
# Sistemi distribuiti e pervasivi

Fabio Brambilla - Simone Malesardi - Federico Piscitelli - Omar Zaher

Matricole Feb - 978989 - 970949 - 970944

Professore Claudio Bettini

A.A. 2020/2021

Ultimo aggiornamento January 22, 2022

# Indice

# Specifiche

Il corso introduce la distribuzione dei sistemi e la loro estensione ai sistemi pervasivi ottenuti dall'inclusione di oggetti smart, IOT, sensori. Con un sistema l'obiettivo è anche quello di non far vedere all'utente tutta la complessità del sistema, far riuscire a far sincronizzare tutti i componenti di un sistema, gestire i guasti.
Creazione di un progetto di un sistema pervasivo distribuito.

Fondamenti del corso:

- organizzazione dei nodi nei sistemi e architettura client-server
- comunicazione, modelli di comunicazione nei sistemi distribuiti
- algoritmi sulla sincronizzazione tra i vari sistemi e sul tempo (mutua esclusione e algoritmi di elezione di un nodo: abbiamo un insieme di nodi, ma chi è il capo?)
- seminario sull'applicazione dei sistemi in Google

Modalità d'esame:

- scritto di fine appello (fine maggio/inizio giugno): domande a scelta multipla e un paio di domande aperte tra cui un esercizio
- progetto individuale

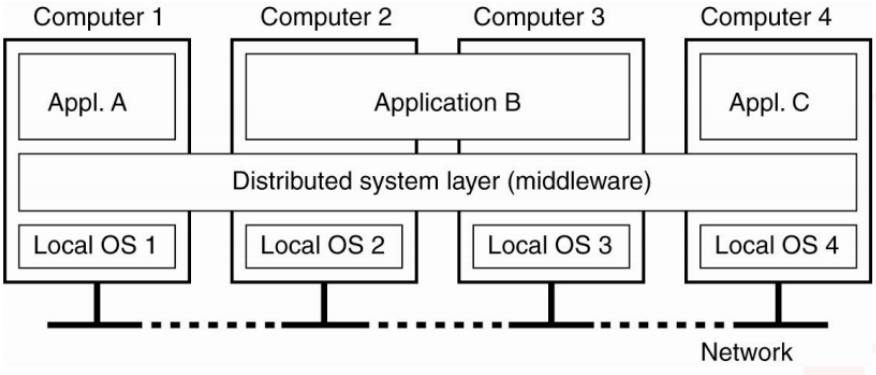
L'esame deve essere concluso prima del nuovo anno accademico. Il progetto viene presentato a fine aprile. Il voto finale è calcolato come la media dei due voti.

# Introduzione

**——— Lezione 1 - 3 marzo 2021 ———**

*Un sistema distribuito è una collezione di computer indipendenti (nodi collegati, che possono comunicare tra loro) che appare ai suoi utenti come un singolo sistema coerente.*

I nodi non sono solo indipendenti ma anche collegati tra loro (non hanno quindi una memoria condivisa e sono collegati in qualche modo).




Dal punto di vista architetturale abbiamo una collezione di PC, ognuno ha un local OS (sistema operativo) e ognuno ha applicazioni che vuol far girare. C'è poi uno strato che permette di far collaborare i vari PC che si chiama middleware, il quale mette a disposizione alle applicazioni la stessa interfaccia.

Si sviluppa un software ad un certo livello che permette di andare a nascondere agli utenti finali gli n computer e farli apparire come se fosse un sistema unico.

Esempi:

- se abbiamo un insieme di PC in rete e un file system condiviso, non solo i file sono condivisi ma anche le risorse di calcolo
- nei multiplayer online games, chi gioca non deve sapere dove lo stato del gioco viene memorizzato
- WWW sistema distribuito che si occupa della gestione distribuita di documenti identificati univocamente con URL. Il web non è esattamente un sistema distribuito. Attraverso l'URL si capisce dove si trova il documento, la trasparenza, quindi, è svanita.
  In un sistema distribuito vero devo avere identificatori di risorse che non danno informazioni sulla locazione dei documenti.

. Lamport, l'inventore di latex, dice "You know you have one when the crash of a computer you've never heard of stops you from getting any work done". Significa che se abbiamo un problema è difficile capire cos'è successo perché in un sistema distribuito è difficile effettuare il debugging.

# Obiettivi di un sistema distribuito

## 2.1 Accessibilità

Un sistema distribuito deve rendere accessibili le sue risorse.

## 2.2 Trasparenza

Ci sono diversi tipi di trasparenza:

- Accesso: nasconde differenze di filesystem e/o come sono rappresentati i dati e come viene effettuato l'accesso ad essi. Esempio: considerando i 4 pc che fanno parte del sistema, voglio far risultare trasparente un accesso ad un file condiviso. Ogni PC però ha un file system diverso (esempio uno EXT4 per Linux, NTFS per Windows...).
- Locazione: non devo sapere dove si trova la risorsa
- Migrazione: nasconde che la risorsa potrebbe spostarsi
- Rilocazione: nasconde che la risorsa potrebbe spostarsi durante l'utilizzo
- Replicazione: nasconde quando una risorsa viene replicata. Questo viene fatto per ridondanza (sopperiamo a eventuali mal funzionamenti) ed efficienza.
- Concorrenza: nascondere che una risorsa potrebbe essere condivisa da molti utenti che eseguono modifiche in maniera concorrente
- Failure: il sistema deve nascondere eventuali fallimenti e ripristinare automaticamente la risorsa

## 2.3 Apertura (Openess)

Un sistema aperto dovrebbe offrire:

- interoperabilità
- portabilità
- estensibilità

L'apertura può essere raggiunta attraverso:

- l'utilizzo di protocolli standard
- pubblicazione di interfacce chiave
- testing e verifica della conformità dei componenti rispetto a determinati standard

## 2.4 Scalabilità

La scalabilità è una caratteristica fondamentale in un sistema distribuito di larga scala. Quello che vogliamo ottenere è che più il sistema è grande (e viene ampliato) più è performante. Bisogna dunque evitare centralizzazione di servizi, dati e algoritmi.

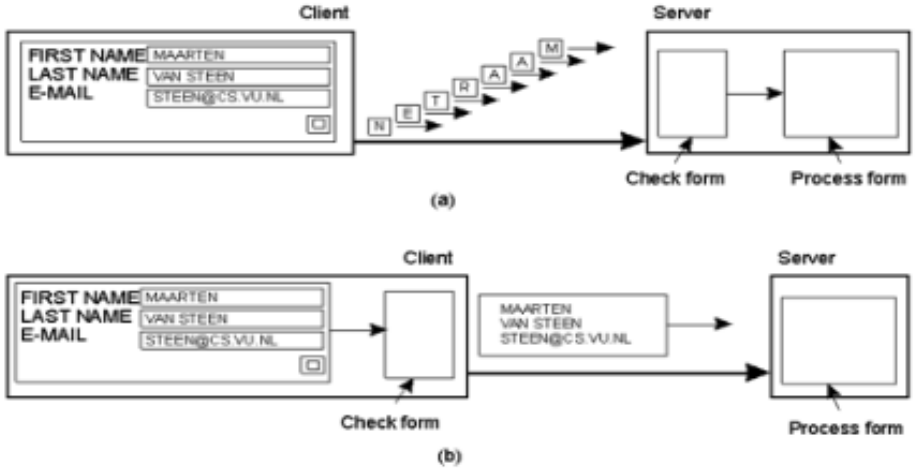
Ci sono diversi problemi in cui si incorre quando si cerca di ottenere la scalabilità:

- centralizzazione dei servizi: singolo server per tutti gli utenti. Esempio: il bonus bici dove si ha un unico server per più utenti e dove la risorsa stava nello stesso posto. Tutti gli utenti nello stesso momento hanno cercato di accedere a quella risorsa.
- centralizzazione dei dati. Esempio: un singolo elenco telefonico on-line
- centralizzazione degli algoritmi: routing basato su tutte le informazioni.

Le caratteristiche degli algoritmi decentralizzati:

- nessuna macchina (nodo) dispone di informazioni complete sullo stato del sistema
- le macchine prendono decisioni basate solo su informazioni locali
- il guasto di una macchina non rovina l'algoritmo
- non esiste l'assunzione di avere un clock globale. Non è possibile coordinare gli orologi sulle macchine, infatti non sono mai sincronizzati perfettamente

*In un contesto client server come faccio a rendere scalabile un sistema?*
Si può delegare del lavoro al client, in modo che aggiungendo client (scalabilità) il lavoro del server aumenti ma sia comunque gestibile (poichè abbiamo scaricato del lavoro sul client).




Un sistema distribuito che usiamo tutti i giorni è il DNS. Quando lanciamo un indirizzo simbolico, prima che il browser vada a fare la richiesta web viene interrogato il DNS, si guarda nella cache, si guarda nel name server che abbiamo ma se quello non è in grado di risolverlo, allora si interrogano altri nodi. Questa procedura è del tutto trasparente.

### 2.4.1 Insidie

Quando si sviluppa un sistema distribuito si possono effettuare delle valutazioni errate come:

- la rete è affidabile
- la rete è sicura
- la rete è omogenea
- la topologia non cambia
- la latenza è zero
- la larghezza di banda è infinita
- il costo di trasporto è zero
- c'è un amministratore

- il debug delle applicazioni distribuite è analogo alle applicazioni standard

## 2.5 Tipi di sistemi distribuiti

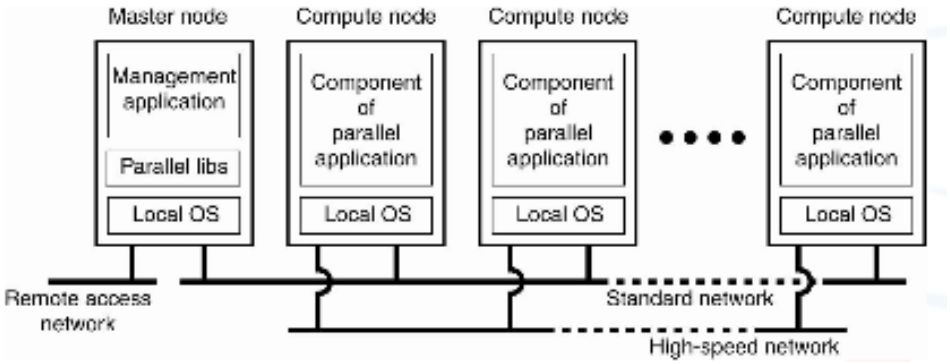
- Distributed Computing Systems:
  - cluster
  - cloud computing
  - edge computing
- Distributed Information Systems:
  - database distribuiti
  - transazioni distribuite
- Distributed Pervasive Systems

### 2.5.1 Distributed Computing Systems

#### 2.5.1.1 Cluster

I cluster sono utilizzati dai principali fornitori cloud.
Un cluster è una collezione di server uguali o simili strettamente connessi da una rete locale ad alta velocità che solitamente hanno lo stesso sistema operativo.
Gli obiettivi principali sono: avere un'attività di elaborazione ad alte prestazioni (distribuisce il calcolo su più nodi) e avere un'alta disponibilità (se un sistema va offline, fornisce comunque la richiesta).
Ci sono due tipi di cluster:

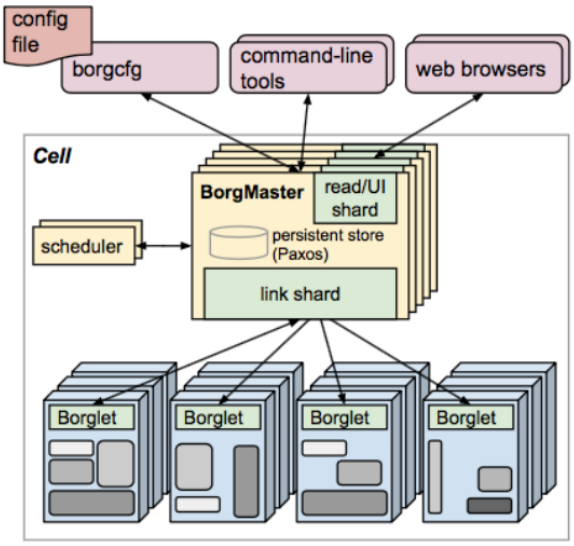
- asimmetrico: è il più utilizzato, qualche nodo ha un ruolo principale (nodo master) rispetto ad altri e ha il compito di distribuire i task ad un insieme di nodi di computazione.
  Esempi: Google Borg, sistema utilizzato internamente da Google per gestire i suoi cluster e Beowulf analogo per Linux.




Ciascun nodo ha un local OS. I nodi sono collegati da high-speed network oltre che attraverso una normale rete.
Il cluster verrà raggiunto dall'esterno attraverso remote access network.
Questi nodi sono tutti uguali, tranne il master che ha un applicazione di management per distribuire il workload sui nodi.

- simmetrici: non c'è un master, tutti i nodi sono allo stesso livello e hanno lo stesso software installato. Questi nodi devono auto-organizzarsi per distribuire il carico e fornire le stesse funzionalità in qualche modo.
  Esempio: MOSIX, che ha molteplici versioni, che in ambito Linux permette di realizzare cluster simmetrico.

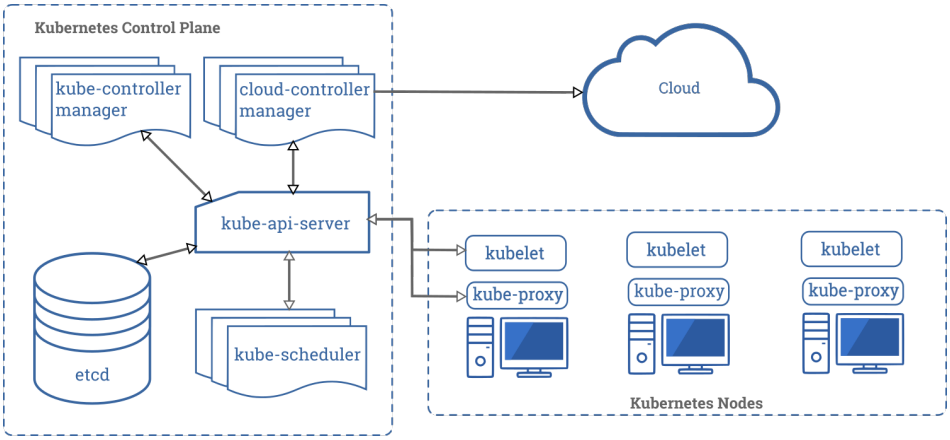
#### 2.5.1.2 Caso di Borg




Ci sono il nodo master (Borg master) e i nodi slave (Borglet) che non sono come il nodo master. I Borglet eseguono parti di applicazioni che vengono scaricati sui nodi

che partecipano (slave) in modo tale che venga distribuita la computazione.
Il master può essere o non essere collegato ad high speed network, poiché la comunicazione più frequente avviene tra i nodi ed è li che si vuole ottimizzare.
Possiamo notare dallo schema che BorgMaster è replicato. Questo perché altrimenti se andasse giù il master si bloccherebbe tutto. I dati fondamentali di cui ha bisogno il master per operare vengono condivisi sulle copie attraverso uno storage distribuito e Paxos garantisce che tutti abbiano la stessa visione sullo stato del sistema.

Da Borg ha preso ispirazione Kubernetes, un sistema di gestione di cluster più sofisticato (sviluppato da Google), che viene utilizzato principalmente in ambienti conteinerizzati. Google ha donato questo sistema alla Linux Foundation ed è quindi open source.




Etcd è un database chiave valore. Borglet - Kubelet.

#### 2.5.1.3 Cloud computing

Il cluster assume che le macchine siano localizzate nello stesso posto, per essere collegate ad una rete ad alta velocità.
Un'infrastruttura cloud invece prevede anche una distribuzione geografica. Le macchine saranno quindi collegate tra loro con canali che utilizzato internet.
Il cloud computer è un modello accessibile in qualunque momento e conveniente che può essere rapidamente fornito e rilasciato con un minimo di sforzo di gestione o interazione con il fornitore di servizi.

#### 2.5.1.4 Caratteristiche cloud

- i nodi sono eterogenei: nei cluster spesso le macchine sono identiche. Nel cloud i nodi possono essere invece molto eterogenei a livello di hardware e a livello di software

- le connessioni di rete sono eterogenee nelle loro capacità e affidabilità
- on-demand self-service: riconfigurare le risorse ad un certo utente in modo semplice
- ampio accesso alla rete: capacità del cloud sono disponibili su tutta la rete accedute da meccanismi standard (trasparenza rispetto all'accesso) tramite API.
- pool di risorse: le risorse di elaborazione del provider vengono raggruppate per servire più consumatori
- rapida elasticità: le funzionalità e le risorse possono essere fornite in modo elastico (dare o restituire rapidamente)
- servizio misurato: chi implementa un cloud deve implementare dei sistemi di misura. Esempio: quanta ram ho usato?

#### 2.5.1.5 Cloud service models

- SaaS: un utente inesperto si appoggia ad un sw fornito dal cloud, ad esempio google docs.
- Paas: una via di mezzo tra IaaS (ho bisogno una macchina, non la voglio comprare ma voglio autonomia su quello che posso installarci, voglio soltanto l'infrastruttura) e SaaS. L'applicazione finale la voglio sviluppare io, però voglio utilizzare db ad alto parallelismo/disponibilità che mi viene fornito. Quindi uso quello del cloud.
- Iaas: sviluppare ed eseguire un software su un'infrastruttura cloud arbitraria

#### 2.5.1.6 Cloud Deployment Models

- private cloud: utilizzato da una singola organizzazione composta da più consumatori, ad esempio l'università di Milano
- community cloud: utilizzato da una specifica comunità di consumatori provenienti da organizzazioni che condividono interessi
- public cloud: uso aperto da parte del pubblico in generale, ad esempio Amazon e Microsoft
- hybrid cloud: posso avere sia dati pubblici che privati. Faccio quindi una soluzione ibrida in maniera da integrare, mantenendo un certa località dei dati e controllo di accesso.

#### 2.5.1.7 Cloud Computing Infrastructure

Switch per gestire una rete ad altà velocità e tecnologie di load balancing con hardware e software molto sofisticati.
Ogni data center ha uno o più cluster di migliaia di nodi collegati tra loro.

#### 2.5.1.8 Edge (Fog) computing

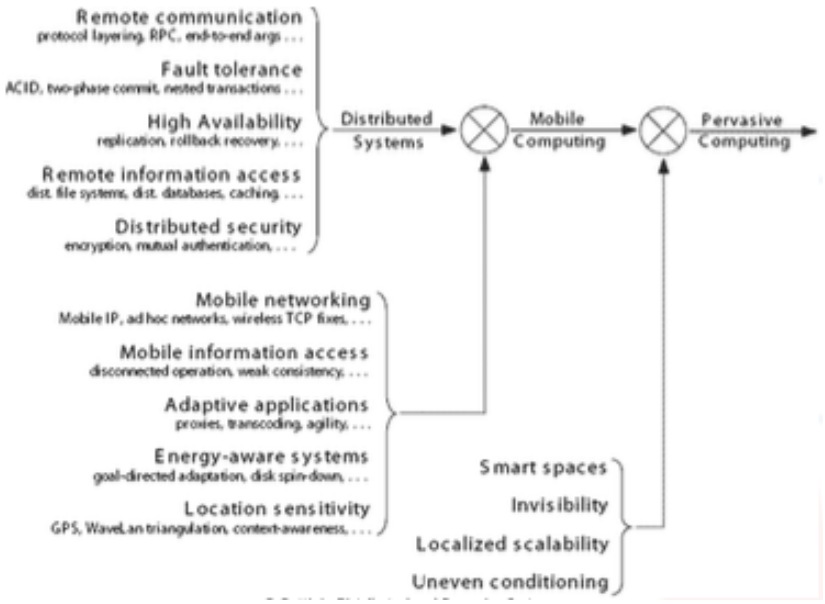
IOT: dispositivi connessi (possono far parte di sistemi disribuiti) spesso con sensori che producono enormi quantità di dati, alcuni dei quali devono essere processati in tempo reale. Quindi la latenza introdotta dal cloud in molti casi si rivela un problema.
L'idea è di avere una struttura gerarchica (anzichè solo cluster dislocati del mondo), in cui abbiamo server vicini a dove i dati vengono prodotti (edge, confine della rete) e questi nodi riescono a connettersi con i dispositivi che producono questi dati e quindi fare processing o preprocessing in tempo reale, in un modo che il cloud generico non riuscerebbe a fare. 5G ha un ruolo chiave in questo ambito, ma da solo non riesce.
I dispositivi di edge possono essere di tipo diverso, ma sostanzialmente sono un primo passaggio da IoT e sensori verso il cloud.
Fog viene usato spesso come sinomino, ma a volte viene utilizzato in maniera differente per indicare i livelli di decentralizzazione in gerarchie a più di 3 livelli (?).

### 2.5.2 Distribuited Pervasive Systems

L'idea generale è che il calcolo sparisca, nel senso che la trasparenza sia totale. Non dobbiamo andare in giro con i portatili poiché troviamo nell'ambiente qualcosa che fa lo stesso servizio.

Un sistema distribuito pervasivo è un sistema distribuito con alcune caratteristiche in più:

- ci sono dei nodi non convenzionali: oggetti con o senza capacità di calcolo (smartphone, IOT, smart application...)
- adattività: abbiamo esteso un sistema per capire cosa succede in real-time, ma da alcuni di questi oggetti cerchiamo di capire cosa sta succedendo e attraverso l'applicazione cerchiamo di capire cosa è meglio fare in quel contesto ovvero in base ai vari dati che sono stati raccolti.




Chi progetta sistemi di pervasive computing deve risolvere tutte le problematiche dei sistemi distribuiti, le problematiche di mobile computing (mobilità dei nodi) e le problematiche tipiche del pervasive computing, come ad esempio smart spaces, invisibility.

Esempi di sistemi distribuiti pervasivi:

- sistemi di ambienti smart: ad esempio un insieme di sensori che ci permette di analizzare il traffico. Anche le macchine sono dei sensori: la macchina potrebbe sfruttare la fotografia per vedere quando un semaforo è verde, ma un'applicazione più efficiente potrebbe essere il fatto che il semaforo abbia un sistema integrato che permette di comunicare un segnale quando diventa verde con tutte le auto.
- e-health care system: esempio, sistemi in grado di fornire dati rispetto a determinate persone . Un altro impatto dei sistemi pervasivi sulla società riguarda la medicina

# Architetture

——— **lezione 2 - 3 marzo 2021** ———

## 3.1 Architettura di un sistema

Principalmente si parla di architetture software e hardware. Un'architettura di un sistema distribuito definisce:

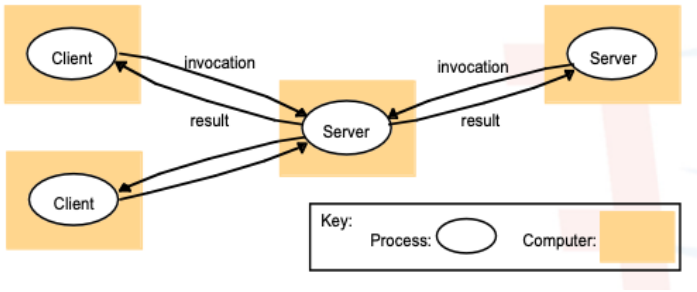
- le entità principali del sistema
- i pattern della comunicazione (client-server, event-bus o pub/sub, peer-to-peer)
- come comunicano (modi con i quali i processi in esecuzione su macchine disclocate in posti diversi comunicano)
- ruolo delle entità (client o server) e come possono evolvere (client diventa server e poi magari ritorna anche client)
- come le entità sono mappate a livello di infrastruttura fisica (potrebbero essere replicati)

## 3.2 Architetture centralizzate

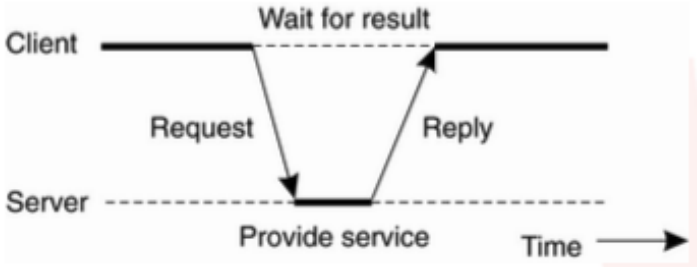
Le due architetture centralizzate più note sono:

- Client-server (basato su una richiesta e una risposta)
- Event bus (basato su eventi che utilizza publish subcribe)

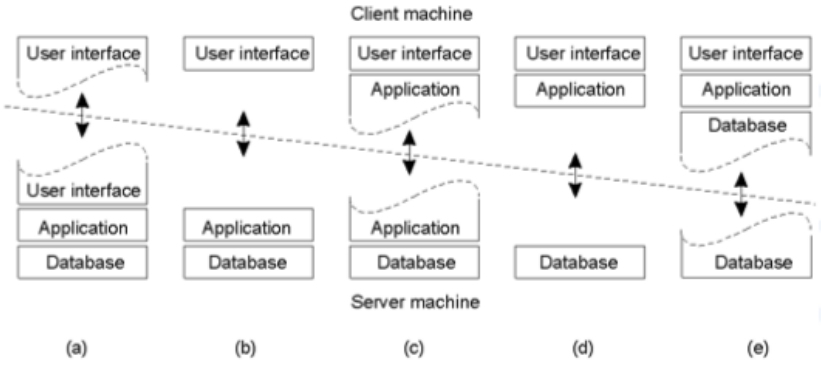
### 3.2.1 Client Server




Abbiamo dei processi sui nodi, i clients invocano una richiesta al server che deve gestire più richieste e deve rispondere a tutti quanti. In questo schema c'è un solo server, ma nella realtà potrebbero essercene più di uno.

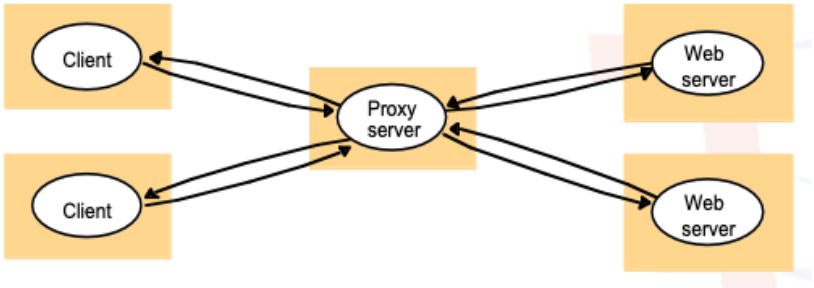



In questo grafico si ha sull'asse delle ascisse lo scorrere del tempo. La linea marcata rappresenta un processo in esecuzione, la linea tratteggiata indica un processo che o non è in esecuzione o sta facendo altro (ad esempio il server potrebbe servire un altro client e il client va in wait in attesa della risposta). L' inclinazione delle frecce di richiesta/risposta sono oblique perché c'è sempre un delay temporale di comunicazione da tenere in considerazione.

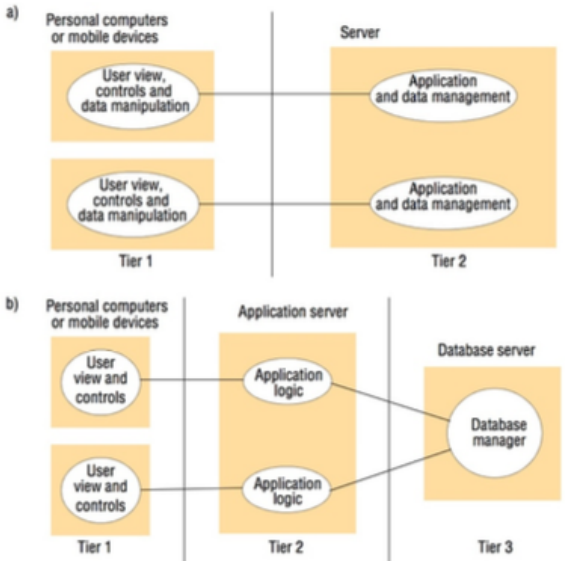



Cosa esegue il server e cosa il client?

Dobbiamo scegliere in fase di progettazione. Possiamo andare a caricare o scaricare il client e il server in base al tipo di componenti che stiamo utilizzando.




Possiamo fare uso di sistemi di caching ad esempio tramite l'utilizzo di proxy server che fanno anche questo servizio.
Il funzionamento è molto semplice: quando viene fatta una richiesta che era già stata fatta recentemente e viene valutato che la risposta alla richiesta è uguale a quella di prima (è ancora fresca) non si va a disturbare ancora il server ma si utilizza un proxy server che ha tenuto in cache la risposta data precedentemente.

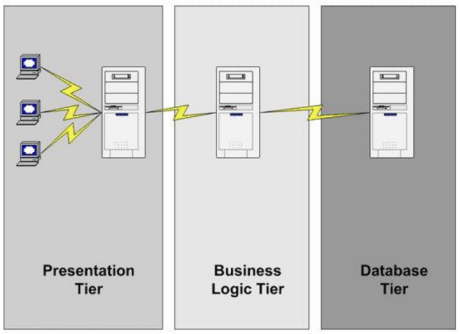



Esistono architetture client-server definite come multitier (multi livello). In queste si aggiunge la gestione di più livelli lato server. In un interazione con il client divido le funzionalità lato server in più livelli ad esempio application server e database server.
Quando parliamo di server multi livello non sempre intendiamo due macchine diverse, possiamo intendere anche due processi diversi sulla stessa macchina.

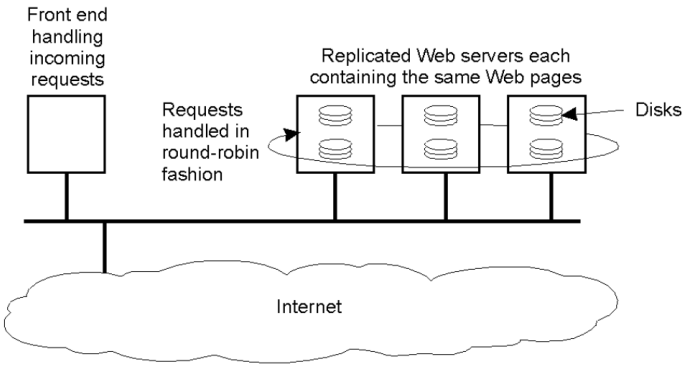
#### 3.2.1.1 Distribuzione

Esistono diversi tipi di distribuzioni che non sono altro che modi per distribuire il carico di lavoro e rendere efficienti i sistemi distribuiti. Le principali sono:

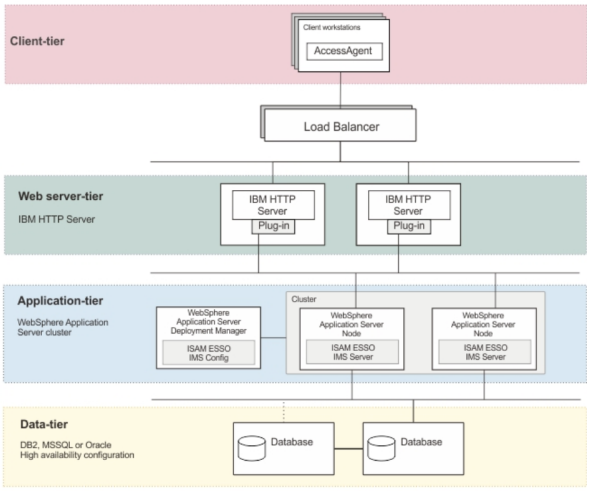
- distribuzione verticale: si ha un server (inteso non per forza come macchina fisica) differente per ogni funzionalità




- distribuzione orizzontale: si ha una stessa funzionalità distribuita (replicata) su nodi diversi

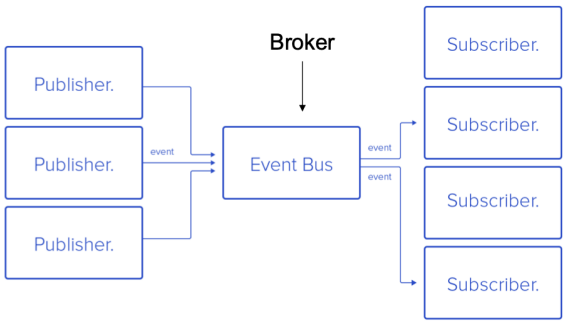



- distribuzione combinata: ogni funzionalità è duplicata su un gruppo di server. In questa distribuzione abbiamo il load balancer che si occupa di smistare il carico di lavoro.




### 3.2.2 Event Bus architecture

È un tipo di architettura molto utilizzato nell'ambito iot.



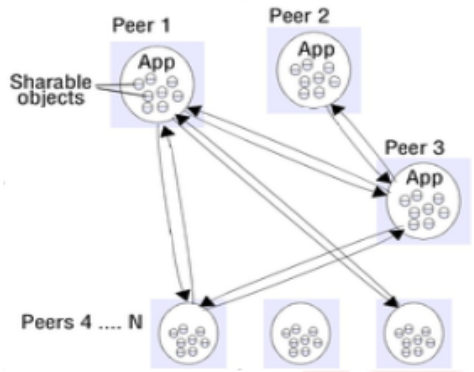

In questa architettura abbiamo tre diverse componenti:

- broker: è un componente software che gestisce l'event bus.
- publisher: client che comunicano che si è verificato un aggiornamento. Esempio: in questa stanza è stato registrato un cambio di temperatura di 1 grado
- subscriber: si "iscrive" a ricevere tutti gli aggiornamenti di uno o più publisher. Esempio: aggiornamenti sui cambi di temperatura.

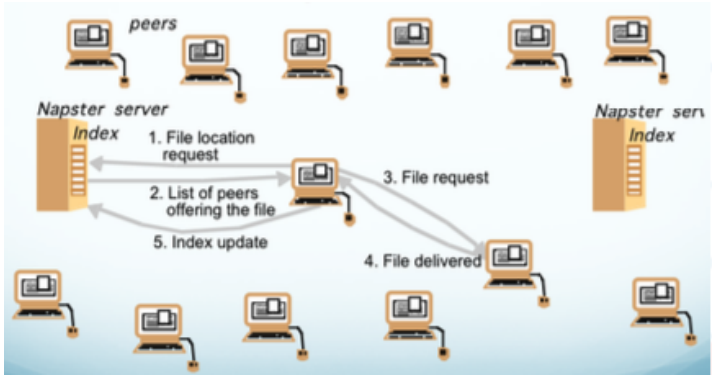
Il broker quindi ha il compito di comunicare ai subscriber interessati gli eventi registrati dai publisher.

## 3.3 Architetture decentralizzate

### 3.3.1 Peer-to-peer




Abbiamo diversi peer (nodi), dove ognuno ha le stesse capacità funzionali degli altri. È un'architettura disegnata in modo che ogni peer abbia risorse condivisibili. Non c'è il coordinatore (load balancer), ma i nodi sono tutti alla pari.




Un esempio di architettura P2P è quello di Napster che è stato il primo sistema che ha dimostrato utilità e efficacia di un sistema distribuito su larga scala per lo scambio di risorse che sono file. È nato più di vent'anni fa ed è stato chiuso per questioni di copyright.
Questo sistema non era peer to peer puro. Come possiamo vedere dallo schema c'è un napster server che mantiene un index. Tutti i nodi hanno sia funzionalità di client e di server ed eseguono tutti lo stesso codice, tranne appunto i server "direcory".
La gestione degli indici è centralizzata. Se un client vuole un certo file, manda richiesta al server iniziale che risponde con una lista di indirizzi di peer che dovrebbero avere quel file. Il peer a questo punto usa uno di questi indirizzi e contatta un altro peer, che si comporta da server. Se ha il file lo restituisce al client (quello che fa la richiesta) come risposta. Il client dice all'indice di aggiornarsi in modo che si tenga traccia che

anche lui adesso ha il file. Viene quindi reso disponibile il file su più punti della rete P2P.

Uno dei problemi principali delle architetture P2P è capire come distribuire le risorse. Gli obiettivi nel distribuire sono:

- raggiungere load balancing quando si accede ai dati, migliorare quindi le performance
- assicurare disponibilità, proteggersi quindi da eventuali guasti
- evitare troppi overhead

#### 3.3.1.1 Generazioni di sistemi P2P

- Prima generazione: primi sistemi di condivisione di file.
  Esempio: Napster.
- Seconda generazione: vengono migliorati i protocolli per aumentare scalabilità, fault tolerance e anonimità.
  Esempi:
  - Gnutella: tentativo di fare un sistema puro che funziona su larga scala, non usa directory service completi e che ha indici parziali e distribuiti.
  - FreeNet: aveva come obiettivo quello di fornire file sharing che garantisse anonimità.
  - bitTorrent: particolarmente efficiente per una caratteristica fondamentale: non viene scambiata la risorsa intera, ma vengono gestiti diverse porzioni di file.
- Terza generazione: viene introdotto il P2P Middleware, un sistema che è una specie di middleware per gestire in modo distribuito e decentralizzato delle risorse.

#### 3.3.1.2 Obiettivi del Middleware P2P

Deve abilitare i client a poter:

- localizzare e comunicare con una risorsa
- aggiungere e rimuovere risorse
- aggiungere e rimuovere peers

in modo completamente trasparente.
Per poter raggiungere questi obiettivi si costruiscono le overlay networks.

#### 3.3.1.3 Overlay Network

Rete logica (virtuale) costruita sopra la vera rete (fisica) che collega i peer.
Criteri di ottimizzazione: scalabilità, load balancing, far comunicare i peer vicini (minore latenza), sicurezza e anonimità. Questa rete logica consente di effettuare in modo efficiente le cose suddette. Sulla rete logica viene seguito il routing delle richieste (a livello applicazione) da client magari anche esterni alla rete agli host/nodi che contengono le risorse che ci interessano). Non è il routing del protocollo IP, è routing a livello di applicazione. È basato non su indirizzi IP, ma su identificatori globali. Le risorse hanno identificatori globali nel sistema distribuito. Il routing deve instradare verso uno dei peer che hanno una replica della risosa (quindi possono averla più di uno). I protocolli di routing overlay devono anche occuparsi di inserire e rimuovere nodi e oggetti. Questo algoritmo è installato su tutti i nodi del sistema.
Esistono essenzialmente due tipi di overlay network:

- strutturate: sono le più utilizzate oggi. Danno garanzie maggiori. In queste reti c'è un algoritmo deterministico con strutture dati particolari per cercare all'interno di questa rete logica (spesso viene gestita tramite tabelle di hashing).
- non strutturate: usano algoritmi probabilistici/randomizzati. Ogni peer non sa esattamente dov'è nella rete logica. Ha una vista parziali di alcuni altri peer che si evolve nel tempo. Siccome è parziale, nel momento in cui un client fa una richiesta a uno di questi peers, o ha la risosa, o chiede ai peer che conosce. Non c'è garanzia che la conosca. Viene richiesto man mano. Si cerca di ricostruire e aggiornare le partial view.

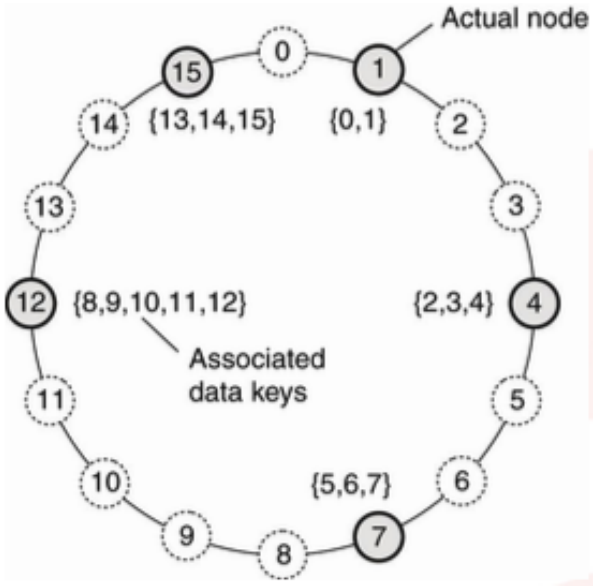
##### 3.3.1.3.1 Strutturate: chord

Parliamo di algoritmi abbastanza complicati. Articolo che spiega in dettaglio nel materiale.
È un sistema basato su DHT (distributed hash table), come la maggior parte di sistemi con overlay strutturate.

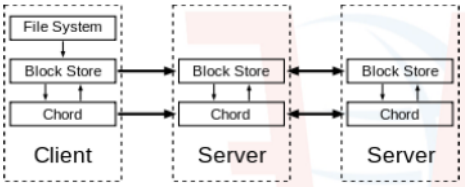
1. Per prima cosa si fissa uno spazio di indirizzamento, tipicamente fatto di 128-160bit (nell'esempio si usano 4 bit)
2. A ciascun nodo è associato come id un indirizzo nello spazio (ad esempio eseguendo l'hashing del suo IP): identifico un processo sulla macchina attraverso una hashing function che prende ip e porta e produce un'indirizzo in questo spazio. Gli actual

node corrispondono ad un processo sulla macchina che partecipa al sistema.

3. Ad ogni dato (risorsa) viene associata una chiave (indirizzo) nello stesso spazio. L'hashing della chiave in questo spazio darà un certo numero. Il discorso dell'hashing dà garanzia di distribuire in maniera uniforme nello spazio.
4. Un dato con chiave K viene mappato nel nodo con il più piccolo ID $\geq$ k. La funzione LOOKUP(k) o "successore", data la risorsa restituisce quale nodo la gestisce.
5. La cosa più difficile è fare efficientemente il LOOKUP(k). Una soluzione un po' furba è trovare un algoritmo che abbia complessità $\log n$ del numero dei nodi. Questo può essere fatto usando DHT, salvando in ogni nodo una finger table.



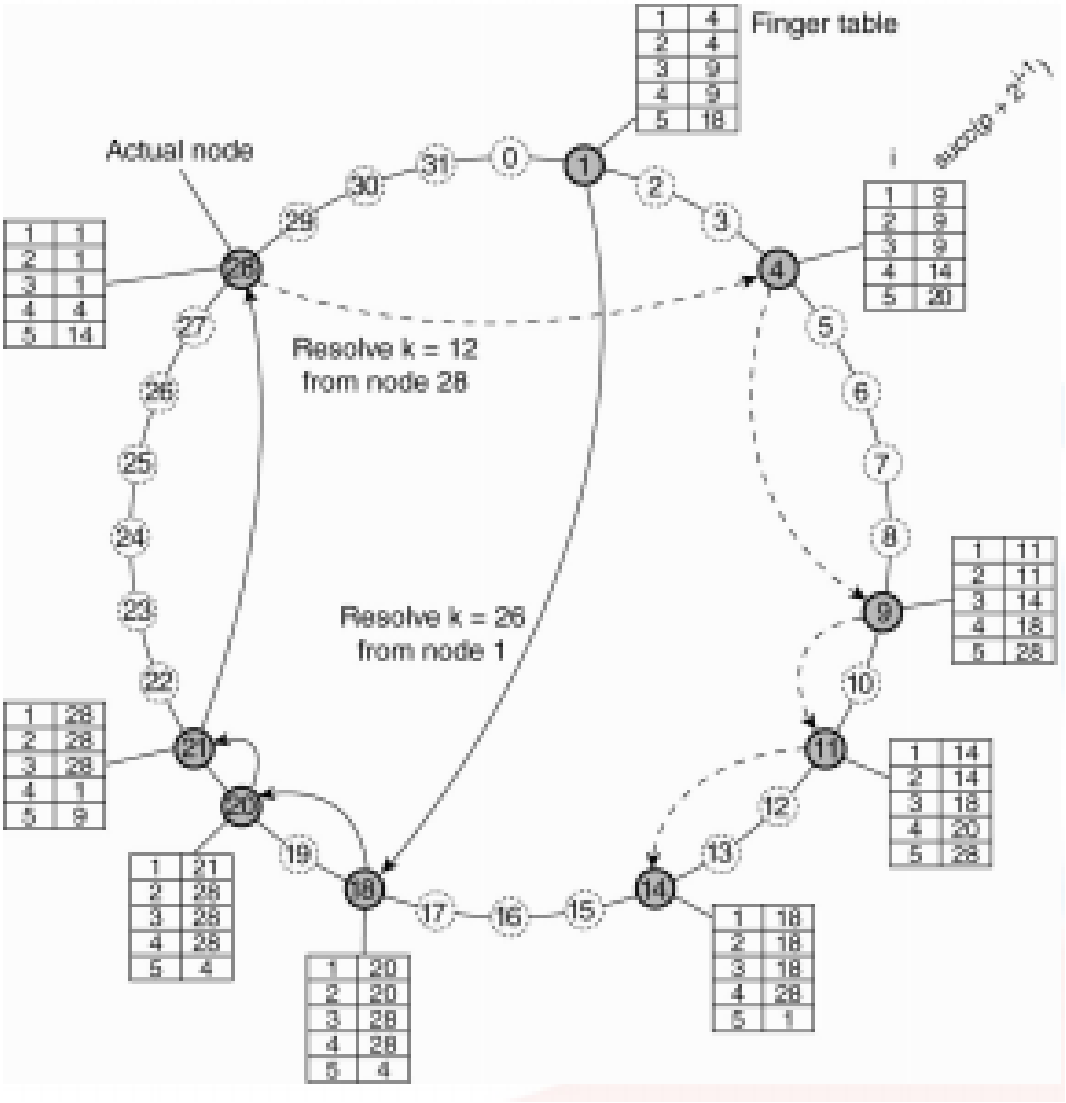

I nodi tratteggiati nell'immagine indicano che non c'è un nodo effettivo, sono disponibili nel caso in cui entrasse un nuovo computer.
Quando entra un nuovo nodo si fa riallocazione delle risorse e si gestiscono i puntatori, in modo tale che un altro non punti più al 12, ma al 10 ad esempio e che il 10 abbia un puntatore al 12. Devo quindi gestire anche i link logici. **Esempio di applicazione**



In questo esempio abbiamo una serie di server e un client che partecipa anche lui a chord. Hanno tutti un livello di middleware chord.
A livello del client abbiamo un'interfaccia al file system.
Il FS in realtà traduce le richieste (le mappa) in operazioni a basso livello sui blocchi (che sono distribuiti su tutte le macchine partecipanti).
Chord è usata per eseguire operazioni sui blocchi, per bilanciare la distribuzione dei blocchi e per trovare il server dove il blocco che serve è memorizzato.
LOOKUP è l'operazione che chord deve fare quando il FS chiede di accedere un blocco e occorre capire dov'è.

#### 3.3.1.3.2 Distributed hash table

Ciascuno dei nodi ha in esecuzione l'algoritmo che stiamo discutendo. Ciascuno ha anche una struttura dati "finger table" che è la tabella hash. Tante piccole parti di hash table sono distribuite nell'anello.
L'obbiettivo è rendere più efficiente la ricerca di una risorsa.

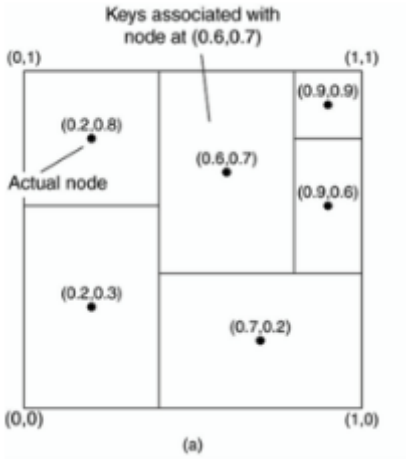



Ogni tabella ha 5 righe in quanto lo spazio di indirizzamento (nell'esempio) è di 5 bit. Nella prima colonna, ogni riga fa riferimento al bit in questione, dunque da 1 a 5. Il valore della seconda colonna è dato dalla formula $succ(p+2^{(i-1)})$, dove i è il valore della prima colonna e p è l'id del nodo.
Esempio: un client si collega al nodo 1 e chiede la risorsa 26. Viene guardata dunque la finger table del nodo 1 alla ricerca dell'elemento j (nella colonna di destra) tale per cui k (26) è $\geq$, ma $\leq$ del successivo (j+1). In questo caso il valore k (26) è maggiore di tutti i valori della finger table della colonna di destra e quindi viene presa in considerazione l'ultima riga della tabella. A questo punto viene eseguito un salto al nodo 18. Si esegue ancora la ricerca e in questa finger table si ferma alla seconda riga in quanto nella terza riga il valore della colonna di destra è $>$ di k(26). Viene dunque eseguito un altro salto

verso il nodo 20. Si esegue la stessa procedura e si salta al nodo 21. A questo punto, dato che già nella prima riga il valore è > di k (26), si evince subito che la risorsa che si sta cercando risiede nel successore del nodo 21. Dunque la risorsa che si sta cercando (26) è contenuta nel nodo 28.

#### 3.3.1.3.3 Strutturate: CAN (Content Addressable Network)

Anche questo è basato su DHT (distributed hash table). Utilizza uno spazio cartesiamo d-dimensionale.




Questo spazio viene partizionato in nodi. In ogni rettangolo ci possono essere più punti. I punti sono associati ad un processo/nodo. I punti che si trovano al centro dell'area gestiscono tutte le risorse che verranno allocate, con delle funzioni che quindi mapperanno le risorse in uno spazio n-dimensionale, nell'area.
Quando arriva un altro nodo e viene allocato in un'area, quell'area viene divisa in due e le risorse di ciascun area sono associate ai rispettivi nodi.

#### 3.3.1.3.4 Limitazioni delle overlay strutturate

Queste finger table vanno manutenute, con anche gli indirizzi e i puntatori. C'è un costo di overhead nel manutenere queste strutture.
Inoltre c'è il problema di organizzazione dei puntatori per essere resistente ai fallimento, occorre capire se un nodo è ancora online, posso fare un timeout, però questo non è banale, il problema potrebbe essere irrisolvibile.

#### 3.3.1.3.5 Non strutturate

Hanno una struttura più semplice, senza DHT e senza manutenzione.
Si costruisce la rete con algoritmi randomizzati dove ogni nodo ha una vista parziale.
Ogni tanto i nodi vicini si scambiano informazioni sui nodi nuovi che hanno conosciuto,

detti anche protocolli di gossiping.
Questo modo di scambiarsi i gossip ha delle componenti randomiche: si sceglie solo una percentuale a caso della view conosciuta da ciascuno.
C'è anche algoritmo di distribuzione delle risorse e anch'esso ha componente randomiche. Possono essere diverse strategie. La ricerca non può dare garanzie fisse (upper bound, che trovi la risorsa, che ritorni più volte la stessa risorsa). L'effort di manutenzione è inferiore e migliora la scalabilità. La ricerca è limitata. Chiedono ai vicini per un numero limitato di hop o per un timeout temporale.

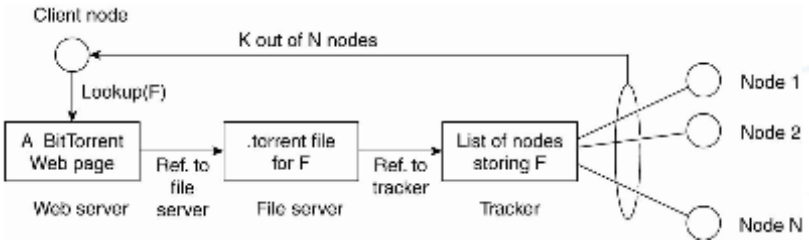
**3.3.1.3.6 Non strutturate: peer sampling** La conoscenza locale dell'overlay è aggiornata basandosi sul servizio di peer sampling: i nodi si scambiano periodicamente le loro viste random e aggiornano le loro viste creando un nuovo campione random. Queste viste random definiscono un overlay network approssimativa

| | Strutturate | Non Strutturate |
|---|---|---|
| Vantaggi | Garantisce di trovare gli oggetti (assumendo che esistano) e può offrire un margine a tempo e complessità a questa operazione. Overhead dei messaggi relativamente basso | Si auto-organizzano e sono resilienti dal fallimento dei nodi |
| Svantaggi | Ha bisogno di manutenzioni frequenti se la struttura è complessa, il che può essere costoso e difficile da ottenere, specialmente in ambienti molto dinamici | Non possono offrire garanzie di trovare l'oggetto. Si raggiunge un eccessivo overhead di messaggi che può influire sulla scalabilità |

**3.3.1.3.7 P2P: Strutturate vs Non strutturate** Ci sono anche soluzioni miste, ad esempio possiamo avere reti private/locali che funzionano in un certo modo con i regular peer e poi c'è un punto superpeer che a sua volta partecipa ad una rete a cui non partecipano i suoi sotto-peer.

## 3.4 Architetture ibride

BitTorrent è una rete peer to peer? Sì. È pura secondo definizione? Non proprio.

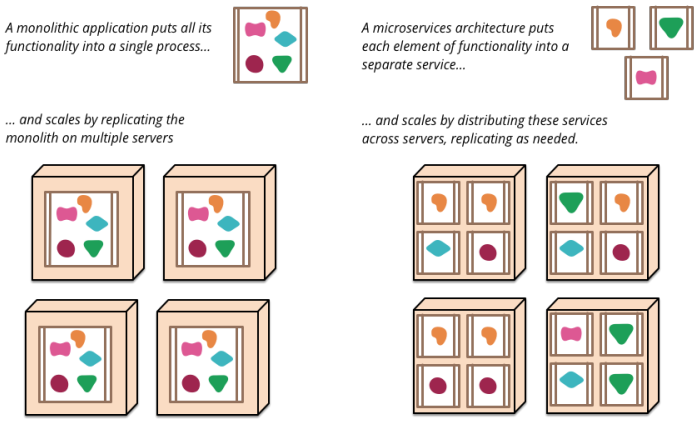



Spesso c'è una componente iniziale che aiuta il resto del sistema a lavorare in modo centralizzato. Un client che vuole accedere ad una risorsa, all'inizio fa un lookup di f non direttamente ai nodi, ma a un webserver per trovare un .torrent, che è un file che non contiene il file, ma un riferimento ai tracker.
Il tracker è un nodo che coordina la distribuzione del file.
Quindi abbiamo all'inizio un'architettura client/server che va a identificare un riferimento, che porta ad un tracker e poi inizia la parte P2P vera e propria.

## 3.5 Microservizi, Container e Servizi Cloud per SD

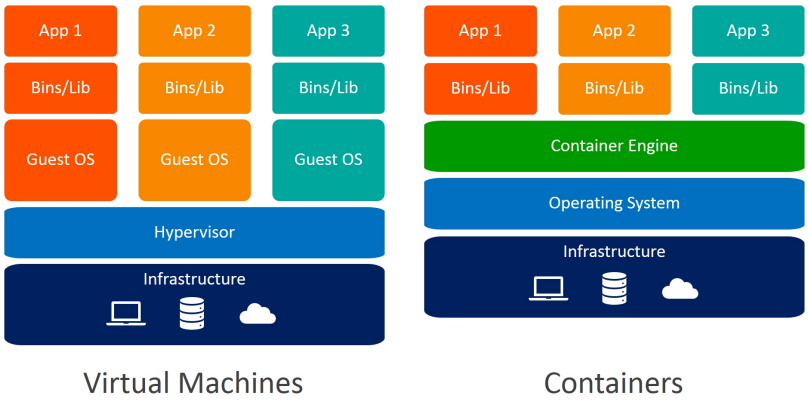
### 3.5.1 Microservizi

Possono essere visti come una specie di raffinamento della distribuzione verticale: si cerca di andare a raffinare ancora di più anche all'interno dell'applicazione, isolando le singole funzionalità e rendendole indipendenti dalle altre. Questi microservizi devono interagire tra loro comunicando attraverso meccanismi leggeri, altrimenti l'overhead vanifica l'idea di separazione delle funzionalità per migliroare qualità.
Quindi si usano API REST o RPC.
I microservizi possono essere scritti in linguaggi di programmazione diversi, a seconda che una funzionalità sia meglio in un linguaggi piuttosto che un altro.
L'architettura a microservizi ha avuto successo perché permette di distribuire anche singoli servizi su macchine diverse, piuttosto che replicare tutto, a seconda delle necessità.
Potrei aver bisogno di replicare di più un componente piuttosto che un altro e così lo posso fare in modo più agile.




### 3.5.2 Containers

Sono un'astrazione a livello applicativo nelle quali l'idea è mettere insieme codice e dipendenze che il codice ha. L'obiettivo è quindi ospitare il codice dei microservizi e associare al microservizio anche le dipendenze di cui necessita, cercando di aumentare la portabilità e riducendo i problemi durante la fase di deployment.
Il container è l'analogo della VM. La VM è l'astrazione di una macchina fisica, mentre il container è l'astrazione a livello di una applicazione (tendenzialmente a livello di un microservizio).




In ambito cloud è utilizzato l'approccio microservizi-container per distribuire su hw eterogeneo e distribuito in rete le funzionalità che vengono offerte come servizi cloud o SaaS.

### 3.5.3 Servizi Cloud per SD

Sono insiemi di servizi costruiti su sistemi distribuiti e vengono applicati in:

- Calcolo e comunicazione
- storage e DBMS
- identità e sicurezza
- management

Offrono diversi tipi di trasparenza.
I principali esponenti in questo campo sono:

- Amazon AWS
- Google Cloud Platform
- Microsoft Azure

Alcuni esempi potrebbero essere:

- Storage
  - Google Distributed File System (GFS, Colossus)
  - Distributed RDBMS/NoSQL DBMS (Spanner)
- Comunicazione
  - Google PUB/SUB
- Computazione e analisi di dati
  - Google DataFlow

Una delle problematiche del cloud è che è difficile fare il porting, appoggiandosi a diverse soluzioni specifiche di un fornitore di cloud.
Se usando i cloud si usano microservizi e container, questo favorisce la portabilità.
Quando si vuole coordinare comunicazioni e gestione di microservizi su container in un architettura a cluster diventa utile Kubernetes (evoluzione di Borg).

# Comunicazione

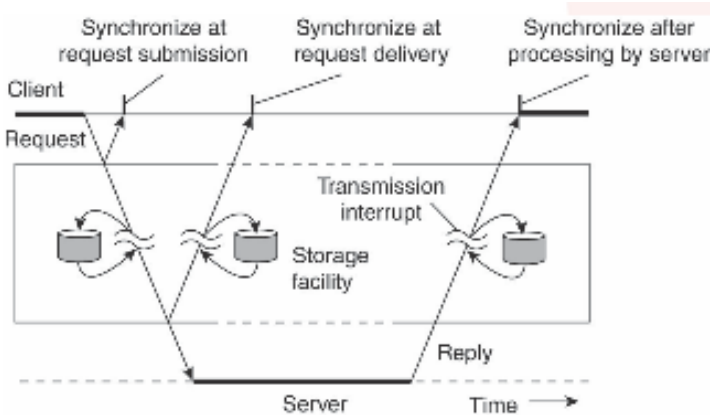
I sistemi distribuiti funzionano tramite lo scambio di messaggi tra processi situati su nodi diversi, eterogenei e distribuiti geograficamente.
Quali sono i modelli di comunicazione?

## 4.1 Middleware Protocols

Il middleware è un livello che fonde sessione e presentazione della pila ISO/OSI e contiene i servizi di comunicazione di alto livello: socket, remote procedure call, ecc.
Nel middleware ci sono sistemi di protocolli che definiscono servizi utili per un ampio spettro di applicazioni (authentication, locking, commit distribuito ad esempio).

## 4.2 Tipi di comunicazione

Adottiamo modello client server, dove anche il client può fungere da server.



Analizziamo la foto: da sinistra a destra scorre il tempo.
Il client è in esecuzione, fa una richiesta al server, con una certa latenza (rappresentata dall'inclinazione della freccia). È possibile che questa richiesta parta e attraversi un sistema che fornisce memorizzazione (es. mando SMS e questo viene memorizzato in server SMS e se il server non è disponibile, lo riceve dopo).
Il client potrebbe chiedere una sincronizzazione a livello di invio della richiesta, sincronizzazione a ricezione della richiesta, oppure sincronizzazione che il server ha effettivamente processato la richiesta.
Ci sono quindi vari tipi di sincronizzazione con il server.

## 4.3 Persistenza e sincronia nella comunicazione

Insieme di post office nei quali la posta viene trasferita tra i vari post office e se il destinatario non è disponibile riesce comunque a rimediare le sue lettere grazie al post office di riferimento.
Viene introdotto così un sistema di memorizzazione intermedio.
La comunicazione può essere anche transiente ossia in completa assenza di metodi per memorizzare messaggi in transito (una telefonata, ad esempio, senza segreteria telefonica)

Abbiamo diverse combinazioni di queste dimensioni:

- Persistente asincrono: processo A (in esecuzione) manda un messaggio al processo B (non in esecuzione), c'è un metodo di memorizzazione intermedio che salva il messaggio. A non aspetta nessun ack da B, e va avanti a lavorare.
- Persistente sincrono: processo A manda messaggio, B manda conferma (anche se non era in esecuzione), A attende conferma.
- Transiente asincrono: A manda messaggio e continua l'esecuzione, il messaggio può essere inviato solo se B è in esecuzione.
- Transiente sincrono:
  - Receipt-based: A manda messaggio e si ferma (aspetta conferma), B è in esecuzione ma sta facendo altro, B comunica che l'ha ricevuto (anche se non lo sta elaborando)
  - Delivery-based: si differenzia dal precedente nel momento di sincronizzazione. Tiene A fermo fino a quando non comncia a dare effettivamente attenzione la richiesta.

– Response-based: è il più comune. Non c'è meccanismo di persistenza. B deve essere attivo. A si ferma finché non riceve il risultato di elaborazione della richiesta.

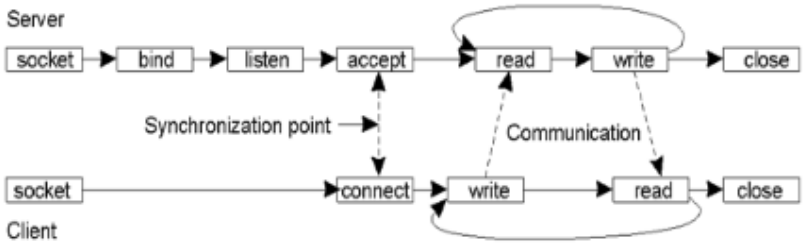
## 4.4 Communication middleware

### 4.4.1 Message oriented transient communication

Socket introdotte con la versione Berkley di Unix.
È un metodo di comunicazione che si appoggia al livello di trasporto (sta sopra). Ha avuto un grandissimo successo ed è passato su tutte le piattaforme.
Oggi sono molto diffuse le web socket, ma tutto deriva da questo modello di base.

Con socket si intende un estremo del canale di comunicazione (ci sarà una socket per il client e una per il server).
Scrivendo programmi con linguaggi a basso livello (C) abbiamo a disposizione diverse primitive:

| Primitiva | Significato |
|---|---|
| Socket | Crea un nuovo endpoint di comunicazione, a livello di s.o. crea uno spazio di memoria e ci associa un numero |
| Bind | Associa una porta del nodo locale ad una socket |
| Listen | Comando che esegue solo il server: annuncia la dipsponibilità di accettare connessioni |
| Accept | Comando che esegue solo il server: blocca il chiamante fino a quando non arriva una richiesta di connessione |
| Connect | Comando che esegue solo il client: avrà come argomenti ip e porta, che corrispondono all'ip del nodo dove sta il server e la porta della socket del server. |
| Send | Invio dati nel canale. |
| Receive | Ricevo dati dal canale. |
| Close | Comando da non dimenticare: rilascia le risorse allocate dal sistema operativo (file descriptor, aree di memoria). |



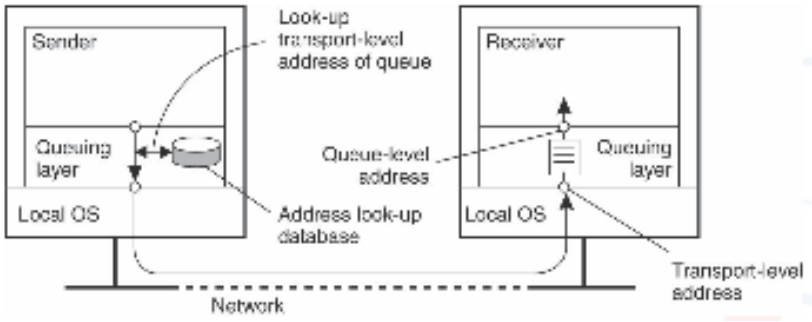

Come possiamo vedere dallo schema sia client che server creano una socket.
La bind lato client viene fatta in modo implicito dal sistema operativo.
Il server fa esplicitamente la bind perché si potrebbe voler usare una porta specifica (mettendo 0 viene scelta una porta qualsiasi, la prima disponibile).
La listen dice al sistema operativo di riservare un'area di memoria in cui posso accodare le richieste, poiché non voglio che queste vengano rifiutate se il server è già impegnato. Gli viene data anche una dimensione di quest'area (quante richieste accodare) e se arrivano altre richieste oltre a quel numero, vengono rifiutate.
La accept prende il primo elemento (FIFO) nella coda della listen e stabilisce la connessione con questo. La accept restituisce un altro socket descriptor (socket di servizio).
Questo server viene implementato come multi-thread, quindi la socket di servizio sarà dedicata alla comunicazione con il client da parte di un thread e la socket principale viene lasciata a gestire le altre richieste.
Dopodiché c'è un ciclo di read e write, come se scrivesse su un file.
Alla fine della comunicazione sarebbe meglio rilasciare le risorse esplicitamente, altrimenti ci pensa sistema quando viene chiuso il processo.

Il modello delle socket è sincrono.
Il server esegue una accept per accettare le connessioni da parte del client che rimane bloccato fino a quando non arriva una connessione.
Il client che esegue una connect deve trovare dall'altra parte il server pronto a rispondere sulla porta sulla quale sta richiedendo la connessione.
Se la connect non trova il server c'è un timeout per cui dopo un tot dà errore. Il client rimane bloccato finché non riesce a fare questa operazione.

### 4.4.2 Message Oriented Persistend Communication: Sistemi di code

Modello di comunicazione persistente, sempre message oriented.
I sistemi che più vengono utilizzati in questi caso sono i sistemi a code. Si hanno due nodi, un sender e un receiver. Ognuno ha un proprio OS, una connessione di rete, con

tutti gli stack di protocolli. A livello middleware c'è uno strato, il queuing layer, dove vengono gestite delle code.
Dal lato local OS c'è un punto di comunicazione nel server e nel receiver che sono un punto di accesso alle code e non più alla socket del processo.
Da parte del codice del server e del receiver il riferimento per inviare e ricevere i messaggi è un punto d'entrata al middleware delle code.

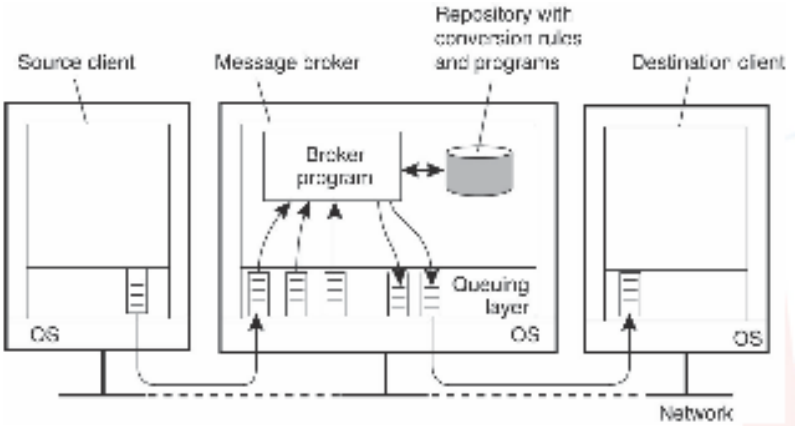



Spesso questo sistema delle code, per implementare la persistenza in modo più affidabile, fa riferimento a nodi intermedi identificati come routers. In pratica viene utilizzato il sistema visto per il pony express: una serie di "uffici postali" intermedi che si occupano dell'instradamento e della gestione dei messaggi per consegnarli al ricevente finale che potrebbe non essere in esecuzione. Questo garantisce persistenza.
Il sender manda un messaggio a una coda che fa riferimento al sistema di routing intermedio e arriverà all'applicazione ricevente tramite una coda alla quale saranno recapitati i messaggi.

| Primitiva | Significato |
|---|---|
| Put | esegue l'append di un messaggio a una coda specifica |
| Get | preleva i messaggi da una coda. È una primitiva bloccante: si blocca fin quando trova un elemento. Se c'è più di un elemento utilizza una politica FIFO |
| Poll | è una get non bloccante. Preleva il primo messaggio presente senza mai bloccare l'esecuzione |
| Notify | installa un handler che può essere chiamato quando un messaggio viene messo nella coda specificata. (Una specie di trigger che può servire per fare l'opposto della poll dall'altra parte) |

#### 4.4.2.1 Message Brokers

Il ruolo dei broker in un message queue system non è soltanto quello di garantire la persistenza nel caso in cui i nodi riceventi siano offline, oppure i processi non siano disponibili, ma potrebbe dover fare delle traduzioni/conversioni.




I nodi in un sistema distribuito possono essere eterogenei e potrebbe essere utile inserire in questo middleware anche un metodo per far parlare nodi che hanno convenzioni diverse.
Il broker oltre a gestire le code per ingresso/uscita di messaggi, potrebbe avere accesso a una repository con delle regole di conversione (traduzioni) da utilizzare per tradurre i messaggi nel caso in cui i due nodi utilizzino standard diversi.

**Quando conviene usare i sistemi a code?**
Conviene usare un sistema a code quando:

- si deve implementare comunicazione asincrona con persistenza
- quando la scalabilità può aumentare ammettendo ritardi in risposte e richieste
- quando il producer è più veloce del consumer
- quando si implementa il pattern di comunicazione publish subscribe

#### 4.4.2.2 Protocolli di messaging

I protocolli a messaggi più comuni sono:

- **XMPP**: utilizzato in precedenza da Google Talk e FB fino al 2014. È basato su XML e permette di strutturare con i tag i contenuti dei messaggi. Supportava la presence information, cioè capire quando un utente è collegato e dare un ack a

chi deve comunicare per vedere chi è presente o no, manutenzione di contact list, ecc. Era aperto.

- **MQTT**: sviluppato da IBM con l'intenzione di creare qualcosa di leggero. L'obiettivo finale era quello di riuscire a far utilizzare questo protocollo anche a client estremamente thin, quindi anche dispositivi IoT. Questi protocolli lavorano sopra TPC/IP. Ogni client ha un ID univoco. MQTT in generale non garantisce persistenza. È stato creato per il modello publish subscribe. Il broker gestisce la coda. Ci sono altri client che si registrano al broker dicendo che vogliono i dati che vengono pubblicati su una certa coda. Usato da FB Messenger. Utilizza lo standard OASIS ed è uno dei principali candidati per comunicazione machine to machine in ambito IOT.
- **AMPQP**: utilizza lo standard ISO. Perfetto per un'ampia gamma di infrastrutture middleware di messaggistica e anche per il trasferimento dei dati peer-to-peer.

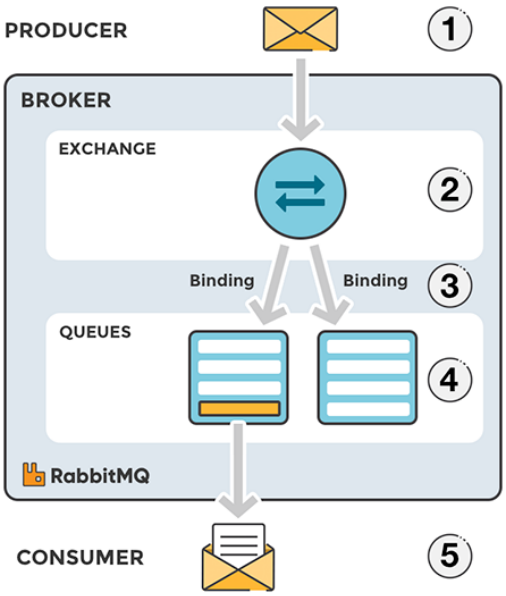
#### 4.4.2.3 Message broker software

I principali software dei message brokers sono:

- Eclipse Mosquitto: supporta MQTT, è leggero e opensource
- RabbitMQ: molto popolare, opensource e supporta AMQP e altri protocolli
- Apache ActiveMQ and Kafka
- Google Cloud Pub/Sub
- Amazon MQ

**4.4.2.3.1 RabbitMQ** È un'implementazione multi-linguaggio dell'AMQP, ma supporta plugin per MQTT ed altro protocolli. Consiste in:

- un broker (distribuito in un cluster o in un insieme di cluster), scritto in Erlang
- client disponibili per vari linguaggi e i client comunicano con il broker utilizzando AMQP




Il message flow classico è: il producer (publisher) manda i dati che produce al broker (nella parte di gestione dei producer), in cui viene rediretto il messaggio a determinate code e dalle code viene instradato verso i consumer (subscriber).
RabbitMQ ha molte funzionalità: per esempio supporta la modalità diretta, cioè il messaggio contiene una chiave/stringa che lo direziona verso una sola coda specifica alla quale c'è un consumer specifico. Questo è un modo per rendere asincrona e persistente una comunicazione 1:1 come quella client-server.
Oppure c'è la funzione opposta in cui il messaggio viene mandato a tutti.
La modalità più usata, anche in MQTT, è invece che il producer pubblica il messaggio specificando un "topic". Ci sono code dedicate a questi "topic" e serve un sistema estremamente efficiente per gestire tutti i subscriber che si iscrivono ad un determinato "topic".
Il passaggio dei dati ai consumer (subscriber) funziona in questo modo: il subscriber si registra al sistema (al broker), ottiene un identificatore e stabilisce una connessione. Nel registrarsi al broker, il processo consumer che gira su un nodo stabilisce una connessione con il broker (esattamente come per la socket) e questa connessione viene poi utilizzata per mandargli i messaggi di sua competenza.
Nel caso in cui il consumer vada offline, chiuda la connessione o ci sia un problema di rete, il sistema deve prevedere dei modi affinché se riprende la connessione e il consumer è registrato già con un certo ID, viene riconosciuto e ricomincia la trasmissione su un nuovo canale.

## 4.5 Remote Procedure Call

Le Remote Procedure Call sono un metodo per introdurre trasparenza di accesso rispetto alla comunicazione via messaggi. L'obiettivo è avere un modo per scrivere programmi (come siamo abituati a scriverli) sul sistema distribuito, senza esplicitare canali tra processi.
Vogliamo comunicare con altri perchè magari alcune operazioni non le sa fare il mio nodo, ma uno più potente sì. RPC nasconde la messaggistica che è necessaria per simulare una chiamata a procedura remota.

### 4.5.1 Chiamata a procedura convenzionale

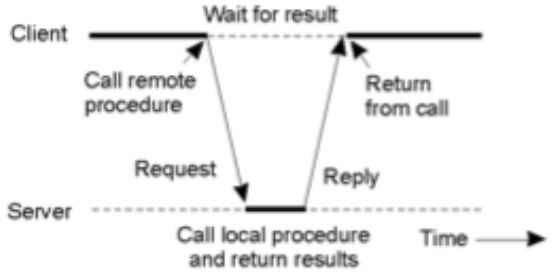
Abbiamo una procedura read(fd, buf, nbytes). Durante la sua esecuzione si popola lo stack mano a mano che si eseguono le istruzioni.
Ad una procedura locale possiamo passare i parametri per:

- valore
- riferimento

Ovviamente non possiamo usare il passaggio per riferimento per una RPC, perchè l'indirizzo di memoria sulla mia macchina sarà diverso rispetto alla macchina remota. Si deve serializzare e passare per valore.
Se ho una grossa mole di dati, questo è molto costoso.

### 4.5.2 Client e Server Stubs

Lato client è come se chiamassi una procedura locale, il middleware si occuperà di tradurre tutto in una richiesta, in una chiamata alla procedura sul server, e alla restituzione del risultato.



### 4.5.3 Passi di una RPC

Sul client abbiamo le varie istruzioni e ad un certo punto nel codice c'è una chiamata ad una funzione che vogliamo far eseguire sul server.
Qui interviene un pezzo di codice che è stato automaticamente generato che non è visto dal programmatore.
All'interno di questa porzione di codice abbiamo la serializzazione dei parametri e l'invio di questi come messaggio (costruito dallo stub).
Dall'altra parte del canale di trasmissione c'è la risalita di tutti i livelli della pila ISO/OSI fino ad arrivare al processo che si è reso disponibile per eseguire quella funzione.
Il server ha un codice che implementa la funzione richiesta, che un programmatore ha scritto, non per rispondere a richieste di altri dall'esterno, ma per eseguire questa funziona in locale.
C'è però come nel client una porzione di codice stub che fa il lavoro di spacchettare il messaggio, costruire la struttura dati, identificare la funzione e fare la chiamata alla funzione come se fosse una chiamata locale.
Il risultato verrà preso dallo stub, messo in un messaggio, e inviato dall'altra parte.
Il client ricostruisce il risultato e riprende la sua esecuzione.
Le stub del client e quelle del server possono essere scritte in linguaggi diversi.

### 4.5.4 Marshalling e Unmarshalling

Sono due procedure, una opposta all'altra:

- Marshalling: formattazione dei parametri di una RPC in un messaggio
- Unmarshalling: estrarre i parametri dal messaggio

Il Marshalling richiede la serializzazione degli oggetti o delle strutture dati (codifica in sequenza di byte che può essere inviata sul canale).
La serializzazione può essere effettuata in binario (come nelle Java RMI o in gRPC) o in stringhe (ad esempio REST WS in JSON).

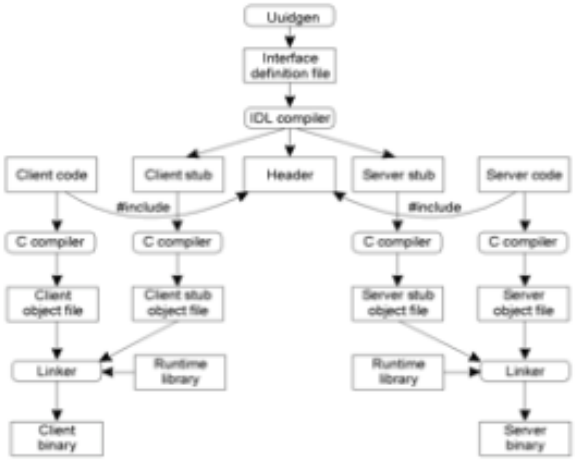
*Se sono in un sistema eterogeneo i nodi interpretano le cose in modo diverso.*

### 4.5.5 RPC sincrone e asincrone

Tipicamente RPC è sincrona. È possibile però chiedere anche una chiamata asincrona, eventualmente, con dei parametri.
C'è la possibilità di ricevere ack a chiamata ricevuta, oppure con la modalità deferred

synchronous, si aspetta un tot fino all'accettazione della richiesta, ma non si aspetta il risultato e si continua a lavorare.
One-way RPC non chiede di eseguire niente al client e il client semplicemente manda un ack. (modalità quasi asincrona)

### 4.5.6 Scrivere un client e un server




Il server per pubblicare metodi remoti deve scrivere una interfaccia in IDL. Caratteristiche di una IDL (deve essere scritta in un linguaggio neutro - esistono piu' compilatori in base al linguaggio che si vuole ottenere, questi compilatori, dato l'IDL creano automaticamente Server e Client Stub):

- Come si chiama il metodo
- Parametri e tipo del metodo

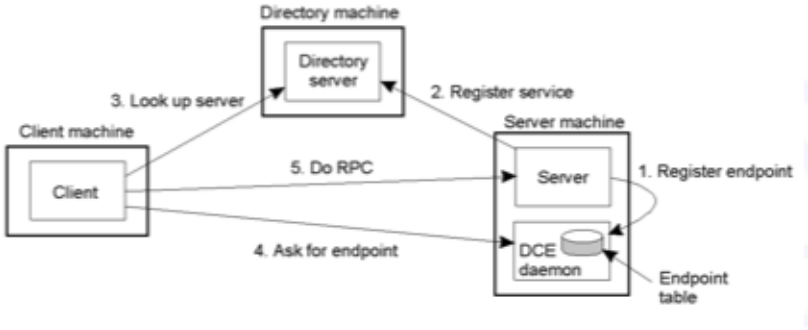
#### 4.5.6.1 gRPC

gRPC è un'implementazione di Google del protocollo RPC. È un framework open source, usato largamente per la sua caratteristica di avere una bassa latenza e un'alta scalabilità all'interno dei sistemi distribuiti. Supporta molti linguaggi di programmazione come C,C++, Java, Python, Go ...
In gRPC viene usato ProtoBuf per la comunicazione.

### 4.5.7 Associare un client a un server

In un sistema distribuito sarebbe opportuno anche avere trasparenza rispetto alla location.

In Corba c'è un modo per scoprire dinamicamente chi è disposto a offrire certe funzioni. Dal punto di vista del programmatore è possibile interrogare un server directory e avere in restituzione l'indirizzo di un nodo che in quel momento è disponibile per quella funzione.




Lato server machine viene comunicato a un processo demone a quale porta si risponde e viene pubblicato il servizio sul directory server.
Il client fa una ricerca di chi offre la funzionalità attraverso il directory server che gli fornisce la porta e l'indirizzo ip del demone sul nodo (non indirizzo del server specifico).
Successivamente il client chiede al demone sulla server machine l'endpoint della server stub.
Il demone guarda se sulla sua macchina qualcuno offre quella funzione, verifica se è il servizio è attivo e se non lo è, lo lancia.
A questo punto il servizio è attivo e gli restituisce la porta.
Finalmente il client può eseguire una RPC.

# Sincronizzazione

Non è sufficiente far comunicare i processi in un sistema distribuito per realizzare le funzionalità del sistema. Ci serve praticamente sempre una qualche forma di coordinamento tra le computazioni che ci sono nei vari nodi e quindi coordinare anche le comunicazioni che occorrono per raggiungere gli obbiettivi.

Gli algoritmi di sincronizzazione prevedono uno scambio di messaggi (solo in questo modo i nodi possono parlare e organizzare il loro coordinamento).

## 5.1 Clock Synchronization

All'interno di un sistema distribuito, ogni nodo è indipendente ed ha quindi un proprio clock interno. Questo clock non è perfetto. Quando ogni macchina ha il proprio clock, un evento che si è verificato dopo un altro evento può comunque essere assegnato ad un timestamp precedente, dunque i sistemi non possono semplicemente basarsi sul confronto dei timestamp.

## 5.2 Physical Clocks

Non è possibile sincronizzare perfettamente gli orologi, poiché ci sono i tempi di latenza della rete. Possono, però, essere sincronizzati con un certo livello di approssimazione rispetto al tempo preciso. Il riferimento principale è l'UTC.

Il modo in cui noi misuriamo il tempo deriva dall'osservazione di fenomeni astronomici. Di fatto, il tempo è la distanza tra due eventi. Gli orologi comuni dividono il giorno solare medio basandosi sull'oscillazione dei cristalli di quarzo. L'orologio atomico, invece, misura il tempo in termini di transizioni di un atomo di Cesio 133. Dal giorno solare medio viene derivato il secondo solare medio, che corrisponde a circa 9 miliardi di transizioni di Cesio.

Una serie di istituti per la misurazione del tempo, che hanno degli orologi atomici, si accordano tra di loro, facendo una media, dopodiché delle stazioni si occupano di distribuire questo tempo quasi perfetto (TAI). Il TAI non è comunque perfettamente preciso, poiché l'universo non è regolare (la rotazione della Terra diminuisce, i fenomeni astronomici mutano, etc). Dunque, questi istituti, quando si accorgono che ci si sta discostando dall'osservazione astronomica, aggiungono dei secondi chiamati leap seconds che non sono periodici. Questo di solito viene fatto a capodanno.

UTC = TAI + leap seconds

L'ideale è che i nodi di un sistema distribuito corrispondano tutti all'UTC, ma questo non è possibile. Nella realtà, tutti gli orologi saranno più lenti o più veloci, quindi dovranno essere sincronizzati più o meno frequentemente in base a quanto divergono dal perfect clock.

## 5.3 Synchronizing Physical Clocks

### 5.3.1 GNSS

È possibile utilizzare i GNSS per ottenere il tempo, poiché i satelliti dei sistemi GNSS hanno a bordo degli orologi atomici. Ogni satellite di un GNSS fa un broadcast di messaggi che contengono il timestamp dell'orologio atomico (nel momento in cui viene inviato il messaggio), oltre all'identificatore del satellite stesso. Il ricevitore ascolta più satelliti e usa metodi di geometria computazionale per determinare la propria posizione (dimensione spaziale) e una dimensione temporanea. Il tempo ottenuto dal GNSS ha una precisione dell'ordine dei micro o addirittura dei nanosecondi (ma dipende in realtà dalla precisione del GNSS).

Il GPS è il sistema di navigazione satellitare globale (GNSS) più utilizzato e riesce a calcolare la posizione guardando quanto tempo impiega il segnale ad essere trasmesso dal satellite al ricevitore e capendo a quante unità di distanza si trova, da qualche parte su una circonferenza. Ricevendo la posizione da un secondo satellite riduce le possibili soluzioni a due punti, ed aggiungendo un terzo satellite deduce più facilmente quale dei due possa essere. Dunque, se il nodo del sistema distribuito ha un GPS, allora ha già un modo per sincronizzarsi in maniera abbastanza precisa con l'UTC.

Complicazioni del GPS:

- ci vuole del tempo prima che i dati sulla posizione di un satellite raggiungano il ricevitore.
- l'orologio del ricevitore generalmente non è sincronizzato con quello del satellite.

- la precisione del GPS è principalmente disturbata nel caso in cui il segnale faccia fatica ad arrivare, anche a causa di fenomenti metereologici. Esistono metodi di aggiustamento che si appoggiano a delle stazioni a terra che, osservando i fenomeni metereologici, forniscono l'aggiustamento.

### 5.3.2 Cristian's algorithm and Network Time Protocol

Se un nodo del sistema distribuito non ha un GPS, il metodo più comune è rivolgersi ad un server. L'algorimo NTP deriva dall'algoritmo di Cristian: si ispira a quello, ma è stato perfezionato per riuscire a stimare meglio i ritardi dovuti alla rete.

Non prevede una banale architettura client-server, bensì prevede una gerarchia di server NTP divisi per strato. Lo strato 1 riguarda gli NTP server che hanno un orologio atomico o sono collegati ad esso con una rete a latenza bassissima. Gli strati inferiori, invece, garantiscono meno precisione.

Il client fa una richiesta ad un server NTP, supposto avere un'ora più o meno precisa, chiedendo l'UTC. L'aspetto complicato di questo algoritmo è calcolare le latenze. L'idea è che nel messaggio di richiesta venga inserito anche il timestamp. Conoscendo i timestamp di invio/ricezione della richiesta/risposta, può essere calcolata la distanza tra invio e ricezione della richiesta e la distanza tra invio e ricezione della risposta, dividendo per 2 (approssimazione, non è detto che siano uguali). Per fare un'approssimazione migliore di queste latenze, viene ripetuta l'operazione più volte. NTP permette di raggiungere un'accuratezza nell'ordine dei millisecondi.

### 5.3.3 Berkeley Algorithm

Fino a qui abbiamo visto due metodi (più utilizzati) per effettuare la sincronizzazione con il tempo reale. Non tutti i sistemi, però, hanno bisogno di questo. Se il sistema deve interagire con misure provenienti dal tempo reale, allora è opportuno, in altri casi, quando il sistema è chiuso, potrei anche accontentarmi del fatto che i nodi siano semplicemente tutti sincronizzati su un tempo.

Uno dei nodi funziona da Time Daemon e manda a tutti quanti (anche a se stesso) il proprio orario. I nodi rispondono con la differenza del loro orologio rispetto a quello che hanno ricevuto. Il Time Daemon calcola quindi una media tra gli orari e comunica a tutti di sistemarsi su questa media.

Non importa che sia l'orario vero oppure no e spesso questo algoritmo viene utilizzato in un ambiente ristretto, dove i nodi sono collegati attraverso una banda larga, per cui la latenza non viene solitamente presa in considerazione. Attenzione, però, che portare indietro un orologio non è una cosa indolore. Quello che avviene, di fatto, è

farlo rallentare finché non è allineato, come se fosse stato portato indietro.

## 5.4 Lamport's Logical Clocks

In molti casi non è necessario che i nodi concordino sul valore dei loro orologi fisici, bensì è possibile semplicemente utilizzare dei contatori o degli orologi logici (possono essere immaginati come dei contatori). Questo è possibile quando nel sistema distribuito è sufficiente avere una conoscenza condivisa sull'ordine parziale degli eventi, cioè quando occorre che il sistema sappia cosa è accaduto prima/dopo di qualcos'altro.

Un evento può essere:

- evento interno: ogni nodo ha un processo in esecuzione e ci sono eventi che riguardano processing locale
- messaggio inviato/ricevuto

Il tempo aumenta sempre, dunque utilizzando un contatore con questa stessa proprietà, otteniamo un clock logico. All'interno di ogni nodo viene utilizzato un contatore intero come orologio logico e viene incrementato ogni volta che si verifica un evento interessante (inizialmente saranno tutti uguali a 0). Osservando lo stato del sistema, in un particolare istante, si potrà notare che gli orologi logici avranno valori diversi.

L'algoritmo di Lamport prevede che:

- se A e B sono due eventi, l'espressione $A \rightarrow B$ denota la relazione (transitiva) A accade prima di B
- C(A) è il valore di clock logico assegnato dal processo in cui si verifica A
- Obiettivo: se $A \rightarrow B$, allora $C(A) < C(B)$, anche se sono su due processi diversi

Se A e B sono eventi sullo stesso processo, il contatore viene semplicemente incrementato. Il problema si pone nel momento in cui A è l'evento di invio da un nodo e B è l'evento di ricezione su un altro nodo. Potrebbero verificarsi molti eventi prima di A e pochi eventi prima di B, avendo così C(A) maggiore di C(B). Ovviamente questo comportamento non va bene, dunque:

- prima di eseguire un evento, incremento il contatore
- quando il processo Pi manda un messagio m a Pj, inserisce il valore del contatore dopo l'incremento
- Pj riceve il messaggio m, osserva il valore del contatore e lo confronta con il proprio. Dopodiché prende il maggiore e lo incrementa di uno

Questo, così come tutto ciò che vediamo nei sistemi distribuiti, si implementa attraverso un middleware.

### 5.4.1 Enforcing Total Order

Alcuni algoritmi vorrebbero un ordinamento totale dei timestamp nel sistema. Per evitare che due valori di clock nel sistema per due eventi diversi siano uguali, viene concatenato l'identificatore del processo (un qualsiasi numero che difficilmente si trovi in altri processi) al contatore. A questo punto, prendendo due valori di timestamp, è possibile stabilire un ordine totale tra questi: si osserva il valore di contatore maggiore/minore e, in caso di uguaglianza, si osserva l'identificatore maggiore/minore. L'obiettivo, quindi, è che ogni evento del sistema abbia associato un timestamp diverso. Attenzione, però, che un ordine totale dei timestamp non significa conoscere la relazione temporale tra ogni coppia di eventi.

## 5.5 Totally Ordered Multicast

Si assume che nessun messaggio venga perso e che i messaggi dallo stesso mittente vengono ricevuti nell'ordine in cui sono stati inviati.

Il processo Pi invia un messaggio $m_i$ con un timestamp a tutti gli altri. Il messaggio stesso viene inserito in una coda locale i. Qualsiasi messaggio in arrivo su Pj viene accodato nella coda j, in base al suo timestamp, e viene inviato un ack di ricezione del messaggio ad ogni altro processo (gli eventi di invio e ricezione di messaggi e ack sono totalmente ordinati con Lamport). Pj passa un messaggio $m_i$ alla sua applicazione se:

- $m_i$ è in testa alla coda j AND
- $m_i$ è stato confermato da tutti gli altri processi

Tutti i processi alla fine avranno la stessa copia della coda locale, quindi tutti i messaggi vengono passati all'applicazione nello stesso ordine ovunque.

## 5.6 Mutual exclusion

In un sistema distribuito, se un insieme di processi in esecuzione su sistemi diversi vogliono accedere ad una risorsa condivisa, occorre effettuare uno scambio di messaggi.

### 5.6.1 A centralized algorithm

Un processo, su uno dei nodi, viene utilizzato per gestire la coda. Gli altri nodi, con la solita modalità client/server, se necessitano di accedere alla risorsa lo richiedono al coordinatore. Se la coda è vuota, il coordinatore consente l'accesso, altrimenti il coordinatore non risponde (si suppone che sia sincrono, per cui il processo si blocca) ed inserisce la richiesta in coda. Nel momento in cui il coordinatore riceve il release da parte del processo che stava utilizzando precedentemente la risorsa, estrae la prima richiesta dalla coda (FIFO) e consente l'accesso. Il processo, a questo punto, si sblocca e riceve la possibilità di utilizzare la risorsa fino a che non la rilascerà.

I problemi di questo algoritmo sono:

- single point of failure: in caso di crash del coordinatore nessuno potrà più accedere alla risorsa, a meno che non sia replicato in qualche modo
- se l'utente non rilascia la risorsa, occorre un meccanismo di timeout (tempo limite di rilascio) dopo il quale viene tolta forzatamente
- bottleneck: passano tutti per lo stesso coordintore

### 5.6.2 A distributed algorithm

Si assume un ordinamento totale dei timestamp e una consegna affidabile dei messaggi, attraverso un meccanismo di ack.

Un processo P che vuole accedere ad una risorsa costruisce un messaggio contenente il nome della risorsa, l'id del processo ed il timestamp corrente (contatore + identificatore) e dopodiché invia il messaggio a tutti i processi, incluso se stesso. Quando un processo Q riceve un messaggio, abbiamo 3 casi:

- se Q non sta utilizzando R e non ha intenzione di farlo, risponde OK a P
- se Q sta utilizzando R non risponde e accoda la richiesta
- se Q non sta utilizzando R, ma ha intenzione di farlo, confronta il timestamp del messaggio con quello della propria richiesta. Se quello nel messaggio inviato da P è inferiore risponde OK a P, altrimenti accoda il messaggio

Dopo aver inviato il proprio messaggio, P attende un OK da tutti i processi prima di accedere a R. Quando P termina di utilizzare R, invia OK a tutti i processi che aveva precedentemente inserito nella sua coda e la svuota.

Si potrebbe anche ipotizzare di utilizzare l'orologio fisico, ma, anche se fossero sincronizzati correttamente, ci si potrebbe comunque trovare nella situazione di avere

più volte uno stesso timestamp, andando ad invalidare l'assunzione, a meno che non si aggiunga al timestamp l'identificatore del processo.

I problemi di questo algoritmo sono:

- la mancata risposta da parte di un processo può essere dovuta ad un suo crash
- coinvolgere tutti i processi di un sistema distribuito optrebbe essere uno spreco di risorse (con l'aumentare dei nodi)

### 5.6.3 A ring algorithm

Questo algoritmo utilizza una rete di collegamento tra i nodi che non rispecchia la rete di collegamento fisica. Partendo da un gruppo di processi non necessariamente ordinati viene costruito un anello logico, assegnando un identificatore univoco a ciascun processo in esecuzione sui diversi nodi del sistema. Dopodiché viene utilizzato un token, rappresentato da un messaggio, per gestire l'accesso alla risorsa. Il token viene fatto ruotare all'interno dell'anello e chi lo ottiene acquisisce la possibilità di accedere alla risorsa. Essendocene uno solo, non potrà mai essere in due nodi contemporaneamente. Quando il processo termina di utilizzare la risorsa passa il token al processo successivo. Non può usare due volte lo stesso token.

I problemi di questo algoritmo sono:

- nel caso in cui si dovesse avere il crash di uno dei nodi si spezzerebbe l'anello. Solitamente per avere un po' più di tolleranza si memorizzano k elementi in avanti
- perdita del token

## 5.7 Election algorithms

Questi algoritmi prevedono una modalità con la quale un insieme di nodi si accorda su quale sarà il coordinatore, eleggendo il processo attivo (in qualsiasi istante potrebbe non esserlo più) con l'identificatore più alto in quel momento. Alcuni algoritmi presumono che ogni processo conosca gli ID e possa comunicare con qualsiasi altro processo. Tuttavia, non possono sapere se un processo è attivo.

Questi algoritmi non perdono di generalità: può essere creato un ordinamento totale utilizzando il criterio che si ritiene più opportuno nella realizzazione degli identificatori.

### 5.7.1 Bully Algorithm

Abbiamo un gruppo di processi, di cui uno con l'identificatore più alto. Nel momento in cui un processo, dopo aver tentato di comunicare con il coordinatore, si accorge che questo non è più attivo (timeout scaduto), indice un'elezione, inviando un messaggio a tutti i processi con ID maggiore del proprio (compreso il coordinatore precedente, che nel frattempo potrebbe essere tornato disponibile), non sapendo se questi sono attivi. Quando un processo attivo riceve un messaggio di elezione, risponde con un messaggio di OK. Se un processo riceve almeno un OK, esce dall'elezione, poiché vuol dire che c'è almeno un processo con l'ID più alto del suo che si occuperà dell'elezione. A questo punto, i processi che hanno dato l'OK manderanno nuovamente un messaggio di elezione ai processi con identificare più alto del loro, ripetendo la stessa procedura vista precedentemente. Alla fine, il processo che non riceverà alcuna risposta entro un certo timeout diventerà il coordinatore e manderà un messaggio in broadcast per informare gli altri nodi.

Potrebbero essere indette più elezioni contemporaneamente, ma di solito dovrebbe vincere lo stesso nodo.

### 5.7.2 A ring-based election: Chang and Roberts algorithm (1979)

Questo algoritmo utilizza una struttura ad anello, in cui i messaggi circolano in senso orario, senza fallimenti, e in cui i processi hanno un identificatore unico. L'obiettivo è eleggere il processo attivo con l'ID più alto (deve essere unico, anche nel caso in cui più elezioni siano indette in modo concorrente).

L'algoritmo funziona nel seguente modo:

- i processi sono contrassegnati da un valore booleano che indica se questi sono partecipanti o non, in maniera tale da poter fermare il prima possibile eventuali messaggi di altre elezioni che porterebbero allo stesso risultato
- inizialmente tutti i processi sono contrassegnati come non partecipanti
- quando un processo Pk si accorge che il coordinatore non sta più rispondendo, indice un'elezione contrassegnandosi come partecipante e inviando al nodo successivo nell'anello un messaggio <ELECTION, ID(Pk)>
- quando un processo Pm riceve un messaggio <ELECTION, ID(Pk)>:
  - se l'ID del processo Pk contenuto nel messaggio è maggiore dell'ID del processo Pm, inoltra il messaggio al processo successivo nell'anello e si contrassegna come partecipante

  - se l'ID del processo Pk contenuto nel messaggio è minore dell'ID del processo Pm, si contrassegna come partecipante e sostituisce l'ID presente nel messaggio con il proprio, inoltrandolo poi al processo successivo nell'anello. Se, invece, il processo era già contrassegnato come partecipante, ferma il messaggio, bloccando così l'elezione
  - se l'ID del processo contenuto nel messaggio è il proprio, vuol dire che non c'è nessun altro processo attivo con ID più grande. Si contrassegna, quindi, come non partecipante ed invia un messaggio <ELECTED, ID(Pm)> al processo successivo nell'anello

- quando un processo Pk riceve un messaggio <ELECTED, ID(Pm)>, si contrassegna come non partecipante, memorizza l'ID del coordinatore e, a meno che non sia lui stesso il coordinatore, inoltra il messaggio al processo successivo nell'anello

I problemi di questo algoritmo sono:

- Errori di gestione: gli errori dovuti al crash dei nodi nell'anello vengono gestiti da ciascun processo memorizzando, non solo l'indirizzo del processo successivo, ma anche alcuni altri che lo seguono nell'anello. Se la comunicazione con il processo successivo fallisce, il messaggio viene inviato al primo tra quelli che lo seguono che è attivo
- Elezioni simultanee: l'uso dello stato partecipante/non partecipante aiuta a estinguere il prima possibile i messaggi non necessari nelle elezioni simultanee

Nel caso peggiore vengono scambiati 3n - 1 messaggi, cioè quando il coordinatore dovrebbe essere il nodo appena prima di quello che ha indetto l'elezione.

Il caso peggiore si verifica quando il processo con l'ID più alto è il processo più vicino in senso antiorario a quello che indice l'elezione. In questo caso abbiamo bisogno di N - 1 messaggi per raggiungere il nodo, altri N messaggi per concludere l'elezione e N messaggi per annunciare il coordinatore, dunque 3N - 1.

# Tolleranza e consenso

La resistenza ai guasti è uno dei parametri di valutazione di un sistema distribuito. Più è alto il numero dei nodi partecipanti (e dunque collegamenti), più è alta la probabilità di malfunzionamenti. Ad esempio è possibile avere ritardi, crash dei processi e nodi compromessi/malevoli. Questi malfunzionamenti vengono chiamati partial failure nel sistema. Un buon sistema distribuito fornisce trasparenza (in questo caso rispetto ai guasti) ovvero deve ovviare al malfunzionamento senza che l'utente possa percepirne gli effetti.

La *Fault tollerance* è collegata alla *dependability*, la quale implica:

- disponibilità: probabilità che il sistema operi correttamente in ogni singolo istante
- affidabilità: il sistema deve riuscire a rimanere attivo (senza guasti) il più a lungo possibile
- safety: nel caso in cui ci fosse un guasto, questo deve essere gestito in modo tale che non si blocchi l'intero sistema. Bisogna dunque replicare le risorse affinchè questi guasti non portino ad eventi catastrofici
- maintaniblity: abilità di riparare facilmente un sistema che ha fallito o ha dei malfunzionamenti

Un aspetto importante è che un sistema può avere alta *disponibilità*, ma bassa *affidabilità*: il sistema risulta dunque quasi sempre disponibile, ma rimane inattivo per pochissimo tempo con una certa frequenza. Può anche avere alta *affidabilità*, ma bassa *disponibilità*. Quello che si cerca di ottenere è dunque un buon comportamento sotto entrambi gli aspetti.

## 6.1 Modelli di fallimento

Ci possono essere vari modelli di fallimento all'interno del sistema:

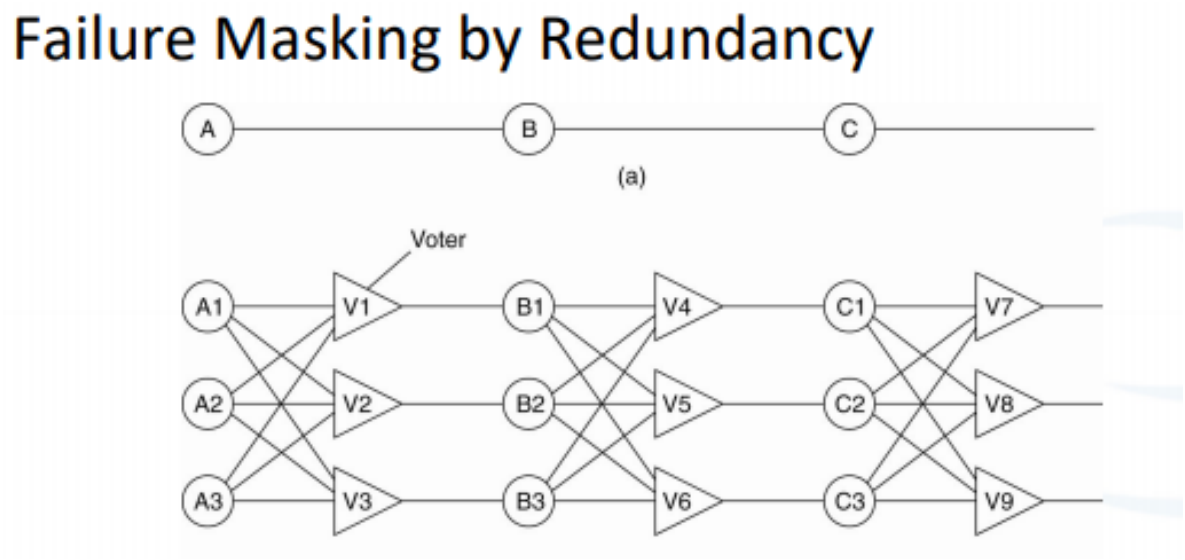
- crash failure: un server (nodo del sistema) che funziona perfettamente smette di funzionare
- omission failure: problemi nel ricevere e/o inviare messaggi
- timing failure: la risposta del server non rientra nel timeout
- response failure: il valore di una risposta non è quello che ci si aspetta oppure il flusso di controllo di un nodo non segue quello previsto
- arbitrary failure: un server potrebbe produrre risposte arbitrarie in momenti arbitrari. Rientra in questo failure anche il Byzantine failure nel quale dei server devono mettersi d'accordo su qualcosa e per via dei ritardi delle risposte e/o dei fallimenti i server potrebbero non riuscire ad arrivare ad uno stato di consenso

### 6.1.1 Mascheramento dei guasti per ridondanza

Per rendere più robusto un sistema rispetto ai guasti, il primo intervento che si utilizza (non solo in ambito di algoritmi) è aumentare i componenti. Un esempio è avere un server di riserva

- Information redundancy: si usano dei bit per cercare di ricostruire il messaggio quando c'è del rumore presente
- Time redundancy: se una transazione viene abortita, può essere ripetuta. L'utente finale vede un po' di ritardo, ma riesce comunque ad ottenere la risorsa
- Physical redundancy: hardware o processi in più al fine di sopperire al mal funzionamento di qualche componente

ESEMPIO:
Abbiamo un circuito con 3 dispositivi che si passano il segnale in sequenza. Se uno di questi funziona male, c'è un'alta possibilità che il circuito dia un risultato scorretto. Uno dei modi è replicarli. Replichiamo quindi A, B e C con altre due copie. Introduciamo dei componenti nuovi come il voter il quae emette un segnale se lo riceve in input dalla maggioranza dei dispositivi ad esso collegato. Se una delle 3 malfunziona, il voter da come output il valore dei due funzionano bene. Anch'essi non sono totalmente affidabili, dunque c'è ridondanza anche sui voter. Un failure tollerato per ogni componente (A, B e C)




### 6.1.2 Process resiliance

Il Process resiliance è possibile dedurlo da ciò che è stato fatto nell'ambito dell'hardware. Possiamo gestire la ridondanza usando gruppi di processi, invece che un singolo processo. Ogni processo riceve i messaggi del gruppo e per fare ciò è necessario un protocollo/meccanismo per fare il join/leave del gruppo. Quanta ridondanza abbiamo bisogno? Dipende dai tipi di guasti:

- crash: il client fa una richiesta al server. Se il server fallisce, il sistema è bloccato. Se abbiamo due processi nel gruppo e uno solo va in crash, l'altro è sufficiente per mascherare il malfunzionamento. Basta che resti attivo un processo nel gruppo per restituire il valore al client. Basta che resti attivo un processo nel gruppo per restituire il valore al client. Se invece, più processi rimangono attivi, restituiranno lo stesso valore al client. Il client gestirà queste risposte tutte uguali e riesce lui a fare da voter. K+1 processi forniscono k fault tolerance
- wrong values: se consideriamo il caso in cui i processi nel gruppo possano dare risposte al client con valori sbagliati, per garantire k fault-tolerance necessitiamo almeno 2k + 1 processi nel gruppo

### 6.1.3 Problemi di accordo

Il consenso è una proprietà in cui si vuole che i processi di un certo gruppo si mettano d'accordo sul valore di un certa variabile. Questi problemi di accordo si chiamano: *agreement problems*. Un esempio di questi problemi è riscontrabile andando ad analizzare lo stato di una macchina: finchè lo stato corrisponde ad un singolo nodo è

chiaro, ma se la macchina è l'immagine di un gruppo di processi nel sistema distribuito, bisogna che tutti i nodi siano d'accordo su quale sia lo stato effettivo della macchina. E' dunque necessario che ogni nodo abbia la stessa informazione e questa deve essere mantenuta consistente. Tra gli agreement problems si trovano:

- consenso: ciascun processo propone un valore e i processi che non sono malfunzionanti dovrebbero mettersi d'accordo su un unico valore
- generali bizantini: un processo (commander) propone un valore e tutti gli altri processi devono concordare su questo valore. Anche il comandante potrebbe non dare il valore giusto duqnue tutti i partecipanti devono mettersi d'accordo e il valore corretto dev'essere uno. Se il comandante non è faulty, tutti devono concordare
- interactive consistency: ogni processo propone un valore e tutti i processi non faulty concordano su un vettore di valori (che contiene cosa ha detto ciascuno). Se ci sono processi che fanno apposta a confondere, occorre che quelli non malevoli capiscano cosa hanno detto tutti quelli non malevoli

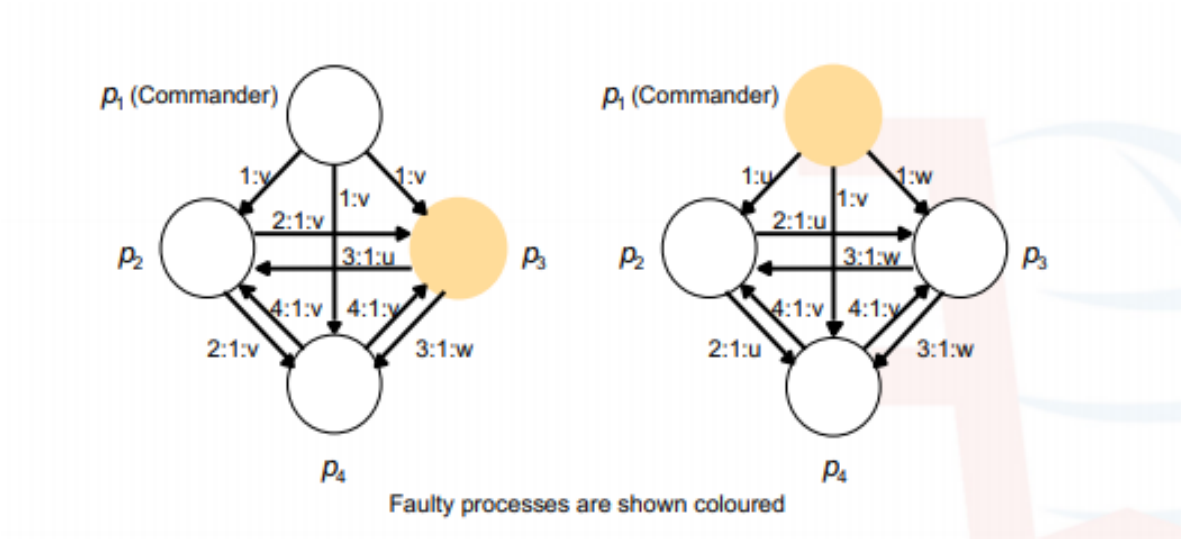
Ognuno di questi problemi risulta molto complesso e a volte anche irrisolvibile. Se si ha una soluzione per uno di questi problemi, è abbastanza facile usare la stessa soluzione per risolvere gli altri due.

#### 6.1.3.1 Risultato dell'accordo bizantino con sistemi sincroni

Nel caso di agreement con malfunzionamenti bizantini servono almeno N = 3k + 1 processi per riuscire ad arrivare al consenso e ottenere k-fault tollerance. Questo vale per sistemi sincroni. Sotto certe assunzioni, si riesce dunque a risolvere il problema.

**Esempio di soluzione per il problema dei generali bizantini per N = 4 e k = 1:**

- Il comandante (uno dei quattro) manda un valore agli altri 3. Ognuno dei 3 manda il valore che ha ricevuto agli altri peer. Nel caso di 4 bastano 2 cicli di questo algoritmo. Nel caso di più di 4, va avanti: chi riceve i messaggi dei peer a sua volta manda il messaggio che ha ricevuto con un numero di round dipendente da N. Nel primo esempio, p2 e p4 ricevono una maggioranza di v. Nel secondo esempio, nessuno dei 3 riceve una maggioranza e l'algoritmo utilizza, quindi, un carattere speciale che è indeterminato, che è impossibile che sia uguale a quello del comandante e riconoscono che il comandante non è affidabile. Dovranno quindi provvedere ad eleggere un nuovo comandante e ripetere il processo.




## 6.2 Sistemi sincroni VS sistemi asincroni

I nodi all'interno di un sistema distribuito si scambiano messaggi affinché emerga una condivisione di valori tra i processi che stanno funzionando correttamente. I sistemi si dividono in sistemi sincroni e asincroni:

- Sincroni: nei sistemi sincroni si assume che l'esecuzione del codice su ogni nodo è limitata in termini di velocità e tempo. Anche i link di comunicazione hanno un delay di tramissione con un certo bound e il clock su ogni nodo può avere un drift rispetto all'utc che però è bounded, poiché ci sono meccanismi di sincronizzazione
- Asincroni: nei sistemi asincroni l'esecuzione su ogni nodo può avere velocità arbitraria, i link di comunicazione hanno diversi delay di trasmissione unbounded e il clock su ogni nodo può essere completamente disallineato

I sistemi che usiamo nella realtà si avvicinano di più ai sistemi asincroni. Il problema è che il coordinamento e l'accordo nei sistemi asincroni è difficile e spesso impossibile. Di fatto molto spesso vengono fatte assunzioni di sincronicità parziale del sistema. E' dunque necessario utilizzare algoritmi che realizzano una forma di sincronizzazione anche se bisogna comunque tenere conto che tante soluzioni falliranno data la natura asincrona del sistema.

### 6.2.1 Impossibilità di accordo nei sistemi asincroni

Teorema FLP (1985): Quando i ritardi nella risposta ai messaggi sono arbitrari non esiste una soluzione garantita per l'accordo bizantino (consenso in presenza di difetti bizantini). Questo significa che se c'è anche un solo nodo nel sistema che ha un comportamento bizantino e siamo nel caso estremo asincrono, quindi con assenza di garanzie, non c'è una soluzione perfetta al problema. Abbiamo dunque una sincronia parziale. (?)

Teorema CAP: Questo teorema afferma che è possibile garantire al massimo due (di tre) proprietà nel caso di r/w su memoria condivisa. Queste proprietà sono:

- Consistenza: Ogni lettura riceve la scrittura più recente o un errore (implica che tutti i nodi vedano gli stessi dati)
- Disponibilità: Ogni richiesta riceve una risposta (sistema operativo in qualsiasi momento)
- Partition tolerance: Il sistema riesce a tollerare un numero arbitrario di messaggi persi (o ritardati)

La partition tolerance accade spesso nei nostri sistemi, quindi la vogliamo. Questo implica che i sistemi reali devono rinunciare a uno degli altri due.

### 6.2.2 Consenso pratico: il protocollo Paxos

Questo protocollo è stato introdotto da Lamport nel 1989 e ad oggi ne esistono diverse versioni. Lo scopo di questo protocollo è risolvere il problema del consenso garantendo la consistenza in una rete di processi inaffidabili, che possono quindi guastarsi.
La versione base di questo protocollo, che comunque è complicatissima, non copre i failure bizantini. Questa versione tollera k-failing nodes avendo N = 2k+1.
Avendo Paxos molte versioni, ne esiste anche una per risolvere il problema bizantino. Dato che nel caso completamente asincrono non è possibile dare una soluzione completamente corretta, vengono fatte delle assunzioni. Anche se sono molto rare, esistono condizioni nelle quali questo algoritmo non riesce ad arrivare al suo obbiettivo. Questo algoritmo è alla base di molti tool moderni.

### 6.2.3 Consenso pratico: il protocollo Raft

Il protocollo Raft nasce da un gruppo di ricerca di Stanford come soluzione alternativa e più semplice al protocollo Paxos.
Raft si pone due obiettivi:

- Rendere più accessibile la comprensione del protocollo
- Rendere più facile l'implementazione e l'applicazione dell'algoritmo di consenso risetto a Paxos.

Raft, come Paxos, ha lo scopo di trovare una soluzione alla distribuzione dello stato delle macchine e avere così un consenso sullo stato del sistema preservando la safety (consistenza). Una caratteristica fondamentale di Raft è che si basa sull'elezione di un leader e questo comporta tutta una serie di problematiche che sorgono nel caso in cui il leader si guasti. Raft non copre nè bizantine failures nè nodi malevoli, ma copre ritardo di messaggi e crash dei nodi. I nodi del sistema credono alla correttezza del leader eletto e tollera k nodi in errore con $N = 2k + 1$.

# DLT e Blockchain

## 7.1 Distributed Ledger Technologies & Blockchain

Il problema del consenso è uno dei problemi fondamentali all'interno della blockchain. Nell'ambito business è considerata, tra le tecnologie emergenti, una tecnologia essenziale.

### 7.1.1 Storia

La nascita della blockchain coincide con la nascita del progetto teorico del Bitcoin (2008). Gli algoritmi che stanno alla base di questa tecnologia invece erano già presenti nell'ambito dei sistemi distribuiti.
Nel 2009 esce una prima implementazione opensource di questo sistema. Da questo momento vengono sviluppate altre varianti. Questo meccanismo va bene non solo per denaro elettronico, ma ha potenziali applicazioni in molti altri ambiti. Potrebbe addirittura essere la base per avere applicazioni trusted distribuite, il cui output è trusted nonostante possano essere eseguite anche da nodi malevoli.
Nel 2012-2013 vengono create altre criptovalute basate su DLT.

### 7.1.2 Perchè blockchain?

La blockchain implementa un registro di cose che avvengono (ad esempio registro delle transazioni immobiliari/catasto) che è:

- immutabile: posso andare a cercare indietro nella storia, ma non posso cancellare informazioni, posso solo aggiungerle
- distribuito: non si ha tutte le informazioni in uno stesso posto come un catasto o un registro centralizzato

- fault tolerant: non soltanto rispetto ad eventuali crash, ma anche rispetto a comportamenti bizantine, poiché non ho fiducia in alcun nodo che partecipa

### 7.1.3 Il modello del sistema DLT

È un sistema distribuito che ha le seguenti caratteristiche:

- controllo decentralizzato, si ha quindi l'assenza di un nodo coordinatore
- i nodi sono gestiti da entità separate che non si fidano gli uni degli altri
- una copia dei record dei dati è salvata in ogni nodo

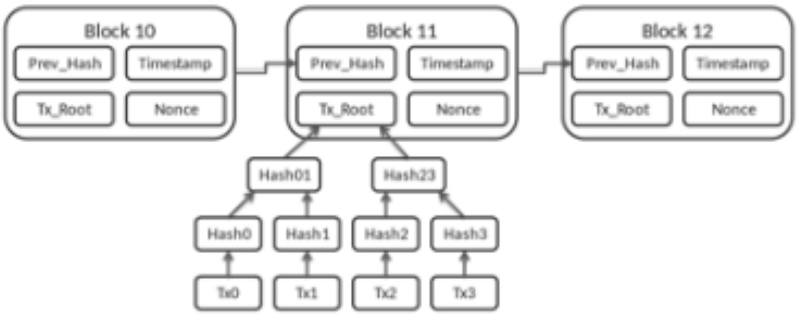
Il problema del consenso, in questo caso, è che i nodi devono essere d'accordo, non sull'ultimo dato memorizzato, bensì sulla storia dei dati.

### 7.1.4 Dati nella Blockchain

Una blockchain è una sequenza storica di transazioni.
Una transazione è un record di dati (in ambito finanziario una transazione è un trasferimento di finanze ad esempio in BTC).

### 7.1.5 Approccio

Ogni transazione che viene inserita viene firmata (tramite criptazione asimmetrica, ossia utilizzando una chiave pubblica e una privata), e questa transazione firmata viene mandata (propagata) a tutti i nodi.
All'interno della blockchain si è identificati tramite la coppia chiave pubblica, chiave privata. Di queste coppie se ne possono avere diverse, quindi all'interno della blockhain si possono avere diverse identità(per questo è un sistema così anonimo).
Quando un nodo riceve una transazione deve validarla (in un ambiente finanziario un nodo potrebbe vedere nella sua storia se si hanno in fondi necessari per eseguire la transazione).
Le transazioni dopo che sono validate sono in stato pending siccome non sono ancora nella parte della catena.




In una blockchain le transazioni sono raggruppate in blocchi (di n numeri di transazioni, dove n può essere diverso da un blocco a un altro) che hanno:

- Timestamp di quando il blocco è stato creato
- Tx_Root è una struttura dati (merkle tree: molto efficiente capire se una transazione è contenuta o meno nel blocco) che serve per memorizzare le transazioni che stanno dentro al blocco. Ogni transazione ha un hash che la rappresenta.
- Nonce: un numero
- Prev_Hash: collegamento al blocco precedente tramite il suo hash

*Ogni nodo partecipante ha una copia della blockchain intera.*

### 7.1.6 Problemi

I sistemi di cui stiamo parlando sono asincroni, difficili da gestire a causa di latenza indecidibile, sincronizzazione imprecisa del clock e nodi malevoli, tutte caratteristiche che comportano:

- l'ordine di arrivo delle transazioni possono essere diversi su diversi nodi
- alcune transazioni potrebbero contraddirsi a vicenda
- nodi diversi possono costruire diversi blocchi
- nodi diversi possono ritrovarsi con catene diverse (non si ha consenso)

### 7.1.7 Consenso nella blockchain

La sfida è quindi avere per ogni nodo il consenso sui blocchi e sulla sequenza di blocchi (sulla catena).
L'idea principale dell'algoritmo è:

1. calcoliamo l'hash di ogni transazione e di un blocco
2. quando calcolo l'hash del blocco inserisco nella funzione di hash anche l'hash del blocco precedente
3. si include un trucco per rendere la computazione dell'hash del blocco molto dispendiosa(processo di mining), ma molto facilmente verificabile (proof of work)
4. se abbiamo un certo numero di nodi che vogliono fare un lavoro li si fanno competere e c'è una ricompensa (valuta di quella blockchain) per il vincitore, che si occuperà di distribuire il blocco calcolato a tutti gli altri nodi (è come eleggere un nodo che impone il suo blocco)

L'hash di un blocco, di fatto, è la sua firma digitale. Modificando qualcosa all'interno della catena, la catena non sarà più valida, poiché occorrerà ri-validare tutti i blocchi che seguono attraverso il mining.
Per controllare se gli hash delle catene sono uguali, controllo l'hash dell'ultimo blocco di ciascuna catena: se sono uguali, ho il consenso sulla catena. Se non ho consenso ho il rischio che nodi diversi abbiano catene diverse.

Perchè viene dato un lavoro difficile al miner?
Perchè ci vorrà del tempo per risolverlo e quindi sarà più difficile avere soluzioni contemporanee.
Questo "puzzle" da risolvere viene calibrato in base alla quantità e alla capacità dei miner. Deve essere un problema risolvibile con algoritmi che operano con tecniche brute force, che si può far diventare più difficile progressivamente e che abbia una certa varianza.
Ma qual è questo problema così difficile da risolvere?

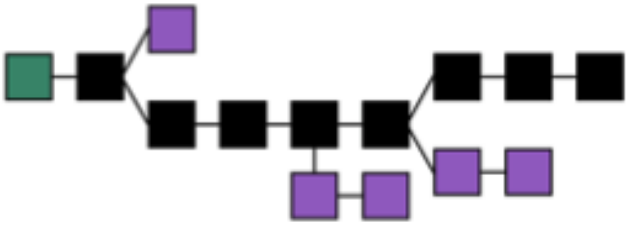
### 7.1.8 Hashing

Abbiamo una funzione di hash crittografica f (SHA-256):

- dove f(A) ha una lunghezza fissa (per esempio 256 bit, indipendentemente dalla lunghezza dell'input A)
- che è resistente alle collisioni (se A diverso B anche f(A) diverso f(B))
- dove è molto difficile trovare A a partire da f(A)
- dove è molto facile calcolare f(A), cioè è facile verificare dati A e B se B = f(A)

### 7.1.9 Algoritmo PoW nella Blockchain

Il compito del miner è trovare il Nonce tale per cui il valore dell'hash finale sia più piccolo di un certo numero, scelto in modo collettivo, affinchè il tempo di risoluzione rimanga costante nel tempo.
Il computer deve provare con un meccanismo di brute force tutti i Nonce fino a quando non trova quello che soddisfa i requisiti.
È un problema progettato per richiedere un certo lasso di tempo, in modo da diminuire al minimo la probabilità che due miner risolvano l'enigma nello stesso momento.
Quando un miner risolve il puzzle lo annuncia a tutti.
Gli altri nodi, quando ricevono un messaggio da un nodo che dice di aver risolto un blocco, se effettivamente è stato verificato, lo aggiungono alla copia locale della loro catena.



La catena può avere dei branch (come possiamo vedere anche dalla foto), poiché siamo in una modalità concorrente, cioè i processi vengono eseguiti su nodi diversi ed è difficile prevedere chi finisce prima o dopo o quasi nello stesso momento.
È possibile che arrivino due blocchi entrambi validati e che abbiano lo stesso indirizzo del blocco precedente, nonostante siano diversi, formando quindi un branch.
La proprietà del Proof of Work fa si che ci sia una catena che si sviluppa più velocemente delle altre e che è quindi la porta ad essere la principale.
I blocchi che rimangono pendenti e che non fanno parte della catena prevalente, devono essere svuotati delle loro transazioni, che vengono rimesse nella pool di transazioni pending, a meno che non facciamo già parte di un blocco nella catena principale.
Saranno poi pescate in successivi tentativi di composizione dei blocchi. (?) Un miner prende delle transazioni dal pool del pending, senza nessuna politica di selezione. Diversi miner possono prendere diverse transazioni. Se due miner sono al corrente di uno stesso blocco e sono d'accordo su di questo, prendono due insieme di transazioni diverse, lo risolvono e quando lo restituiscono al nodo vengono aggiunti entrambi generando il branch.(?)

### 7.1.10 Proprietà della blockchain

- Assumendo che il maggior numero dei nodi stanno lavorando sulla stessa catena, quella che cresce più velocemente sarà la più lunga e la più veritiera.
- Un nodo malevolo se vuole modificare la transazione di un nodo intermedio deve anche ri-minare tutti i blocchi successivi e deve anche prevalere su tutti gli altri nodi della rete.
- Il meccanismo della blockchain è sicuro finchè più del 50% del lavoro dei miner è onesto

### 7.1.11 Limiti del Proof of Woork

Ci sono tantissime critiche a questo algoritmo di consenso, per cui sono emerse altre proposte.
Non è il PoW che fa funzionare la blockchain, è l'algoritmo di consenso in generale ad essere importante.
I principali difetti sono:

- consumo di energia e risorse (circa 1 miliardo di euro al giorno)
- numero di transazioni al secondo limitate

### 7.1.12 Proof of Stake

È detto proof-of-stake (PoS, vagamente traducibile in italiano come "prova che si ha un interesse in gioco") un tipo di protocollo per la messa in sicurezza di una rete di criptovaluta e per il conseguimento di un consenso distribuito. È basato sul principio che a ogni utente venga richiesto di dimostrare il possesso di un certo ammontare di criptovaluta. Si differenzia dai sistemi proof-of-work che sono basati su algoritmi di hash che validano le transazioni elettroniche. Peercoin è stata la prima criptovaluta ad introdurre sin dal lancio il sistema Proof of Stake senza mai implementarlo completamente. Altre note implementazioni del PoS sono BitShares, Nxt, BlackCoin e Cardano.

#### 7.1.12.1 Varianti per la selezione di un blocco

Ogni qualvolta un nuovo blocco viene aggiunto alla blockchain, deve essere scelto il creatore del blocco successivo. Dato che quest'ultimo non può essere l'account che possiede la maggiore quantità della criptovaluta (altrimenti questo creerebbe tutti i blocchi), sono stati escogitati diversi metodi di selezione.

**7.1.12.1.1 Selezione casuale (random)** Nxt e BlackCoin utilizzano una funzione casuale per predire il generatore del blocco successivo, impiegando una formula che cerca il valore hash più basso rapportato alla dimensione della somma in gioco. Dato che la conoscenza delle somme è pubblica, ogni nodo della rete può predire - con ragionevole accuratezza - quale account si aggiudicherà il diritto di forgiare un nuovo blocco.

**7.1.12.1.2 Selezione basata sull'anzianità** La PoS di Peercoin mescola la selezione casuale con il concetto di "anzianità", un numero ottenuto tramite il prodotto del numero di monete per il numero di giorni in cui tali monete sono state possedute. Le monete che non sono state spese per almeno 30 giorni competono per la creazione del blocco successivo. Gli ammontari di monete più anziani e più grandi hanno una maggiore probabilità di firmare il blocco successivo. Eppure quando un ammontare di monete è utilizzato per firmare un blocco, questo ammontare deve ricominciare con "anzianità zero" e quindi aspettare almeno altri 30 giorni prima di poter firmare un altro blocco. E inoltre la probabilità di trovare il blocco successivo è massima dopo 90 giorni, per prevenire che somme consistenti e molto "anziane" possano dominare la blockchain. Questo processo mette in sicurezza la rete e produce gradualmente nuova valuta nel corso del tempo senza consumare una potenza computazionale significativa. Gli sviluppatori di Peercoin sostengono che questo renda più difficile attaccare la rete dato che cade il bisogno di piattaforme centralizzate di mining e inoltre acquistare più di metà delle monete è probabilmente più costoso che acquisire il 51% della potenza di hashing della proof-of-work.

**7.1.12.1.3 Selezione basata sulla velocità** Il concetto di PoS di Reddcoin basata sulla velocità rivendica di incoraggiare la movimentazione di moneta piuttosto che il suo accumulo.

**7.1.12.1.4 Selezione basata sul voto** Invece di utilizzare solamente il concetto di posta in gioco (stake), i creatori dei blocchi possono essere selezionati mediante votazione. BitShares utilizza un sistema che comprende 101 delegati e sceglie casualmente tra essi.[1] Il voto della comunità aumenta l'incentivo dei creatori dei blocchi ad agire responsabilmente, ma al contempo apre alla prospettiva di scenari di sybil attack - come ad esempio nell'eventualità che un singolo utente impersoni i primi cinque delegati.

### 7.1.13 Raft

Raft è algoritmo per il consenso che è stato progettato per essere facile da capire. È l'equivante a Paxos a livello di fault-tolerance e performance.
Questo protocollo scompone il codice in sottoproblemi relativamente indipendenti tra

loro e affronta ogni “pezzo” singolarmente.
Lo scopo principale di Raft è quello di rendere il consenso disponibile ad un pubblico sempre più vasto e che quest'ultimo sarà in grado di sviluppare nuovi sistemi basati sul consenso.
Raft si occupa della replica dei log. Intuitivamente, se si riesce a far apparire la stessa identica sequenza di log sulla maggior parte delle macchine, si è riusciti a far sì che quel cluster di macchine concordi su qualcosa.
In effetti, questo è chiamato trasmissione dell'ordine totale.
Se si riesce a far apparire ogni voce del log nella stessa posizione in un cluster di macchine, si può usarlo per implementare il consenso.
Ad esempio, se una macchina vuole proporre un valore e supponiamo che la sua ultima voce di registro sia nella posizione i, può provare a far sì che altre macchine mettano questo nuovo valore in i + 1 . Dopo che la maggior parte delle macchine nel cluster ha replicato quel valore in i + 1 , ora chiamiamo quel valore impegnato in i + 1 . Questo è effettivamente lo stesso che proporre un valore e farlo accettare da altri nei termini di Paxos.
Raft è un protocollo basato su leader. Nel suo normale corso operativo, un solo leader sarà eletto dal cluster di nodi. Gli altri nodi sono seguaci. Il leader accetta le richieste di scrittura del cliente e le replica ai follower.
Se si vogliono avere informazioni più approfondite su Raft si può leggere questo articolo che ne spiega in dettaglio il funzionamento.

### 7.1.14 Smart Contracts

La blockchain è utile per salvare cose che vanno oltre a transazioni finanziarie. Ad esempio Ethereum è stata progettata per gli smart contracts.
Gli smart contract sono porzioni di codice software definiti come contratti self-executing. Sebbene originariamente sono stati proposti come versione digitale di contratti legali, possono essere dei programmi software generali.
Ogni contratto è salvato nella blockchain, diventando così immutabile.
Ogni nodo può verificare se le condizioni del contratto sono soddisfatte e eseguire determinate azioni (l'output deve essere validato in maniera distribuita). Alla fine tutti i nodi devono essere d'accordo sullo ”stato” risultante.
Un esempio di applicazione potrebbe essere una piattaforma di crowdfounding senza autorità centrale.

### 7.1.15 Permissionless vs Permissioned DLT

Le blockchain di Bitcoin ed Ethereum sono chiamate permissionless poichè:

- si ha un DL decentralizzato che traccia tutte le transazioni

- non ci sono terze parti fidate
- si ha accesso incondizionato al ledger
- si ha la validazione delle transazioni e la generazione di nuove monete da parte dei miners
- si ha la pseudo-anonimità dei partecipanti
- le transazioni sono immutabili

Esistono anche DLT permissioned, che vengono sfruttate soprattutto in ambito business dove ci sono molte istituzioni che usano questo meccanismo in un insieme trust limitato. In queste DLT:

- si ha un DL decentralizzato che traccia tutte le transazioni
- ci sono una o più terze parti fidate
- si ha un accesso condizionato al ledger
- si ha la validazione delle transazioni e la generazione di nuove monete da parte dei miners
- si conosce l'identità dei partecipanti
- le transazioni sono immutabili

#### 7.1.15.1 Hyperledger Fabric

Hyperledger nasce nel 2015 come consorzio di industrie che ha lo scopo di sviluppare una blockchain open-source per il business, hostata da Linux Foundation.
Fabric è uno dei sottoprogetti, originariamente gestito da IBM) che prevedeva:

- permissioned DLT
- architettura modulare in grado di adattarsi a diversi requisiti: transazioni private, contratti confidenziali, diversi protocolli di consenso
- app (contratti) distribuite in linguaggi di programmazione generici
- nessuna dipendenza da criptovalute native

# Large Scale Data Storage and Processing on Google's Distributed Systems

La missione di Google è organizzare l'informazione globale affinché sia universalmente accessibile. I sistemi di Google memorizzano enormi quantità di dati, ad esempio:

- indici web
- archivio cache delle pagine web
- tutti i video di YouTube
- informazioni di Maps e Street View
- dati Gmail
- Google Photos
- Google Drive

## 8.1 Storage systems

Negli anni in Google sono state sviluppate molte diverse tecnologie di archiviazione. Codice e implementazione sono proprietari, ma i risultati sono spesso condivisi con la comunità di ricerca sotto forma di articoli e conferenze. Molte alternative open source sono state sviluppate da altre aziende, spesso anche con l'aiuto di Google.

Esempi di queste tecnologie sono: Google File System, Bigtable, Spanner e F1.

## 8.2 Protocol buffers

Protocol buffer è una rappresentazione strutturata di dati, molto utile in applicazioni che prevedono il passaggio di dati da un server all'altro in maniera efficiente dal punto di vista del networking e del processing.

Definire un messaggio Protocol buffer è un po' come definire una struct o una classe. Utilizza una propria sintassi, ma viene compilato in classi native, supportando la maggior parte dei linguaggi. Tutto questo è open source.

Si preferisce a JSON e XML per motivi di efficienza, compattezza e sicurezza rispetto alla retrocompatibilità.

## 8.3 GFS - Google File System

Google File System è un file system distribuito in modo massiccio e tollerante ai guasti che archivia e recupera i dati in modo efficiente.

Fornisce primitive di accesso POSIX (open, read, write, close) ai file ed essendo distribuito consente l'accesso a diversi client simultaneamente. Inoltre, per garantire fault tolerance, archivia più copie dei dati su macchine diverse (ridondanza).

### 8.3.1 Colossus

L'implementazione originale di GFS ha ormai più di 15 anni. Per soddisfare i nuovi requisiti, attualmente Google utilizza una nuova tecnologia denominata Colossus che:

- utilizza la tecnologia Bigtable per archiviare i metadati
- aumenta i limiti di dimensione dei file
- migliora la latenza
- diminuisce l'utilizzo dello spazio di archiviazione, includendo nuovi protocolli per la ridondanza dei dati (codifica Reed-Solomon, aumenta la ridondanza dei dati senza, però, dover necessariamente impiegare la stessa mole di dati su disco).

## 8.4 Bigtable

Bigtable è il primo dei cosiddetti database NoSQL, database che violano il paradigma di SQL e le sue tipiche proprietà. Introduce diverse assunzioni al fine di riuscire ad archiviare moli di dati enormemente maggiori, mantenendo però una certa efficienza

in termini di latenza. Consiste in una mappa chiave-valore ordinata, distribuita e multidimensionale, progettata per memorizzare miliardi di record e scalare su migliaia di macchine in uno stesso datacenter. Inoltre fornisce delle utility per replicare dati su diversi datacenter, permettendo la comunicazione tra due istanza di Bigtable situate in datacenter differenti. Inizialmente era basato su GFS.

Ogni riga è composta da un numero arbitrario di colonne e per ogni coppia riga-colonna sono associate diverse celle. Infatti, Bigtable fornisce anche la dimensione temporale: data una riga e una colonna, possono essere salvate più celle, ciascuna annotata con un certo timestamp. L'idea è che si possano avere diverse versioni di un certo dato. Inoltre, tutti i tipi di dato sono stringhe e la conversione, così come la scelta del timestamp, è lasciata allo sviluppatore.

Queste tabelle vengono divise in intervalli di record (gruppi di record adiacenti da circa 100-200MB di dati) chiamati tablet. I tablet sono distribuiti tramite diversi tablet server, ciascuno dei quali è responsabile di circa 100 tablet. In un'istanza di Bigtable si possono avere chunk di tabelle di diversi servizi, ciascuna con le proprie regole. Il sistema deve garantire:

- Fast recovery: le 100 tablet che un server (andato offline) aveva vengono rese disponibili dal sistema. Gli altri server cominceranno a contendersele e verranno distribuite, un tablet per ciascuno. Il master mantiene un indice di tutti i tablet disponibili e dei server che li hanno in carico. Quando un server non risponde più, il master sa quali tablet sono rimasti "orfani" e contatta tutti gli altri oppure distribuisce in maniera opportunistica. La macchina che gestisce i tablet ha solamente il compito di mantenere e aggiornare questi tablet su GFS, ma, se va offline, il dato è già altrove.
- Fine-grained load balancing: spostare i tablet da una macchina sovraccarica. Il master prende decisioni in merito al bilanciamento del carico.

Problemi di Bigtable:

- non supporta le transazioni: l'atomicità è garantita solo a livello di riga. I Bigtable replicati sono coerenti solo alla fine, cioè se un client scrive all'interno del record un nuovo dato, un altro client non potrà mai vedere un valore intermedio nella scrittura
- non supporta le tabelle relazionali: per applicazioni che richiedono schemi complessi e in evoluzione, Bigtable può essere difficile da usare

Altri sistemi sono stati realizzati a partire da Bigtable cercando di risolvere questi problemi, ad esempio Megastore per supportare le transazioni e la replica.

## 8.5 Spanner

Per diversi anni i ricercatori hanno studiato soluzioni più SQL-like senza dover sacrificare prestazioni in termini di scala, potendo così gestire la stessa quantità di dati. Un nuovo sistema, denominato Spanner, è stato studiato e sviluppato per anni per fornire maggiori garanzie di Bigtable rispetto alle transazioni e alla replica globale.

Spanner è un database multiversione distribuito che supporta le transazioni (ACID), la schematizzazione delle tabelle e un data model semi-relazionale. Utilizza il linguaggio SQL ed impone che le tabelle abbiano una sorta di gerarchia. Spanner è stato progettato per scalare su milioni di macchine, attraverso centinaia di datacenter, e salvare trilioni di righe di database.

- mantiene il versionamento dei dati, anche se il timestamp è gestito dall'engine piuttosto che dal programmatore (il timestamp di ogni versione è quello di commit)
- la distribuzione globale dei dati attraverso i datacenter è configurabile, ad esempio i dati possono essere spostati automaticamente più vicino agli utenti che li utilizzano più frequentemente
- fornisce forti garanzie di consistenza:
  - esterna di read e write, ossia per l'osservatore esterno ciascuna scrittura avviene in maniera atomica, quindi non potrà leggere dati parziali
  - possibilità di accedere ad una versione del database ad un certo punto nel tempo, con la garanzia che sia consistente in maniera globale

Spanner è costituito da un'unica istanza globale distribuita attraverso i datacenter (Bigtable, invece, prevede un'istanza differente per ciascun datacenter). Introduce un secondo livello di master: ciascun datacenter è costituito da una zona e per ciascuna zona si ha una pool di master locali, avendo quindi master distribuiti globalmente all'interno di ciascun datacenter. I dati sono distribuiti su server chiamati spanserver, ciascuno dei quali contiene 100-1000 tablet replicati attraverso diversi datacenter (zone).

Per garantire che il database evolva in maniera indipendente, il locking distribuito avviene a livello di ciascuna tablet, dunque tutte le altre possono essere scritte/lette senza problemi. Ad ogni transazione, il timestamp del commit e l'ottenimento del lock viene confermato tra le zone attraverso un protocollo Paxos. Spanner è dunque riuscito ad adattare i protocolli di consenso a livello globale. Per evitare i possibili errori sulla precisione del tempo è stata definita una particolare rappresentazione del tempo che aggiunge incertezza (TrueTime), attraverso la quale il lock viene richiesto per un intervallo di tempo intorno a x. Questo garantisce che il sistema di lock abbia

performance di latenza accettabili.

## 8.6 F1

Inizialmente Spanner non prendeva in considerazione diversi aspetti aggiuntivi di un database SQL, per potersi focalizzare sull'avere un engine che potesse essere utilizzato per un diverso spettro di applicazioni. Tuttavia certe applicazioni avevano bisogno di dati particolarmente complessi dal punto di vista delle relazioni, dunque il team responsabile di Spanner cominciò a lavorare su un layer addizionale di accesso a Spanner, chiamato F1, che introducesse alcune feature, in particolare:

- schema relazionale: un engine che consentisse di
  - mantenere gli indici consistenti
  - utilizzare tipi di dato diversi dalle stringhe, in particolare protocol buffer
  - evolvere lo schema del database senza causare un particolare downtime
- interfacce multiple: SQL, key/value R/W, MapReduce
- notifiche di cambiamento: un engine di notifica nel momento in cui certi dati nel database cambiano. Questo fu inizialmente sviluppato on top of Spanner e successivamente fu introdotto all'interno di Spanner per migliorare le performance e togliere un livello di indirection

F1 introduce un pool di worker per l'esecuzione distribuita di SQL ed effettuare ciò che Spanner non è in grado di fare, utilizzando quest'ultimo per lo storage dei dati su GFS. Il server di F1 è stateless.

Una delle caratteristiche di F1, come visto precedentemente, è la possibilità di utilizzare protocol buffer come tipo di dato all'interno del database. Nel momento in cui vengono serializzati, questi vengono archiviati come blob in Spanner. F1 estende la sintassi di SQL per la lettura di campi nidificati, ossia per fare riferimento ai valori dei campi del protocol buffer salvati all'interno di una colonna. Protocol buffer consente di semplificare schema e codice, poiché le app utilizzeranno gli stessi oggetti.

## 8.7 MapReduce

Il primo fondamentale contribuito da parte dei ricercatori di Google per il processing e la generazione di grandi set di dati è stato il modello di programmazione MapReduce. È stato progettato per funzionare con pool di macchine eterogenee, semplicemente connesse. MapReduce è basato su alcuni paradigmi di programmazione abbastanza

comuni in programmazione funzionale e in alcuni linguaggi di programmazione. Può essere applicato a un'ampia varietà di problemi e consente il calcolo parallelo di problemi nella seguente forma:

- map: (k1, v1) → list(k2, v2)
- reduce: (k2, list(v2)) → list(v2)

Dove k1 e v1 sono una mappa valore-chiave di input e v2 è l'output desiderato. Si presuppone, quindi, di ricevere un problema definito dal programmatore in termini di una o più operazioni di map, ovvero un'operazione che dato un insieme di chiavi-valori genera una lista di chiavi-valori, e una operazione di reduce, ovvero un'operazione che data una chiave e una lista di valori associata a quella chiave è in grado di computare un certo risultato. MapReduce, se si è in grado di modellare il problema in questi termini, può garantire la distribuzione del problema su diverse macchine. MapReduce raggruppa quindi i record emessi da varie funzioni di map distribuite che hanno la stessa chiave ed invoca la funzione reduce.

Il sistema, inoltre, esegue un importante passaggio addizionale tra Map e Reduce, chiamato informalmente Shuffle. Shuffle è l'operazione di raggruppamento che MapReduce esegue per tutti i valori associati ad una chiave, cosi che si ottenga un record per ciascuna chiave con la lista dei valori associati ad essa.

- map: (k1, v1) → list(k2, v2)
- shuffle: list(k2, v2) → (k2, list(v2))
- reduce: (k2, list(v2)) → list(v2)

Da un punto di vista architetturale, quando si scrive un programma MapReduce, quello che si ottiene è un binario che, a seconda di come viene invocato dalle diverse macchine (quali parametri), fa assumere alla macchina una diversa funzione nel sistema distribuito. Questo file viene distribuito su tutte le macchine che si vuole includere nella computazione. Una macchina assumerà il ruolo di master e tutte le altre assumeranno il ruolo di worker, pronte ad effettuare un lavoro su indicazione del master.

L'interfaccia MapReduce fornisce delle librerie per poter leggere da differenti tipologie di datasource, ad esempio MapReader, per leggere da directory GFS e assegnare a ciascun worker un file differente, e BigtableMapReader, che istruisce i worker a leggere diversi tablet. Si ha quindi una libreria di lettura che distribuisce il carico di lavoro dell'input attraverso diversi worker, che invocano la funzione di Map sull'input, salvando l'output in alcuni file intermedi in un formato facilmente recuperabile per chiave. Nella parte di Reduce il worker recupera le emissioni per una certa chiave ed invoca la reduce, scrivendo in output alla fine.

Ci sono molteplici applicazioni, di cui semplici esempi sono:

- Distributed Grep: trovare una parola/espressione regolare all'interno di molti file, utilizzando solo la funzione di map, che cerca la parola e se viene trovata emette il risultato, ed il resto lo fa il sistema
- Count URL clicks: contare quante pagine sono state visitate avendo il log di un webserver. Basta emettere un numero per ogni pagina nel log e con la reduce vengono raggruppati tutti i numeri, ottenendo il punteggio
- Reverse Web-Link graph: data una pagina nel grafo del web, calcolare tutte le pagine che hanno un link verso quella pagina. Viene distribuito il grafo del web su diverse macchine, ciascuna delle quali fa la scansione di una pagina e se trova il collegamento emette una coppia chiave-valore, dove la chiave è la destinazione dell'URL. Nella reduce viene raggruppato il tutto

È possibile concatenare più MapReduce per eseguire algoritmi più complessi, ma può iniziare a diventare ingestibile.

## 8.8 FlumeJava

FlumeJava è un framework e un nuovo modello di programmazione che evolve MapReduce, in cui vengono fornite delle API (primitive e funzioni costruite sopra a queste) per fare computazioni distribuite:

- libreria Java per la scrittura di pipeline parallele ai dati
- classi per raccolte (possibilmente enormi) immutabili
- metodi per operazioni parallele ai dati
- crea il grafo di esecuzione implicitamente tramite lazy execution

Si occupa anche di ottimizzare la computazione cercando di capire le possibili scorciatoie (fonde operazioni parallele ai dati e forma MapReduce) senza compromettere la validità dei dati ed è, inoltre, un executor che esegue il grafo di esecuzione ottimizzato e si occupa di garbage collection, parallelismo delle attività, tolleranza ai guasti e monitoraggio. Le primitive delle operazioni di base sono:

- ParallelDo(DoFn)
- GroupByKey
- CombineValues(CombFn)
- Flatten

FlumeJava non fa nulla dal punto di vista della gestione dei dati fin quando non viene eseguita la funzione run. Viene inizialmente definito il grafo di computazione, in cui ogni trasformata è un nodo. Ciascuna operazione di liberia può essere mappata su una MapReduce, ad esempio la funzione count in realtà esegue una map, una gbk e una combine. In questo modo è possibile ottenere un grafo di computazione più granulare. A questo punto, Flume può iniziare a mappare il problema in termini di successioni di MapReduce, cercando di individuare in maniera furba le funzioni che possono essere incorporate all'interno di una stessa fase di MapReduce.

## 8.9 MillWheel

Flume e MapReduce sono estremamente efficaci per l'elaborazione dei dati in batch e ciò significa che l'unico modo safe per elaborare nuovi dati di input è rielaborare tutto il set di dati di input, rieseguendo lo stesso MapReduce.

Un team di sviluppo all'interno di Google ha sviluppato una soluzione simile a MapReduce per il caso continuo: MillWheel, un sistema per l'elaborazione di grossi flussi di dati (milioni di eventi al secondo). Il framework di MillWheel consente la creazione di sistemi di analisi in streaming, fornendo un modello di programmazione e un sistema di esecuzione:

- il codice definito dall'utente viene eseguito in Computations
- le Computations sono collegate tramite connessioni chiamate Streams
- i record in ingresso vengono trasmessi come tuple (chiave, valore, timestamp)

L'elaborazione e la distribuzione del processo sono per chiave. I calcoli possono accedere e modificare lo stato della chiave del record corrente, ad esempio per confrontare un nuovo input con un input precedente, impostare ed elaborare i timer, per effettuare ri-computazioni in maniera asincrona nel futuro, e produrre record.

Ciascuna computazione viene distribuite attraverso diverse macchine. Il criterio di distribuzione è molto simile a quello di Bigtable: dato l'insieme delle possibili chiavi in input, le chiavi vengono divise in range e ciascuna macchina si prende cura di un range di chiavi. Quando il coordinatore riceve un dato in input, a seconda della chiave, sa quale computazione eseguire. Se la computazione produce dati in output, avviene nuovamente la distribuzione, cercando quale istanza della computazione successiva ha in carico il range a cui appartiene la chiave, per eseguire la fase successiva della computazione (tramite RPC).

Ogni computazione nel grafo rappresenta l'elaborazione di uno spazio delle chiavi (potrebbero essere diverse funzioni utente). I collegamenti sono percorsi shuffle su

cui scorrono quadruple (chiave, valore, timestamp, numero di sequenza). I timestamp sono determinati dalla sorgente dei dati, non da MillWheel. Inoltre, ogni computazione è suddivisa in intervalli attraverso molti worker. Una singola chiave viene gestita da un singolo operatore alla volta, consentendoci di aggiornare in maniera consistente lo stato persistente per ciascuna chiave.

# Pervasive computing

Il pervasive computing, anche detto ubiquitous computing, rappresenta un'area dell'informatica che è l'intersezione di diverse altre aree fondamentali tra cui:

- reti
- sistemi
- algoritmi
- architetture
- intelligenza artificiale

Gli elementi che caratterizzano il pervasive computing sono smart objects che possono diventare nodi di un sistema distribuito.

"The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it." - Mark Weiser.

Con la parola pervasive si intende la volontà di avere un ambiente che, tramite una modifica attuata dal cumputing, riesca a rispondere alle nostre esigenze. I nodi del sistema che permettono di fare un sensing dell'ambiente e di cambiarlo, sono dispositivi harware e software distribuiti.

## 9.1 Sistema distribuito pervasivo

Un sistema distribuito pervasivo è un sistema distribuito con altre caratterisiche:

- i nodi del sistema sono nodi non convenzionali ovvero sono dei dispositivi con altre funzionalità principali (es. elettrodomestici con capacità di calcolo e comunicazione).
- il sistema è adattivo: la logica del sistema considera il contesto attraverso un ascolto dei parametri dell'ambiente e adatta il sistema per fornire un servizio migliore.

Questi sistemi sono dotati di alta volatilità poiché:

- Potrebbero essere soggetti a fallimenti di dispositivi e di comunicazione molto più che nei tradizionali sistemi distribuiti poiché i nodi potrebbero essere veramente tanti (sensori) e i sistemi di comunicazione diversi.
- In alcune situazioni la banda per la comunicazione può essere limitata
- Aumentando il numero dei dispositivi, aumenta la probabilità che questi si associno o si dissocino dal sistema distribuito (sistema più dinamico che richiede meccanismi di associazione/disassociazione particolari).

## 9.2 Mobile computing

Il pervasive include anche la mobilità attraverso dispositivi integrati in spazi e ambienti differenti (ad esempio l'abitazione, l'ufficio, la città) e spesso questi dispositivi risultano invisibili (non hanno un'interfaccia diretta). Occorre dunque gestire gli smart spaces, quindi stabilire quale tipo di reti instaurare tra questi oggetti, come farli comunicare e in che modo gestirli. Questi sistemi crescono molto velocemente ed è fondamentale, al fine di gestirne la scalabilità, cercare di formare una rete tale per cui nodi vicini comunichino in maniera diretta al fine di ottimizzare lo scambio di messaggi all'interno dell'intera rete.

*Uneven conditioning*: occorre convivere in un ambiente in cui cambia la connettività, in cui alcune cose sono più smart di altre. Occorre progettare un sistema in grado di funzionare anche quando è sconnesso o quando non vi sono condizioni ottimali, che si adatti anche in reti di sensori più primitivi.

Anche nel mobile computing vi sono risorse limitate e problematiche da gestire:

- tipi di interfacce diverse
- varianza nella connettività (ip che cambia)
- possibilità di lavorare in modo sconnesso (con memoria locale e meccanismi di sincronizzazione non appena la connessione è disponibile) e mobile data management

Principali argomenti di ricerca:

- Rete
- Mobile information access
- Mobile data management (privacy e sicurezza, mobile cloud services, LBS)
- Positioning (indoor, outdoor, prossimità, tracking)
- Software (App e mobile services design, develompent and testing, scalabilità)

## 9.3 Pervasive Computing

Il pervasive computing spazia tra nodi mobili, computer embedded, attuatori e sensori connessi ad internet che ascoltano ed influenzano il mondo fisico.

Esempi di pervasive systems:

- smart environment systems
  - Smart home services
  - smart energy management
  - smart transportation (usando crowdsensing attraverso smartphone e sensori)
- e-Health systems per
  - tele-healthcare
  - independent living and ageing well
  - accessibility

## 9.4 IoT

Nell'IoT (Internet of things) sono inclusi quei dispositivi connessi ad internet che sono pensati per fare sensing o interventi. Nell'IoT ci sono moltissimi dispositivi che non hanno un indirizzo IP, partecipano a delle sotto-reti e utilizzano Internet attraverso un gateway. Nonostante il pervasive computing comprenda anche l'internet of things, non tutto ciò che è IoT è da considerare pervasive computing. E' bene sapere che, quando introduciamo nodi in un sistema distribuito, necessitiamo di un coordinamento per fare comunicare i vari dispositivi distribuiti nel modo corretto. Nella maggior parte dei casi questo coordinamento viene gestito in maniera centralizzata.

### 9.4.1 Smart

Molti produttori di tecnologia spacciano loro prodotti di fascia medio-alta come smart. Ma cosa è veramente "smart"?
Di fatto un dispositivo per essere considerato smart non basta che sia connesso ad internet e che sia comandabile da remoto. Un dispositivo smart deve infatti rispettare delle caratteristiche per essere definito tale:

- Connesso ad internet: ovviamente la prima caratteristica è quella più comune, ovvero necessita di una scheda di rete che gli consenta di connettersi in rete così da poter essere acceduto dall'esterno, invocare servizi, etc..
- Capacità di calcolo: il dispositivo smart deve essere in grado di svolgere pre-computazione di dati o utilizzare protocolli per inviare e ricevere dati dal cloud
- Servizi context-aware personalizzati: la vera caratteristica principale di un dispositivo smart è quella di riuscire a capire il contesto in cui si trova e offrire servizi personalizzati in base alla situazione. Ad esempio una macchinetta del caffè che capisce dal tono della tua voce che tipo di caffè vuoi (macchiato, espresso..)

#### 9.4.1.1 "Smart" appliances

Il campo smart si sta sempre più diffondendo in ogni settore, cercando di rendere "smart" sempre più oggetti e contesti. Ad esempio:

Smart vehicles, le quali integrano diverse compinenti:

- centinaia di sensori diversi
- telecamere
- 2+ leader che ricostruiscono in 3d l'ambiente

- 4+ radar
- gps
- attuatori che in base ai dati rilevati e ad un algoritmo decisionale, modificano il comportamento dell'auto in base ai dati rilevati. In questo campo è sempre più necessario l'introduzione del 5g, così da riuscire in tempo reale a far comunicare l'auto con gli altri veicoli e con l'ambiente circostante. Tutti questi dati contribuiscono ad un sistema distribuito.

Altro esempio è quello del sensor network in cui rientrano, ad esempio, tute piene di sensori per la lettura dei parametri vitali.

## 9.5 Applicazione ed ambiti del pervasive computing

Abbiamo visto dunque come il pervasive computing sia un incrocio di diverse aree dell'informatica. Questo ambito è tutt'ora in via di sviluppo e ancora da esplorare a fondo. Ad esempio, alcune nuove questioni ancora da esplorare sono:

- come usare smart spaces in modo utile ed efficiente e dunque adattare il sistema in base al contesto
- cercare di rendere i dispositivi sempre più invisibili e integrati all'interno dell'ecosistema
- dotare ogni nodo di un indirizzo e capacità specifiche e rendere l'integrazione e la comunicazione tra i vari dispositivi il più trasparente possibile. Ad esempio, entrando in una camera di hotel, il proprio telefono che conosce le tue abitudini si associa ai vari componenti smart della stanza e li dirige per far si che la tua esperienza sia il più personalizzata possibile

Ecco alcuni ambiti del pervasive computing:

- modellare le attività umane di alto livello
- crowd sensing
- energia
- pervasive health, tramite dispositivi indosabili o nell'ambiente e l'interazione dei due

- trasporti e ottimizzazione della mobilità
- edge computing, cercare di spostare il calcolo sui dispositivi vicini, soprattutto quando è oneroso
- security e privacy, sono molto facili da attaccare dal punto di vista del software e dell'harware su cui girano. Privacy: le attività svolte ad esempio in casa sono private e se sono condivise in cloud, qualunque attacco al cloud va ad impattare sui dati.

# Acquisizione e gestione dei dati dai sensori

## 10.1 Trasduttore

Sia i sensori che gli attuatori sono dei trasduttori, ossia dei dispositivi che trasformano una forma di energia in un'altra forma di energia. Un esempio banale è il microfono che riceve in input delle onde sonore e manda alle casse un segnale elettronico. Le onde sonore che escono dalla bocca vengono dunque digitalizzate (microfono) e inviate ad un altro trasduttore (cassa) che ritrasforma le onde digitalizzate in onde sonore. Nell'immagine il microfono funge da sensore (trasduttore di input) e le casse da attuatori (trasduttore di output).

## 10.2 Funzionamento dei sensori

Il compito di un sensore è quello di osservare e catturare un fenomeno fisico (onde del parlato, inquinamento in una città, ecc..) per poi tradurlo in un segnale elettrinico. Il sensore è una componente che risiede a bordo di un dispositivo il quale lo integra assieme ad altre componenti.

## 10.3 Principali tipi di sensori

Sensori fisici:

- Sensori di movimento o inerziali: misurano le forze di accelerazione e rotazione rispetto ai diversi assi e dunque monitorano in che modo il dispositivo si muove nello spazio
- Sensori ambientali: Misurano parametri ambientali (fenomeni fisici) in ambiente domotico, come ad esempio la temperatura, la pressione, l'illuminazione, l'umidità, ecc
- sensori di posizione: misurano la posizione fisica del dispositivo. Questa categoria contiene sensori di orientamento e magnetometro

Sensori virtuali:

- Un dispositivo è veramente smart se è in grado di comprendere il contesto e adattarsi al fine di offrire l'obiettivo finale del sistema. In questa acquisizione di contesto partecipano i sensori posizionati sul dispositivo (fisici), ma magari si vuole sapere anche che tempo fa fuori e cosa c'è vicino al soggetto. Un esempio sono le google places api come sensori virtuali. Cosa c'è a queste coordinate? Per la risposta utilizzo una comunicazione remota tramite API. Vengono dunque ottenute informazioni tramite un web service che funge da sensore di contesto. Un'altra tipologia di sensori virtuali sono quei sensori che sono posizionati nell'ambiente attorno al dispositivo in uso e che possono essere utilizzati come se fossero installati sul dispositivo stesso. I sensori virtuali vengono considerati (dal prof) parti del sistema di sensing nonostante siano remoti.

### 10.3.1 Accelerometro e gyro

Se i sistemi distribuiti si sono cominciati a vedere anche nella pratica per lo sviluppo delle reti, i sistemi pervasivi sono stati abilitati dallo sviluppo della sensoristica. Il grande salto è stata la miniaturizzazione dei dispositivi e quindi una diminuzione dei costi.

#### 10.3.1.1 MEMS

I MEMS (Micro Electro-Mechanical Systems) sono stati inventati da un'azienda nei pressi di Milano. Queste componenti harware sono state inventati per caso mentre si stavano svolgendo degli studi sulle cartucce delle stampanti. I MEMS sono dispositivi elettronici implementati su chip con una parte meccanica. Utilizzano una parte meccanica (massa e molle) per misurare l'accelerazione. Lo spostamento delle molle è misurato da una parte elettrica che misura una variazione del campo magnetico che viene generato attraverso dei condensatori. Le invenzioni di queste micro componenti ha rivoluzionato l'ambito della sensoristica e ha permesso l'implementazione a basso costo per vari dispositivi.

### 10.3.2 Sensori ambientali

Sensori in grado di misurare, ad esempio, il passaggio di acqua all'interno di una tubazione o il consumo energetico, etc.

### 10.3.3 Biosensori e biosegnali

Questa tipologia di sensoristica ha visto una nascita grazie ai MEMS per poi evolversi con dispositivi in grado di flettersi, prendendo anche la forma di cerotti o addirittura venendo integrati (sensori e attuatori) direttamente all'interno del corpo.

### 10.3.4 Indossabili (wearables)

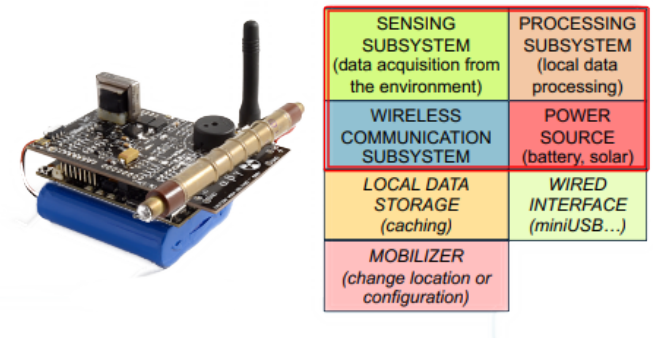
L'espansione della sensoristica e la miniaturizzazione ha permesso un'integrazione dei dispositivi indossabili in tutti i livelli. Ad esempio ci sono già stati progetti importanti della Levis con altre aziende per la creazione di indumenti sensorizzati. Ad esempio il vestito deve capire il contesto e adattare le proprietà del proprio tessuto in base all'utilità del momento. Oppure cambiare colore e proprietà. Questo ambito è ancora agli albori ed è dunque in fase di iniziale espansione.

### 10.3.5 Beacons BLE

I beacons sono piccoli dispositivi che emanano un segnale bluetooth low energy (BLE) i quali possono servire per localizzare un oggetto all'interno del range della tecnologia bluetooth che utilizzano. Questi dispositivi possono venire impiegati per localizzare un utente con uno smartwatch/smartphone anche in assenza di GPS, facendo triangolazioni del segnale bluetooth. La dimensione è data dalla batteria, poiché la parte circuitale è miniaturizzata.

### 10.3.6 Sensing device

Il problema di questi dispotivi di sensing è che non hanno una sorgente permanente di energia e necessitano di batterie. Questi dispositivi hanno una parte di sensing subsystem, che comprende i sensori, e una power source. Questi dispositivi hanno anche le caratteristiche di un sistema distribuito: processing subsystem e wireless comunication subsystem. Hanno inoltre una minima capacità di memorizzare dati, interfacce wired per aggiornare il firmware etc, ma le cose fondamentali sono le 4 precedenti, che caratterizzano ogni dispositivo di questo genere.




### 10.3.7 Smartphone based sensing

Cosa è possibile fare con l'integrazione dei dati dei sensori a bordo dei dispositivi? Quando si sviluppano applicazioni di ogni genere bisogna avere coscienza di tutti i dati che si possono avere a disposizione. E' dunque necessario capire quali sensori sono utilizzabili dal device che l'utente avrà a disposizione perché l'applicazione sarà tanto migliore quanto sarà la sua integrazione con i sensori messi a disposizione.

Tramite gli smartphone e i sensori posti su di essi è possibile fare inferenze su:

- attività (seduto, camminando, incontra amici)
- mood (felice, triste)
- abitudini (in palestra, bar, lavoro)
- ambiente circostante (rumoroso, caldo, luminoso)

### 10.3.8 SmartWatch sensing

Questo tipo di sensing si è poi spostato anche sugli smartwatch che sono in grado di captare molte informazioni: battito puntuale, varianza nel battito, ossigenazione del sangue, tracking di attività, rumorosità dell'ambiente circostante, ecc. Molte di queste informazioni sono utilizzate poi da applicazioni lato smartphone che consigliano, in base ai valori registrati, alcune accortezze da seguire per migliorare lo stato di salute.
(Per capire in maniera concreta come ci si sta muovendo in questo settore, è bene informarsi sul PNRR).

### 10.3.9 Attuatori

Particolare tipo di trasduttore che riceve in input un segnale elettrico (dalla rete wireless) e lo trasforma in un'azione fisica. Necessitano di energia ed uno degli obbiettivi è ridurre al minimo i consumi (Esempi di azioni che compiono: Accendere la luce, aprire una porta...).

### 10.3.10 Reti con sensori e attuatori

Zigbee, z-wave e thread hanno un range tipico piuttosto limitato (tipicamente un ambiente casalingo), anche se possono essere estese. Hanno data rate piuttosto limitati e l'obbiettivo per cui sono state create queste tecnologie di rete è minimizzare il consumo. Funzionano a determinate frequenze (zigbee e thread usano la stessa del wifi, mentre z-wave utilizza frequenze diverse per EU e US). La topologia della rete che prevale è *mesh*, in cui tutti i dispositivi vengono associati al gateway e vengono riconosciuti come partecipanti al sistema. Il dispositivo che partecipa, anche se viene spostato in una zona della casa in cui magari non riesce a raggiungere direttamente il gateway, ha comunque la possibilità di comunicare tramite procedure di heal network in cui i nodi cercano di comunicare in una modalità p2p per raggiungere il gateway utilizzando gli altri nodi come ponte. Questo rende il sistema più scalabile e più affidabile. Sono tutte e 3 principalmente per home automation e sono concorrenti diretti. Sono state proposte varianti quali z-wave lr che dovrebbe quadruplicare il range e decuplicare il

numero di dispositivi associabili nella rete. Zigbee e thread sono open source, z-wave è proprietaria. Thread (zigbee e zwave non associano un indirizzo IP al dispositivo, associano un indirizzo interno alla rete e possono essere considerati in internet attraverso l'indirizzo IP dell'abitazione, passando il comando al gateway) è stato proposto da grosse aziende ed ha l'obbiettivo di rendere ogni dispositivo IP addressable. Questa tecnologia è ottimizzata per IoT. I dispositivi in thread sono divisi in router eligible e lan device: quindi alcuni possono fare da coordinatore ed altri no, favorendo l'affidabilità. Esiste un'iniziativa che si chiama CHIP project (connectedhome over ip) coordinata da zigbee alliance ma a cui partecipano anche googl, amazon e apple e l'obiettivo è risucire a far interoperare dispositivi fatti da aziende diverse in una maniera quasi plug-and-play. Questo vuole essere fatto utilizzando tecnologie thread, ble e wifi integrati in una stessa rete.

| Technology | Range (typical/max) | Max data rate | Power consumpt. | Frequency range (GhZ) | Topology, Max nodes | Main Apps |
|---|---|---|---|---|---|---|
| Zigbee | 10/20 m | 20/ 40/ 250 kbps | Low | 2.4 | star/mesh, 65k | HomeAut, smartGrid, RC |
| Z-Wave | 10/30 m | 9.6/40/100 kbps | Low | 0.868 EU, 0.908 US | mesh, 256 | HomeAut, Security |
| Thread | 10/20 m | 20/40/.../250 | Low | 2.4 | Mesh, IPv6 addressable, 256+ | HomeAut, IOT |
| Bluetooth Low Energy (BLE) | 6/100 m | 1-2 Mbps | Low | 2.4 | scatternet | Audio, persDevice, healthcare, beacons |
| WiFi (802.11n/ac) | 30/200 m | 100-1000 Mbps | High | 2.4/5 | star | LAN |

#### 10.3.10.1 Base stations / Border routers

In queste reti c'è una base station (nodo principale) che ha della potenza di calcolo e che memorizza dati in un database, acquisendoli dai sensori. Può agire come gateway tra reti (zigbee, etc) e wifi/rete internet.

#### 10.3.10.2 Discovery and pairing

Aspetto importante degli ambienti sensorizzati è il discovery and pairing, poiché questi smart devices possono frequentemente apparire e sparire negli smart spaces e questa

cosa dev'essere gestita. C'è anche una parte di netwoork bootstrapping che registra il nodo nella rete assegnandogli un indirizzo unico. L'associazione è piuttosto faticosa e particolarmente importante per quanto riguarda reti z-wave e zigbee e questa si vorrebbe semplificare con thread e nuove soluzioni. Una questione critica dell'associazione è cercare di far associare solo i dispositivi nello spazio a loro designato, anche per questioni di sicurezza. Per questo motivo si usano metodi che sfruttano sensori a bordo del dispositivo. Scatternet: insieme di pico net connesse tra di loro. In un bluetooth lo smartphone fa da gateway bluetooth. Dall'altra parte viene visto il dispositivo bluetooth (telefono) che fa da master e le pico net fanno da slave (es. cuffie bt).

### 10.3.11 Data acquisition

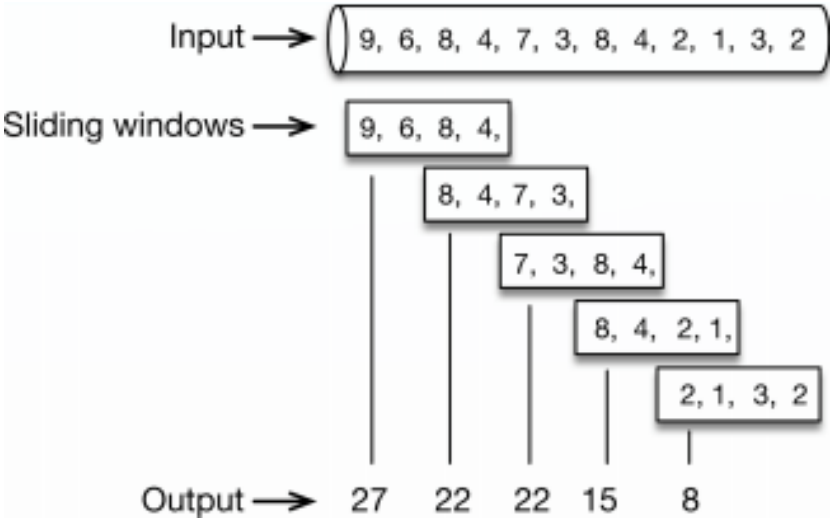
Al fine di cambiare la temperatura in un appartamento (esempio) interrogo dei sensori. Rispetto alle query base, in questo tipo di sistemi vengono vengono utilizzate contiuos query: Ad esempio voglio sapere una certa cosa, ma il fenomeno che voglio sapere è in continua evoluzione, allora si vorrebbero effettuare delle interrogazioni che continuano a chiedere una cosa. Questo ovviamente è particolarmente oneroso in termini di chiamate. Es. Voglio sapere in maniera continua quante persone ci sono in piazza d'uomo. Modo banale: ripeto la stessa query ripetutamente in ogni momento. Questo però è troppo oneroso. In questo ambito c'è una forte caratterizzazione spazio-temporale dei dati. Il problema è come interrogare di continuo senza sprecare risorse. Metodi base

- batch processing: bufferizzare i dati acquisiti dal sensore sul dispositivo ed inviare i dati alla base station o ad un server remoto. Nella base station arriverrano i dati acquisiti dai sensori e verrà effettuato un processing. Questo permette una risposta precisa, mandando i dati così come sono stati acquisiti dal dispositivo. Non viene fatto processing sul sensore. C'è però un alto costo di comunicazione. La base station riceve i dati con ritardi dovuto alla latenza etc.
- sampling: Non vengono inviati tutti i dati acquisiti, ma vengono inviati dati campionati. Il campionamento è fatto secondo la distribuzione attesa dei risultati in modo che siano significativi per il fenomeno che si vuole catturare. Es. Nell'audio sampling è il processo di misurare l'audio continuo ad una certa frequenza. Si considera la distribuzione attesa e lo scopo finale dei dati e così si decide il campionamento. Non offre risposta precisa, ma fornisce garanzie sull'errore che viene introdotto. Se si riesce a fare un campionamento meno frequente si risparmia nella batteria.
- Sliding windows: Si prende un gruppo di letture consecutive di dati e la si considera come una finestra temporale (es. finestra di 3 secondi, guardo tutti i dati, e calcolo qualcosa). Questo è il primo metodo in cui effettivamente c'è del calcolo sul dispositivo che ospita il sensore. Es: Calcolo la media delle letture e

comunico alla base station il valore dell'operatore di aggregazione applicato sulla finestra. La base station riceve uno stream continuo (periodicamente) di valori approssimati con un delay poiché deve aspettare la fine della finestra, calcolare il valore aggregato e inviarlo.

#### 10.3.11.1 Overlapping - Sliding windows

Questi operatori aggregati sono chiamati features. Le window in realtà sono spesso sovrapposte (non comincia una finestra nel momento in cui finisce quella prima, bensì comincia prima che la precedente finisca). Il vantaggio della sovrapposizione consente di catturare qualcosa che sta al confine tra due finestre. Ho una maggiore qualità del dato trasmesso. Nella stragrande maggioranza dei dati inviati da sensori si utilizzano sliding windows.



# Context awareness