un messaggio di elezione a tutti i processi con id maggiore del suo per indire una nuova elezione (il messaggio elezione non deve essere mandato a tutti i processi ma solo a quelli con id maggiore perché fa parte delle assunzioni suddette, vogliamo che venga eletto uno con l'id più elevato del suo). Se nessuno risponde allora pi si autoproclama vincitore facendo il bullo, mandando un messaggio a tutti i processi con id minore del suo dicendo che ora è lui il coordinatore. Se però nel sistema c'era pk che riceve il messaggio di pi, pk manda un messaggio a pi che ne blocca l'elezione, e pk indice l'elezione sua di pk quindi rimandando un messaggio a tutti gli id minori dicendo che sta diventando il coordinatore. Quindi pi in questo secondo caso "abbandona" la sua preparazione a diventare coordinatore. Nel momento in cui un processo ha subito un crash può tornare in esecuzione nel momento in cui non conosce quello che è l'attuale coordinatore all'interno del sistema e quindi può proporre una nuova elezione.



**Assunzione importante - il SD è sincrono e usa un meccanismo di time out per identificare un processo faulty!**

Nel caso è il processo 1 ad accorgersi del crash, tutti indicano l'elezione, quindi 1 manda n-1 messaggi, 2 manda n-2 messaggi e così via.

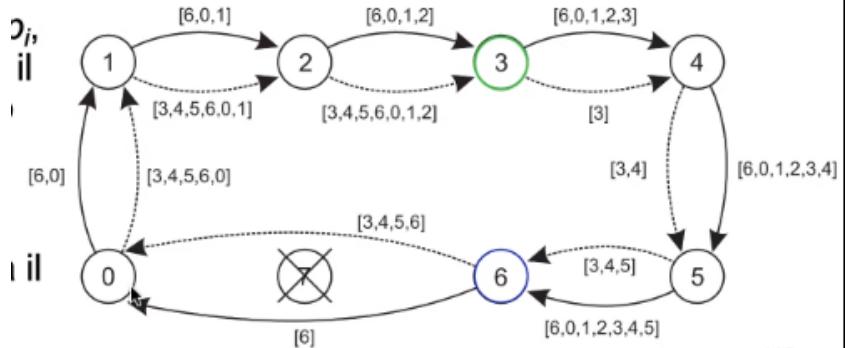
Il **costo di comunicazione** sarà un  $O(n^2)$  nel worst case.

## 2.16.2. Algoritmo di Fredrickson & Lynch

I diversi processi che partecipano all'elezione sono organizzati logicamente in un anello unidirezionale. Ciascun processo conosce solo l'id del processo che lo segue nell'anello. Se un processo si accorge che il coordinatore attuale è crashato (3 si accorge del crash di 7 per esempio). Allora 3 inizia a far circolare sull'overlay network il messaggio elezione dove all'interno inserisce il suo id. Il successivo processo che riceve il messaggio di elezione inserisce a sua volta l'id inoltrando il messaggio. 6 prova ad inoltrarlo al processo 7 ma si accorge che non risponde e lo manda al successivo quindi a 0. Per essere tollerante ai guasti in questo algoritmo è quindi necessario conoscere non solo il successore ma anche il successore del successore. Quel messaggio tornerà quindi al processo 3, che vedendo già presente nel messaggio il suo id capisce di averlo già avuto tra le mani, quindi a questo punto esamina quale è l'id più alto scritto sul messaggio e manda un messaggio per informare tutti i processi di chi è il nuovo coordinatore. Se insieme al processo 3 si accorge anche il processo 6 del crash di 7? Ci saranno due messaggi in circolo, come nell'esempio. Anche se due processi mettono in circolazione il messaggio, comunque alla fine solo 1 diventerà il nuovo coordinatore proprio perchè si eleggerà il maggiore.

La lista è incrementale man mano che si va avanti.

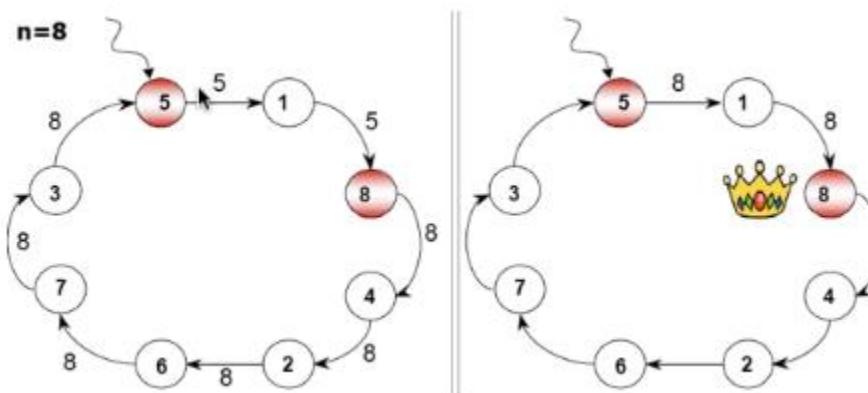
Il **costo** è di  $O(2N)$



## 2.16.3. Algoritmo di Chang & Robert

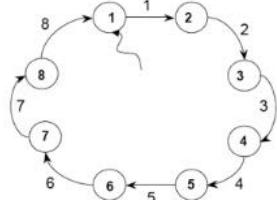
Anche questo funziona ad anello, Chang e Robert ottimizza il numero di messaggi che viene scambiato rispetto ai precedenti. Il messaggio di elezione che circola qui contiene solo l'id di un processo. Ciascun processo può trovarsi in due stati: partecipante o non partecipante all'algoritmo. L'algoritmo è composto da due passi, nel primo viene fatto circolare il messaggio di elezione e nel secondo viene comunicato chi è il coordinatore nuovo, e questi due sono in comune con gli algoritmi precedenti.

Nel primo passo supponiamo che il processo pj sia il processo che riceve il messaggio di elezione, valuta l'id nel messaggio con il proprio, se è maggiore inoltra al processo successivo a lui nell'anello il messaggio e marca il suo stato come partecipante. Se l'id nel messaggio era minore di quello di pj, se lo stato del processo era non partecipante, allora pone il contenuto del messaggio pari a j e inoltra il messaggio al suo vicino. Se j riceve il suo stesso id nel messaggio capisce che è lui il nuovo coordinatore perchè il messaggio ha percorso tutto l'anello. A quel punto il leader si marca come non partecipante e manda un messaggio dove dice che lui è il nuovo leader scrivendo il suo id nel messaggio. Quando un processo riceve un messaggio di elezione, si marca come non partecipante, registra l'id eletto e inoltra il messaggio non modificandolo. Quando un messaggio eletto raggiunge il leader, scarta il messaggio e l'elezione è over.



Il **costo di comunicazione** dipende da come sono organizzati nell'anello i processi rispetto al proprio id. Il caso migliore in termini di comunicazione si verifica quando i processi sono ordinati dal più piccolo al più grande e l'anello viene percorso in senso antiorario.

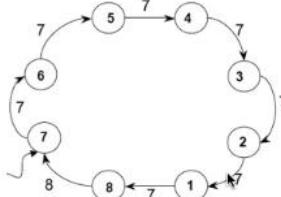
- Worst case:  $O(N^2)$
- Best case:  $O(N)$
- On average:  $O(N \log N)$



**Best case:**  $p_1$  starts the election in clockwise direction and process ids are organized in clockwise direction

$$(2^N - 1) + N \text{ messages}$$

$$\Rightarrow O(N) \text{ cost}$$



**Worst case:** the ring is in clockwise direction, process ids are organized in anti-clockwise direction and all the processes decide to hold an election at the same time

$$(N + (N-1) + \dots + 1) + N \text{ messages}$$

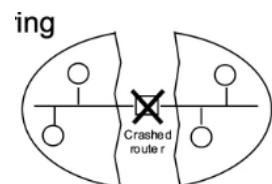
$$\Rightarrow O(N^2) \text{ cost}$$

## 2.17. Proprietà degli algoritmi di elezione:

- Assunzione comune a tutti gli algoritmi di elezione: la comunicazione è affidabile (messaggi non persi o corrotti)
- Algoritmi di elezione ad anello: Funziona sia per comunicazione sincrona che asincrona, per qualunque quantità di processi e non richiede a nessuno di loro di conoscere quanti sono all'interno dell'anello.
- Tolleranza ai guasti rispetto al fallimento dei processi: Cosa succede se un processo crasha durante l'elezione? Dipende dall'algoritmo e dal processo crashato, potrebbero servire meccanismi aggiuntivi come la riconfigurazione dell'anello.
- Tuttavia c'è una cosa importante da considerare:

Che cosa accade in presenza di una partizione di rete? Quindi nel caso in cui viene interrotta la comunicazione tra due parti del sistema, quindi i nodi sono in grado di comunicare ma solo all'interno di una delle due parti in cui il sistema è stato suddiviso. Questa partizione si può verificare anche molto spesso nelle reti a larga scala, quindi è fondamentale che gli algoritmi siano tolleranti alle partizioni di rete.

Per gli algoritmi visti finora se accade ciò, non viene più garantita la mutua esclusione, perché i nodi delle due parti eleggeranno un proprio leader ciascuna, quindi serviranno degli algoritmi di riconciliazione per far ricomunicare tutti i nodi fra loro. L'algoritmo di Paxos e quello di Raft che sono algoritmi di consenso distribuito che possono essere usati come elezione, sono tolleranti alle partizioni di rete.



## 3. Consistenza e replicazione

### I pro della replicazione: perchè i dati vengono replicati?

- Per aumentare la disponibilità del sistema distribuito quando i server falliscono o la rete viene partizionata.
  - $p$  = probability that 1 server fails
  - $p^n$  = probability that  $n$  servers fail
  - $1-p^n$  = availability of service/system with  $n$  servers
    - If  $p=5\%$  and  $n=1 \Rightarrow$  service is available 95% of time
    - If  $p=5\%$  and  $n=3 \Rightarrow$  service is available 99.9875% of time
- Per aumentare la tolleranza ai guasti del sistema distribuito
  - Under the fail-stop model (see slides on fault tolerance), if up to  $k$  of  $k+1$  servers crash, at least one is alive and can be used
  - Protect against corrupted data
- Per migliorare le performance del sistema distribuito grazie a scalabilità (dimensionale o geografica)

### Svantaggi introdotti dalla replicazione:

avere più copie dello stesso dato che possono essere distribuite su un sistema a larga scala significa che bisogna mantenere consistenti fra di loro le diverse repliche. Il sistema dovrà garantire che tutte le repliche siano aggiornate. La latenza per garantire la consistenza fra diverse parti del sistema distribuito non è trascurabile. Bisogna capire quando e come aggiornare le repliche tenendo conto del problema legato alla latenza. Parliamo infatti ora di:

### 3.1. Problemi di consistenza

Come mantenere consistenti le repliche fra di loro?

Le operazioni conflittuali fra le repliche che causano problemi di consistenza, che non vengono dalle operazioni di lettura effettuate, ma da quelle di scrittura. Possiamo avere conflitti di tipo

- **Read write:** operazione di lettura e scrittura concorrenti sullo stesso dato
- **Write write:** più operazioni di scrittura concorrenti sullo stesso dato

Garantire un ordinamento globale delle operazioni è costoso e compromette quindi la scalabilità del sistema. La soluzione consiste nell'allentare i vincoli di consistenza per evitare una sincronizzazione globale ed ottenere un sistema "consistente" ma efficiente. Si cerca un tradeoff, un compromesso fra consistenza ed efficienza. Ci sono dei **modelli di consistenza** dove vengono rilassati determinati vincoli sulla consistenza avendo via via consistenza più debole che garantiscono però più efficienza.

- **Archivio di dati distribuito:** insieme di spazi di memorizzazione fisicamente distribuiti e replicati su molteplici processi. Esempio sono i file system distribuiti, basi di dati distribuite, cloud storage
- **Modello di consistenza:** è un contratto tra l'archivio di dati distribuito e i processi che stabilisce che se i processi rispettano un certo insieme di regole, l'archivio assicura un funzionamento corretto

Nello specifico, vediamo i modelli di consistenza.

Tutti i modelli di consistenza tentano di restituire come risultato di un'operazione di lettura di un dato l'ultima operazione di scrittura sul dato; differiscono in come viene determinata l'ultima operazione di scrittura e rispetto a chi.

Differenziamo due famiglie di consistenza:

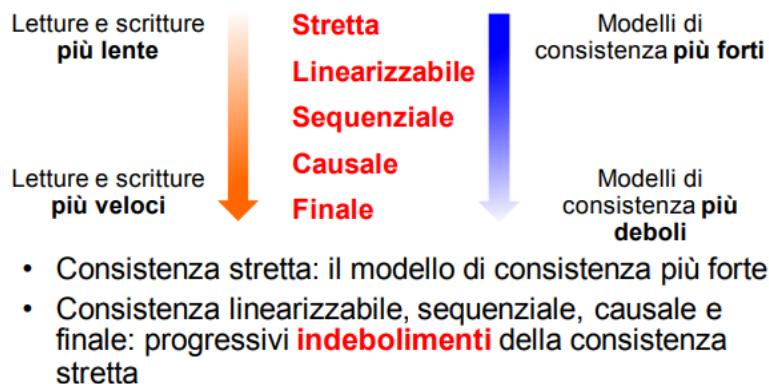
- **Modelli di consistenza data centrici:** obiettivo di fornire una vista di un archivio di dati consistente a livello di sistema
- **Modelli di consistenza client-centrati:** obiettivo di fornire una vista dell'archivio di dati consistente a livello di un singolo client. Viene garantita una politica di gestione della consistenza più veloce ma meno accurata rispetto alla data centrica

Non esiste in assoluto un modello di consistenza giusto o sbagliato, ne esistono molteplici; quale scegliere dipende dai requisiti di consistenza. Se parliamo di traffico aereo per esempio serve la consistenza più stretta possibile, che ci dà la garanzia massima di consistenza. Se invece abbiamo per esempio un file in un sistema di storage cloud che contiene informazioni non critiche si può tollerare un certo grado di inconsistenza.

I modelli descrivono come e quando le diverse repliche dell'archivio di dati vedono l'ordine delle operazioni; le repliche devono accordarsi su quale sia l'ordinamento globale delle operazioni prima di renderle permanenti.

### 3.2. Modelli di consistenza data centrici

Si tratta dei principali modelli di consistenza basati sull'ordinamento delle operazioni di **read** e **write** su dati condivisi e replicati. In generale a seconda di quale viene utilizzato si ottiene un rilassamento progressivo della garanzia sulla consistenza, a fronte di maggior efficienza. Di modelli data centrici ne esistono molte tipologie, noi analizziamo la tipologia che prevede l'ordinamento delle operazioni; in particolare scrittura e lettura di singole variabili.

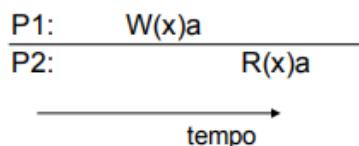


- Consistenza stretta: il modello di consistenza più forte
- Consistenza linearizzabile, sequenziale, causale e finale: progressivi **indebolimenti** della consistenza stretta

La notazione che viene utilizzata per rappresentare il comportamento dei processi che, accedendo

- $W_i(x)a$ : 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  $b$

all'archivio di dati distribuito, eseguono le operazioni di lettura e scrittura sui dati condivisi è:



### 3.2.1. Consistenza stretta: il modello ideale

La consistenza stretta è il modello più stringente di consistenza, è il modello ideale, che prevede che qualsiasi lettura su un determinato dato  $x$  ci restituisce un valore che corrisponde al risultato più recente della scrittura su  $x$ .

Consistenza stretta		Violazione della consistenza stretta	
P1:	$W(x)a$	P1:	$W(x)a$
P2:	$R(x)a$	P2:	$R(x)NIL$ $\downarrow$ $R(x)a$

È come se nel sistema ci fosse un'unica copia del dato  $x$  → *la scrittura sul dato viene vista istantaneamente da tutti processi.*

Operazione write → eseguita istantaneamente su tutte repliche come *una singola operazione atomica*.

Per realizzare consistenza stretta ci serve ordinamento temporale assoluto di tutti gli accessi all'archivio di dati. Serve quindi un *orologio globale*.

Considerando che quindi l'implementazione richiede che nel momento in cui viene effettuata una write essa venga propagata immediatamente su tutte le repliche dell'archivio distribuito rispetto a quella su cui è avvenuta la scrittura, è **necessario che tutti i processi non potranno effettuare operazioni di lettura finché non sarà finita la scrittura**, così da garantire che tutti leggano sempre la stessa cosa. L'implementazione richiede un'accurata sincronizzazione fisica fra i diversi nodi dell'archivio di dati distribuito.

### 3.2.2. Consistenza sequenziale: indebolimento di quella linearizzabile

Il risultato di una qualunque esecuzione è uguale a quello ottenuto se le operazioni (di read e write) da parte di tutti i processi sull'archivio di dati fossero eseguite

- secondo un qualche ordine sequenziale
- e le operazioni di ogni singolo processo apparissero in questa sequenza nell'ordine specificato dal suo programma

Operazione di lettura e scrittura vengono viste da tutti i processi secondo un qualche ordine di sequenza → tutti processi vedono la stessa sequenza di operazioni.

Quando i processi sono in esecuzione concorrente, qualunque alternanza (interleaving) di operazioni è accettabile (purché rispetti l'ordine di programma), ma tutti i processi vedono la stessa alternanza di operazioni.

È un indebolimento della consistenza stretta: non esiste un tempo assoluto reale rispetto al quale assegniamo un timestamp alle singole operazioni

- Non assume un tempo reale
- Ma preserva l'illusione di avere a che fare con una singola copia

### Esempi consistenza sequenziale:

- **Esempio che soddisfa la consistenza sequenziale:** sequenza di operazioni valida in un archivio di dati sequenzialmente consistente

P1:	W(x)a		
P2:	W(x)b		
P3:		R(x)b	R(x)a
P4:		R(x)b	R(x)a

Identifichiamo l'**interleaving**, ovvero la sequenza di operazioni che include tutte le operazioni presenti nel diagramma che rappresenta l'archivio di dati distribuito. L'ordine delle operazioni rispetta il vincolo che debba essere mantenuto l'ordine per ogni singolo processo. Esistono 4 interleaving ammissibili in questo caso, ed in questo caso ne è soddisfatto almeno 1.

Gli interleaving ammissibili in questo esempio, quindi che rispettano l'ordine di programma dei singoli processi sono

- $W_2(x)b R_3(x)b R_4(x)b W_1(x)a R_4(x)a R_3(x)a$
- $W_2(x)b R_4(x)b R_3(x)b W_1(x)a R_4(x)a R_3(x)a$
- $W_2(x)b R_3(x)b R_4(x)b W_1(x)a R_3(x)a R_4(x)a$
- $W_2(x)b R_4(x)b R_3(x)b W_1(x)a R_3(x)a R_4(x)a$

È qui infatti possibile **ordinare scrittura - lettura di a b o viceversa** in uno almeno di questi interleaving in questo esempio, garantendo l'ordinamento visto dai processi. Lo scopo è proprio garantire, al fine di considerare un interleaving valido, che globalmente sia eseguita prima scrittura e poi lettura di qualcosa, per tutti.

- **Esempio di operazioni non valida in un archivio di dati sequenzialmente consistente:** qui viene violata la consistenza sequenziale.

Qui non riusciamo a trovare un interleaving ammissibile delle operazioni che vediamo.

Per p3 prima deve essere letto b e poi a, quindi non va bene. Per p4 prima dobbiamo avere scrittura di xa e poi di xb, quindi neanche il secondo va bene. Deve essere una consistenza sequenziale, quindi ogni processo deve poter avere sequenzialità rispetto alle operazioni che esegue, su quelle avvenute prima.

P1:	W(x)a		
P2:	W(x)b		
P3:		R(x)b	R(x)a
P4:		R(x)a	R(x)b

- Non esistono interleaving ammissibili, ad es.

- $W_1(x)a R_4(x)a R_3(x)a W_2(x)b R_3(x)b R_4(x)b$  viola l'ordine di programma di P3
- $W_2(x)b R_3(x)b R_4(x)b W_1(x)a R_3(x)a R_4(x)a$  viola l'ordine di programma di P4

### 3.2.3. Consistenza linearizzabile

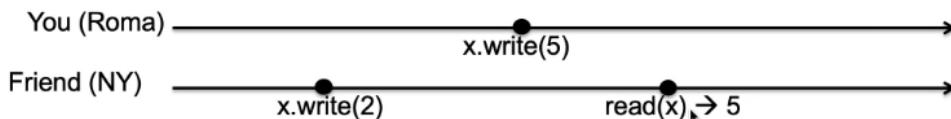
Ogni operazione di lettura e scrittura appare nel sistema come se avesse effetto istantaneamente in un momento preciso tra il momento in cui l'istruzione ha inizio e quello in cui viene completata. Abbiamo l'uso di un clock fisico per effettuare la sincronizzazione. La consistenza linearizzabile soddisfa il vincolo della sequenziale ed in più introduce un ordinamento rispetto al tempo globale. Se 1 precede 2 temporalmente allora nella sequenza dell'interleaving (operazioni viste dall'archivio di dati distribuito) l'operazione 1 deve precedere la 2. La linearizzabile dà l'illusione di avere una copia per ogni singolo client; inoltre, una operazione di read ritorna la write più recente, indipendentemente dal client, in accordo con il loro ordinamento.

La consistenza linearizzabile è più debole della stretta in quanto la linearizzabile non garantisce nessun ordine sulle operazioni sovrapposte. Si può implementare una particolare strategia di ordinamento; finchè ne esiste un singolo, l interleaving ordering per le operazioni sovrapposte, va bene. La linearizzabilità è un rilassamento di consistenza stretta.

Rispetto alla sequenziale invece, entrambe prevedono un single client single copy semantic e:

- Con la consistenza sequenziale: libertà di fare l'interleave delle operazioni provenienti da clients differenti, finchè l'ordine da ogni client sia preservato.
- Con la consistenza linearizzabile: l'interleaving tra tutti i client è molto determinato sulla base del tempo.

#### 3.2.3.1. Performance di linearizzabilità:



- Come implementare la linearizability?
  - Tutti i client mandano tutte le read/write operations al datacenter irlandese
  - Il datacenter irlandese le propaga al datacenter del North Carolina
  - Una richiesta non ritorna finchè tutta la propagazione è effettuata
  - Garantisce la correttezza? Simp (si)
  - E la performance? Nop (no)
- Linearizability tipicamente richiede la sincronizzazione completa di multiple copie prima di scrivere l'operazione di ritorno
- Ha meno senso in settings mondiali, ma ha più senso in settaggi locali

#### 3.2.3.2. Performance di consistenza sequenziale:

La consistenza sequenziale è programmer-friendly, ma difficile da implementare in maniera efficiente: le scritture dovrebbero essere applicate nello stesso ordine sulle differenti copie per dare l'illusione di una copia singola.

Per implementare la consistenza sequenziale servirebbe un sequencer globale(centralizzato) o un protocollo multicast totalmente ordinato (decentralizzato).

Esistono poi delle consistenze ancora più rilassate rispetto alle precedenti, che sono utilizzati per avere miglior performance, minori costi e maggior disponibilità che sono consistenza causale e finale, perdendo l'illusione di single copy.

### 3.2.4. Consistenza causale

Nella causal consistency ci interessa l'ordine delle operazioni di write relazionate causalmente, mentre nella consistenza finale finchè si può dire che tutte le repliche prima o poi convergeranno alla stessa copia, va bene.

Causale è modello di consistenza data centrico dove operazioni di scrittura in causa effetto fra di loro devono essere viste dai processi nello stesso ordine: vogliamo che tutti vedano prima la causa e poi l'effetto. Le operazioni di scrittura contemporanee possono essere viste in ordine diverso da processi diversi.

In una relazione di causa effetto:

- Read seguita da write sullo stesso processo: la write è potenzialmente causalmente correlata con la read
- Write seguita da read dello stesso valore su processi diversi: la read è potenzialmente causalmente correlata con la write.
- Per esempio, se P1 scrive x e P2 legge x e usa il valore letto per scrivere y, la lettura di x e la scrittura di y sono causalmente correlate

Se due processi scrivono simultaneamente due variabili, le due write non sono causalmente correlate (write concorrenti)

La consistenza causale quindi indebolisce quella sequenziale in quanto andiamo a distinguere le operazioni di causa effetto da quelle che non lo sono, garantendo stesso ordine visto da tutti solo per le prime.

**Esempi di consistenza causale:**

**1. Esempio di sequenza valida in un archivio di dati causalmente consistente, ma non in un archivio sequenzialmente consistente**

- W\_2(x)b e W\_1(x)c sono write concorrenti: possono essere viste dai processi in ordine differente
- W\_1(x)a e W\_2(x)b sono write in relazione di causa/effetto

P1:	W(x)a		W(x)c
P2:		R(x)a	W(x)b
P3:	R(x)a		R(x)c R(x)b
P4:	R(x)a		R(x)b R(x)c

**2. Esempio di sequenza di operazioni non valida in un archivio di dati causalmente consistente**

- W\_1(x)a e W\_2(x)b sono in relazione di causa/effetto: devono essere viste da tutti i processi nello stesso ordine

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

**3. Esempio di sequenza di operazioni valida in un archivio di dati causalmente consistente**

- Ma non valida in un archivio sequenzialmente consistente

P1:	W(x)a		
P2:		W(x)b	
P3:			R(x)b R(x)a
P4:			R(x)a R(x)b

Con la consistenza causale non abbiamo più l'illusione di avere una singola copy:

- Le scritture concorrenti possono essere applicate in ordini differenti sulle copie
- Le scritture causalmente relazionate devono essere applicate nello stesso ordine per ogni copia

Per via dei requisiti rilassati, la latenza è più trattabile della consistenza sequenziale. Serve un meccanismo per tenere traccia dei write causalmente relazionato. Tra i meccanismi che ci permettono di tenere traccia di relazione di causa effetto abbiamo i clock vettoriali; utilizzando il concetto di clock vettoriali si può implementare la consistenza causale; altrimenti si può tracciare grafico che tenga traccia di quali operazioni dipendano da altre e quali.

Consistenza	Descrizione
Stretta	Tutti i processi vedono gli accessi condivisi nello <b>stesso ordine assoluto di tempo</b>
Linearizzabile	Tutti i processi vedono gli accessi condivisi <b>nello stesso ordine</b> : gli accessi sono ordinati in base ad un timestamp globale (non unico)
Sequenziale	Tutti i processi vedono gli accessi condivisi <b>nello stesso ordine</b> ; gli accessi non sono ordinati temporalmente
Causale	Tutti i processi vedono gli accessi condivisi <b>correlati causalmente nello stesso ordine</b>

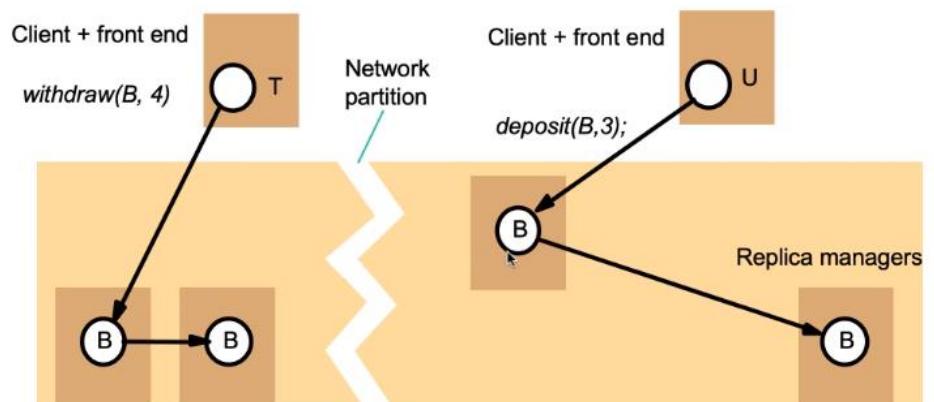
### 3.2.5. Teorema CAP

Possiamo rilassare ulteriormente questi modelli di consistenza? Sì, nella consistenza finale, popolarizzata dal **teorema CAP** (consistency availability partition tolerant ovvero tollerante alle partizioni di rete). Il teorema CAP nasce dall'osservazione di cosa accade a consistenza e disponibilità quando avviene partizione di rete in un sistema distribuito che suddivide in due sottoinsiemi i nodi del sistema. I nodi continuano ad operare ma i due sottoinsiemi non sono più in grado di comunicare tra loro per la partizione.

Supponiamo di avere un database distribuito che viene utilizzato da un'applicazione bancaria e supponiamo che i diversi server replicati siano indicati dalla lettera B, che rappresenta il conto corrente su cui due utenti stanno operando. Consideriamo la seguente partizione di rete

Effettuando deposito e ritiro soldi da entrambe le parti è facile avere inconsistenza, bisogna quindi avere un meccanismo di aggiornamento delle repliche.

In presenza di una partizione di rete, per mantenere le repliche consistenti, è necessario bloccare il sistema in attesa dell'aggiornamento delle repliche; dal punto di vista esterno, infatti, il sistema risulterà non disponibile. Se il sistema non viene bloccato e vengono servite le richieste da due partizioni, allora le repliche di sicuro divergeranno: in questo caso il sistema sarà disponibile ma non consistente. Quale scegliere? Il teorema CAP affronta questo dilemma.



### Enunciato del teorema CAP:

Ogni sistema di nodi interconnessi da una rete che condividono dati può avere al massimo 2 su 3 proprietà fra consistenza, disponibilità e tolleranza alle partizioni.

- **Consistency (C):** have a single up-to-date copy of data  
*"All the clients see the same view, even in presence of updates."*
- **Availability (A) of that data (for updates)**  
*"All clients can find some replica of data, even in presence of failure."*
- **Tolerance to network partitions (P)**  
*"The system property holds even if the system is partitioned."*

Di solito si fissa la P come proprietà imprescindibile per poi scegliere fra A e C: quest'ultima è una scelta di design progettuale. Si fissa la P in quanto nei sistemi distribuiti le partizioni avvengono spesso per router outages, tagli dei cavi sottomarini, DNS fuori uso, anche all'interno di un datacenter nonostante sia meno frequente. Avere una partizione fa perdere i messaggi da un nodo all'altro.

In generale se si tratta di un sistema che predilige la availability allora si sceglie di rilassare la consistenza, scegliendo quindi il modello di consistenza finale.

Per uno sviluppatore, lavorare con un CP significa che il sistema potrebbe non essere disponibile in alcuni momenti perché sta facendo operazioni di scrittura per mantenere consistenza. Così lo sviluppatore deve decidere cosa fare con il dato che stava scrivendo qualora non fossero disponibili le scritture. Se sta lavorando con un sistema AP sa che il sistema accoglierà sempre le sue operazioni di scrittura, ma deve tener conto che quando fa operazioni di lettura potrebbe prelevare un dato non consistente.

### Esempio di CAP:

Abbiamo due repliche che si separano, una a Londra di Ann e una a Mumbai di Pathin. Ann e Pathin stanno cercando di prenotare una stanza di hotel a New York, ed avviene la partizione di rete. C'è una sola stanza disponibile, cosa accade?

Se abbiamo un sistema CA, in presenza della partizione di rete nessuno dei due potrà prenotare la camera perchè non c'è tolleranza alle partizioni.

Se abbiamo CP Pathin che ha il server master dalla sua parte potrà prenotare la camera. Ann non potrà invece.

Se abbiamo AP entrambi potranno prenotare la stanza, causando una situazione di overbooking.

Il teorema CAP non parla esplicitamente di latenza, nonostante sia cruciale per comprendere la sua essenza. La latenza non è pari a 0 ed ogni forma di consistenza richiede la comunicazione tra i processi. Più forte è la consistenza, più forte sarà la latenza introdotta per garantirla.

### 3.2.6. Consistenza finale

Modello di consistenza in cui abbiamo mancanza di operazioni di aggiornamento simultanee. In generale si può ritrovare all'interno di un archivio di dati distribuito caratterizzato da mancanza di aggiornamenti simultanei (conflitti write-write) o comunque loro facile soluzione in caso di conflitto o da forte prevalenza di letture rispetto alle scritture (mostly read). In sistemi come questo si adotta spesso un modello di consistenza rilassato, detto consistenza finale (eventual consistency)

- **Cosa garantisce:** se non si verificano aggiornamenti, tutte le repliche (distribuite geograficamente) diventano gradualmente consistenti entro una finestra temporale (detta **inconsistency window**). In assenza di fallimenti, l'ampiezza dell'inconsistency window dipende da: latenza di comunicazione, numero di repliche, carico del sistema

Un esempio è costituito dal Domain main system, un sistema di storage distribuito dove può succedere che alcuni client vedano una risoluzione hostname - indirizzo IP che non è la stessa offerta dal name server autoritativo, ma di un server intermedio che ha nella propria cache la risoluzione disponibile e non scaduta.

La consistenza finale è adottata nel caso in cui il sistema sia AP, dove le due repliche sono sempre disponibili per la lettura ma possono portare ad una possibile inconsistenza. Nel caso di consistenza forte (tipo la linearizzabilità) c'è consistenza, ma le repliche non sono accessibili fino alla fine del loro aggiornamento.

- **Vantaggi** della consistenza finale sono che è semplice, poco costoso, garantisce letture e scritture veloci sulla replica locale.
- **Svantaggi** sono che con questo modello si perde l'illusione di avere una singola copia del sistema (come in quello causale), rendendo possibile l'inconsistenza per scritture conflittuali.

I sistemi di storage usano infatti **sistemi di riconciliazioni** per risolvere i conflitti derivanti da scritture conflittuali.

Tecniche di riconciliazione di versioni conflittuali dello stesso dato:

- **Strategia last write wins**, ovvero vince l'ultima scrittura. Vedendo un timestamp fisico basato su un clock fisico sarebbe difficile tenere traccia dell'ordinamento, quindi si usa un clock vettoriale per determinare quale sia la versione più recente (confrontando i due clock vettoriali e vedendo quale dei due è maggiore dell'altro). Se risultano concorrenti bisognerà trovare un'altra soluzione.
- **Strategia a livello applicativo** dove un'applicazione si occupa di risolvere il conflitto, come nel caso di Amazon Dynamo (da non confondere con Dynamo DB che è un servizio di storage).

Il modello della consistenza finale è largamente utilizzato nei SD a larga scala, soprattutto per i servizi cloud di storage, come aws s3, dropbox, ma anche nei datastore noSQL come Cassandra dove si offre all'utente la scelta di consistenza.

Il sistema distribuito sarà più veloce ma il costo di garantire un maggior livello di consistenza ricade sullo sviluppatore dell'applicazione (deve valutare se un'eventuale inconsistenza sia accettabile o meno).

### 3.3. ACID vs BASE

ACID e BASE rappresentano due politiche di design ad estremi opposti nello spettro di consistenza/disponibilità.

**ACID (Atomicity, Consistency, Isolation, Durability)**: è l'approccio tradizionale per indirizzare il problema della consistenza in un sistema di gestione di database relazionale. È un approccio pessimistico poiché previene l'occorrenza dei conflitti dando per scontato che accadranno. ACID non scala bene quando tratta grandi quantità di bytes.

**BASE (Basically Available, Soft state, Eventual consistency)**: approccio ottimistico in quanto lascia i conflitti avvenire, ma li identifica ed esegue azioni per riordinarli. È **basically available** poiché il sistema risulta disponibile la maggior parte del tempo ma potranno esistere sottosistemi temporalmente non disponibili. **Soft state** poiché la persistenza dei dati è lasciata nelle mani dell'utente che deve preoccuparsi di aggiornarli. **Eventually consistent** poiché alla fine il sistema converge sempre ad uno stato consistente. L'approccio base è spesso usato nei datastore noSQL

## 3.4. Modelli di consistenza client-centrici

Obiettivo di fornire consistenza di accesso ad un archivio distribuito a livello del singolo client. Dai modelli client centrici non vengono offerte garanzie di consistenza da parte degli accessi degli altri client. Abbiamo 4 modelli consistenza client centrici:

- Consistenza **monotonic-read** (o **read-after-read**)
- Consistenza **monotonic-write** (o **write-after-write**)
- Consistenza **ready-your-write** (o **read-after-write**)
- Consistenza **writes-follow-reads** (o **write-after-read**)

Esempio: archivio di dati distribuiti acceduto da un'utente con un dispositivo mobile. Spostandosi accederà a due repliche del sistema differenti. Mentre si sposta potrebbe essere accaduto che le scritture effettuate su A potrebbero non essere ancora propagate su B quando vi accede, dando inconsistenza. Quali sono le garanzie offerte all'utente mobile dipende proprio da quale modello si adotta.

Nei modelli di consistenza client-centrici usiamo la seguente notazione:

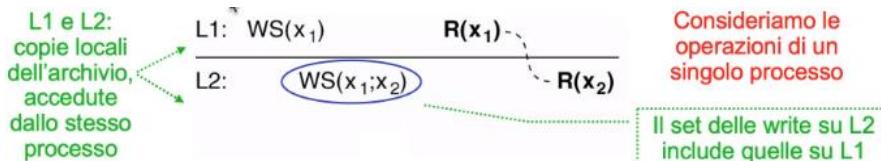
- $X_i[t]$  : versione di  $x$  sulla replica locale  $L_i$  al tempo  $t$
- $WS(x_i[t])$ : sequenza di operazioni di scrittura (Write Set, WS) su  $L_i$  che hanno portato come risultato a  $X_i[t]$
- $WS(x_i[t_1]; x_j[t_2])$ : le operazioni in  $WS(x_j[t_2])$  sono state eseguite anche sulla replica  $L_j$  in un tempo successivo  $t_2$  (Questo significa che  $WS(x_i[t_1])$  è parte di  $WS(x_j[t_2])$ )
- Per brevità non indichiamo il tempo  $t$

Assumiamo che i dati possano essere modificati solo dal processo proprietario (assenza di conflitti write-write)

### 3.4.1. Read after read (monotonic read)

Se un processo legge il valore di un dato  $x$ , qualunque successiva operazione di lettura su  $x$  da parte di quel processo restituirà sempre quello stesso valore o un valore più recente

- In un archivio di dati che fornisce la consistenza monotonic-read



- In un archivio di dati che non fornisce la consistenza monotonic-read

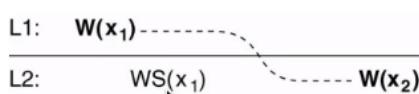


Esempio di monotonic read: se stiamo leggendo il calendario accedendo a diverse repliche del servizio.

### 3.4.2. Write after write (monotonic write)

Un'operazione di scrittura da parte di un processo su un dato  $x$  viene completata prima di qualunque operazione di scrittura successiva su  $x$  da parte dello stesso processo. L'archivio garantisce la serializzazione delle scritture per il singolo client

- Archivio dati che fornisce la consistenza monotonic-write



- Archivio di dati che non fornisce la consistenza monotonic-write

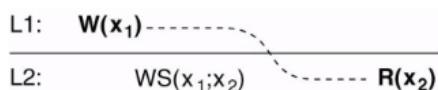


Esempio di monotonic write: aggiorniamo un programma su un certo server, tutti i componenti della nostra applicazione dipendono dal linking che abbiamo fatto, monotonic write garantisce di mantenere versioni di file replicati nell'ordine corretto su ogni server.

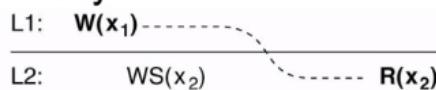
### 3.4.3. Read after write (read your writes)

Analogo della consistenza causale per i data centrici ma qui per il singolo client. L'effetto di un'operazione di scrittura da parte di un processo su un dato  $x$  sarà sempre visto da una successiva operazione di lettura di  $x$  da parte dello stesso processo.

- Archivio di dati che **fornisce** la consistenza



- Archivio di dati che **non fornisce** la consistenza read-your-writes

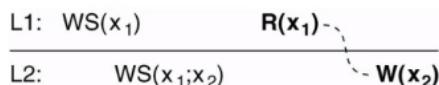


Esempio di read your writes: quando aggiorniamo al pag web, se il browser adottasse questa mostrerebbe la versione più recente di una pagina richiesta anzichè quella che ha nella cache.

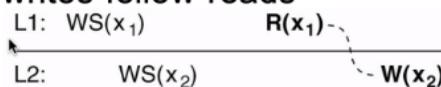
### 3.4.4. Consistenza writes-follow-reads

Un'operazione di scrittura da parte di un processo su un dato  $x$  che segue una precedente operazione di lettura di  $x$  da parte dello stesso processo ha luogo sullo stesso valore di  $x$  che è stato letto o su un valore più recente

1. Archivio di dati che **fornisce** la consistenza writes-follow-reads



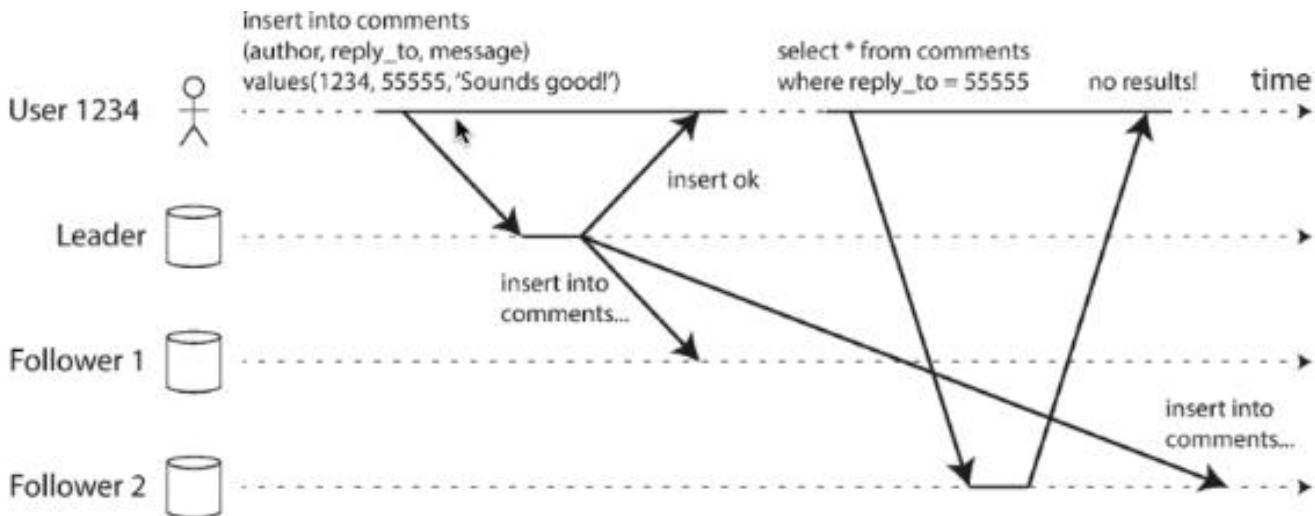
- Archivio di dati che **non fornisce** la consistenza writes-follow-reads



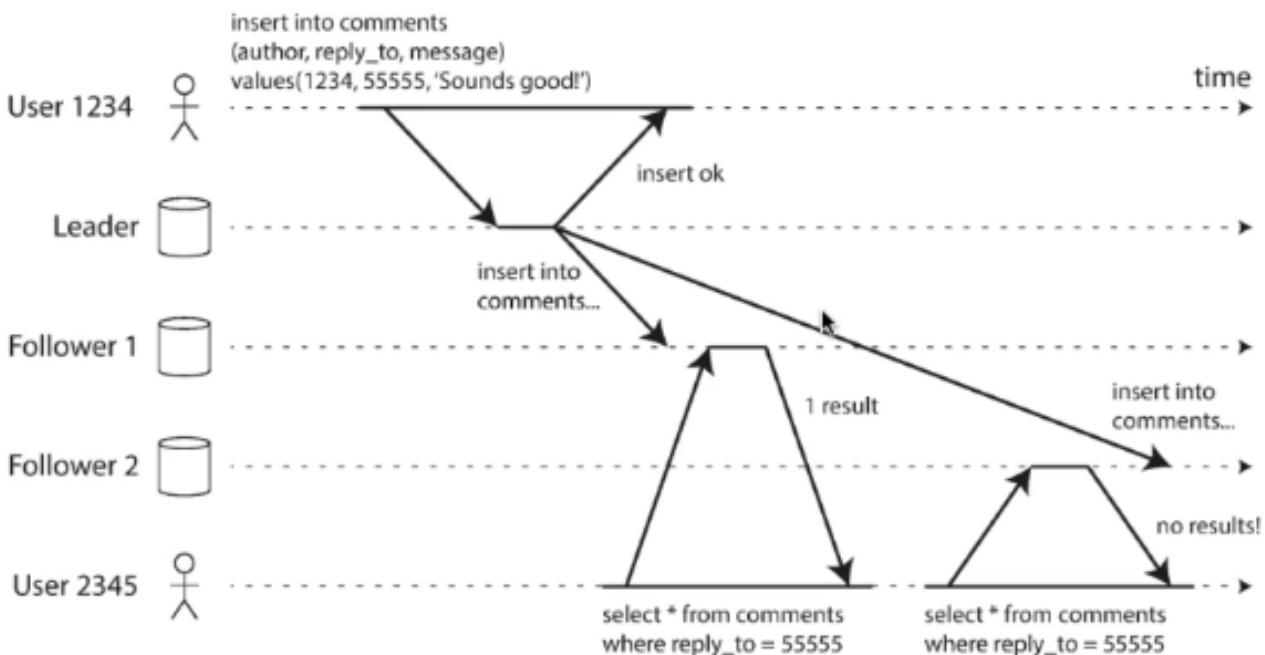
Esempio writes-follow-reads: se un utente legge un articolo A in un newsgroup e scrive un commento B relativo ad A, writes-follow-reads garantisce la scrittura di B su ogni copia del newsgroup solo dopo la scrittura di A (una lettura "tira" la corrispondente operazione).

### 3.4.5. Esempi di consistenza client-centrica

- Un utente esegue una write, seguita da una read di una vecchia replica. In questo esempio viene violata read after write in quanto effettua una scrittura quando una lettura non è ancora stata propagata. Quando il follower 2 legge non vede ancora il 555 scritto all'inizio.



- Un utente prima legge da una replica fresca, poi da una vecchia replica. Qui è violata la monotonicità rispetto alle operazioni di lettura.



### 3.5. Protocolli di consistenza

Consideriamo due diverse famiglie di protocolli per il mantenimento della consistenza:

- Protocolli per consistenza linearizzabile e sequenziale:
  - **Primary based:** le operazioni di scrittura vengono effettuate solo su una replica, chiamata replica finale o leader o master. Questa replica primaria si assicura che le scritture vengano propagate in modo ordinato su tutte le altre repliche.
  - **Replicated write:** le operazioni di scrittura vengono eseguite su molteplici repliche
- Protocolli per consistenza client-centrica.

### 3.5.1. Protocolli primary based

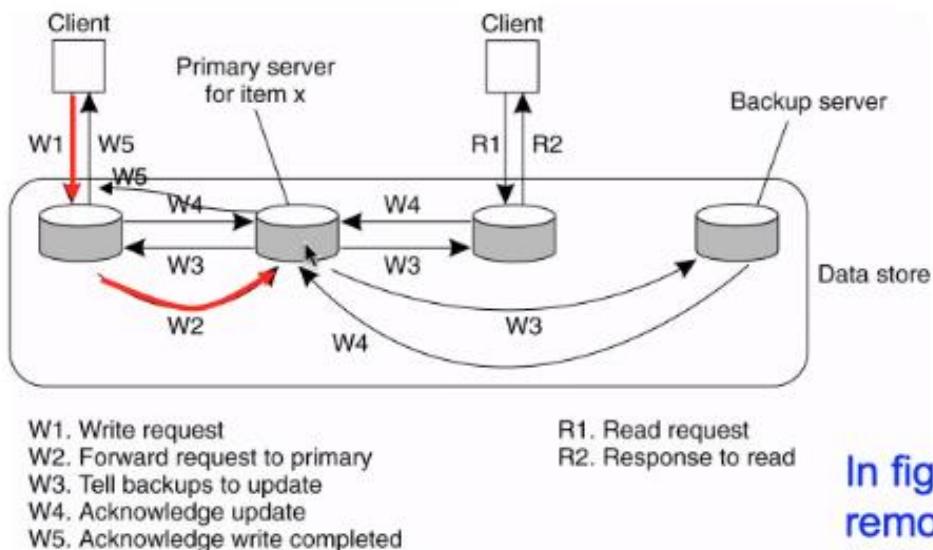
Anche chiamati protocolli di replicazione passiva, l'idea è che ad ogni dato associamo una replica primaria che è appunto il leader, il quale ha il ruolo di coordinare le operazioni di scrittura su tutte le altre repliche secondarie. L'operazione di lettura del dato può essere eseguita su ciascuna replica. Questi protocolli si dividono in:

- **Remote write:** l'operazione di scrittura del dato viene inviata alla replica primaria (che può essere remota rispetto al client), che poi inoltra alle repliche secondarie coordinandone l'aggiornamento.
- **Local write:** la copia primaria del dato migra dalla replica primaria a quella locale per cui il client sta accedendo per fare l'operazione di scrittura e la replica locale inoltra alle altre repliche l'operazione di scrittura.

#### 3.5.1.1. Primary-based: protocolli remote-write

I protocolli tipicamente usati nei DB distribuiti in alcuni data store NoSQL, in MQS e file system distribuiti, per i quali si richiede un elevato grado di tolleranza ai guasti. Gli **svantaggi** sono: lentezza in caso di repliche distribuite geograficamente e scarsa scalabilità all'aumentare del numero di repliche

**Si nota la lentezza nell'eseguire le operazioni dove la velocità dipende dalla replica più lenta, ovvero il critical path.**



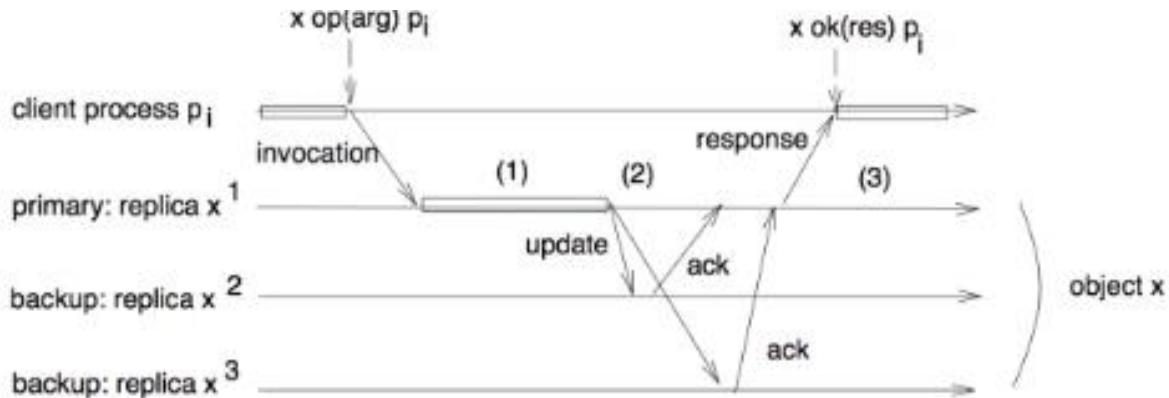
Remote write è bloccante, in quanto il client rimane bloccato finché non riceve il messaggio di risposta. In alternativa si può pensare ad un protocollo non bloccante in cui la replicazione sulle altre repliche è fatta in modo asincrono, ovvero la replica primaria risponde al client che la scrittura è stata completata solo sul primario e successivamente in modo asincrono la replica primaria si occupa di fare l'aggiornamento su tutte quante le altre repliche. Non è garantito che l'ordinamento delle scritture delle repliche sia uguale a quello temporale.

Nel caso in cui viene usato remote write in versione bloccante il modello di consistenza è la linearizzabilità.

L'aggiornamento delle repliche secondarie da parte della replica primaria è offerto dal meccanismo del **log shipping**. L'aggiornamento della replica primaria è effettuato in modo **bloccante** o **non bloccante** per il client.

### Il bloccante (o replicazione sincrona):

- La replica primaria notifica al client che la scrittura è stata completata su tutte le repliche
- Modello di consistenza: linearizzabilità
- **Vantaggi:** maggiore tolleranza a guasti (repliche sincronizzate), incluso crash della replica primaria
- **Svantaggi:** lentezza (il client attende l'aggiornamento di tutte le repliche)

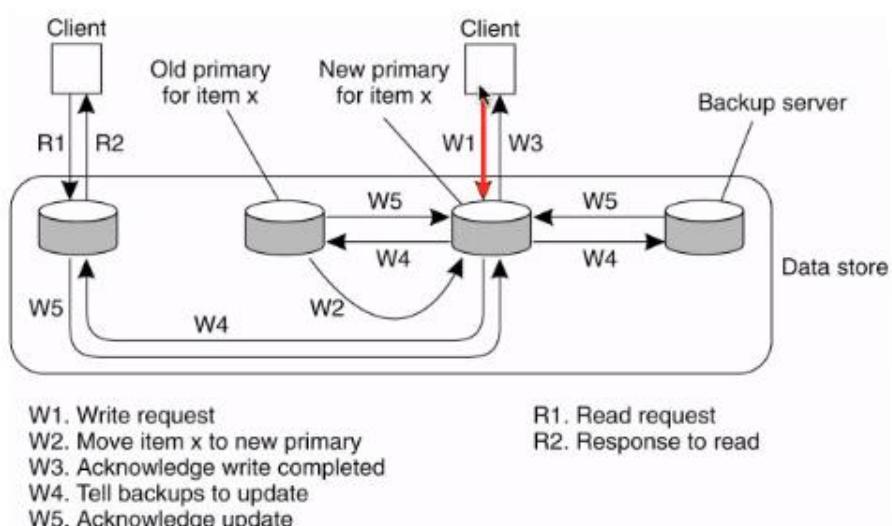


### Il non bloccante (o replicazione asincrona):

- La replica primaria notifica il client che la scrittura è stata completata solo su di essa
- Modello di consistenza: sequenziale
- Vantaggi: minore attesa da parte del client, più adatto per repliche in numero elevato e distribuite geograficamente
- Svantaggi: minore tolleranza ai guasti e perdita della linearizzabilità

#### 3.5.1.2. Primary-based: protocolli local-write

La replica primaria migra rispetto alla posizione del client; se un utente richiede la scrittura di un dato  $x$ , viene fatta primaria la replica con la quale ha parlato il client, in maniera non bloccante viene fatto il write in locale su quella replica per poi essere propagato sulle altre



W1. Write request  
 W2. Move item  $x$  to new primary  
 W3. Acknowledge write completed  
 W4. Tell backups to update  
 W5. Acknowledge update

R1. Read request  
 R2. Response to read

### 3.5.2. Protocolli replicated write

La scrittura non avviene più in modo centralizzato e poi propagato sulle rimanenti, ma la scrittura può essere eseguita su molteplici repliche. Abbiamo 2 approcci

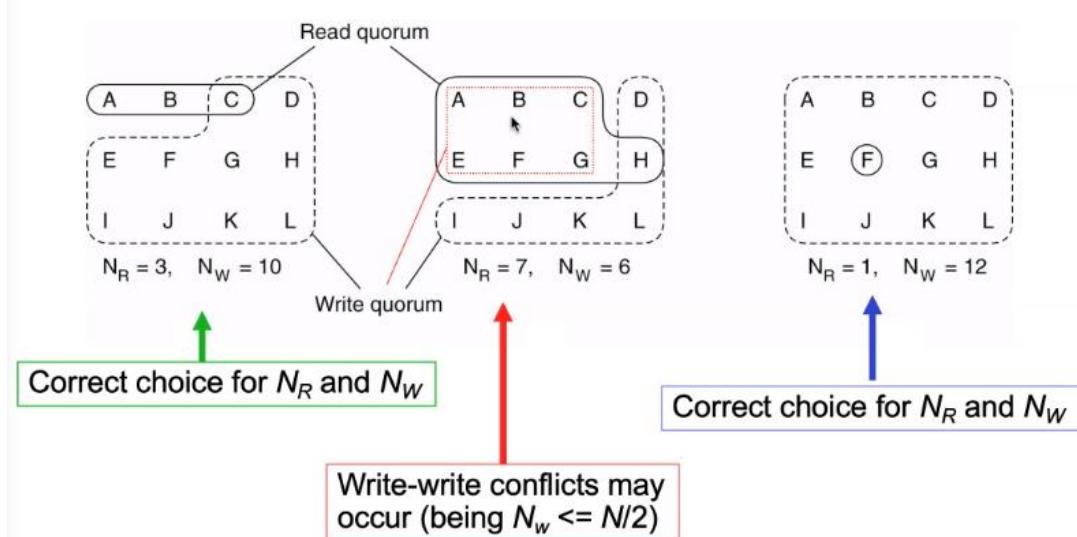
- **Replicazione attiva:** nel momento in cui una replica riceve un'operazione di scrittura la manda in multicast a tutte le repliche, così che tutte vadano ad eseguire le stesse operazioni. Questo sistema si usa per garantire che le scritture vengano fatte nello stesso ordine su ogni replica: per garantirlo si usa il multicast totalmente ordinato implementato in modo decentralizzato usando un clock scalare, oppure tramite un sequencer centralizzato. La soluzione dal multicast soffre di problemi di scalabilità in sistemi a larga scala in caso decentralizzato, problemi di scalabilità e single point of failure in caso centralizzato. Consistenza sequenziale: stesso ordine su tutte le repliche, ma può essere diverso da quello temporale
- **Protocolli basati su quorum:** per effettuare un'operazione di lettura o scrittura viene richiesto un quorum di voti. Nel momento in cui un processo vuole effettuare un'operazione di lettura richiede un quorum per la lettura così da garantirsi la lettura dell'ultimo dato scritto. Nel quorum di scrittura infatti è prevista l'inclusione di un numero di versione, così da capire quale versione verrà letta.

Per evitare conflitti lettura scrittura si applicano delle condizioni, ovvero la somma del quorum di lettura e del quorum di scrittura deve essere maggiore di  $N$  (totale repliche); per evitare conflitti scrittura scrittura il quorum dei write deve essere maggiore della metà delle repliche.

$$\begin{aligned} a) \quad & N_R + N_W > N \\ b) \quad & N_W > N/2 \end{aligned}$$

Se le condizioni non sono soddisfatte il modello si degrada fino alla consistenza finale, se sono soddisfatte c'è consistenza sequenziale.

In generale vediamo con un esempio a cosa servono queste condizioni:



La lettura nel terzo caso risulta particolarmente veloce in quanto se ne deve occupare una sola replica, le operazioni di scrittura sono molto lente invece perché bisogna ottenere il quorum da tutte le repliche. Questa prende il nome di **ROWA** (read one write all).

Altri esempi sono:

1. **RAWO** quando  $Nw = 1$  e  $Nr = N$ , la quale permette scritture molto veloci ma letture lente; inoltre è possibile avere delle scritture conflittuali in quanto non è rispettata la seconda condizione.
2. Altro esempio è la conformazione **MAJORITY** quando  $Nw = Nr = N/2+1$  che rispetta entrambe le condizioni e garantisce elevata disponibilità.

I protocolli basati su quorum sono molto utilizzati in quanto configurando  $Nw$  e  $Nr$  è possibile lasciare allo sviluppatore la scelta sul grado di consistenza che gli interessa, spaziando da una consistenza forte ad una debole come modello.

### 3.5.3. Protocolli per la consistenza client-centrica

Le garanzie vengono date al singolo client più che essere garanzie a livello di sistema.

L'implementazione della consistenza client-centrica richiede di definire il set delle operazioni rilevanti per ciascun processo, e l'insieme delle operazioni di scrittura eseguite da quel processo. Facendo passare il write set tra repliche locali verso la quale il client si sposta o perchè è un utente mobile o perchè durante l'utilizzo dell'applicazione viene collegato a repliche differenti, portandosi dietro il write set di scrittura o lettura si può implementare un diverso tipo di consistenza.

Per la consistenza monotonic-read; le operazioni che devono essere effettuate sono fatte o sull'ultimo dato letto o sul più recente. Quando il processo si sposta da una replica all'altra quindi passa il write set delle proprie operazioni di scrittura. Se alla replica manca qualche scrittura, contatta le altre repliche per essere aggiornata sulle operazioni mancanti prima di eseguire l'operazione richiesta dal processo c. Dopo ciò, il write set di scrittura sarà aggiornato. Date le grandi dimensioni eventuali dei write set e read set, l'implementazione più efficiente prevede l'utilizzo di timestamp vettoriali.

**Esempio:** a partire da dicembre 2020 Amazon S3 offre consistenza forte read after write per le operazioni di PUT e DELETE per i bucket degli utenti, sia per operazioni di scrittura su nuovi oggetti e sovrascrizioni di oggetti esistenti sia per operazioni di delete. In realtà questo non risolve i problemi di inconsistenza in quanto S3 non supporta il locking degli oggetti per scritture concorrenti, quindi fra 2 put vince quella con timestamp più recente. Questi problemi derivano dal fatto che s3 è client centrico.

Alcuni noSQL offrono modelli di consistenza a scelta del programmatore. Ad esempio DynamoDB di Amazon prevede consistenza finale o consistenza forte per le letture. Cassandra provvede una consistenza quorum-based, dove i quorum possono essere scelti dal programmatore su misura.

## 3.6. Replicazione

Andiamo a replicare delle risorse, in particolare ad esempio i dati. Bisogna scegliere dove posizionare le repliche, come propagare i contenuti quando c'è un aggiornamento e cosa propagare.

Per il posizionamento delle repliche è un problema largamente studiato nei sistemi distribuiti non solo quando abbiamo repliche di dati ma anche quando abbiamo repliche di risorse computazionali. Ad esempio Kubernetes ha al suo interno delle repliche sui quali verranno pubblicati i pod. È importante sapere che il problema del posizionamento è NP-hard, quindi è un problema che richiede

un tempo computazionale elevato in caso in cui sia di grandi dimensioni. Non avendo a disposizione soluzioni concrete quindi si ricorre ad euristiche per individuare in modo sub ottimo ma velocemente una soluzione così da riconfigurare dinamicamente la posizione delle repliche.

**Sistema di coordinate di rete:** idea di andare a modellare internet come se fosse uno spazio geometrico caratterizzando la posizione di un nodo rispetto alla posizione che occupa nello spazio geometrico. In questo modo si possono stimare le distanze nel sistema per capire le latenze di cui è caratterizzato. A tal proposito esiste **l'algoritmo di Vivaldi**, un algoritmo decentralizzato basato su gossiping, il quale permette facilmente di stimare la latenza fra i nodi.

## 4. Tolleranza ai guasti

Andremo a studiare il consenso distribuito per affrontare i guasti all'interno di un sistema. Analizziamo subito un concetto fondamentale;

**Dependability:** abilità di un sistema di fornire un servizio che può essere considerato fidato in maniera giustificata. È importante in quanto ciascun componente può dipendere da altri componenti in un sistema distribuito.

Proprietà di un sistema dependable:

- **Disponibilità** - assicura che il sistema sia pronto per essere usato in un istante di tempo t. Si può anche definire come mean time to failure fratto la somma fra mean time to failure e mean time to repair. Il denominatore sarebbe il mean time between failures. Si misura con la scala dei 9 per definire quanto in percentuale sia disponibile (99%, 99.99%, ecc).
- **Affidabilità** - assicura che il sistema sia funzionante senza guasti in maniera continuativa, ovvero per quanto tempo sarà up and running. È definita anche come la probabilità condizionale che il sistema sia funzionante nell'intervallo [0,t) sapendo che era funzionante al tempo 0. L'affidabilità è diversa dalla disponibilità, in quanto se il sistema non funziona un millisecondo di ogni ora, abbiamo una ottima disponibilità per il bassissimo tempo di downtime, ma una scarsa affidabilità in quanto il mean time between failure è di un'ora, relativamente basso.
- **Safety** – se il sistema smette di operare correttamente non succede nulla di catastrofico per l'utente ed il sistema.
- **Manutenibilità** - misura la facilità con cui il sistema può essere riparato dopo un guasto.
- **Integrità** – assenza di alterazioni improprie del sistema.

### 4.1. Failure, error, fault: differenze

- **Failure (fallimento):** si verifica quando il comportamento di un componente del sistema devia rispetto alle specifiche.
- **Error (errore):** può determinare un fallimento in quanto un componente presenta una deviazione di stato rispetto ai previsti.
- **Fault (guasto):** è la causa dell'errore, sono dovuti ad una distrazione del programmatore.

Sulla base di essi, definiamo alcuni strumenti utili per la dependability:

- Prevenzione di guasti ad esempio migliorando la progettazione
- Tolleranza ai guasti (fault tolerance) ovvero il sistema continua a funzionare anche in presenza di guasti in qualche componente (i guasti vengono mascherati).
- Rimozione dei guasti
- Predizione dei guasti

### 4.1.1. Tipi di failure

I componenti del SD possono fallire in modi differenti:

- **Crash** – il componente si arresta, ma fino a quel momento aveva lavorato correttamente.
- **Omissione** – il componente non risponde ad una richiesta, per esempio perchè il server è down.
- **Fallimento nella temporizzazione** – il componente risponde ma ci mette più dell'intervallo prestabilito.
- **Fallimento nella risposta** - il componente risponde ma la risposta che fornisce non è corretta.
- **Fallimento arbitrario o bizantino** – il componente può produrre una risposta arbitraria con tempi arbitrari.

I crash sono i fallimenti più innocui, quelli bizantini i più gravi.

### 4.1.2. Modelli di failure

nei SD risulta difficile distinguere se un componente ha subito un crash oppure se è solo molto lento. Per esempio, il processo P aspetta da Q una risposta in un sistema distribuito asincrono.

Distinguiamo i seguenti modelli di failure:

- Fallimento **fail stop**: Q ha subito un crash e P può scoprire il fallimento tramite timeout o preannuncio del componente che fallisce.
- Fallimento **fail silent**: Q ha subito un crash o un'omissione ma P non può distinguerli.
- Fallimento **fail safe**: Q ha subito un fallimento arbitrario ma senza conseguenze.
- Fallimento **fail arbitrary**: Q ha subito un fallimento arbitrario non osservabile.

## 4.2. Rilevamento dei fallimenti

Per mascherare i fallimenti, bisogna innanzitutto rilevarli tramite il meccanismo di **failure detection**, il quale può essere applicato nelle seguenti modalità:

- **Attiva**: invio di un messaggio e timeout per rilevare se un processo è fallito (una delle soluzioni più usate, adatta per fallimento fail-stop).
- **Passiva**: attesa di ricevere un messaggio.
- **Proattiva**: come effetto collaterale dello scambio di informazioni tra vicini (ad es. disseminazione delle informazioni basata su gossiping).

Il timeout è facile da impostare nei sistemi sincroni a differenza di quelli asincroni.

**Implementazione pratica per failure detection:** per rilevare se un processo è appena crashato, esiste un modello generale così costituito:

- Ogni processo possiede un modulo di rilevamento del fallimento
- Un processo P invia un messaggio di probe a un altro processo Q per una reazione: se Q reagisce allora è considerato in vita per P, altrimenti se non risponde in una unità di tempo t, si sospetta che possa aver crashato.

Nell'implementazione pratica quello che si fa è: se P non riceve un "heartbeat" da Q entro un tempo t, P sospetta di Q. Se però Q risponde al messaggio dopo il tempo t, P smette di sospettare di Q e incrementa il valore del timeout t.

## 4.3. Ridondanza

tecnica principale per mascherare i guasti, la quale può essere declinata su tre diversi assi:

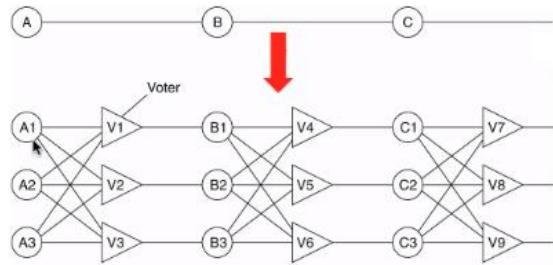
**Ridondanza delle informazioni:** Quando per esempio si usa un bit di parità o un codice di auto-rilevazione degli errori, ovvero ci sono entità che segnalano guasti

**Ridondanza nel tempo:** viene eseguita un'azione e se necessario, verrà rieseguita nel tempo. Questa ridondanza è utile se il guasto è intermittente, se è permanente questa ridondanza non mi permette di mascherare il guasto.

**Ridondanza fisica:** si aggiungono attrezzature o processi extra, applicabile sia a livello hardware che software.

### 4.3.1. Esempio di ridondanza fisica: Ridondanza tripla modulare

Ciascun componente viene replicato 3 volte e viene introdotto un meccanismo di voto a maggioranza. I 3 componenti replicati eseguono un'operazione, il cui risultato viene sottoposto ad una votazione per produrre un unico output. Se uno dei tre componenti replicati fallisce (singolo fallimento di tipo arbitrario), gli altri due possono mascherare e correggere il guasto



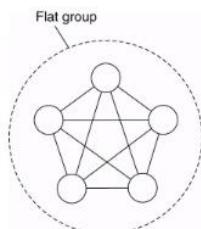
## 4.4. Resilienza dei processi

In ingegneria la resilienza è la capacità di un materiale di resistere a forze di rottura. Nei SD corrisponde alla capacità del sistema distribuito di fornire e mantenere un livello di servizio accettabile in presenza di guasti e minacce alla normale operatività. Per mascherare il guasto di un processo in un SD si può replicare e distribuire la computazione in un gruppo di processi identici. Questi gruppi possono essere organizzati in modo flat o gerarchico

- **Gruppo flat**

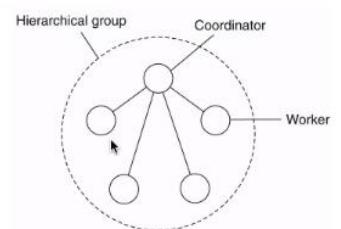
Nel secondo caso è tollerante ai guasti se viene rieletto il coordinatore al crash.

- Adatto per tollerare guasti (simmetria e assenza di single point of failure)
- Maggiore overhead perché il controllo è completamente distribuito (più difficile da implementare)



- **Gruppo gerarchico**

- Singolo coordinatore
- Tollerante ai guasti?
- Relativamente semplice da implementare



## 4.5. Modelli di replicazione: passiva vs. Attiva

Due modelli alternativi:

- Replicazione passiva
- Replicazione attiva

**Replicazione passiva:** gruppo di processi organizzati in modo gerarchico. Una sola replica primaria esegue le azioni sui dati mentre le altre repliche (passive) servono in caso di guasto. Una sola replica è corretta in quanto le altre possono anche non essere aggiornate (repliche calde o fredde). Possibile scollamento dello stato tra replica primaria e repliche secondarie, quindi esiste un **checkpoint** per tenere aggiornato lo stato delle repliche secondarie. Se la replica primaria subisce un crash, le altre repliche eseguono un algoritmo di elezione per individuare il nuovo coordinatore. Un esempio è costituito dai protocolli primary-based per la consistenza.

**Replicazione attiva:** il gruppo dei processi è organizzato in modo flat, ed esiste un coordinamento tra le repliche attive; il costo del suddetto coordinamento aumenta con la scala del sistema e la complessità delle politiche di coordinamento. Un esempio sono i protocolli replicated-write per la consistenza.

## 4.6. Gruppi e mascheramento dei guasti

Si consideri un gruppo **flat**, un gruppo composto da N processi è **k-fault tolerant** se può mascherare k guasti concorrenti (**k** è detto **grado di tolleranza ai guasti**). Quanto deve essere grande un gruppo k-fault tolerant dipenderà dal modello di failure. I principali modelli di failure sono:

- Fallimento **fail-stop o fail-silent** ( $N \geq k+1$  processi): nessun processo del gruppo produrrà un risultato errato, quindi il risultato di un solo processo non guasto è sufficiente.
- Fallimento **arbitrario**: il risultato del gruppo è definito tramite un **meccanismo di voto** ( $N >= 2k+1$ ) processi. Abbiamo bisogno di  $2k+1$  processi non guasti in modo che il risultato corretto possa essere ottenuto con un voto a maggioranza.

Assunzioni di cui tener conto:

1. **Tutti i processi sono identici**
2. **Tutti i processi eseguono i comandi nello stesso ordine**, per essere certi che tutti i processi facciano esattamente la stessa cosa

## 4.7. Consenso nei sistemi distribuiti

Assunzione: consideriamo ora che i **processi del gruppo non siano identici**, ovvero che ci sia una computazione distribuita. L'obiettivo è che i processi non guasti del gruppo devono raggiungere un consenso (**accordo**) unanime su uno stesso valore (es. il prossimo comando da eseguire) in un numero finito di passi, nonostante la presenza di processi guasti. Degli esempi di consenso sono l'elezione di un coordinatore, mutua esclusione, commit di una transazione. I tipi di guasti si suddividono in guasti non bizantini e bizantini.

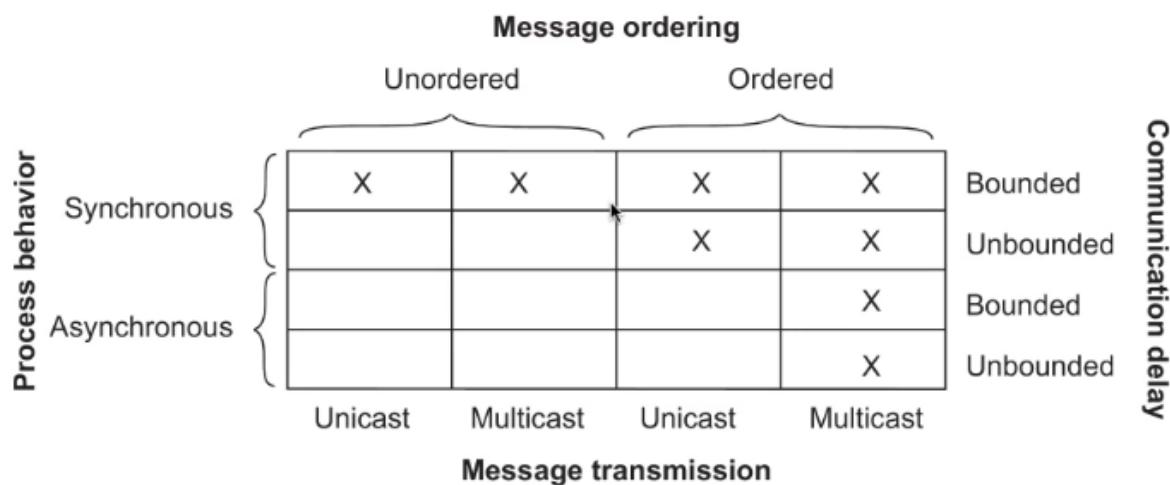
**Differenze fra agreement e consenso:** nel corso abbiamo trattato questi due termini come sinonimi, ma per la comunità teoretica dei sistemi distribuiti i due termini sono correlati a problemi molto simili ma non identici. I problemi di tipo agreement sono riferiti ai problemi in cui un singolo processo possiede il valore iniziale, invece i problemi di tipo consensus si riferiscono ai problemi in cui tutti i processi che hanno un valore iniziale.

Gli algoritmi per il consenso che andremo a prendere in considerazione sono **Paxos**, **Raft**, **generali bizantini**, dei quali i primi due sono problemi di consensus, e l'ultimo è un agreement problem.

#### 4.7.1. Condizioni necessarie al consenso:

Per determinare le condizioni necessarie a raggiungere il consenso in un sistema distribuito bisogna analizzare:

- le **diverse tipologie dei processi**: se i processi sono sincroni operano in modalità **lock step**, ovvero eseguono i stessi passi dell'algoritmo a breve distanza l'uno dall'altro, non stanno facendo la stessa cosa nello stesso istante: possono avere velocità che si discostano fra di loro ma non si disallineano fra loro (esiste un  $c$  maggiore uguale 1 tale che se un processo ha eseguito  $c+1$  passi, ogni altro processo ha eseguito almeno 1 passo).
- i **ritardi di comunicazione**: possono essere limitati o illimitati
- l'**ordinamento dei messaggi**: può differire in quanto possono essere consegnati nello stesso ordine in cui sono stati ordinati (FIFO), oppure in ordine differente.
- la **trasmissione dei messaggi**: unicast o multicast.



#### 4.8. Teorema dell'impossibilità di FLP (Fischer, Lynch and Patterson)

È impossibile raggiungere il consenso in un sistema distribuito in presenza anche di un solo processo guasto. Non esiste un algoritmo distribuito in grado di risolvere il problema del consenso nel caso in cui all'interno di un modello asincrono esiste la possibilità che un singolo processo possa crashare. FLP in particolare dimostra che ogni algoritmo fault-tolerant di risoluzione del consenso ha sempre dei casi in cui l'esecuzione non termina mai, ma questi casi sono estremamente rari.

Il **problema della membership** è ciò che rende difficile il raggiungimento del consenso, ovvero sapere chi è che fa parte di un gruppo all'interno del sistema distribuito asincrono. Questo perché in un SD asincrono le velocità di esecuzioni dei processi non sono vincolate, e non siamo in grado di identificare i fallimenti in modo affidabile (un processo lento potrebbe apparire come guasto).

Il fatto è che i sistemi asincroni che vediamo noi non sono gli stessi che avevano usato FLP nel loro teorema. Noi abbiamo SD parzialmente sincroni, ovvero per la maggior parte del tempo il sistema è sincrono, ma accade che durante il funzionamento si comporti in modo asincrono in quanto non ci sono vincoli sul tempo. Si introduce una randomizzazione sul tempo di timeout sui messaggi di probes per avere maggiore garanzia che si è verificato un evento di crash.

## 4.9. Algoritmo di Paxos

Serve a raggiungere il consenso in un sistema distribuito in presenza di guasti. Se eseguito su un insieme di  $N$  processi può tollerare fino a  $k$  fallimenti, dove  $N$  maggiore di  $2k+1$ , in sistemi asincroni.

È l'algoritmo principale utilizzato nei SD asincroni. Le assunzioni del modello sono:

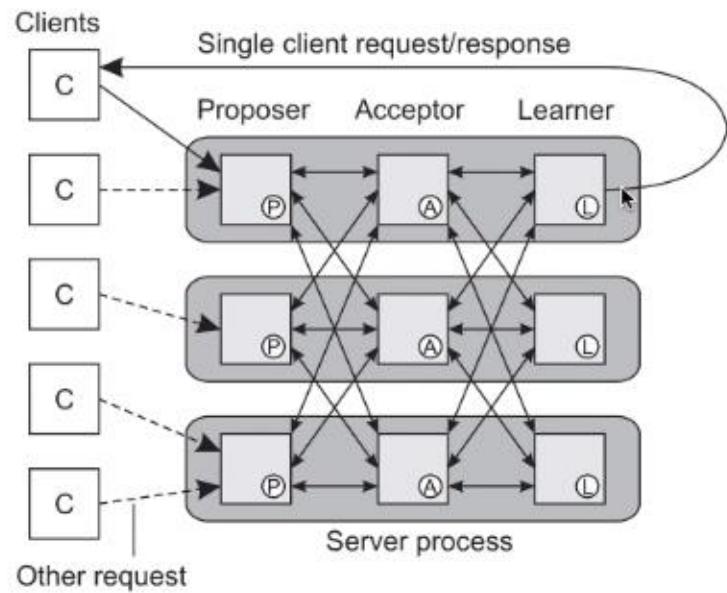
- Il sistema è parzialmente sincrono (o asincrono) soggetto a guasti non bizantini.
- I processi comunicano per scambio di messaggi
- La comunicazione può essere non affidabile (i messaggi possono essere persi o duplicati) e lenta.
- Per quanto riguarda i processi, serve che siano noti a priori quali fanno parte della membership; essi operano a velocità arbitraria e sono di tipo fail-stop (non possono verificarsi failures arbitrari). Qualora falliscano possono essere restartati, e possono ricordare alcune informazioni se fatti ripartire dopo il failure. Infine, non devono raggiungere risultati errati.

L'algoritmo di Paxos è basato su quorum. L'obiettivo è che l'insieme dei processi raggiunga il consenso su un singolo valore proposto. Se viene scelto un valore gli altri processi lo impareranno. I processi votano su ciascuna proposta di valore, e ad ognuna di queste è associato un numero di processo univoco (il quale viene assegnato questo numero di processo è uno dei punti dibattuti dell'algoritmo di Paxos). Una proposta verrà approvata se una maggioranza dei processi la vota. I processi che partecipano al protocollo devono essere noti a priori abbiam detto, quindi la cardinalità della membership è nota a priori, così come la dimensione della maggioranza. Un processo che sta valutando due differenti proposte approva quella col numero di versione più alto associato. L'algoritmo **soddisfa la proprietà di safety** (il consenso viene raggiunto solo su un valore che viene proposto, viene scelto solo un valore e non esiste un processo che impari che un valore è stato scelto a meno che non venga scelto per davvero) ma **non garantisce la liveness** (alla fine viene sempre scelto un valore proposto e quando un valore viene scelto, un processo può impararlo). L'algoritmo di Paxos non garantisce la liveness perché ne possono esistere versioni che non terminino mai, ma la probabilità che quelle versioni si verifichino è talmente bassa che consideriamo che l'algoritmo di Paxos termina. Questo perché come abbiamo visto i nostri sistemi la maggior parte del tempo funzionano come sincroni e a volte sono asincroni, e non sempre asincroni come ci dice il teorema di impossibilità di FLP.

Perciò diciamo che ci viene sempre garantita la safety ed il tradeoff è sulla liveness.

Nell'algoritmo ogni processo può assumere 3 differenti ruoli: **proposer** (propone un valore), **acceptor** (coloro che votano), **learner** (i processi che imparano quale valore è stato scelto). Un singolo processo può svolgere anche tutti e tre i ruoli, ma per semplicità ipotizziamo ogni processo possa svolgere un singolo ruolo.

Se abbiamo un solo processo che fa il ruolo di acceptor è la soluzione più semplice ma non è distribuito il sistema, perché se fallisce quello è game over. Bisogna avere acceptor multipli, ciascuno dei quali può accettare un valore che è stato proposto. Quando gli acceptor raggiungono una maggioranza semplice allora quel valore verrà scelto. Due maggioranze devono avere almeno un acceptor in comune, perché essi possono accettarne solo uno alla volta, così facciamo in modo che diversi gruppi concordano lo stesso valore.



Per distinguere proposte diverse, gli acceptor necessitano di un numero di proposta. Ogni proposta sarà costituita dal numero di proposta e dal proposed value. Possono esistere multiple proposte per lo stesso valore in quanto proposte da diversi proposer. Altra assunzione è che i processi devono avere a disposizione uno storage persistente stabile che in caso di fallimento preservi lo stato del protocollo per recuperare le info dopo essere tornato funzionante in caso di crash.

L'algoritmo di Paxos adotta un approccio basato su molteplici round, molteplici interazioni dell'algoritmo. Quando in un round viene raggiunta la decisione da parte degli acceptor, nei round successivi ci si assicura che anche tutti gli altri raggiungano il consenso su quel determinato valore. Ogni round è composto da due fasi, una di **prepare** (i proposer prendono il valore da proporre, si bloccano le vecchie proposte non completate) ed una di **accept** (si chiede agli acceptor di accettare un valore specifico).

#### 4.9.1. Fase 1: prepare

- Il proposer seleziona il valore che vuole proporre e il numero di proposta, ed invia una richiesta di prepare ad una maggioranza degli acceptor.
- Quando un acceptor riceve una richiesta di prepare con numero di richiesta pari a N. Se questo è maggiore di quelli a cui ha già risposto, l'acceptor promette di non accettare proposte con numeri di sequenza inferiore e risponde al proposer alla richiesta con il valore più alto che ha eventualmente accettato ed il valore della proposta che ha appena accettato. In un round precedente potrebbe già aver accettato una proposta, quindi potrebbe già essere stato raggiunto il consenso su un determinato valore.

#### 4.9.2. Fase 2: accept

- Se il proposer riceve una risposta alla sua richiesta di prepare per la proposta numerata N da una maggioranza di acceptor, manda una richiesta di accept ad ognuno di quegli acceptor con la proposta numerata N con un valore V corrispondente al valore della proposta numerata più grande tra le risposte della fase 1.
- Se un acceptor riceve una richiesta di accettare la proposta N, la accetta a meno che non aveva già risposto ad una richiesta di prepare che aveva un numero più grande di N.

#### 4.9.3. Proprietà di Paxos

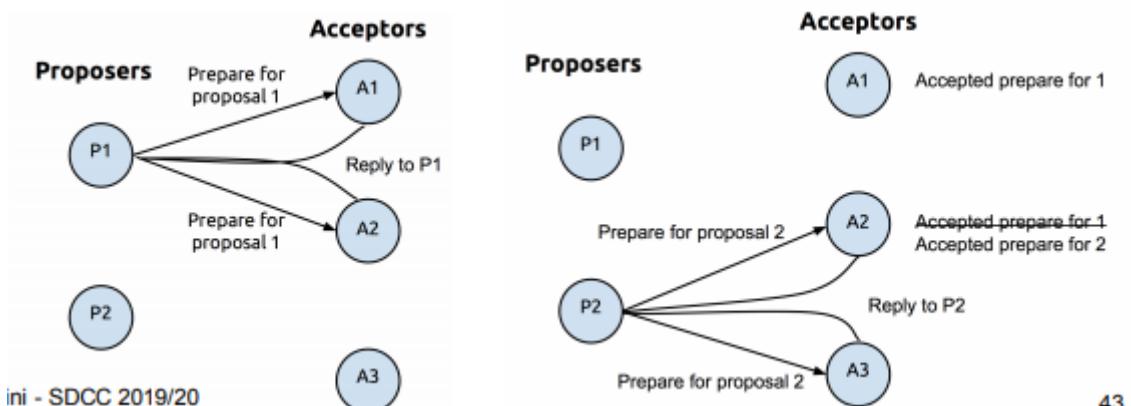
1. Ogni numero di proposta è unico.
2. Ogni due set di acceptors deve avere almeno un acceptor in comune.
3. Il valore inviato in fase 2 è quello relativo alla proposta con numero più alto delle risposte della fase 1.

#### 4.9.4. Esempio di Paxos

Abbiamo 5 processi: i proposer sono p1 e p2, gli acceptor sono a1, a2 e a3. L'obiettivo è quello di riuscire a garantire che tutti gli acceptor (che sono complessivamente 3) raggiungano la maggioranza su un valore.

##### Primo round: prepare phase

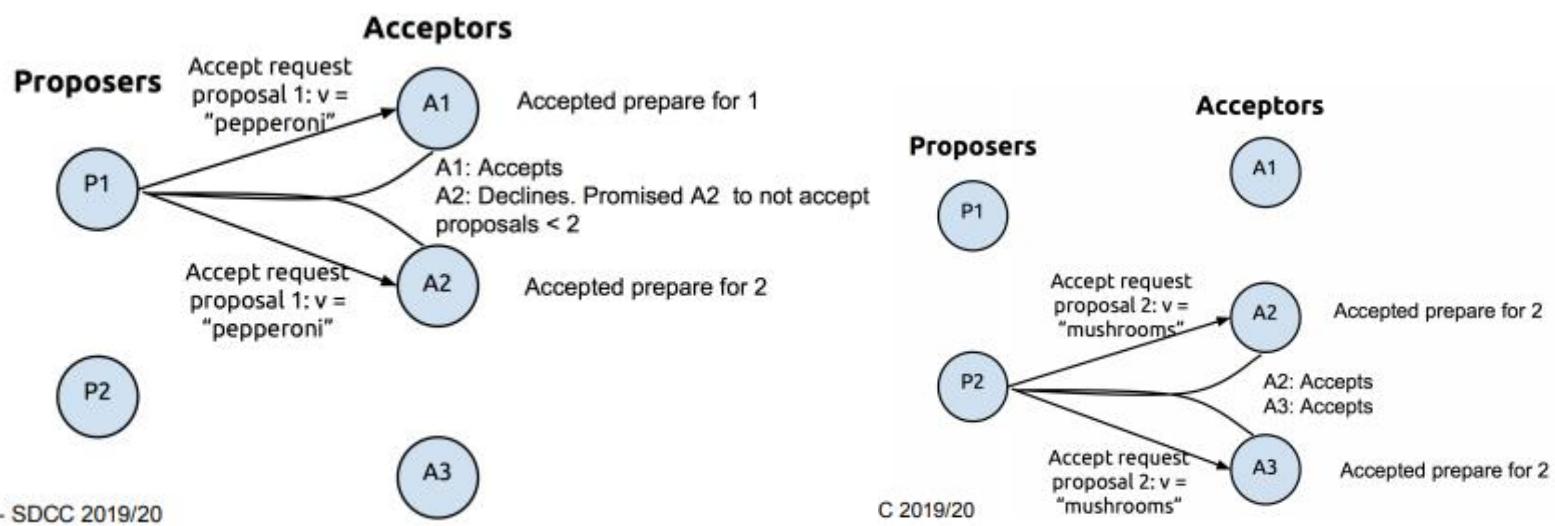
- p1 manda una richiesta di prepare per la proposta 1 ad un sotto insieme di acceptor a1 e a2
- a1 e a2 rispondono a p1 che sono all'inizio dell'algoritmo e che non hanno visto nessun valore finora
- p2 manda la sua richiesta di prepare per la proposta 2 ad a2 e a3.
- a2 e a3 rispondono a p2.



43

##### Primo round: accept phase

- p1 ha ricevuto la risposta da a1 e a2, nessuno gli ha segnalato un valore su cui scegliere, così p1 sceglie il valore di accept da inviare ad a1 e a2, facciamo che quel valore è peperoni. Quindi gli invia valore e numero di sequenza, quindi 1-PEPERONI.
- a1 accetta la richiesta di prepare proveniente da p1; a2 invece non la accetta perché aveva promesso a p2 che non avrebbe accettato richieste con numero di sequenza minore di 2: declina quindi la richiesta di p1.
- p2 non aveva ricevuto suggerimenti sul valore da scegliere, quindi invia la richiesta di accept 2-MUSHROOM; a2 e a3 accettano entrambi la richiesta stavolta.



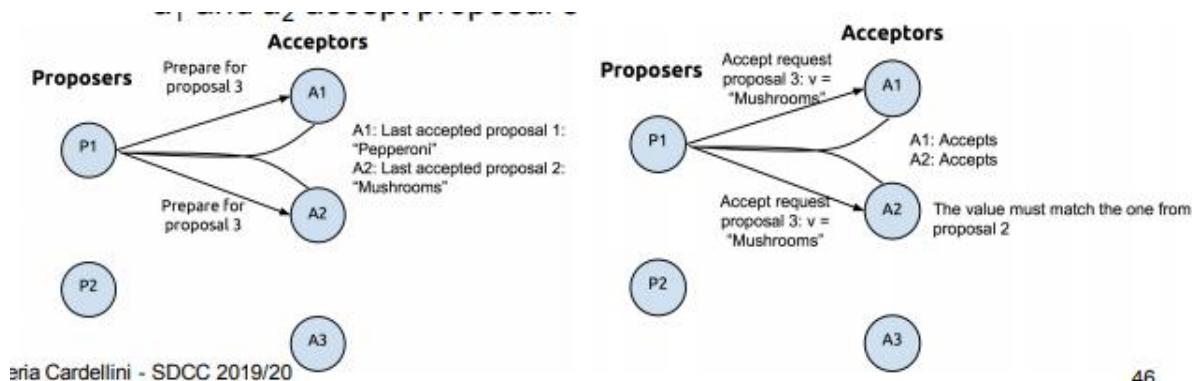
72

### Secondo round: prepare phase

- p1 manda richiesta di prepare ad a1 e a2 perchè la sua proposta era stata rifiutata e ci riprova incrementando il numero di sequenza che diventa 3. a1 aveva accettato il numero di sequenza 1 (peperoni), a2 aveva accettato numero di sequenza 2 (mushroom) e gli rispondono.

### Secondo round: accept phase

- P1 invia la richiesta di accept ad a1 e a2 per 3-MUSHROOM. A1 e a2 accettano entrambi, a2 accetta perchè il valore è uguale al precedente e pechè l'id è maggiore.

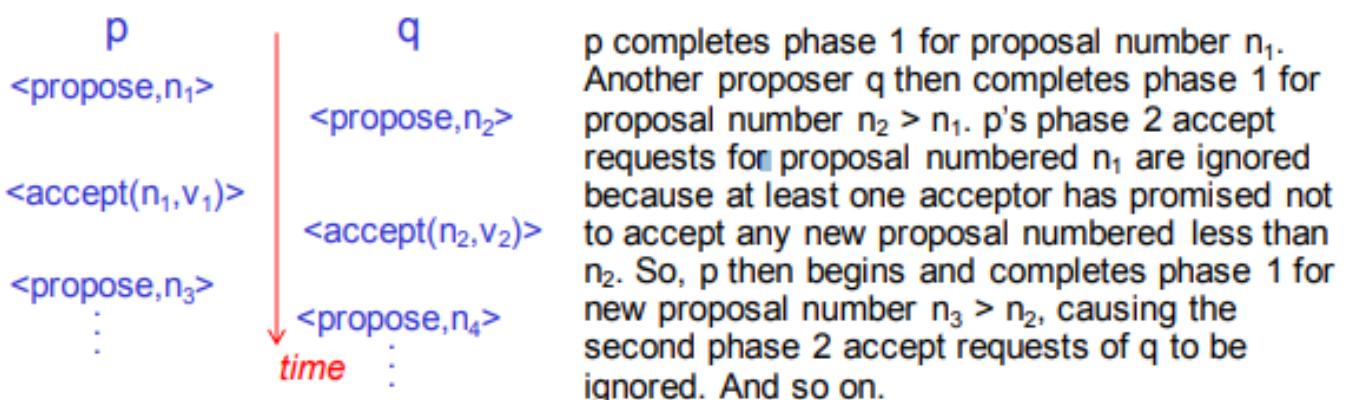


Per quanto riguarda i learner, ci sono diverse opzioni per imparare un chosen value.

Ogni acceptor quando accetta una proposta informa tutti gli altri learners (problema è che ci sono un sacco di messaggi scambiati così), o volendo può mandare ad un distinguished learner (problema del collo di bottiglia). Soluzione intermedia, si sceglie di mandare ad un set di distinguished learners. Limitiamo così i messaggi scambiati e no single point of failure.

Paxos non ci da garanzia sulla liveness, in quanto l'algoritmo potrebbe non terminare mai se due proposer propongono valori conflittuali fra di loro mandando in stallo il protocollo. Una situazione adatta all'esempio è:

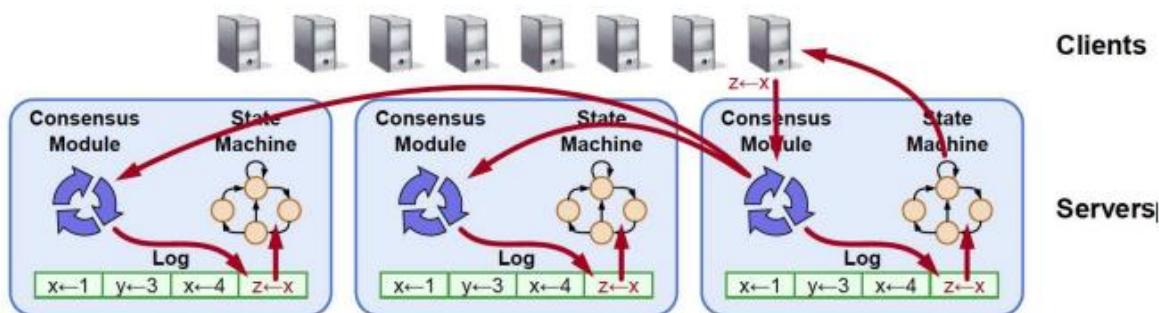
Supponiamo di avere due proposer, p e q. Supponiamo che p che completa la fase 1 con n1 e q completa la fase 1 con n2. Nella fase di accept p chiede l'accept di (n1, v1) e q di (n2, v2).



Per ovviare a questa situazione, molte implementazioni di Paxos fanno sì che ci sia solo 1 proposer attivo e gli altri proposer inviano le proposte solo quando il proposer attivo subisce il fallimento e tramite elezione viene promosso un nuovo leader.

Paxos è l'algoritmo usato nella state machine replication.

La **state machine replication** è un approccio generale in cui vogliamo effettuare computazione distribuita su diversi nodi di un SD, Paxos è proprio usato in questa. Nella SMR si utilizza una variante dell'algoritmo di Paxos chiamata **multi-paxos** in cui c'è un singolo leader che effettua molteplici round, quindi la fase di prepare avviene solo nel primo round, per poi avere solo fase di accept nei round successivi.



## 4.10. RAFT (Replicated And Fault Tolerant)

Designato per essere più semplice da capire e spiegare rispetto a Paxos. Questo algoritmo di RAFT viene progettato per essere direttamente implementabile a differenza di Paxos e verificato senza prove di correttezza che in Paxos sono sofisticate.

Il modello del sistema considerato è simile a Paxos, quindi i messaggi possono essere rallentati o persi e non abbiamo fallimenti bizantini. È equivalente al multi-paxos, 1 fase prepare e poi tutti accept con 1 leader. A differenza di Paxos i problemi sono decomposti in sottoproblemi relativamente indipendenti.

RAFT è un protocollo basato sull'idea della state machine replication. La macchina a stati è una macchina deterministica che specifica il comportamento desiderato nella sua interezza. La state machine

RAFT è uno state machine replication protocol:

- La state machine è un programma deterministico che specifica il comportamento desiderato del cluster come un insieme.
- La state machine processa una sequenza di comandi, i quali sono dati dai client del cluster.
- I client esterni interagiscono con il sistema come se ci fosse un singolo nodo che sta running una singola copia della state macchine.
- Ogni nodo in un Raft cluster simula la copia della state machine
- L'obiettivo del protocollo: mantenere la consistenza tra le copie.

L'algoritmo di RAFT ha **2 componenti fondamentali**: uno per l'**elezione del leader** (scegliere 1 dei server che dovrà agire come leader, che quando fallirà verrà eletto un altro) e uno per la **log replication** (operazione normale che permette di far sì che la macchina di RAFT sia aggiornata e consistente sulla maggioranza dei server che compongono il cluster: replicazione del log sugli altri server evitando le inconsistenze).

Cosa intendiamo per il consenso distribuito?

Partiamo facendo un esempio, prendiamo un singolo nodo che dobbiamo considerare come un database server che immagazina un singolo value.

Dall'altro lato abbiamo un client che può interagire con il DB potendo mandare un valore a tale server. Per arrivare all'accordo, o consenso, sul valore trasmesso dal client è facile con un solo nodo. Quindi la vera domanda da porsi è, con una molitudine di nodi server come arriviamo al consenso? Questa tematica racchiude il problema del consenso distribuito. **RAFT** è un protocollo per implementare il consenso distribuito.

Per comprendere appieno come funziona, sappiamo che un nodo può essere in 3 stati: **Follower** state, **Candidate** state o **Leader** state.

Tutti i nodi iniziano con lo stato di **Follower**. Se un Follower non viene ascoltato da un leader, diventerà un candidato. Il candidato successivamente richiede i voti dagli altri nodi, i quali risponderanno con il loro voto. Il candidato diventerà il nuovo leader se riceve la maggioranza dei voti. Questo processo è chiamato **Leader Election**. Tutti i cambiamenti ora verranno gestiti dal nuovo leader. Ogni cambiamento viene aggiunto come una entry nel log del nodo. Questo log entry è momentaneamente uncommitted così da non aggiornare il valore del nodo. Per fare il commit dell'entry, il primo nodo replica tale entry ai nodi followers, aspettando finchè la maggioranza dei nodi scrive la entry. Adesso la entry è finalmente committed sul nodo leader e il suo stato verrà aggiornato. Successivamente il leader deve avvisare i nodi follower che la entry è stata committed. Il cluster ha ora il consenso dello stato del sistema. Questo processo è chiamato il **LOG REPLICATION**.

#### 4.10.1. LEADER ELECTION:

Nel protocollo RAFT ci sono 2 timeout settings che controllano le elezioni. Il primo è chiamato election timeout, il quale descrive l'ammontare del tempo di attesa di un follower prima di diventare un candidato, un valore che oscilla tra i 150ms ai 300ms. Quindi ricapitolando, dopo il passare di questo timeout, succederà che un follower diventerà un candidato e inizierà una nuova election term. In primis, si voterà da solo, successivamente manda le richieste di voto agli altri nodi. Se per qualsiasi motivo non riceve risposta da un nodo, automaticamente il suo 'astenersi' verrà tradotto come un voto per il candidato, e il nodo resetta il timeout election. Se tale candidato raggiungerà la maggioranza dei voti diventerà leader. Il leader inizia a mandare ai suoi follower delle append entries, a intervalli di tempo specifici chiamati heartbeat timeout. I followers poi rispondono al messaggio di append entries. Questo election term continuerà finchè un follower smette di ricevere un heartbeat e diventerà un candidato.

Vi è un caso particolare, che descrive il momento esatto in cui due candidati iniziano a mandare le richieste di voto allo stesso momento. Per questo preciso caso, si andrà a restartare il timeout per 'sperare' che un nodo diventerà candidato in modo singolare.

#### 4.10.2. LOG REPLICATION:

Abbiamo detto che dopo aver eletto un nodo leader sarà necessario replicare tutti i cambiamenti all'interno del nostro sistema di nodi. Questo procedimento verrà gestito dall'append entries che usa gli heartbeats.

Il processo logico parte da un client, il quale manda un cambiamento al leader che lo appende sul suo log. Successivamente il cambiamento viene mandato ai suoi followers con il prossimo heartbeat. Una entry sarà committed solo quando la maggioranza dei followers ne saranno a conoscenza e avranno risposto. Ora per praticità, mandiamo un comando per aumentare il valore di 2. Dopo i processi descritti in precedenza il nostro valore sarà aggiornato nel sistema. RAFT riesce a essere consistente anche in caso di partizioni di rete. Per questo esempio usiamo 5 nodi e un client.

Separiamo tramite una network partition i nodi A, B dai nodi C, D, E. Con questa divisione dei nodi, si andranno a formare due leader diversi. Un client ora vuole mandare alla coppia A, B un valore = 3. Tramite i processi scritti sopra, il sottosistema A, B si aggiornerà con tale valore. Quello che succede però, è che B non potrà uncommittare la sua entry, in quanto non è abile nel comunicare con la sottoparte C, D, E. Un altro client birichino, cercherà di modificare il valore della sottorete C, D, E con un 8, con un successo perché riesce a replicarlo alla maggioranza. Ora successivamente aggiustiamo

la network partition, così che il nodo B vedrà l'election higher term and step down. Ora i nodi A e B fa l'undo sul suo uncommitted entries e si matcha con il log leader. Il nostro log ora è consistent per tutto il nostro cluster.

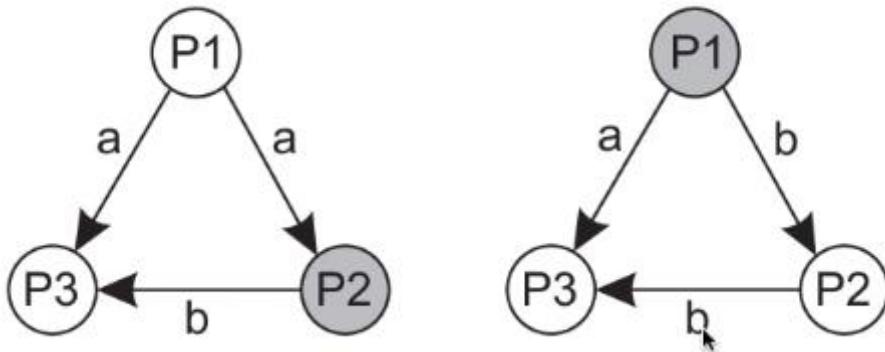
L'algoritmo di RAFT garantisce la safety e una garanzia di liveness se ci sono pochi fallimenti nel sistema ed è tollerante alle partizioni di rete.

Sia Paxos (poiché il criterio di voto è a maggioranza e nelle iterazioni successive vengono considerati anche i nodi che erano della minoranza) sia RAFT sono tolleranti alle partizioni di rete. Mentre i primi due affrontavano i guasti non bizantini, ora vediamo uno scenario di algoritmo per far fronte a fallimenti bizantini.

## 4.11. Problema dell'accordo bizantino

In presenza di fallimenti bizantini, affinché il sistema possa sopravvivere e continuare a funzionare correttamente per raggiungere l'accordo distribuito, servono almeno  $3k+1$  processi. Se sappiamo che  $k$  processi sono soggetti a fallimenti bizantini ci serve che il nostro SD sia costituito da almeno  $N=3k+1$  processi. Si tratta del problema dei generali bizantini definito da Lamport.

L'idea è che si vuole raggiungere l'accordo se attaccare una città oppure ritirarsi tra un gruppo di generali fedeli, essendoci  $k$  generali traditori. Occorreranno  $2k+1$  generali fedeli, poiché se non ci sono più dei  $2/3$  di generali fedeli è impossibile raggiungere l'accordo. P3 non riesce a capire chi sia il traditore tra p1 e p2. Nell'esempio di 3 generali ed 1 traditore non si riesce a raggiungere il consenso.



Assunzioni del problema sull'accordo bizantino:

- Processi sincroni fra loro
- Comunicazione unicast (vengono inviati singoli messaggi verso ognuno dei processi con cui si vuole comunicare)
- Ordinamento dei messaggi FIFO
- Ritardi limitati

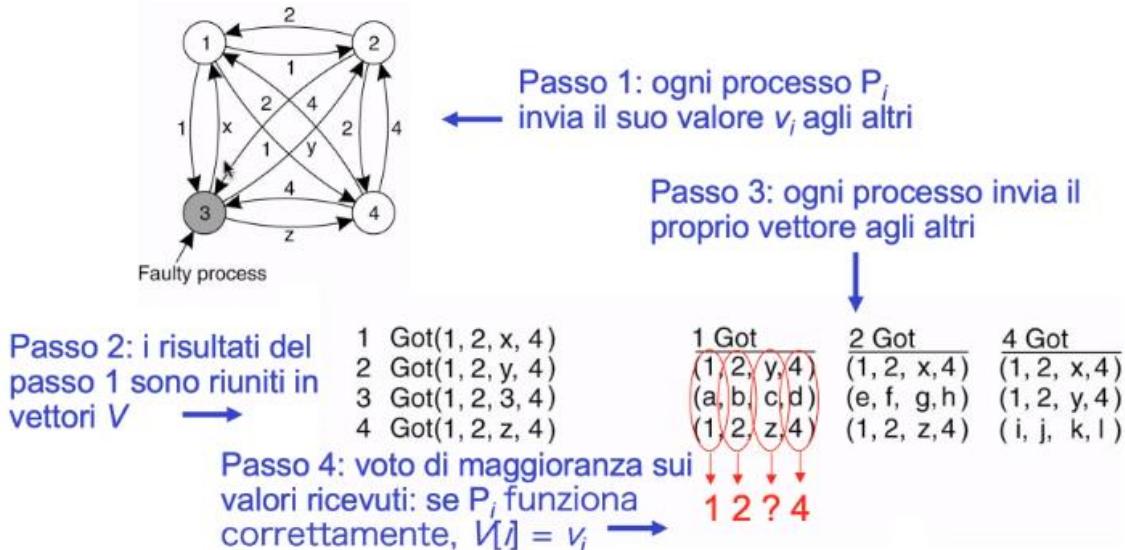
Supponiamo di avere  $N$  processi, dove ognuno fornisce un valore  $v_i$  agli altri processi. L'obiettivo è quello di far costruire ad ogni processo un vettore  $V$  di dimensione  $N$  tale che se il processo  $i$  è non guasto, allora  $V[i] = v_i$ , altrimenti  $V[i]$  non è definito ( $v_i$  è come se fosse la forza della truppa del generale  $i$ ).

### Esempio

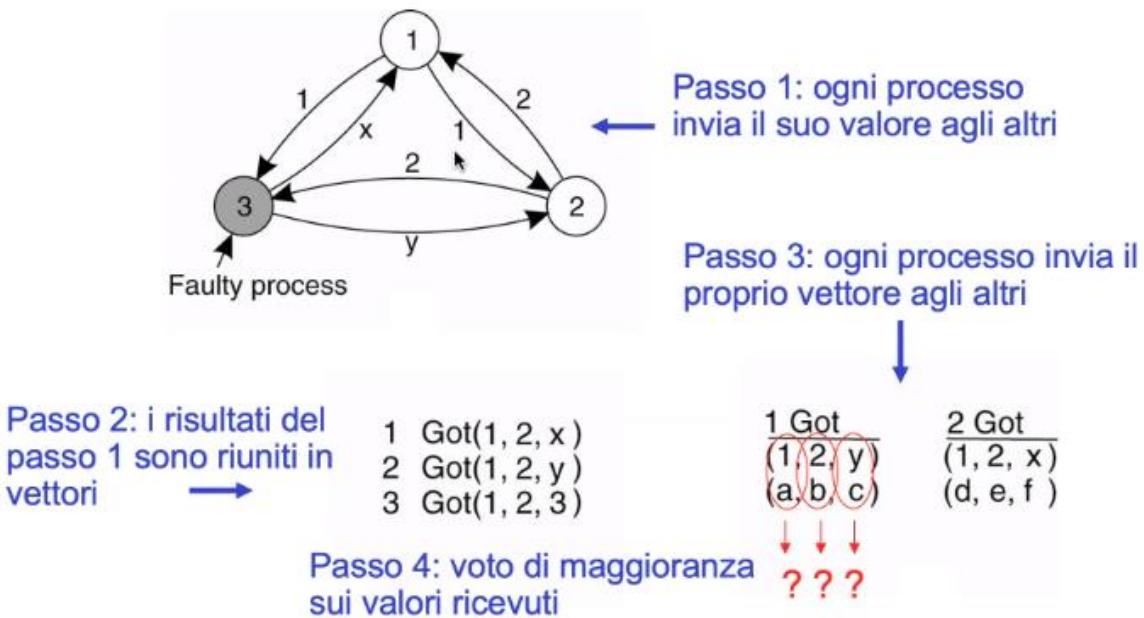
Supponiamo di avere 4 processi e vediamo come essi tollerano la presenza di un guasto bizantino.

Abbiamo  $k=1$  quindi serve avere  $2k+1$  processi non soggetti a guasti bizantini, quindi 3.

Assumiamo che il valore comunicato da ogni processo corrisponde al valore nel seguente esempio ( $v_i = i$ );



Perché con 2 processi che funzionano correttamente ed 1 processo fraudolento  $N=3$ ,  $k=1$  non si riesce a raggiungere l'accordo?



2 processi non sono sufficienti a raggiungere l'accordo in presenza di un processo soggetto al guasto bizantino in quanto non si riesce ad arrivare a conoscenza di quali valori sono scritti nel vettore.

## 4.11.1. Algoritmo di Lamport per l'accordo Bizantino (Oral Message OM)

### L'algoritmo OM(0):

1. Il comandante manda il suo valore a tutti i tenenti.
2. Ogni tenente usa il valore che riceve dal comandante, o usa il valore RETREAT se non riceve nessun valore.

### L'algoritmo OM( $k$ ), $k > 0$ :

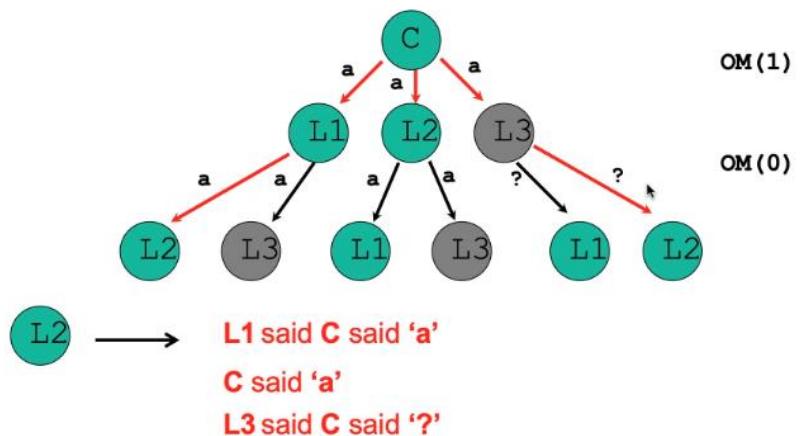
1. Il comandante manda il suo valore a tutti i tenenti
2. Per ogni  $i$ , sia  $v_i$  il valore che il tenente  $i$ -esimo riceve dal comandante, oppure RETREAT se non ha ricevuto nessun valore. Il tenente  $i$ -esimo finge di essere il comandante nell'algoritmo OM( $k-1$ ) per mandare il valore  $v_i$  agli altri  $N-2$  tenenti.
3. Per ogni  $i$  e ogni  $j \neq i$ , sia  $v_{ij}$  il valore che il tenente  $i$ -esimo ha ricevuto dal tenente  $j$ -esimo nello step 2 (utilizzando OM( $k-1$ )), oppure RETREAT se non ha ricevuto nessun valore. Il tenente  $i$ -esimo usa il valore  $\text{majority}(v_1, \dots, v_{N-1})$

### Esempio:

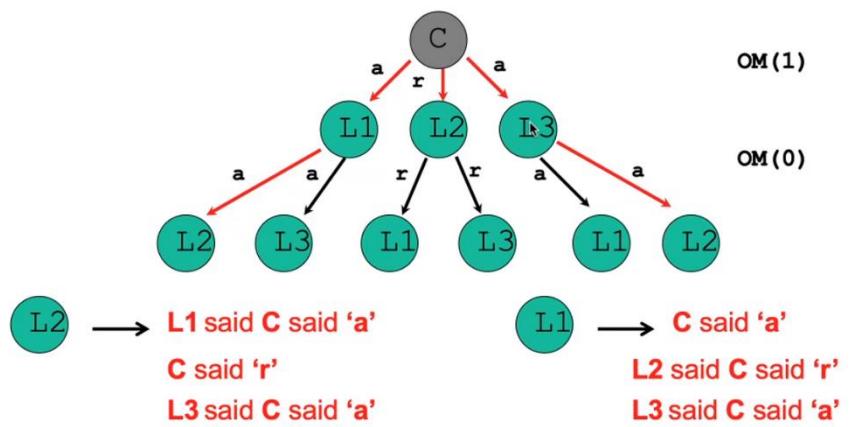
Vediamo un esempio per capire il suddetto algoritmo. Supponiamo che il comandante sia 1 e che i colonnelli siano 3 (Lieutenant 1 2 e 3).

Nel passo 1 il comandante invia a ai 3 colonnelli, poi ciascun colonnello agisce come se fosse un comandante ed invia il suo valore a tutti gli altri.

Ora i colonnelli applicano la maggioranza e si raggiunge il consenso sul valore a:



E se il traditore fosse il comandante? Ipotizziamo che comunica di attaccare a L1 e L3 ma comunica di ritirarsi a L2. Secondo gli stessi procedimenti di prima, vengono comunicati i seguenti valori per poi applicarvi la maggioranza.



#### 4.11.2. La tolleranza ai guasti bizantini a livello pratico

Performance dell'algoritmo per problemi bizantini generici:

- L'algoritmo usa  $k+1$  round sincroni: abbastanza lento
- Esiste un numero esponenziale di messaggi  $O(N^k)$  usati dall'algoritmo

Il protocollo della tolleranza ai guasti bizantini è stato considerato per molto essere troppo costoso per essere applicato a livello pratico.

- I bitcoin usano la tolleranza ai guasti bizantini oggi.

### 5. Problema delle commit distribuite

Problema che si verifica tipicamente nei database distribuiti in cui abbiamo una transazione distribuita fra un insieme di processi e si vuole la decisione se rendere permanente la transazione oppure gli effetti di quest'ultima debbano essere annullati. In letteratura vengono definiti gli algoritmi quindi di commit che permettono di raggiungere le decisioni di commit o di abort. Questi algoritmi si chiamano **commit a 2 fasi** e **commit a 3 fasi**. Si chiamano protocolli a più fasi proprio perchè il protocollo è suddiviso in una prima fase di voto e una seconda di decisione.

Il primo che prevede due fasi ha il problema che non è soddisfatta la liveness in quanto è un protocollo bloccante.

Per questo è stato introdotto un commit a 3 fasi che risolve il problema del blocco che si verificava nel commit a 2 fasi, però non è un protocollo tollerante a partizioni di rete: in presenza di partizioni di rete permette di fare il commit o l'abort di una transazione anche se qualche processo non ha partecipato alla votazione.

Per risolvere il problema delle commit distribuite si può utilizzare sia Paxos che RAFT in quanto tollerano le partizioni di rete. In Paxos finchè c'è maggioranza di processi che funzionano e che possono comunicare fra di loro possono raggiungere l'accordo e appena viene meno la partizione di rete, i restanti nodi che non potevano comunicare prima si vanno a riallineare sulle decisioni prese sugli altri nodi (questo perchè Paxos è a più round) e stesso discorso vale per l'algoritmo di RAFT.