

Il Grande Librone di Sistemi Distribuiti e Pervasivi



A.A. 2019/2020

Sommario

Sommario	1
Introduzione	4
Tipi di sistemi Distribuiti (e Pervasivi)	8
Cluster	9
Grid	9
Cloud computing	10
Edge (o Fog) Computing	10
Modelli di comunicazione	11
Socket	12
Comunicazione persistente Message-oriented	13
RPC (Remote Procedure Call)	14
Data streams	15
Architetture dei Sistemi Distribuiti	15
Architetture centralizzate	15
Event-based communication pattern	17
Architetture Decentralizzate	18
Architettura Peer To Peer (P2P)	18
Funzionamento di Peer to Peer	18
Obiettivo del Peer to Peer	18
Problemi del Peer to Peer	19
Overlay Networks	19
Finger Tables	20
Simulazione di un lookup di una risorsa in Chord	20
CAN-Content Addressable Network	21
Svantaggi structured overlays	21
Unstructured overlays: applicazione	21
Architetture Ibride	22
Superpeers	22
Sistemi distribuiti collaborativi (Collaborative Distributed Systems)	22
Introduzione ai microservizi, containers e cloud services relativi ai Sistemi Distribuiti	22
Containers	23
Servizi di piattaforme cloud (Cloud Platform Services)	23
Kubernetes	24
The (old) Google File System (GFS)	24
Architettura e funzionamento GFS	24
Perché GFS?	25
Problemi di sincronizzazione	25

Sincronizzazione usando Orologi Fisici	25
NTP-Network Time Protocol	26
Algoritmo di Berkeley	27
Sincronizzazione usando Orologi Logici	27
Algoritmo di Lamport	27
Multicasting con ordinamento totale	30
Mutua Esclusione	30
Algoritmi distribuiti: Ricart e Agrawala	31
Algoritmo ad anello	32
Algoritmi a elezione	32
Algoritmo Bully	33
Algoritmo di elezione ring-based	34
Fault Tolerance e Consenso distribuito	35
Concetti base della fault tolerance	35
Il teorema di CAP per i DBMS distribuiti	35
Tipi di guasto	36
Ridondanza (Redundancy)	37
Quanta ridondanza è necessaria?	38
Consenso nei sistemi malfunzionanti	38
Byzantine Agreement (accordo bizantino)	38
Impossibilità dell'Agreement in sistemi asincroni	39
Altri topic importanti	39
Google: una lezione sui SD	40
Organizzare l'informazione	40
Protocol Buffers (open source)	40
GFS - Google File System	41
Architettura GFS	41
Colossus	41
Bigtable	41
Spanner	42
F1	43
MapReduce	43
FlumeJava	43
MillWheel	44
DLT e Blockchain	44
Modello del sistema DLT	44
Blockchain	44
Struttura della Blockchain	45
Consenso nelle blockchain	45
Algoritmo Proof Of Work (PoW)	46
Smart contract	47

Tipi di DL	48
Introduzione al computing pervasivo	48
IoT	49
Perché “Smart”?	51
Sistemi Pervasivi: sensori e acquisizione dati	51
Introduzione ai sensori	51
Trasduttori:	51
Trasduttori: I sensori	52
Esempio giroscopio	52
Trasduttori: attuatori	52
Componenti di un sensore	52
Networking con sensori e attuatori	53
Stazioni base / Router di confine / Controller	53
Tipi di sensori	54
Gestione e utilizzo dei dati dei sensori	55
Discovery e pairing del dispositivo	55
Acquisizione dei dati	55
Metodi base	55
Batch processing	55
Sampling	55
Sliding windows	56
Overlapping Sliding Windows	56
Metodi avanzati	57
In-network query processing	57
Duty Cycling	57
Metodi Mobility-based	57
Metodi Model-based (a.k.a. “Data-driven”)	58
Sensor Data Cleaning	59
Model-based Data Acquisition - Acquisizione dei dati	60
Compressione model-based	61
Poor Man’s Compression MidRange (PMC-MR)	61
Approssimazione Multimodello	62
Context awareness	62
Cos’è il contesto?	62
Definizione di Contesto	62
Cosa contiene un contesto?	63
Adattività	64
Tipi di adattività	65
Ottenere i dati dal contesto	65
Metodi di Inferenza	66
Activity Recognition	67

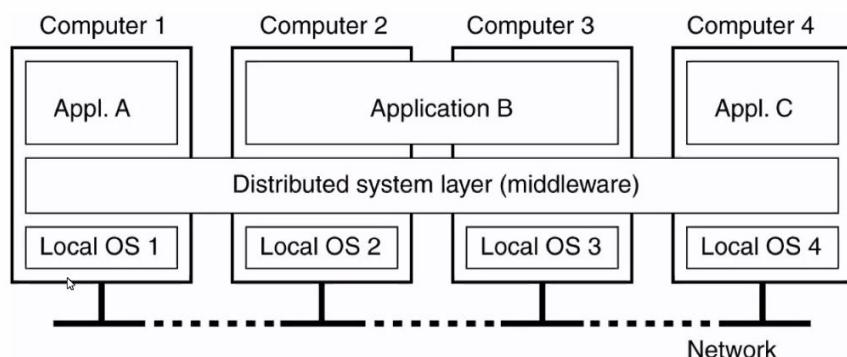
Ragionamento Ibrido nell'activity recognition	67
Rappresentazione del contesto	68
Modelli di tipo flat	68
Modelli DB-based	69
Modelli ontologici	70
Incertezza	71
Supporto all'incertezza	73
Supporto al Middleware (Context Provisioning)	74

Introduzione

Che cos'è un Sistema distribuito?

Un sistema distribuito (SD) è una collezione di computer indipendenti tra loro che appaiono agli utenti come un singolo sistema coerente. Per realizzarlo occorre progettare il sistema operativo, il database e i sistemi di comunicazione in modo differente rispetto a quelli centralizzati. La coerenza di un SD è molto importante, ed è la parte di progettazione più difficile.

Un sistema distribuito è visto come un middleware:



Nella foto abbiamo una rete a cui sono connessi 4 nodi, che rappresentano 4 computer differenti. Ognuno di essi ha un OS locale, su cui girano delle applicazioni. Un SD si interpone tra l'OS locale e le applicazioni, per dare l'illusione che i 4 nodi siano in realtà un unico nodo su cui girano tre applicazioni.

Ad esempio, quando interagisco con GDocs, interagisco con l'editor, che può girare su una o più macchine trasparentemente a me.

Un Sistema Distribuito ha le seguenti caratteristiche:

- **Rendere trasparenti le risorse (Transparency):**

La trasparenza è fondamentale negli SD, perché permette all'utente di credere di stare utilizzando una sola applicazione che gira su una sola macchina, e non l'intero sistema.

Di seguito vi sono tutte le proprietà che un SD per garantire trasparenza, esse sono tutte desiderabili e quindi non può soddisfarle tutte:

- **Accesso:** nascondere le differenze nella rappresentazione dei dati e il metodo per accedere a una risorsa;
- **Ubicazione** (Location): nascondere dove è memorizzata una risorsa;
- **Migrazione:** nascondere l'eventuale spostamento di una risorsa;
- **Rilocazione** (Relocation): nascondere lo spostamento di una risorsa mentre viene utilizzata;
- **Replicazione** (Replication): nascondere che una risorsa è replicata in più luoghi diversi (per esigenze di efficienza / fault tolerance o ridondanza);
- **Concorrenza:** l'utente non deve accorgersi che ci sono altri utenti che usano la stessa risorsa, sia in fatto di consistenza (non ci devono essere incongruenze tra files) che di performance (non devo perdere performance ad es. replicando troppi dati). La principale differenza tra la concorrenza distribuita e quella centralizzata è che nel primo caso non vi è una memoria condivisa, quindi deve esserci un continuo scambio di messaggi al fine di mantenere la sessione;
- **Fallimento** (Failure): nascondere il fallimento/recupero di una risorsa (ad es. un NAS che funziona anche quando un disco si rompe);
- **Apertura (Openness):** un sistema distribuito aperto dovrebbe offrire
 - interoperabilità;
 - portabilità;
 - estensibilità;
 che possono essere ottenute da:
 - l'uso di protocolli standard
 - pubblicazione di interfacce chiave (API);
 - testing e verifica della conformità dei componenti del sistema con gli standard pubblicati.
- **Scalabilità:**
 la scalabilità è la proprietà che permette di rendere un SD più fruibile, ci sono sistemi scalabili e non. Per ognuno di questi ci sono dei problemi che vanno risolti:
 - ✗ Servizi centralizzati: se uso un solo server per tutti i servizi, al primo crash esplode tutto. Ho bisogno di più server decentralizzati.
 - ✗ Dati centralizzati: Sempre **SPoF**: se uso un solo DB su una sola macchina per memorizzare i dati, se crasha perdo tutti i dati, la soluzione è quindi "dividere" le risorse in più parti distribuite tra più device, ogni richiesta andrà a recuperare quella parte nel dispositivo in cui è memorizzato (data replication).
 - ✗ Algoritmi centralizzati: Anche se si ha una completa definizione dello stato del sistema, fare routing basandosi su informazioni complete non è plausibile per i SD: i nodi connessi potrebbero disconnettersi dalla rete invalidando il processo di mappatura della stessa. Occorre dunque decentralizzare gli algoritmi con tecniche specifiche.

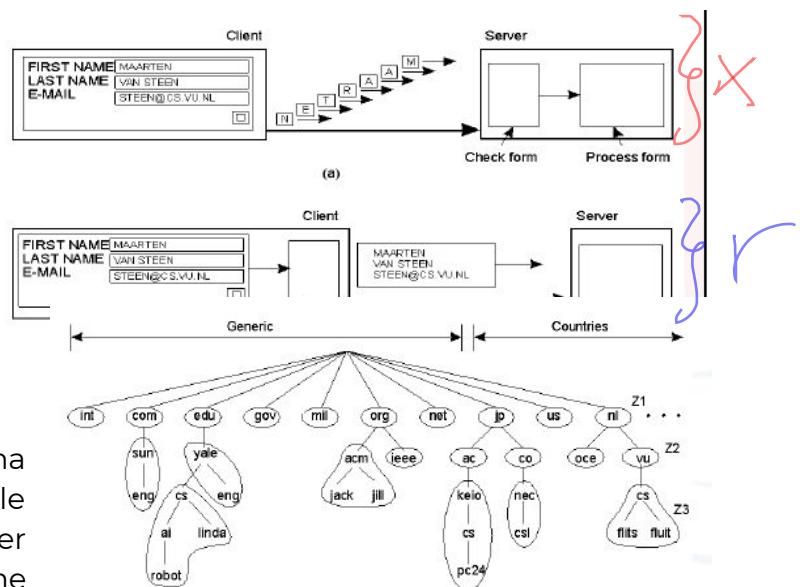
Gli algoritmi **decentralizzati** hanno diverse caratteristiche:

- Ogni macchina esegue lo stesso codice delle altre: non c'è un coordinatore (in realtà si, ma vedremo in seguito);
- i nodi prendono decisioni (ad es. per fare routing) usando informazioni incomplete sulla rete (ad es. conosce solo i nodi adiacenti ad esso e non tutti quanti i nodi). In questo modo, il fallimento di una delle macchine non "rovina" l'algoritmo, in quanto viene sostituita da un'altra che fa la stessa cosa.
- Questo **decentralizzazione** crea nuovi **problemi di sincronizzazione** tra le macchine, in quanto non si può fare un'assunzione implicita che esista un clock comune a tutte le macchine!
- Riassunto scalabilità:** evito di centralizzare servizi, dati e algoritmi, ma li distribuisco su più macchine, così evito anche il SPoF (single point of failure).

Tecniche di scaling:

Si noti una grande differenza nel lasciare a un client il compito di fare check dei form invece che al server: se lo facessimo fare al server sarebbe avremmo molti più dati da processare e quindi più costo in spazio e tempo del server. Invece nel caso B) possiamo vedere come ciò che ci arriva è già compilato e computato dal client, perciò non dobbiamo sprecare risorse extra del server. Il caso b) è meglio di a) per la scalabilità!

Un esempio comune di sistema scalabile è il DNS: tante macchine forniscono un servizio unico con dati distribuiti. All'inizio funzionava con un singolo file, che non è una soluzione scalabile. Ora le macchine collaborano tra loro per risalire all'IP assegnato al nome simbolico.



Pitfalls comuni in cui un progettatore di SD non deve cadere:

- La rete è sicura;
- La rete è affidabile;
- La rete è omogenea;
- La topologia della rete non cambia;
- La latenza è sempre 0;
- La banda è infinita;
- Il costo del trasporto è 0;
- C'è un solo amministratore di rete;
- Il debug è come quello per le app normali

(Nel progetto di SDP sarà fondamentale la gestione della concorrenza!)

Tipi di sistemi Distribuiti (e Pervasivi)

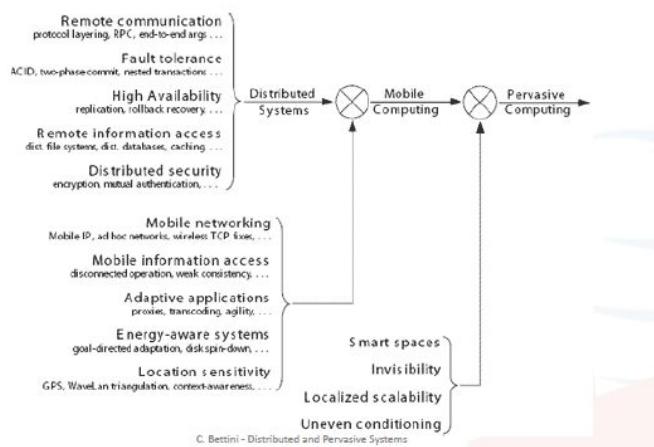
1. Sistemi Distribuiti per il Computing
 - a. Clusters
 - b. Cloud Computing
 - c. Edge Computing
2. Sistemi Informativi Distribuiti (Non trattati nel corso)
 - a. Distributed Databases
 - b. Distributed Transactions
3. Sistemi Distribuiti Pervasivi

Un Sistema Distribuito Pervasivo è un sistema che include dei nodi non convenzionali, come smartphone, network di sensori, ecc. Esso deve essere adattivo, ovvero deve adattare il suo comportamento analizzando il contesto al fine di perseguire il goal richiesto.

Ad esempio il mio telefono mi fornisce diverse informazioni in base a dove mi trovo. Sistemi distribuiti e pervasivi sono ad esempio quelli per smartHome, autonomous driving, ecc.

Nella figura si nota come il mobile e il pervasive computing siano evoluzioni dei SD. Essi aggiungono nuovi e differenti problemi (ad es. la gestione dell'indirizzo mobile degli smartphone). Nel caso del pervasive computing, l'utente si muove in uno spazio smart con devices invisibili (senza interfaccia), questi devono comunicare localmente per essere scalabili. Infine, vi è un problema di cloud conditioning: i devices sono diversi e non eterogenei. Nella slide si vede come i nodi \otimes indicano una complessità dei problemi aggiunti moltiplicata rispetto al passo precedente.

Passiamo ora a descrivere i tipi di sistemi di computazione distribuita, che vedremo nel dettaglio durante il corso:



Cluster

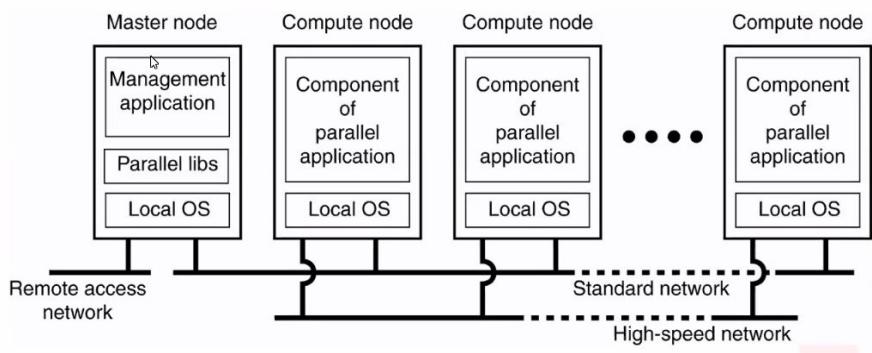
I cluster sono insiemi di computer indipendenti aventi caratteristiche simili, spesso con lo stesso OS, vicini gli uni agli altri e collegati tramite una rete locale ad alta velocità, in questo modo si hanno delle garanzie di latenza. I goal principali da raggiungere sono:

- a. Calcoli ad alta performance;
- b. Alta disponibilità;

I cluster vengono usati di solito per il calcolo scientifico, in quanto unendo macchine simili tra loro possiamo effettuare calcoli complessi in modo rapido.

Esistono due tipi di cluster:

- **Asimmetrico:** Un nodo master controlla e coordina tutti gli altri nodi, un esempio è Google Borg in cui tutti i device controllati da un singolo master sono contenuti in una cella avente la grandezza media di 10K computer, inoltre per mantenere il servizio vi sono più master che entrano in gioco nel caso in cui cada chi ha il comando.
- **Simmetrico:** Tutti i nodi hanno lo stesso SW e possiedono dei modi per controllare l'interazione, un esempio è il P2P.



Nella figura vediamo un esempio di cluster asimmetrico.

Grid

I grid sono cluster legati ad altri cluster, formando una griglia di dispositivi.

I nodi sono più eterogenei in hardware e in software rispetto a quelli in un cluster (un nodo potrebbe essere un telescopio spaziale usato come risorsa condivisa, ma anche NAS supercomputer ecc). Il cloud computing nasce da un'idea usata nelle grid systems: si fornisce solamente la potenza di calcolo che è richiesta dall'utente (on Demand), come nella rete elettrica si fornisce solo l'energia richiesta.

Cloud computing

Evoluzione del concetto del Grid system, come **Software as a service** (≠ dalle licenze!), permettendo ubiquità, convenienza e accesso on-demand alle reti al fine di condividere le risorse. La chiave del cloud computing è l'**elasticità**: la potenza di calcolo viene scalata in modo automatico, in base alle mie necessità. Il provider possiede diverse pool di calcolatori che distribuisce agli utenti. Un altro punto fondamentale è l'**accessibilità continua attraverso la rete**.

Il cloud pone diversi servizi a disposizione:

- **Software as a service**: software offerto via cloud ad un utente, come GDocs. Non devo gestire aggiornamenti e macchine.
- **Platform as a service**: infrastruttura software con la quale è possibile sviluppare applicativi, ad esempio un database scalabile di amazon che uso come pezzo della mia app. L'accesso a questa infrastruttura è garantita da API esterne la cui gestione non è a carico del programmatore;
- **Infrastructure as a service**: ambiente completo di sviluppo e deployment di applicazioni, come le macchine di AWS, preferibilmente espandibile e scalabile.

Ed esistono diversi tipi di cloud:

- Privato: creato ed usato per una singola organizzazione/azienda.
- Ibrido: parte di servizi su un cloud privato, l'altra su pubblico (ad es. registrazioni lezioni di Zoom sono su cloud privato).
- Community cloud: cloud disponibile solamente a una certa community/pool di aziende.
- Pubblico: accessibile a tutti!

La tecnologia per gestire il cloud è un'infrastruttura che connette più cluster tra loro permettendo lo switching e il balancing.

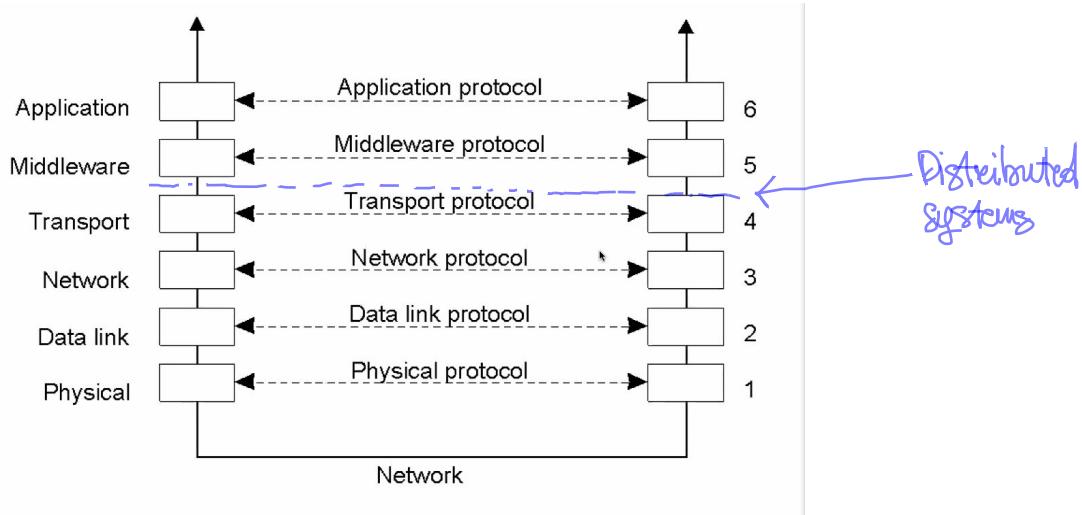
Edge (o Fog) Computing

riguarda i dispositivi che producono dati in massa, la loro elaborazione deve quindi essere real time, eseguita il più vicino possibile a dove vengono prodotti.

Es. Collezione di sensori IoT devono comunicare velocemente, senza passare dal cloud ma attraverso reti più vicine. In questo modo evito dei ritardi grossi di latenza perché non passo dal cloud. Si chiama Fog perché uso un livello di decentralizzazione tra cloud e Edge, una sorta di "nebbia" (fog) tra le due.

Modelli di comunicazione

La maggior parte dei sistemi distribuiti risiede fra il livello di trasporto (TCP/UDP) e il livello del middleware.

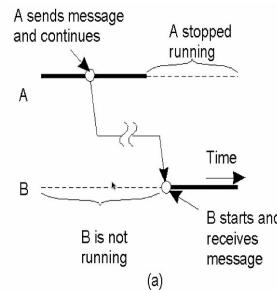


Lo stack dei sistemi di rete.

Persistenza: Sistema che garantisce la consegna di un messaggio a destinazione. Nel caso il destinatario non possa riceverlo, posso rimandarlo un po' più tardi o consegnarlo a un buffer del destinatario. Il suo duale è la **transienza** (Se il server è giù non ricevo risposte).

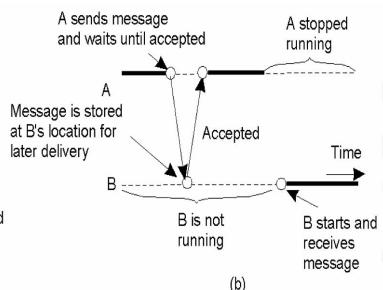
a. **sistema persistente asincrono:**

non aspetta alcuna risposta, c'è un sistema di memorizzazione nel mezzo.



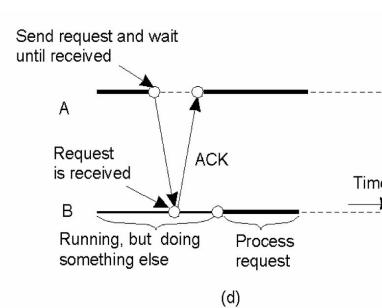
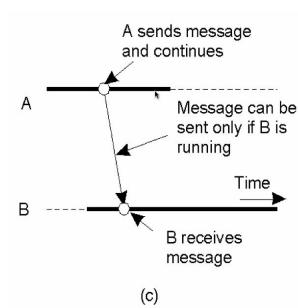
b. **sistema persistente sincrono:**

in questo caso non continuo l'esecuzione finché il messaggio non viene accettato, nel caso in cui B sia spento, un sistema di comunicazione esterno permette ad A di continuare l'esecuzione (come una "cassetta della posta" esterna a B).



c. **Comunicazione transiente asincrona.**

Esiste una primitiva che conserva i pacchetti;



d. **Comunicazione transiente sincrona.**

Aspetto che il server mi notifichi che ha ricevuto la mia

richiesta. Di solito il server riceve il messaggio dal client, ma gestisce la richiesta in un altro momento dato che potrebbero essercene altre in esecuzione.

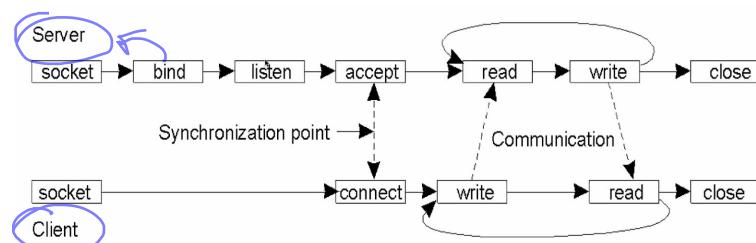
- message based*
- e. **Comunicazione sincrona alla consegna del messaggio.** Il client attende fino a quando il server gli notifica che ha accettato la sua richiesta
 - f. **Comunicazione transiente sincrona response based.** Il client è contento quando il server finisce di processare la sua richiesta.
-

Socket

Il socket è un sistema di comunicazione transiente message-based, che sta sopra al livello TCP/IP.

Per aprire una connessione fra due nodi, entrambi devono aprire una socket. Una socket usa le seguenti primitive:

- **Socket:** inizializza un nuovo punto di comunicazione creando un file descriptor indicante un endpoint, esso è utilizzato per la comunicazione effettiva;
- **Bind:** viene attaccato un indirizzo locale al socket tramite TCP/IP, quindi assegno una porta della macchina all'endpoint creato in precedenza;
- **Listen:** Prepara un'area di memoria per accodare i messaggi e le richieste di connessione. Al loro arrivo, vengono accodate in ordine di arrivo;
- **Accept:** blocca l'esecuzione del codice finché nella coda del listen non arriva una richiesta di connessione;
- **Connect:** cerca di stabilire una connessione a un host (IP) e a una porta;
- **Send, Receive:** invio e ricezioni di messaggi tra una e l'altra macchina;
- **Close:** chiude la connessione tra i due endpoint;

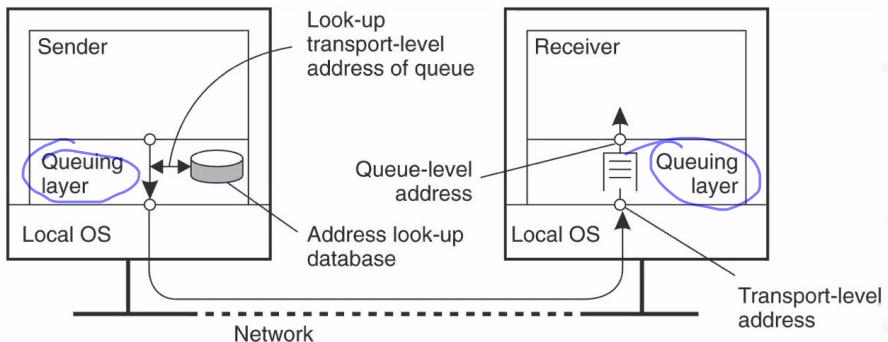


Schema di connessione fra socket. La bind del client è effettuata in automatica dal sistema.

Il sistema che usa le socket per comunicare è **transiente** perché la Connect non funziona se il sistema è offline, inoltre è message-based a causa delle primitive di comunicazione Send e Receive. E' possibile notare che non è presente la system call `bind()` nel client, questo perché il sistema operativo si preoccupa di assegnare un numero di porta quando la socket viene creata, rendendola di conseguenza non necessaria.

Comunicazione persistente Message-oriented

Paradigma di comunicazione comodo e molto usato per avere sistemi di persistenza nella rete.



Nella foto vediamo un sistema di code. I gestori di code sono gestiti sia dal receiver che dal sender, ma anche dal router a livello intermedio. Il grosso vantaggio di questi router è che il messaggio mantiene la persistenza anche se il sender è spento, cioè è possibile grazie ai sistemi a code, dei meccanismi che si mettono in mezzo a mittente/destinatario per interfacciarsi col livello Trasporto. In questi sistemi c'è inoltre un meccanismo aggiuntivo che funge da smistatore di messaggi al fine di gestire meglio la rete ma soprattutto la persistenza. Altro vantaggio dei router è che possono fare da interprete per sistemi che parlano in modi diversi (broker). Le operazioni fondamentali sono :

- **PUT**: appende un messaggio ad una coda;
- **GET**: blocca la coda per prendersi un messaggio in testa;
- **POOL**: controlla se è presente un messaggio;
- **NOTIFY**: installa un handler che verrà chiamato quando un messaggio viene inserito nella coda specifica.

Quando è meglio usare questi sistemi rispetto ai socket?

- Quando si vuole comunicazione asincrona;
- Quando la scalabilità può essere aumentata, ammettendo dei delay in richiesta e risposta;
- Quando il produttore è più veloce del consumatore;
- Quando ho bisogno di implementare il paradigma publish-subscribe;

In Java si può usare ad es JMS (Java Messaging Service).

Alcuni protocolli che usano questo paradigma sono:

- XMPP: sistema open, usato per lo più per instant messaging;
- MQTT: modello publish/subscribe, molto leggero, usato da FB messenger e per i sistemi di IoT;

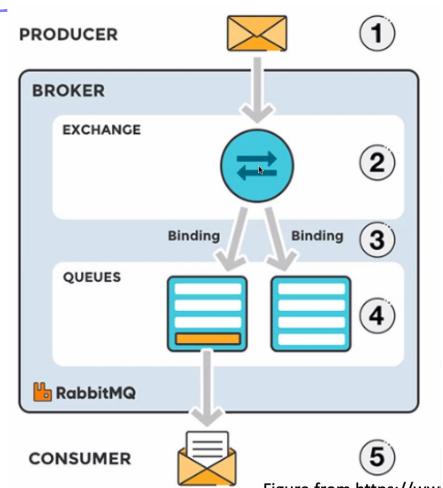


Figure from <https://www>

- AMQP: general purpose, usatissimo, è implementato open source da **RabbitMQ** che supporta più linguaggi e protocolli. Ha infatti un broker al suo interno, e per far comunicare le varie componenti usa AMQP, come vediamo nella figura sotto;

RPC (Remote Procedure Call)

Sarebbe molto bello riuscire a distribuire l'esecuzione di calcoli e procedure su vari nodi nel sistema, senza dover gestire lo scambio di messaggi a basso livello, in modo trasparente. In pratica, dovrei poter avere una chiamata a una funzione distribuita come se fosse una chiamata a una funzione locale.

Un esempio: sto calcolando, e voglio richiamare una moltiplicazione fra matrici in un nodo specializzato che lo fa in maniera particolarmente efficiente.

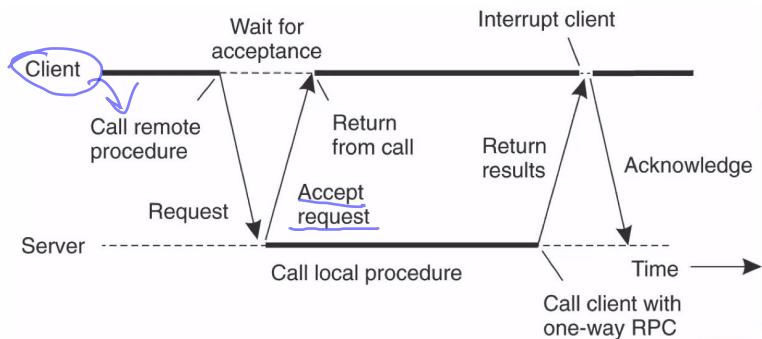
Ciò pone alcuni problemi: come faccio a chiamare una funzione con parametri passati per riferimento(es. l'indirizzo del buffer)?

Sostanzialmente non si può, devo passare per valore, o comunque convertirli e serializzarli (**Marshalling**) e farli deserializzare al server (**Unmarshalling**).

Nella figura vediamo uno schema di chiamata a funzione remota sincrono. si può anche fare in modo asincrono.

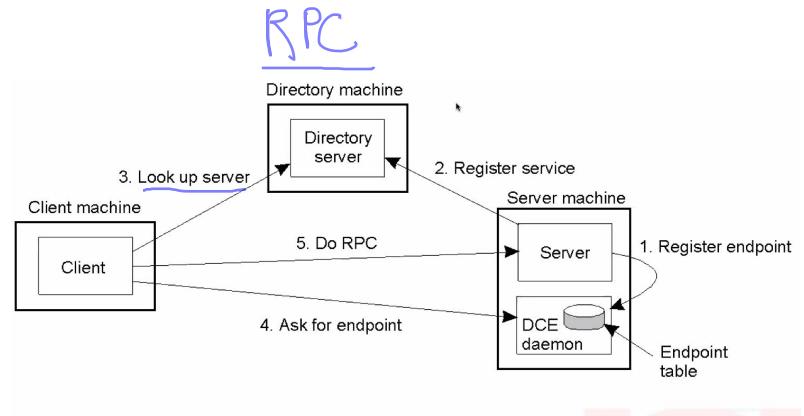
Per ottenere un risultato trasparente, uso i cosiddetti **stubs**, delle porzioni di codice in grado di chiamare la procedura remota inviando un messaggio al server, ricevendo una risposta una volta che quest'ultimo l'ha calcolata. Ma come faccio a fare il binding senza sapere IP, porte ecc? Esiste un server pubblico, il **directory service**, conosciuto sia dal client che dal server, al quale il client chiede l'indirizzo del server richiesto. Una volta scoperto l'indirizzo, contatto il daemon che gira nel server e lo risveglia in modo che mi possa restituire il suo endpoint. Questa cosa si chiama **binding RPC**, come mostrato nella figura sotto.

Un meccanismo simile a RPC è la Remote Object Invocation, la differenza sta nel fatto che i metodi chiamati in remoto vanno a modificare lo stato di un oggetto anziché restituire un risultato.



Data streams

Insieme di dati coerenti che arrivano da una sorgente utilizzate per trasferire i pacchetti nel minor tempo possibile e con la miglior qualità del servizio, un fattore importante per garantire tutto questo è la loro distanza. Ad esempio, se sono in streaming da un media server, avrò più canali audio e video che dovrò sincronizzare, ad es. con dei buffer.



Architetture dei Sistemi Distribuiti

Cosa definisce un'architettura di un sistema distribuito?

- le entità principali del sistema: quali sono i processi o thread che girano sui nodi, quali sono i nodi sensori e attuatori nei sistemi pervasivi, gli oggetti e componenti, ecc.
- il pattern di comunicazione usato: es. publish/subscribe, o socket...
- la comunicazione: definisce i ruoli delle varie entità nell'applicazione;
- come le entità vengono mappate nell'infrastruttura fisica, ad es. replicazione, clustering, caching, ecc.

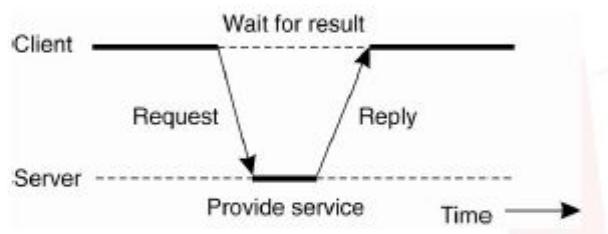
I tre diversi tipi di architetture che vedremo si distinguono in base a come assegnano i ruoli alle componenti che li compongono:

1. Architetture centralizzate;
2. Architetture decentralizzate;
3. Architetture ibride;

Architetture centralizzate

Le architetture centralizzate seguono l'ormai visto e rivisto paradigma **client-server** basato sul sistema di request-reply: quando il client fa una richiesta, si mette in attesa della risposta dal server (**sincrono**).

Vediamo nella figura come funziona una normale interazione sincrona tra client e server.

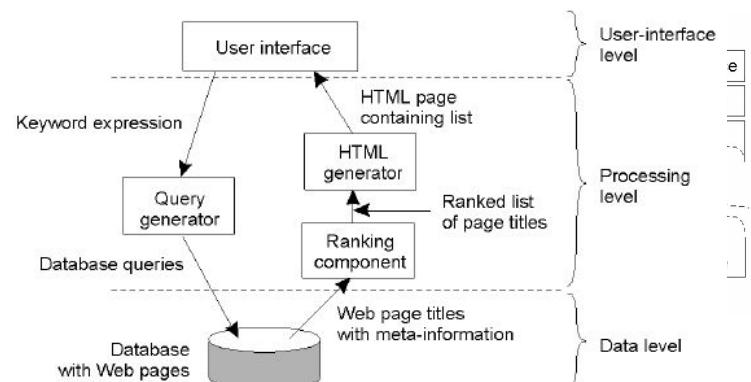
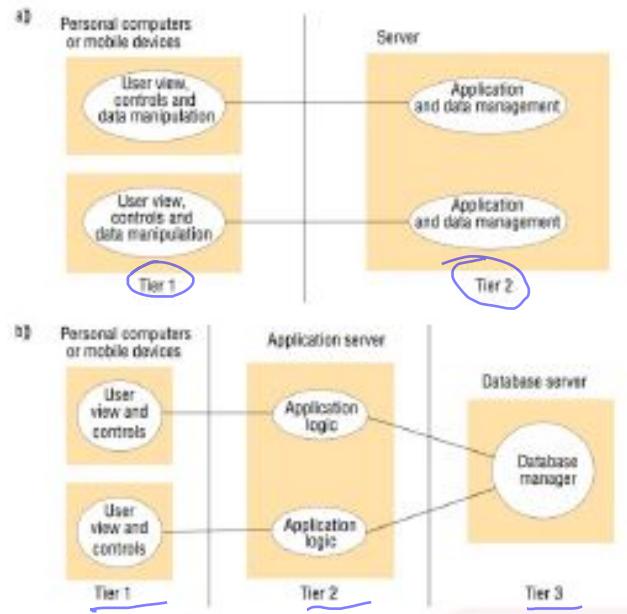


Le architetture client server sono multi-tier, come nella figura sottostante, dove possiamo vedere un esempio di 2-tier (a in figura) e 3-tier (b in figura).

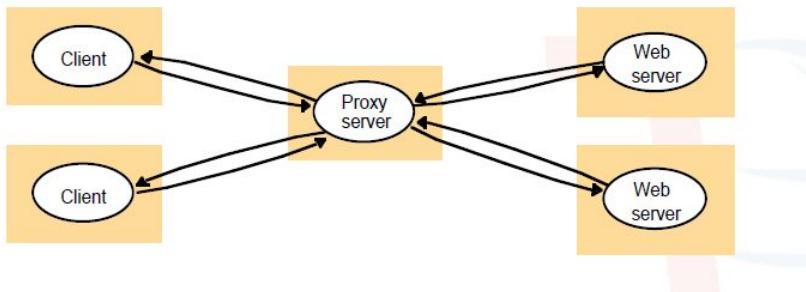
Nel 2-tier ho i client che si collegano direttamente al server, nel 3-tier i client si collegano a un application server a cui mandano le richieste. Quest'ultimo è collegato a un DB server, e in questo modo l'application server fa da client al DB, e da server all'user client.

Sulle slides possiamo vedere un esempio di architettura per un motore di ricerca. Posso avere anche architetture un po' più stratificate (ad esempio PC Utente + server con più istanze di un sito web + Database), come mostrato nella figura.

E' possibile inoltre organizzare i client e i server in diversi modi, dividendo la computazione tra client e server. In base alla situazione, posso decidere di lasciar fare tutto al server e lasciare sul client solo la UI (1° caso a sinistra chiamato dumb client), fino a posizionare la UI, l'applicazione e una parte di DB (opportunamente sincronizzata con quella sul server) a lato client mentre a lato server mantenere il DB (o una parte di esso), usata spesso nel caso di app mobile con DB consultabili anche in offline.



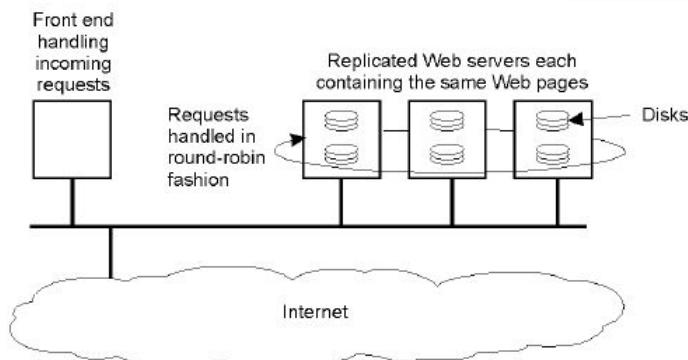
Per ottimizzare la latenza e diminuire il carico sui server, posso avere nel mezzo dei proxy server che fanno caching dell'informazione.



Le funzionalità del client-server possono essere distribuiti in più modi:

- Distribuzione verticale: assegno a ogni server una funzionalità diversa (assegno un nodo a ogni livello, es: i 3-tier).
- Distribuzione orizzontale: distribuisco la stessa funzionalità su un solo livello: ogni server ha quindi la stessa funzionalità da gestire.

Le distribuzioni sono ortogonali tra loro: dentro a una distribuzione verticale posso avere una distribuzione orizzontale, ad es. posso avere tanti server che replicano più pagine web.



Event-based communication pattern

client $\xrightarrow{\text{event - based}}$ event based

Oltre al paradigma di request-reply, vi è un altro paradigma client-server: l'event based. Esso utilizza il meccanismo di publish/subscribe, dove alcune componenti generano degli eventi e inviano a chi si è messo in ascolto su di essi (subscribe) un messaggio (publish) di notifica che l'evento è avvenuto. L'ascoltatore reagirà di conseguenza quando riceverà il messaggio, ciò vuol dire che questo tipo di comunicazione è asincrona, l'ascoltatore infatti può momentaneamente svolgere un altro compito finché non riceve il messaggio.

Architetture Decentralizzate

Disclaimer: nella realtà si usano sistemi ibridi, con alcuni componenti centralizzati e decentralizzati

Architettura Peer To Peer (P2P)

Insieme di nodi “paritari”: tutti i nodi si comportano sia da client che da server, e ogni utente contribuisce condividendo le risorse richieste (es. file condivisi). In questo modo non esiste nessun sistema centralizzato da cui dipendono le operazioni (P2P puro).

Un esempio di servizio che utilizza P2P (non puro, ha delle unità centralizzate replicate su molti directory server) è Napster, un vecchio sistema di music sharing. E' stato il primo esempio di funzionamento di P2P, smontando le teorie che sostenevano fosse impossibile implementarlo e mostrandone l'efficienza in quanto milioni di utenti lo hanno usato.

Sono poi nati altri protocolli per aumentare alcuni aspetti, come Gnutella (completa decentralizzazione, gossip protocols), FreeNet (privacy), bitTorrent (scalabilità, i files sono frammentati in pezzi che posso scaricare anche da chi non ha finito di scaricarli del tutto).

Esiste anche una terza generazione di protocolli, più general purpose, che non si limitano a una funzione singola ma forniscono più servizi e API, come Pastry, Tapestry, Chord ecc.

Funzionamento di Peer to Peer

1. Un peer richiede al server che vuole una certa risorsa.
 2. Il server risponde con una lista di peer che possiedono il file.
 3. Il peer sceglie uno dei peer sulla lista e gli richiede il file.
 4. il peer risponde col file.
 5. Il peer che ha ricevuto il file notifica al server che ha terminato la ricezione, che lo salva nella lista dei peer che possiedono quel file.
-

Obiettivo del Peer to Peer

- Un client deve, in maniera trasparente:
 - Trovare e comunicare con ogni risorsa (lookup).
 - Aggiungere o rimuovere risorse (è il sistema che decide dove ubicarla, per scalabilità ecc.).
 - Aggiungere o rimuovere peers, a prescindere dal sistema, senza dover riconfigurare la rete. *potrebbe essere molto costoso*

Problemi del Peer to Peer

- Come possiamo distribuire i dati su vari host, per distribuire il carico equamente e diminuire la latenza globale, aumentando la disponibilità delle risorse evitando l'overhead?
- Come possiamo farlo in modo da raggiungere gli obiettivi sopracitati?

La risposta è usare le Overlay Networks.

Overlay Networks

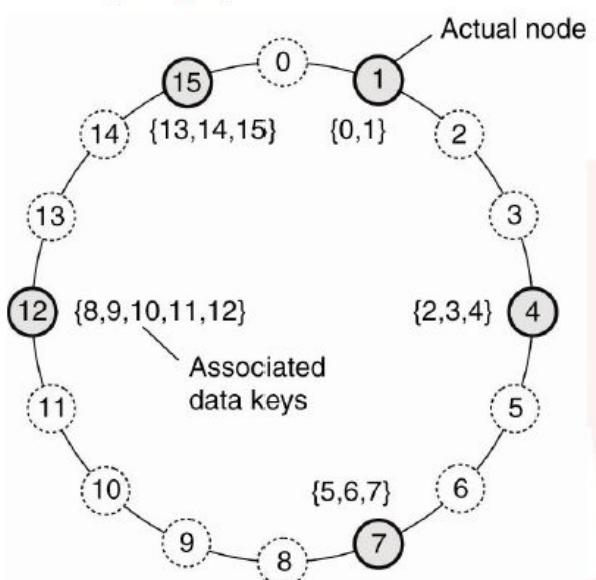
Una Overlay Network è una rete logica costruita sopra ad una rete fisica (ad esempio sopra internet). Organizzo i peer in una rete di questo tipo seguendo un certo schema logico, ad esempio ad anello (chord). In questo modo, pur essendo in rete collegati in modo diverso, grazie agli algoritmi distribuiti (routing overlay algorithms) che sfruttano la struttura della rete logica possono individuare e consegnare le risorse al richiedente.

Il routing avviene dunque a livello logico, non fisico (a livello di IP): si basa invece su identificatori globali, diversi dalla coppia (IP, porta), che mi indirizzano verso la risorsa (o una sua replica) richiesta.

Ci sono due tipi di Overlay Networks:

- **Structured:** La rete di overlay viene costruita in modo deterministico, spesso con hash tables, per ottenere un routing efficiente. Ogni peer conosce l'intera struttura della rete.
- **Unstructured:** La rete di overlay viene costruita in modo non deterministico, usando algoritmi randomizzati (la struttura cambia dinamicamente). Ogni peer conosce solamente i suoi vicini, e perciò possiede una conoscenza parziale del mondo, e propagherà la richiesta della risorsa agli altri peer con cui comunica (se non la possiede lui). In questo caso si risolve il problema dell'overhead per ricreare la rete se questa cambia frequentemente, ma c'è bisogno di un modo per fermare la propagazione della richiesta quando questa è arrivata (introducendo un TTL nella richiesta per limitare il numero di hop che essa fa).

Un esempio di sistema che usa una rete di overlay strutturata è Chord, che utilizza una rete ad anello basata sulle DHT (Distributed Hash Table). Si fissa uno spazio di indirizzamento da 1 a N bit (di solito 128-160, nella foto N=4 bit), e a ciascun nodo assegno uno degli indirizzi in questo spazio. Ad ogni risorsa si assegna un indirizzo preso dallo stesso spazio di indirizzamento usato per i nodi, ed è gestita (memorizzata) nel nodo che



possiede l'id più piccolo maggiore di quella risorsa (ad es. il nodo 4 in figura possiede le risorse 2, 3 e 4, ma non la 5 che è maggiore di 4 e quindi va a finire nel nodo più piccolo maggiore di essa, cioè 7).

Il lookup di una risorsa è descritto dalla funzione

$$\text{Lookup}(k) = P$$

dove k è l'id della risorsa ricercata e P è l'id del peer che la possiede, chiamata anche $\text{succ}(p)$.

Finger Tables

Per rendere il lookup di una risorsa efficiente (descritto dalla funzione $\text{lookup}(k)$), utilizzo una tabella di hash distribuita (DHT) memorizza in ogni nodo una finger table contenente i puntatori a dei nodi della catena per non rifare sempre il giro che contiene fino a m righe, dove m=numero dei bit nella chiave di hashing.

L'-esimo elemento della FT del nodo n conterrà il successore $\text{succ}(n + 2^{i-1}) \bmod 2^m$.

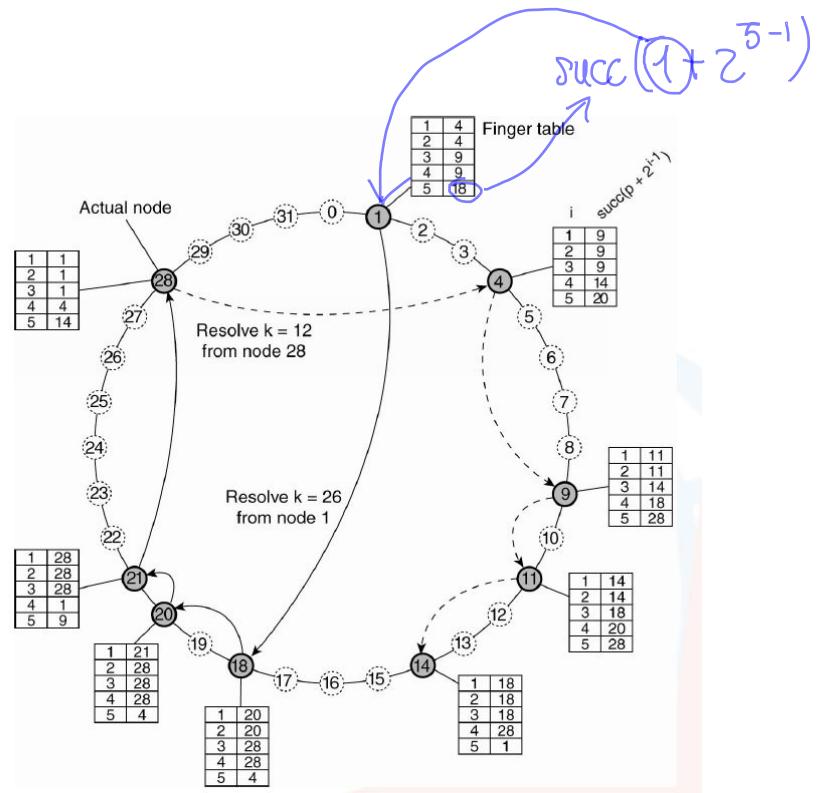
La prima riga di una FT del nodo n sarà il successore immediato di n.

Simulazione di un lookup di una risorsa in Chord

Quando un nodo vuole fare il lookup di una chiave k , passerà la sua richiesta al nodo più vicino che precede o segue k nella sua finger table. Questo passaggio della richiesta avviene finché un nodo che ha ricevuto la richiesta scopre che k è memorizzata nel suo successore immediato (memorizzato nella prima riga della sua FT): quel nodo avrà al suo interno k .

Nell'esempio, il nodo 1 richiede al nodo 18 di trovare $k = 26$, dato che nella sua FT il nodo 1 ha come elemento più vicino a 26 l'ID 18 (la scelta di prendere l'ID che precede o segue k dipende dalla FT).

Il nodo 18 vede che il nodo immediatamente precedente a 26 è 20 nella sua FT, perciò inoltra il messaggio al nodo 20, che fa la stessa cosa con 21, che si accorge che il prossimo nodo che ha sulla FT è 28, l'immediato successore di 26. Quindi invia la richiesta a 28, che avrà al suo interno $k = 26$.



updated and Pervasive Systems

CAN-Content Addressable Network

Un altro esempio di rete strutturata è CAN, dove i nodi hanno indirizzi bidimensionali, e ogni nodo gestisce i punti che cadono nella sua regione.

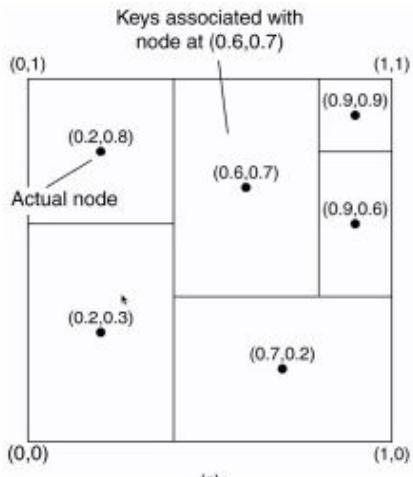


Figure 15: Schema di CAN

Svantaggi structured overlays

Lo svantaggio degli overlay network strutturati è nella mancanza di flessibilità nella rimozione e aggiunta dei nodi in quanto l'operazione di creazione della rete è molto dispendiosa (assegnazione indirizzi, creazione tabelle, distribuzione risorse...) , nonché della riassegnazione di risorse e nella resilienza al fallimento (se faccio un sistema distribuito, voglio essere trasparente al fallimento).

Unstructured overlays: applicazione

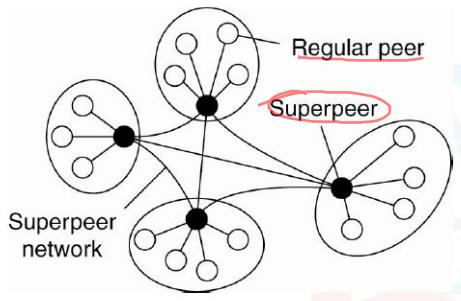
Negli unstructured, invece, si utilizza il **peer sampling**: i peer si scambiano la loro random view (ovvero la visione della rete, che un singolo nodo possiede) con dei conoscenti a caso, definendo una rete di overlay approssimata. La stessa cosa si fa (in parte) anche per le risorse.

Architetture Ibride → used in real life

Sono architetture che utilizzano entrambe le architetture client-server e P2P.

Superpeers

E' possibile avere dei super peer che gestiscono una certa porzione di peer, tipo server.



Sistemi distribuiti collaborativi
(Collaborative Distributed Systems)

Sono dei SD che utilizzano i peer per scambiare i dati, ma che gestiscono il bootstrapping usando il protocollo client-server.

Introduzione ai microservizi, containers e cloud services relativi ai Sistemi Distribuiti

Un microservizio è un caso più specifico di distribuzione verticale delle funzionalità: ognuna è effettuata singolarmente da un singolo processo.

I microservizi devono essere aggiornabili e rimpiazzabili indirettamente dagli altri, perciò sono "impacchettati" con tutti i package necessari per farli funzionare indipendentemente dagli altri. Inoltre, essi comunicano tra loro tramite meccanismi lightweight, come API REST o Remote Procedure Call (RPC).

I microservizi permettono di ottenere una focused scalability: si attua sui microservizi specifici che devono essere scalati, quindi la scalabilità è concentrata su quelli e non devo per forza scalare tutti gli altri microservizi.

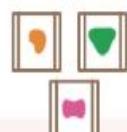
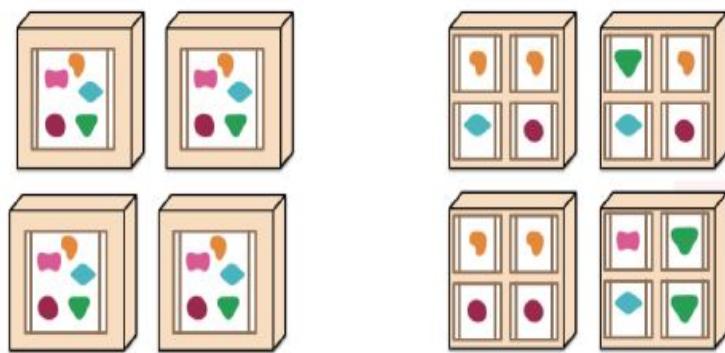
Inoltre, i microservizi possono essere scritti con diversi linguaggi di programmazione.

A monolithic application puts all its functionality into a single process...

... and scales by replicating the monolith on multiple servers

A microservices architecture puts each element of functionality into a separate service...

... and scales by distributing these services across servers, replicating as needed.



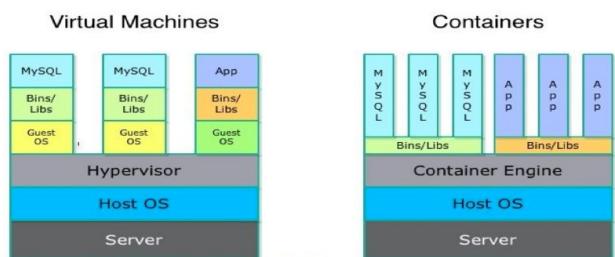
Containers

I container sono un'astrazione del layer applicativo che impacchetta assieme codice e dipendenze. Si effettua quella che chiamiamo una virtualizzazione dei processi nell'user space.

I goal dell'uso di container sono:

- offrire supporto sufficiente per l'architettura a microservizi;
- Incrementare la portabilità del SW, inserendo facilità di migrazione;
- ridurre i problemi che si presentano a tempo di sviluppo.

I containers non sono una virtual machine, qui si virtualizza solamente l'applicazione e tutte le librerie da cui dipende anziché l'intero sistema operativo, mettendo un container engine (come ad es. Docker) sopra il layer dell'OS della macchina host, come vediamo in figura, in cui girano i container. Ne risulta che un container è molto più leggero di una VM, e permette una portabilità maggiore in quanto non è OS-bound.



I container da soli non sono un sistema

distribuito, in quanto girano su una sola macchina. L'idea è quella di avere un sistema distribuito in cui ogni nodo ha una versione del container engine (es. Docker), con un nodo che fa da orchestrator che decide quali nodi hanno quali container su di essi. In questo modo posso distribuire un servizio su larga scala perché è molto leggero (frammentato nei vari container e spartito fra molti nodi).

Servizi di piattaforme cloud (Cloud Platform Services)

I cloud platform services sono un set dei seguenti servizi, costruiti su un SD:

- Computing e Comunicazione;
- Storage e DBMS;
- Identità e Sicurezza;
- Management;

che offrono diversi tipi di trasparenza. I maggiori esponenti sono AWS (Amazon Web Services), Google Cloud platform, Microsoft Azure.

Esempi di servizi forniti da Google Cloud:

- Servizi di **Memorizzazione**:
 - Google Distributed File System (GFS, Colossus, ...);
 - DBMS distribuito RDBMS/NoSQL DBMS;
- Servizi di **Comunicazione**:
 - Google PUB/SUB:
 - Message-oriented middleware sul cloud
 - Supporta la comunicazione tramite scambio di messaggi asincrono multi-a-molti;
- Servizi di **Computazione e Data Analysis**:

- Google DataFlow: si occupa di processing di stream di dati;

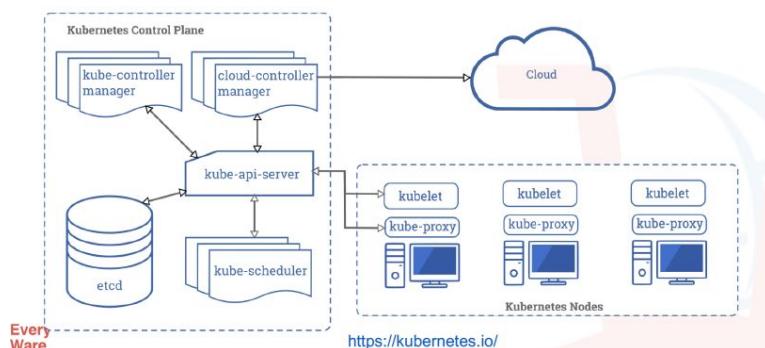
Il problema delle soluzioni cloud sono l'estensibilità e la portabilità, in quanto sono soluzioni sviluppate su sistemi proprietari e non di pubblico dominio.

Usando cloud container e i microservizi si può però aumentare la portabilità, così come l'utilizzo di tools per orchestrare e distribuire applicazioni containerizzate su molteplici macchine, come Kubernetes.

Kubernetes

Kubernetes è una piattaforma pensata per mantenere e gestire i containers su dei cluster, facilitandone la configurazione dichiarativa e l'automazione.

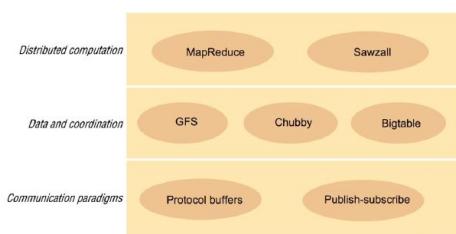
I componenti di un'applicazione sono detti **pods**, e ogni cluster ha dei nodi lavoratori (worker nodes) che fanno da host per i pods. Un pannello di controllo si occupa poi di gestire i pods e i worker nodes.



Prima di passare alla sincronizzazione, parliamo di un ultimo esempio: il vecchio Google File System.

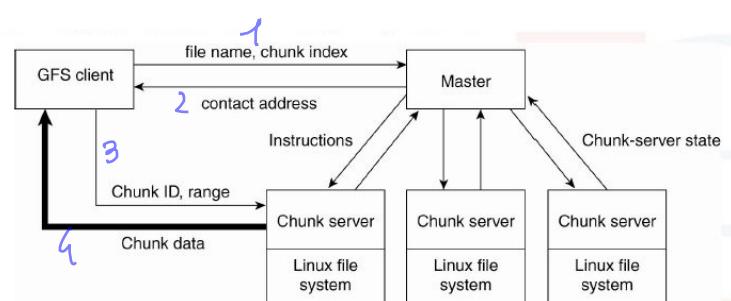
The (old) Google File System (GFS)

GFS è un “SD diverso dai normali SD”, in quanto usa un’organizzazione diversa da quella degli altri SD. L’idea del GFS è di avere un master che comunica col client. Il master gestisce dei server separati, chiamati **chunkserver**. I file vengono spezzettati in chunk, e replicati in ognuno di questi chunkserver. In questo modo il master deve gestire allocazione e inserimento di nuovi chunk.



Architettura e funzionamento GFS

Il client richiede il chunk che gli serve al master. Questo non risponde con il chunk, ma un **contact address** che il client usa per contattare il chunkserver che possiede la risorsa che cerca.



Questa cosa è stata rivoluzionaria nei FS distribuiti.

Perché GFS?

1. permetteva la memorizzazione di files molto grandi;
2. faceva uso dei cluster di computer “commodity” (con high failure rate)
3. I file venivano perlopiù letti, non scritti;
4. Permetteva un throughput di dati molto elevato.

Problemi di sincronizzazione

Risolvere i problemi di sincronizzazione è cruciale, dal momento tutti i device di un SD devono accordarsi nei task per raggiungere obiettivi comuni oltre che parlarsi. La maggior parte dei metodi di sincronizzazione si basa sulla sincronizzazione degli orologi (fisici o logici).

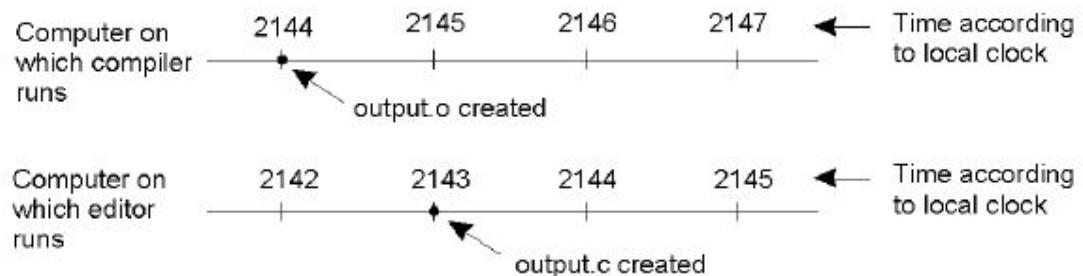
Non meno importante è il problema della mutua esclusione distribuita, che a differenza di quella dei processi, non è risolvibile tramite tecniche tipo i semafori dei sistemi operativi e quindi bisogna ottenerla attraverso lo scambio di messaggi. Nel caso degli algoritmi a elezione, un processo ha il ruolo di eleggerne un altro per continuare il suo lavoro.

Sincronizzazione usando Orologi Fisici

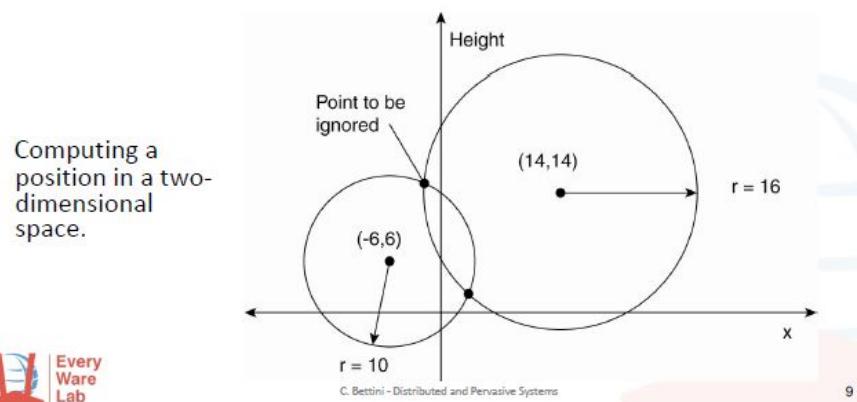
La sincronizzazione attraverso gli orologi funziona utilizzando i timestamp: ogni nodo ha infatti il proprio orologio e assegna un timestamp a ogni evento che accade. **Il timestamp rappresenta quindi l'istante temporale corrispondente al verificarsi dell'evento** e quindi **la sincronizzazione vera e propria avviene facendo un confronto tra essi**, c'è però un problema: gli orologi fisici di ogni nodo potrebbero trovarsi a segnare istanti differenti.

Un orologio fisico è un circuito che tiene il tempo attraverso le frequenze generate dalle oscillazioni di un cristallo di quarzo; tra queste c'è anche quella in linea col nostro concetto di tempo: l'UTC, un'utilità derivata dalla misura dell'orologio atomico Cesio 33 TAI (i cui secondi hanno una lunghezza costante a differenza di quelli solari) a cui è stata aggiunta una variazione per riallineare il tutto col giorno solare (che cambia continuamente), proprio per questo motivo a volte si tolgoni dei secondi all'orologio (l'aggiornamento avviene in modo irregolare).

When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

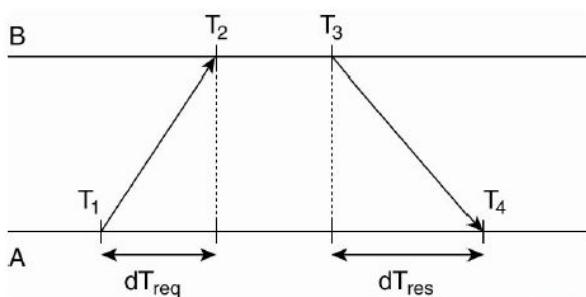


L'obiettivo è quindi ottenere l'UTC più preciso possibile; in questo modo però non si considera che i vari nodi potrebbero andare a velocità differenti: in tal caso si riallineano i valori, utilizzando appositi algoritmi come quelli del GNSS (Global Navigation Satellite System). Questi sono sistemi che hanno orologi atomici a bordo, e fanno un broadcast dei messaggi che contengono le informazioni sul timing. I ricevitori GNSS ascoltano da più sorgenti e calcolano la posizione esatta facendo una differenza tra il tempo misurato di tutte le sorgenti (caso specifico del GPS), oppure con la trilaterazione per gli altri sistemi. La trilaterazione serve a calcolare la posizione dei satelliti e la loro deviazione di tempo UTC: si misura la differenza del tempo dei satelliti e la si utilizza come raggio di una circonferenza, il valore preciso è quindi dato dal punto di intersezione.



NTP-Network Time Protocol

Un metodo alternativo di sincronizzazione esterna è il Network Time Protocol (NTP), o algoritmo di Christian: l'idea è quella di chiedere il tempo agli altri nodi; Il tempo che misuriamo non corrisponde al tempo effettivo su quella macchina, dato che non si tiene conto della latenza della rete.

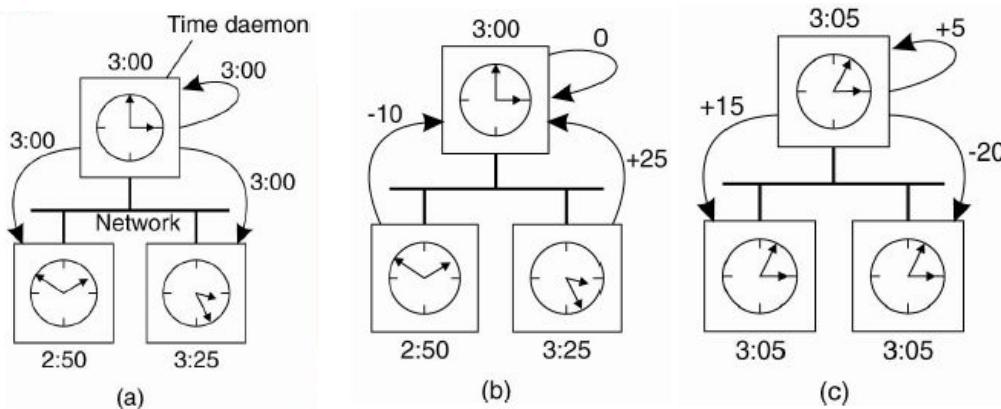


Non esiste, quindi, una sincronizzazione perfetta ma è comunque possibile ottenerne una accettabile cercando di calcolare la latenza per poi sommarla al tempo ricevuto.

Algoritmo di Berkeley

Quando invece si vogliono sincronizzare i nodi interni a una rete locale, si utilizza l'algoritmo di Berkeley:

- Un time daemon **chiede** in broadcast il tempo inviando il proprio;
- tutti gli altri nodi rispondono con la differenza tra il tempo ricevuto dal daemon e il loro tempo;
- Il time daemon effettua una media tra i valori ricevuti e invia agli altri nodi un valore che indica di quanto devono andare avanti o indietro.



Sincronizzazione usando Orologi Logici

In molti casi non è necessario l'allineamento usando i clock fisici esterni: potrebbe bastare anche solo un ordinamento parziale. Gli orologi logici indicano quando un evento avviene prima di un altro, l'obiettivo è infatti assegnare un valore di clock di invio/ricezione di messaggi in modo da rispettare un ordinamento. Il lavoro più importante sulla sincronizzazione tramite clock logica si attribuisce a Lamport, che creò l'algoritmo di Lamport.

Algoritmo di Lamport

L'algoritmo di Lamport si basa sulla seguente affermazione:

"se un evento a viene prima di un evento b ($a \rightarrow b$), allora anche il suo clock viene prima di quest'ultimo".

L'obiettivo di Lamport è **assegnare un valore di clock C(a) condiviso da tutti ad un evento di invio/ricezione di un messaggio in un sistema, in modo che se vale $a \rightarrow b$ allora $C(a) < C(b)$.**

Preso da wikipedia:

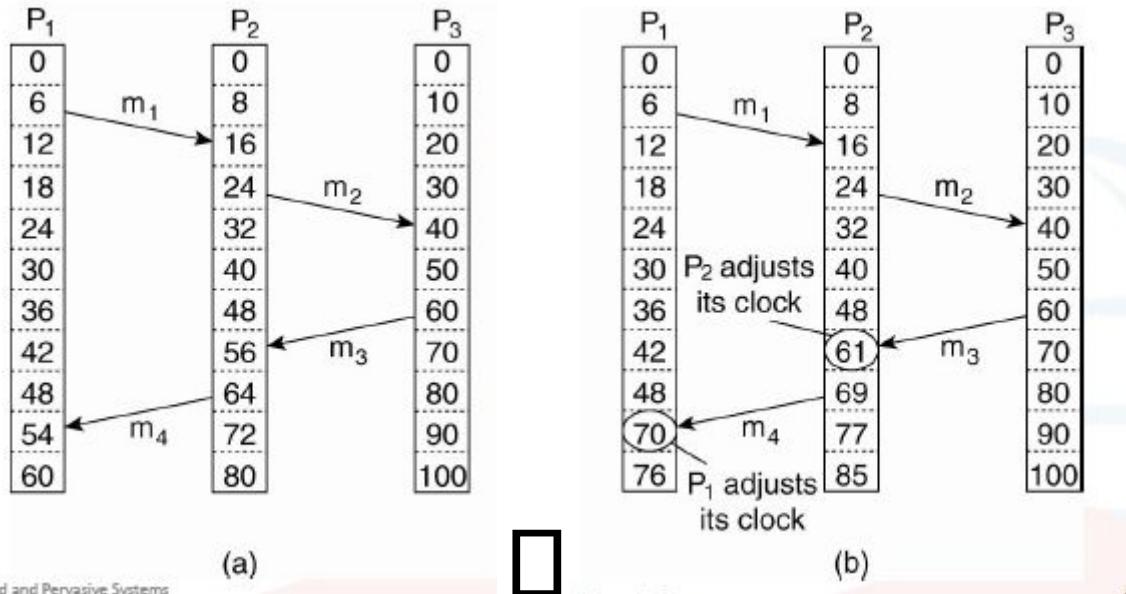
"For any two events, a and b, if there's any way that a could have influenced b, then the Lamport timestamp of a will be less than the Lamport timestamp of b. It's also possible to have two events where we can't say which came first; when that happens, it means that they couldn't have

affected each other. If a and b can't have any effect on each other, then it doesn't matter which one came first."

Il funzionamento di Lamport è il seguente:

- Ogni messaggio porta il timestamp segnato dal mittente;
- Se all'arrivo il clock del destinatario è minore o uguale a quello ricevuto dal mittente, viene reimpostato a quel valore incrementato di 1. Vengono anche aggiornati tutti i timestamp successivi degli eventi di quel processo.

Esempio: vi sono 3 processi su 3 nodi, ognuno col suo clock, che funzionano a ratei differenti. Bisogna assicurare l'invio/ricezione dei messaggi.

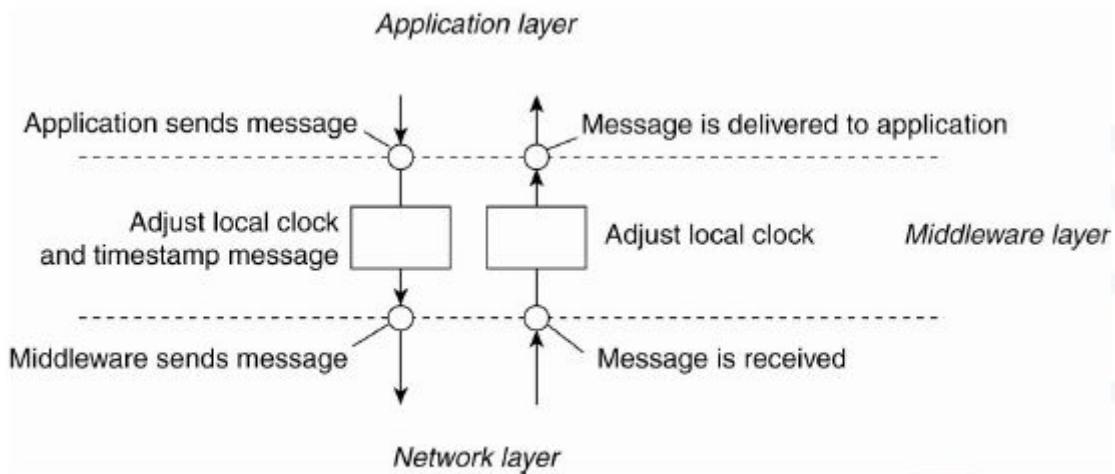


Al tempo a) vedo che $TS(m_3)$ ricevuto da P₂ è > di $C(P_2)$. A tempo b) setto $C(P_2)=TS(m_3)+1$; Stessa cosa con m_4 e P₁: $ts(m_4)>C(P_1)$ ($64>54$): perciò $C(P_1)=ts(m_4)+1$.

Spiegazione scritta:

Nel caso dei messaggi m₁ e m₂ non vi è alcun problema, dato che il timestamp di P₁ al momento dell'invio è inferiore a quello di P₂ in ricezione. Stessa cosa tra P₂ e P₃ col messaggio m₂. Nel caso di m₃ e m₄, il timestamp di chi invia è superiore a quello di chi riceve: quindi il destinatario cambia il proprio clock, in questo modo tutti gli eventi successivi verranno traslati ogni volta che avvengono modifiche sull'orologio.

L'algoritmo di Lamport (e i suoi derivati) vengono implementati a livello di middleware, dato che l'applicazione non deve accorgersene per potersi occupare della lettura dei messaggi.



The positioning of Lamport's logical clocks in distributed systems

Per aggiornare il contatore C_i per il processo P_i :

1. Prima di eseguire un evento, P_i esegue $C_i \leftarrow C_i + 1$;
2. Quando il processo P_i invia un messaggio m al processo P_j , setta il timestamp $ts(m)$ a C_i dopo aver eseguito lo step 1.
3. Alla ricezione del messaggio m , il processo P_j aggiusta il suo contatore locale in questo modo:

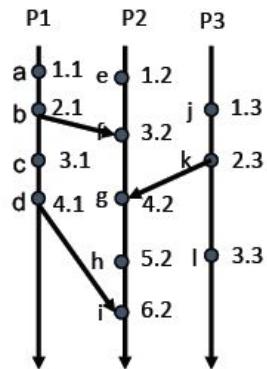
$$C_j \leftarrow \max\{C_j, ts(m)\}$$

4. Dopo questa operazione, P_j esegue il primo step e invia il messaggio all'applicazione.

Nel caso si volesse forzare un ordine totale (ad esempio quando abbiamo più processi che mandano segnalazioni allo stesso timestamp), basta modificare Lamport assegnando l'ID del processo a un evento (un numero unico può essere associato a ciascun processo nel sistema).

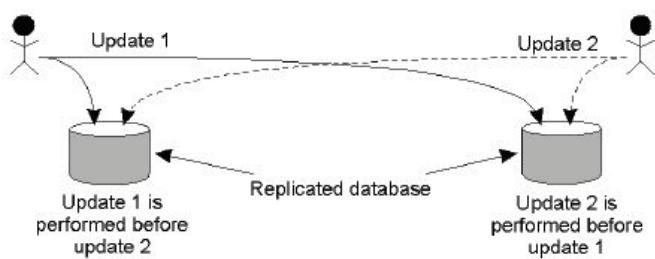
1. P_i associa un timestamp all'evento e con $C_i(e).i$;
2. Possiamo quindi dire che $C_i(a).i$ avviene prima di $C_j(b).j$ **se e solo se:**
 - $C_i(a) < C_j(b)$ oppure:
 - $C_i(a) = C_j(b) \&& i < j$;

Nella foto vediamo che l'ordine creato da Lamport è definito non più solo dal timestamp, ma anche dal process ID. In questo modo processi che agiscono allo stesso tempo di clock hanno comunque un modo per essere ordinati (a parità di clock si esegue l'evento col process id più basso). Gli eventi viaggiano a velocità diverse (perché vediamo d = 4.1 prima di I=3.3?) perché P1, P2 e P3 hanno velocità di clock diverse.



Multicasting con ordinamento totale

Mettiamo caso che vi siano due client concorrenti A e B che vogliono caricare dei dati in due copie diverse di un DB replicato. Se usassimo Lamport in ordine parziale, non riusciremmo a caricare i dati nello stesso modo (nel DB 1 arriverebbe prima la richiesta di A, nel DB 2 quella di B), facendo diventare il DB inconsistente. Per implementare il multicast con ordine totale c'è bisogno di assumere che non ci siano messaggi persi, e che i messaggi dello stesso mittente vengano ricevuti nell'ordine in cui sono stati inviati.



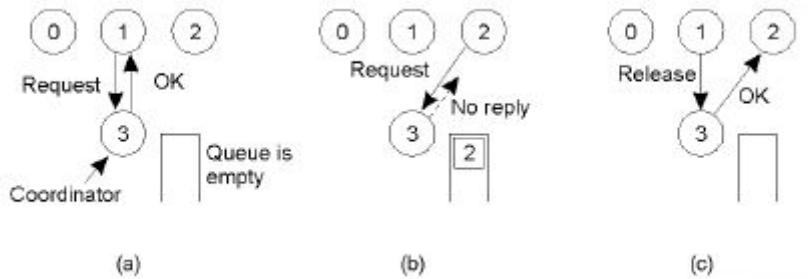
Funzionamento:

- Il processo P_i invia il messaggio m_i , munito di timestamp a tutti gli altri processi. Il messaggio viene messo in una coda locale i ;
- Qualsiasi messaggio in entrata di P_j è accodato nella sua coda j in base al timestamp, e viene inviato un ACK di avvenuta consegna a tutti gli altri processi (gli eventi di send e receive dei messaggi e ACK sono ordinati totalmente con Lamport).
- P_j passa poi il messaggio m_i all'applicazione sovrastante se:
 - m_i è in testa della coda j E
 - m_i è stato ACKed (quindi ricevuto) da tutti gli altri processi;
- Alla fine tutti i processi avranno quindi la stessa copia della coda locale, perciò tutti i messaggi sono passati all'applicazione nello stesso ordine in tutti i nodi.

Mutua Esclusione

La mutua esclusione indica come i processi gestiscono le regioni critiche. Una possibile soluzione è usare un algoritmo centralizzato:

- vi è un processo coordinatore che gestisce gli accessi alle risorse e decide di fare passare altri processi in base alle richieste che riceve.
- se la risorsa è libera, allora viene bloccato l'accesso ad altri finché non conclude, altrimenti la richiesta viene accodata, una volta liberata si notifica al primo processo in coda che la risorsa è nuovamente disponibile.



Questo algoritmo porta dei problemi: vi è single point of failure dal momento che il processo coordinatore può crashare, oltre a un collo di bottiglia causato dal coordinatore per via della natura centralizzata dell'algoritmo, e infine la perdita di messaggi dovuta a malfunzionamenti della rete che non permette di capire se un processo è in attesa di acquisire la risorsa o meno.

Algoritmi distribuiti: Ricart e Agrawala

Nei SD non vi è alcun coordinatore e quindi tutti i nodi si devono arrangiare per gestire questo problema. Una possibile soluzione è utilizzare il timestamp dei messaggi, assumendo che vi sia un ordine totale degli eventi e un sistema di ACK:

- 1) se un processo P vuole utilizzare una risorsa R, esso manda un messaggio contenente:
 - a) il nome di R;
 - b) l'id di P;
 - c) il timestamp al momento dell'invio
 E invia il messaggio in broadcast a tutti i nodi (anche a se stesso!)
- 2) Quando un processo Q diverso da P riceve il messaggio, possono verificarsi tre casi:
 - a) Se Q non sta usando R e non ne ha bisogno, manda un ok a P;
 - b) Se la sta utilizzando, accoda la richiesta e non risponde;
 - c) Se invece Q vuole utilizzare R ma non ha ancora ricevuto l'accesso, confronta il timestamp della sua richiesta con quello del messaggio di P. Se quest'ultimo risulta più piccolo, allora Q manda un ok a P, altrimenti accoda il messaggio;
- 3) Una volta che P ha ricevuto l'ok da parte di tutti i processi, allora accede a R bloccando l'accesso agli altri. Un volta finito, P avvisa tutti i processi della sua coda con un ok per indicare che R è di nuovo libera.

L'unico problema di questo algoritmo è il numero di messaggi che vengono inviati a causa del broadcast.

Vediamo un esempio:

- a) i processi 0 e 2 vogliono accedere R allo stesso tempo. P0 manda in broadcast un messaggio con ts=8, P2 uno con ts=12.
- b) P0 ha il timestamp più piccolo, e vince la corsa. Aggiunge la richiesta di P2 alla sua coda locale e non risponde. P1 manda l'OK a entrambi visto che non vuole usare R. P2 compara il timestamp del suo messaggio con quello di 2 e manda l'OK.

- c) Quando p0 ha finito di usare R, manda OK a P2 (l'unico in coda), che avendo ricevuto l'OK da P0 e P1 può usare R.

Questo tipo di algoritmo porta due problemi: non ci potrebbe essere una risposta se uno dei processi crasha, portando quindi a un'attesa indefinita. Inoltre, chiedere l'OK a tutti i nodi di un SD è uno spreco di risorse. Cerchiamo quindi un altro modo.

Algoritmo ad anello

Ogni processo è un nodo di una rete logica ad anello e ognuno punta al suo successivo. Il suo funzionamento è simile a quello delle reti token ring: un token viene fatto girare tra tutti i nodi e solo chi lo possiede ha il permesso di inviare i messaggi, esso deve continuare a girare e non può essere trattenuto da un nodo più del tempo necessario a usare la risorsa.

I problemi in questo algoritmo riguardano i possibili crash dei processi (rompendo la rete di conseguenza) e la perdita del token. In quest'ultimo caso bisogna stare attenti a non generarne un secondo. Il numero dei messaggi in questo algoritmo tende a infinito, dal momento che il token deve continuamente girare per far funzionare tutto.

Funzionamento:

- All'inizio, il token viene generato dal processo 0;
- Il token viene inviato al processo $(i+1)\%n$, dove n è il numero di processi;
- se uno di questi è interessato a una risorsa, può accedervi quando riceve il token e lo manda al successivo quando ha finito. Uno stesso token non può essere utilizzato più di una volta.

Algorithm	Messages to acquire/release R	Delay to acquire R (num. of messages)	Problems
Centralized	3	2	Crash of coordinator
Distributed	$2(n - 1)$	$2(n - 1)$	Crash of any process
Distributed as ring	1 to ∞	0 to $n - 1$	token loss, crash of a process

Algoritmi a elezione

Gli algoritmi a elezione permettono ai processi di eleggere un coordinatore per la gestione di tutti i messaggi.

Solitamente viene eletto il processo con l'identificatore più alto: in tal caso non c'è perdita di generalità.

Per esempio, il processo P_i con il carico computazionale più leggero può essere eletto settando il suo $ID(P_i)$ a $< \frac{1}{load(P_i), i} >$, dove l'ID i è usato come criterio di ordinamento secondario dove due processi hanno lo stesso carico.

Molti algoritmi di questo tipo assumono che ogni processo conosca il proprio id e che possano comunicare con qualsiasi altro processo, tuttavia in questo modo non è possibile capire se un processo è attivo o meno in quel momento.

Algoritmo Bully

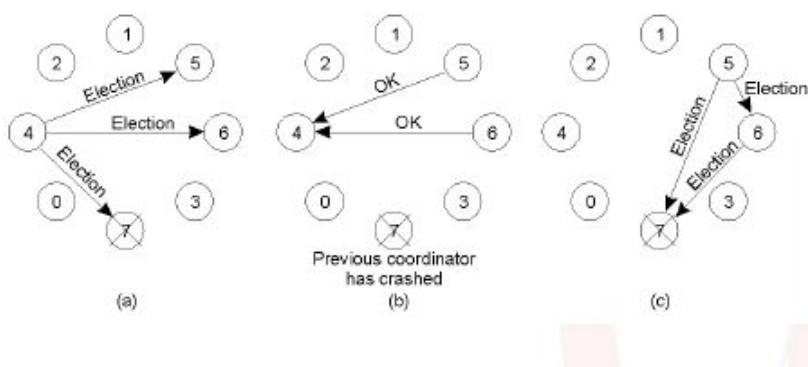
Nell'algoritmo bully, un processo contatta tutti gli altri con id più alto del suo al fine di capire chi è il coordinatore e se questo è attivo o meno.

Se uno di questi non risponde, i coordinatori controllano se effettivamente è crashato o meno.

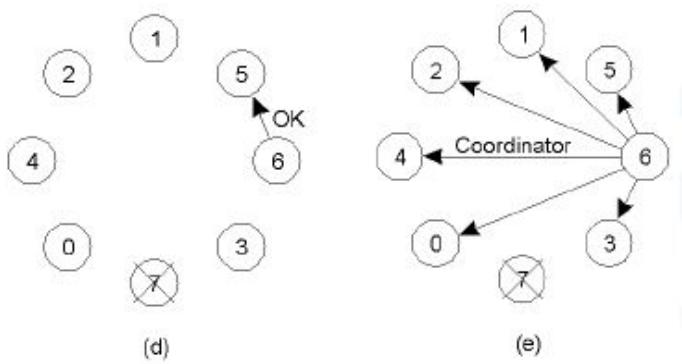
Nel caso sia il coordinatore a non rispondere, viene rifatta l'elezione: il processo che ha scoperto che il vecchio coordinatore era down invia un messaggio di elezione a tutti quelli con id più alto, i quali rispondono con un OK nel caso siano intenti a partecipare all'elezione candidandosi come nuovo coordinatore. Nel caso in cui un processo non riceva alcun ok, significa che esso è quello con id più alto e quindi diventa il nuovo coordinatore.

Questo procedimento viene effettuato da ogni processo, quindi può capitare che vi siano elezioni concorrenti.

- a) Process 4 contacts the old coordinator; it understands that it is not active anymore and starts an election by sending a message to all processes with an ID greater than their own
- b) Processes 5 and 6 answer telling 4 that they will take care
- c) Both 5 and 6 start an election



- d) Process 6 tells 5 that it will take care of the election
- e) Process 6 wins and tells everybody



Algoritmo di elezione ring-based

Nel caso i nodi non abbiano accesso agli indirizzi di tutti gli altri, si può optare per un elezione ad anello: in tal caso si assume che tutti gli n processi siano logicamente ordinati in una rete ad anello; Si assume inoltre che un processo P_i può comunicare solo col suo successivo, precisamente a P_j con $j=(i+1)\%n$.

L'obiettivo è trovare il processo avente l'id più alto per renderlo coordinatore.

Se il vecchio coordinatore crasha:

- 1) Tutti i nodi vengono marcati come non partecipanti all'elezione;
- 2) Quando un processo P_i capisce che il coordinatore non risponde, fa partire un'elezione, diventa partecipante e invia un messaggio al suo successivo;
- 3) Alla ricezione del messaggio di elezione, P_j (con $j=(i+1)\%n$) confronta il proprio id con quello nel messaggio e se quest'ultimo è maggiore lo inoltra al successivo così com'è, altrimenti si segna come partecipante e invia un messaggio al suo successivo.
- 4) Nel caso in cui il messaggio ricevuto da P_j abbia lo stesso id del nodo da cui lo ha ricevuto, significa che questo processo è quello con id maggiore e quindi diventa coordinatore;

L'elezione ring based gestisce molto bene le failures date dal crash dei nodi nell'anello, memorizzando non solo l'ID del prossimo processo, ma anche quelli dei nodi che sono qualche posizione più avanti (Non solo P_i , ma anche P_{i+1} e P_{i+2}). Se la comunicazione col prossimo processo fallisce, il messaggio è mandato al primo dei processi attivi dopo a quello successivo.

Nel caso di elezioni concorrenti, esse sono gestite dallo stato partecipante/non partecipante, che aiuta ad estinguere in fretta i messaggi inutili di elezioni concorrenti: se il messaggio arriva a un nodo già partecipante, quest'ultimo non viene più inoltrato.

Il caso peggiore di questo algoritmo avviene quando l'assenza del coordinatore viene notata dal processo avente id minore (cioè il nodo più vicino in senso antiorario al coordinatore). In tal 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. Perciò in totale nel caso peggiore servono $3N-1$ messaggi.

Nel caso migliore invece l'elezione verrà fatta partire dal processo con id maggiore (quindi si fa un solo giro).

Fault Tolerance e Consenso distribuito

Concetti base della fault tolerance

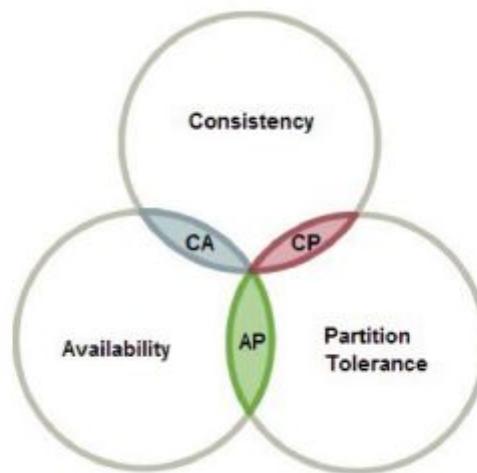
Essere fault tolerant è fortemente legato a ciò che chiamiamo sistemi dependable. La dependability implica le seguenti proprietà:

- Disponibilità (Availability): la probabilità che il sistema operi correttamente in qualsiasi momento (oppure: il sistema è molto probabilmente disponibile quando mi serve);
- Affidabilità (Reliability): l'abilità del sistema di funzionare correttamente per un lungo intervallo di tempo; Posso essere disponibile senza essere affidabile (es. reinizializzo il sistema ogni giorno per 2 minuti);
- Sicurezza (Safety): I malfunzionamenti non portano a fallimenti catastrofici (es. crash INPS non è sicuro), perciò trovo in modo safe di gestire il crash;
- Manutenibilità (Maintainability): l'abilità di "riparare facilmente" un sistema malfunzionante, ad es. l'hot swap di HDD rotti;
- Consistenza (Consistency): l'abilità di ricevere a ogni lettura la scrittura o l'errore più recente di un documento;
- Partition Tolerance: l'abilità di limitare i messaggi persi o in ritardo a un certo numero.

Il teorema di CAP per i DBMS distribuiti

non posso avere le seguenti tre proprietà assieme, posso sceglierne solo 2:

- Consistenza: tutti i nodi vedono gli stessi dati (slides: ogni lettura riceve la scrittura più recente o un errore-->implica che tutti i nodi vedono lo stesso dato).
- Availability: Ogni richiesta riceve una risposta. Implica che il sistema deve essere operativo il 100% del tempo.
- Partition tolerance: Il sistema tollera un numero arbitrario di messaggi persi (lo si fa aumentando di molto le repliche dei dati).
- Di solito si sceglie di rinunciare (in percentuale) a availability/consistency.



Tipi di guasto

Tipo di failure	Descrizione
Crash failure	il caso “migliore” dei failure: il sistema semplicemente si blocca, ma funziona normalmente quando lo si riattiva.
Omission Failure	un server non riesce a rispondere alle richieste in arrivo, o non riconoscendo i messaggi ricevuti (receive omission), oppure non inviando i messaggi ai richiedenti (send omission).
Timing failure	il server risponde alle richieste troppo tardi, e fa scattare il timeout.
Response failure	il server restituisce un valore sbagliato e non fa esattamente quel che deve fare. E’ molto difficile da individuare in quanto il sistema funziona correttamente, ma dà risultati errati che se ignorati possono essere pericolosi (pensiamo all’output del sistema di raffreddamento di una centrale nucleare: che succede se per caso il sistema invia un output di raffreddamento inferiore a quello effettivamente necessario? esatto, Chernobyl).
Arbitrary failure	il server produce risposte arbitrarie in tempi arbitrari. Errore molto grave perché invia dati che non dovrebbe inviare a momenti inaspettati, e normalmente è causato dall’intrusione di nodi malevoli nel sistema.

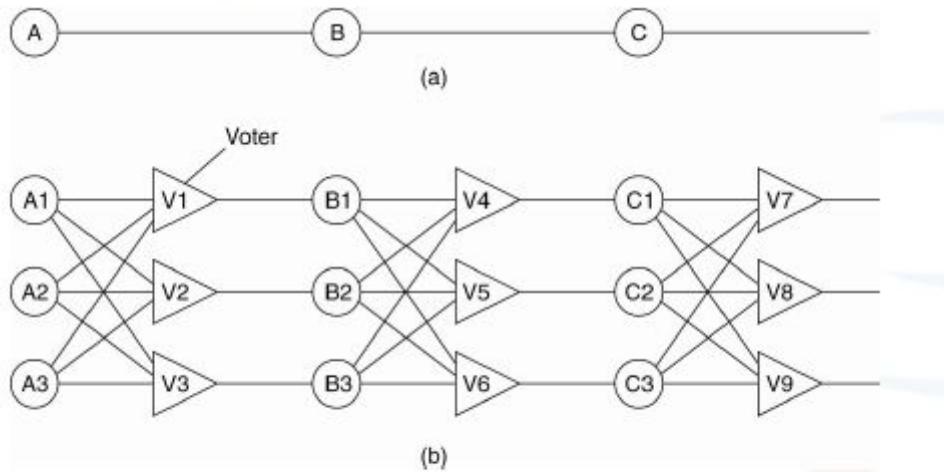
Come possiamo risolvere (almeno in parte) l’insorgenza di questi failures? La soluzione più comune è applicare la **ridondanza**.

Ridondanza (Redundancy)

Può essere:

- **Information redundancy:** aggiungo dell'informazione aggiuntiva per recuperare i messaggi quando c'è rumore sulla rete. I codici di Hamming per la correzione dell'errore, come quelli di Hamming, sono un esempio di information redundancy.
- **Time redundancy:** (o ridondanza delle operazioni) una transazione viene ripetuta un paio di volte se abortita per un fail di un'azione al suo interno (in quanto una transazione deve essere intera per funzionare, se la lascio a metà non va bene). Per le RPC questo sistema richiede un ulteriore livello di attenzione: se sono idempotenti, allora possiamo usare la time redundancy (ad es. la open()), altrimenti no!
- **Physical redundancy:** Aggiungo dispositivi fisici (extra HW) o software per recuperare dal fail di un componente, magari aggiungendo un'unità che processa quel tipo di esecuzione in maniera veloce, o un nuovo sensore per velocizzare l'acquisizione ecc.

Nella figura vediamo un esempio di failure masking su tre nodi che si scambiano l'informazione (da A a B e da B a C) usando la ridondanza.



- How many faulty devices are tolerated?

Voglio replicare il nodo A: prendo tre copie di A, poi metto dei **voter** che ricevono l'output da tutte le copie di A, e mette in output la maggioranza tra queste (se ad es. A1 e A2= 10, mentre A3=5, il voter prende 10). Se non c'è maggioranza manda output "sconosciuto". Ne consegue che per ogni nodo devono funzionare almeno due nodi replica, così se uno si rompe ne ho comunque due tra cui fare la maggioranza. In questo schema si tollera solo un faulty device.

La ridondanza può anche essere applicata ai processi, chiamata **process resilience**:

1. La ridondanza si fa creando un gruppo di processi identici, che usano lo stesso dato.
2. Ogni processo riceve i messaggi del gruppo.
3. Occorre implementare un protocollo e un meccanismo per far uscire/entrare i processi dal gruppo.

Quanta ridondanza è necessaria?

possibile domanda d'esame!

Ad es. nella process resilience: quanti processi vanno messi in un gruppo?

Se k è il numero di processi che falliscono:

- Se ho solo il caso del crash da monitorare, allora bastano $k+1$ processi che danno k fault tolerance (ne basta uno attivo x mandare avanti il sistema);
- Se ho response failure, mi servono almeno $2k+1$ processi, e il client decide votando: i server inviano al client le loro risposte, e il client decide in base alla maggioranza.
- Se c'è bisogno di consenso nel gruppo (le decisioni vanno prese all'interno del gruppo) la cosa è più complicata. Secondo Bettini servono, in una situazione normale senza nodi malevoli, $3k+1$ processi. In alcuni casi ciò è impossibile, in particolare quando la maggioranza dei nodi sono malevoli (dirigono la maggioranza verso di loro).

Consenso nei sistemi malfunzionanti

Il consenso nei sistemi malfunzionanti è un problema difficile, la cui soluzione dipende da diversi fattori:

- Il sistema è sincrono o asincrono?
- Il delay della comunicazione è legato a qualche proprietà o è arbitrario?
- L'invio dei messaggi è ordinato (prima 1 poi 2) o no?
- La trasmissione dei messaggi avviene in unicast o multicast?

Ci sono diverse tecniche per gestire il consenso, una di queste è il

Byzantine Agreement (accordo bizantino)

E' una soluzione che risolve sia il timing che il response failure. E' stato sviluppato da Lamport.

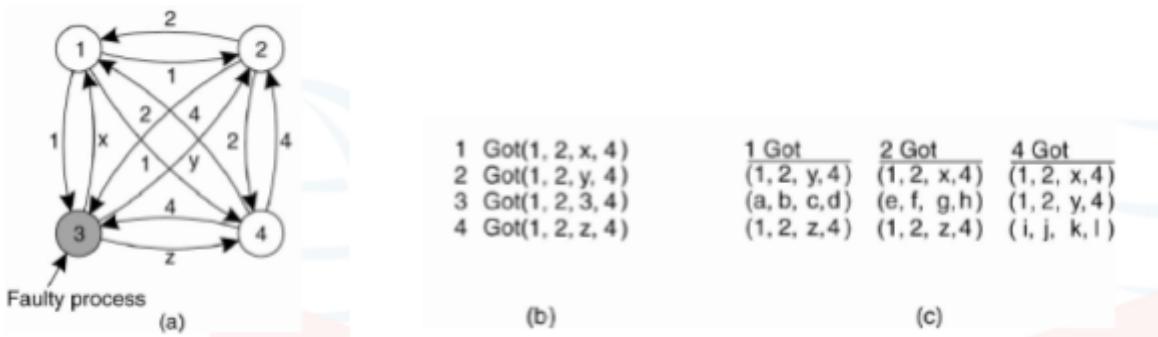
Condizioni per il funzionamento:

- Ho N processi, di cui K sono di tipo "bizantino" (malevoli o malfunzionanti);
- Ognuno produce un valore $V[i]$;
- Gli $N-K$ nodi funzionanti devono accordarsi su un vettore corretto, isolando i dati errati;
- In questo caso, per far funzionare l'agreement servono almeno $3K+1$ processi funzionanti, con $k=$ processi malfunzionanti.
- Assunzioni fatte da Lamport per il funzionamento: processi sincroni, comunicazione unicast, bounded delay, ordine dei messaggi preservato.

Funzionamento:

1. Ogni processo manda il suo valore computato agli altri (in unicast reliable). Il nodo bizantino potrebbe mandare valori sbagliati.
2. Ogni processo assembla un vettore con i valori ricevuti dagli altri;
3. Ogni processo manda la sua visione agli altri processi. Il nodo malevolo può mandare valori errati anche in questa fase (la ricezione si fa con la funzione `got(vettore)`).
4. Ogni processo esamina l'i-esimo elemento dei vettori ricevuti. Se un valore ha la maggioranza sugli altri, viene messo in un nuovo vettore. Se non c'è maggioranza, mette un valore "unknown".

Qui sotto metto la simulazione dei processi usando N=4 processi di cui K=1 faulty, e vediamo che al punto c) troviamo un consenso sui valori v1, v2 e v4 dei vettori dei processi non malfunzionanti.



Il Byzantine Agreement necessita di almeno $3k+1$ processi, dove k =processi faulty.

Impossibilità dell'Agreement in sistemi asincroni

Quando abbiamo un sistema con delay arbitrari come quello asincrono, non esiste nessuna soluzione garantita all'Agreement (nemmeno per il multicast totalmente ordinato), anche se fallisce un solo nodo. Esistono però soluzioni per sistemi parzialmente sincroni, che possono essere usate per modellare sistemi nella pratica e non solo nella teoria.

Altri topic importanti

(solo slides, non ci sono negli appunti. potrebbe essere facoltativo)

- Reliable client-server communication
 - Example: RPC in presence of failures
- Reliable group communication
 - Example: Atomic multicast
- Distributed commit
 - Generalization of atomic multicast to arbitrary operations performed by members of a group (all-or-none semantics).

Google: una lezione sui SD

Lezione a cura di Dario Freni, Software Engineer di Google.
Dariofreni@google.com

La missione di Google è quella di organizzare l'informazione mondiale in modo universale. Abbiamo quindi un'enorme mole di dati da gestire in modi diversi.

Vedremo 3 aspetti:

- Organizzare l'informazione
 - Come si salvano? In che modo le gestiamo?
- Processare l'informazione
 - Enorme problema di scalabilità
 - Soluzioni per processare i dati
- Rendere sempre disponibile l'informazione.

Organizzare l'informazione

I DBMS non bastano assolutamente per gestire tutti questi dati. Google vuole avere un link di virtualmente ogni pagina esistente sul web, e nuove pagine escono con una frequenza inaudita (es. Youtube 7 anni fa è arrivato a 48 ore di video uploadate in un'ora!).

Pensiamo a Street View: vi è un numero enorme di strade che vanno salvate e organizzate nel modo corretto. Non solo, pensiamo all'enormità di dati scambiati con Gmail.

Nel corso degli anni, Google ha sviluppato svariate tecnologie per gestire l'enorme mole di dati, alcune di esse poi vennero usate da sistemi open source.
<https://research.google.com> per vedere i papers)

Vediamone alcuni.

Protocol Buffers (open source)

E' il sistema con cui Google rappresenta i dati, una sorta di JSON/XML. La definizione dei messaggi viene compilata all'interno di classi native, e poi compilato in una struttura più compatta, inviato e alla fine spacchettato dal ricevente. In questo modo i dati possono essere inviati sulla rete in binario, in modo da renderli molto più leggeri di JSON/XML. La differenza sostanziale con JSON e XML è che **il messaggio viene compilato**.

ESEMPIO con JAVA:

```
syntax = "proto2";
package tutorial;
option java_package = "com.example.tutorial";
option java_outer_classname = "AddressBookProtos";
message Person {
    required string name = 1;
    required int32 id = 2;
    optional string email = 3;
```

```

enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
}
message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
}
repeated PhoneNumber phones = 4;
}
message AddressBook {
    repeated Person people = 1;
}

```

GFS - Google File System

Google lo descrive come un “massive distributed and fault tolerant File System” per lo storage di grosse moli di dati in un SD. Utilizza le classiche chiamate POSIX open(), read(), write(), close() per interagire coi files. Supporta inoltre l’accesso concorrente al FS e prevede la replicazione dei dati su varie macchine per preservarli. Abbiamo quindi un tradeoff tra replicate e fault tolerance.

Architettura GFS

I client comunicano con delle macchine master. Quando vogliono accedere a un dato, i client chiedono la sua posizione al master che indica al client il chunkserver su cui è salvato, all’interno di uno dei chunk memorizzati su quel master. I chunk sono quindi replicati in altri chunk server. E’ buona pratica per il master indirizzare la richiesta verso il chunkserver più “libero”.

Colossus

Sistema evoluzione di GFS che usa la tecnologia Bigtable per evitare problemi di memorizzazione dei file, incrementando la dimensione massima memorizzabile e diminuendo la latenza per il retrieving.

Colossus usa i codici di Reed-Solomon per avere un maggior controllo sulla perdita dell’errore e risolvere la replica corretta dei files.

Bigtable

Precursore del movimento NoSQL, ha anche implementazioni open source.

Si tratta di una mappa di sorting multidimensionale e distribuita, disegnata per memorizzare miliardi di righe, scalandole su migliaia di macchine con connessione ultraveloce (macchine nello stesso datacenter).

Fornisce la replicazione di più dati su datacenter multipli.

Una tabella Bigtable è una tabella del tipo righe-colonne, dove le righe sono rigorosamente delle stringhe memorizzate in un certo ordine.

A ciascuna riga è associato un insieme di celle che contiene una famiglia di colonne (esempio di MongoDB). Ogni tabella può essere divisa in tablets: ognuna di esse contiene un range di righe contigue, e si cerca di farle avere una dimensione attorno ai 100/200 MB, ma il parametro può essere caricato in base alla cache (cioè: quante tablet posso caricare in cache in maniera efficiente senza sovraccaricare il tablet server?).

Le celle sono utilissime perché permettono di risparmiare spazio, con il quale è possibile salvare versioni diverse di uno stesso dato nel tempo (per ritornare ad es. a una versione precedente). Ogni server contiene circa 100 tablets. Nel caso una macchina fallisca, le sue tablet vengono prese dalle altre macchine ancora attive (trasparenza rispetto a guasti e migrazione), cosa che può avvenire anche per motivi di load balancing (fine grained load balancing). Le macchine vengono scelte in base alla memoria disponibile, la latenza media, dimensione cache, ecc.

Quando una macchina va giù, i dati vengono migrati in questo modo:

1. Si chiude la connessione coi client della macchina faulty;
2. Le librerie lato client si occuperanno di recuperare i nuovi server dove ci sono i dati migrati (o copie di quei dati); Si pagano i retry in tempo;

E' intuitivo come in realtà esistano dei master replicati per recuperare più in fretta i dati (anche se in realtà c'è un sistema di lock che dice in quale server sono associati i dati, detto Chubby simile a un FS);

Bigtable è un ottimo sistema, ma garantisce atomicità solo a livello di riga, perciò non supporta eventuali transazioni (atomicità richiesta su + righe). Non supporta neppure le tabelle relazionali, perciò non si usa per sistemi in continua evoluzione e molto complessi (come ad es. maps). Per ovviare a questo problema sono stati usati dei workaround costruiti su Bigtable ma che aumentavano latenza e diminuivano lo storage massimo dei file.

Spanner

Più o meno gli stessi scopi di bigtable. Si tratta di un database semirelazionale, usa SQL in ibrido con altre tecniche. Si tratta di una master table con dei figli ben precisi per evitare relazioni a caso (es. customer è master, tab figlia= credit card). Garantisce le proprietà ACID. (Proprietà ACID)

Spanner scala su milioni di macchine e miliardi di milioni (trillions) di dati!

Come bigtable i dati hanno un timestamp di commit che può essere distribuito globalmente, e garantisce una certa consistenza rispetto ad un certo timestamp (strong consistency). Supporta il paradigma Map-Reduce.

Il lock durante una transazione non è sul DB ma sulla serie di tablet richieste nella transazione, e l'utente ha un tempo max per fare la transazione, calcolato in base al timestamp di quest'ultima. In realtà si usa la tecnica del **TrueTime**: si dà un tempo massimo più grande del timestamp, in modo da permettere ai pc di sincronizzare gli orologi logici.

F1

Si posiziona sopra Spanner nello stack, e mirato a certe caratteristiche di velocità. Le colonne sono espresse in termini di protocol buffer, cioè salvo i dati in un'unica colonna, e poi ci entro estraendo il dato da essa (Nested Fields) ma si possono fare interrogazioni con SQL. L'architettura permette di distribuire le query tra i vari workers e quindi eseguirle in modo distribuito.

MapReduce

Modello di programmazione per processare e generare grossi dataset. Utilissimo per i SDP, ed è applicato per risolvere molti problemi. Consente un calcolo parallelo di

$$\text{map} : (k1, v1) \rightarrow \text{list}(k2, v2)$$

$$\text{reduce} : (k2, \text{list}(v2)) \rightarrow \text{list}(k2, v2)$$

Cioè mappare una coppia chiave-valore su una lista di coppie chiave-valore, e poi ridurre la suddetta coppia chiave-lista di valori in una lista di chiavi-valore.

Es. contare le occorrenze di ciascuna parola in un set di documenti;

Map emette un 1 simbolico per ogni parola letta, creando una lista di (parola, sequenza di 1). La reduce viene chiamata una volta per ogni parola, prendendo la lista di valori associata ad essa e riducendola a un solo conteggio. (ad es. se ho lista ("pippo", 1,1,1,1) diventerà ("pippo", 5).

Shuffle=funzione che converte una lista (chiave, valore) in una lista di chiavi univoche con lista di valori associata.

Schema:

Input-->map-->Intermediate-->reduce-->output

I worker della reduce leggono i dati intermedi e invocano la reduce su di essi, e scrivono l'output sul file.

MapReduce è estremamente flessibile e adattabile a moltissimi algoritmi:

- GREP distribuito (map only, emette i risultati se 1 keyword è trovata o no);
- Count URL clicks: input: web logs, map fa <URL,1>, reduce fa <URL, count>;
- Reverse Web-link Graph: data 1 pagina, quante puntano a quella pagina?
 - Map: scan di tutti i target link da ogni source page:
 - <Target, source>
 - Reduce: concatena la lista di tutte le sources:
 - <target, list(source)>

Può diventare inmantenibile se concateniamo troppe operazioni di map-reduce in algoritmi troppo complessi. Questo problema si risolve usando FlumeJava.

FlumeJava

Sono un set di librerie e API pensate come alternativa a MapReduce per costruire pipeline per computazioni parallele. Funzionano creando un grafo di esecuzione implicitamente via lazy execution, ottimizzando e fondendo i MapReduce ecc.

MillWheel

Flume e MapReduce vanno bene per dati in batch, ma sono il massimo se devo operare su degli stream nell'ordine di milioni di eventi al secondo. Per questo si usa il framework MillWheel. E ' usato ad esempio per aggiornare l'indice velocemente per i siti di news. Il paradigma ricorda un po' quello di MapReduce, usando il solito trick di suddividere il carico per chiave.

DLT e Blockchain

DLT (Distributed Ledger Technology) è un registro distribuito corrispondente a una buzzword avente moltissime applicazioni. Un esempio di DLT è Blockchain, essa nasce nel 2008 col paper dei bitcoin ma la prima implementazione avviene per l'anno successivo. Oltre a essa, negli anni successivi nacquero altri DLT basati su criptovalute come Ethereum, tuttavia questi registri non hanno applicazione solamente in questo ambito, Carrefour ad esempio la utilizza per controllare la filiera del pollo.

Modello del sistema DLT

DLT è caratterizzato da un modello che permette un controllo decentralizzato; i nodi che lo compongono vengono eseguiti da differenti entità, tra loro però non vi è fiducia dal momento che vi possono essere nodi malevoli.

Ogni nodo contiene inoltre una copia dei dati, che vanno dunque sincronizzati.

Da queste caratteristiche viene fuori il cosiddetto problema del consenso: i nodi infatti devono trovare un consenso sulla storia dei dati, nel caso della blockchain questo è dato dai blocchi e dal loro ordine.

Blockchain

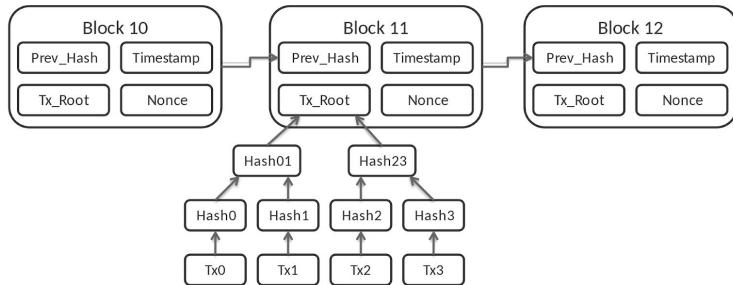
Una blockchain è un sistema basato su DLT, che ha come unità base la transazione. Una transazione è un insieme di dati (come ad esempio un trasferimento di valuta), e ognuna di esse viene memorizzata dalla blockchain in un apposito registro.

L'idea normalmente usata è effettuare le transazioni con nodi inaffidabili andando da un coordinatore della blockchain (approccio utilizzato dalle banche). Per cercare di mantenere la qualità e la decentralizzazione senza utilizzare un coordinatore, si usa un sistema di firme digitali: le transazioni vengono firmate digitalmente e propagate ad altri nodi, che hanno sistemi al loro interno per effettuarne la validazione.

Tutto ciò comunque non basta a rendere una transazione completa, dal momento che le transazioni sono pendenti (non fanno parte della catena) fino alla convalida.

Struttura della Blockchain

Le transazioni nelle blockchain sono raggruppate in blocchi aventi un timestamp, l'intera storia dei dati è memorizzata nel blocco attraverso un Merkle tree, un struttura ad albero molto efficiente che permette di controllare eventuali tracce attraverso dei codici hash. All'interno di ogni blocco sono inoltre presenti un riferimento al suo precedente.



Una blockchain non è esente da problemi:

- Dal momento che i nodi sono distribuiti geograficamente, vi può essere latenza imprevista nella rete;
- Ne consegue che le transazioni potrebbero arrivare in un ordine differente rispetto a quello inserito;
- Inoltre, alcune transazioni possono andare in conflitto;
- Infine, nodi differenti possono mettere insieme le transazioni nell'ordine ricevuto per creare differenti blocchi e creando differenti catene di conseguenza, in tal caso allora è necessario il consenso per risolvere i conflitti tra versioni diverse della chain.

Consenso nelle blockchain

Per ottenere il consenso nella blockchain, l'idea è calcolare l'hash di ogni transazione e blocco, per ognuna si aggiunge l'hash del precedente e, al fine di tenere i nodi bloccati per un po' dopo la transazione, si include un trucco per rendere il calcolo complicato ma comunque facile da risolvere. L'obiettivo infatti è far concorrere i nodi in modo da non farne vincere due nello stesso momento, dando un reward a chi vince, come una sorta di algoritmo a elezione per l'imposizione dei propri blocchi.

L'idea di utilizzare le funzioni di hash è data dai seguenti motivi:

- dato un testo di lunghezza variabile, venga restituito un testo di lunghezza fissa;
- due testi differenti danno come risultato hash differenti;
- deve essere computazionalmente difficile trovare due hash uguali per due testi differenti;
- deve essere computazionalmente difficile trovare un testo a partire dal suo hash;
- l'hash deve essere facile da calcolare;

Algoritmo Proof Of Work (PoW)

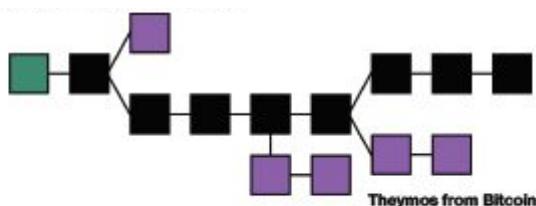
Ogni blocco contiene il suo hash riferito ai dati e un nonce, un numero che viene utilizzato una e una sola volta.

Verificare un blocco significa che tutti gli altri nodi devono risolvere un “puzzle” mentre computa l'hash del blocco. Un nodo miner deve trovare quindi un numero (il nonce incluso nei dati del blocco) tale che l'hash del blocco abbia una particolare proprietà (ad esempio far apparire quattro zeri all'inizio del blocco). Per risolvere il puzzle dunque serve un approccio brute force, che porta via tempo. L'idea della bruteforce è che così è molto improbabile che due miner risolvano il loro giochino allo stesso tempo, evitando così la concorrenza. Il miner che risolve per primo il puzzle riceve un “reward”, che può essere il diritto a salvare la sua versione della chain.

Tutti i blocchi della blockchain devono essere verificati dal momento che ognuno ha l'hash del suo precedente (a eccezione del primo), quindi se avvengono dei cambiamenti in un blocco, esso si invalida così come tutti i suoi successivi e quindi bisogna ricomputarli, questo approccio permette di rendere “vera” questa catena.

All'interno di una stessa blockchain potrebbero esserci più catene differenti tra peer: per risolvere si fa un voto di maggioranza, scegliendo la catena condivisa da più peer.

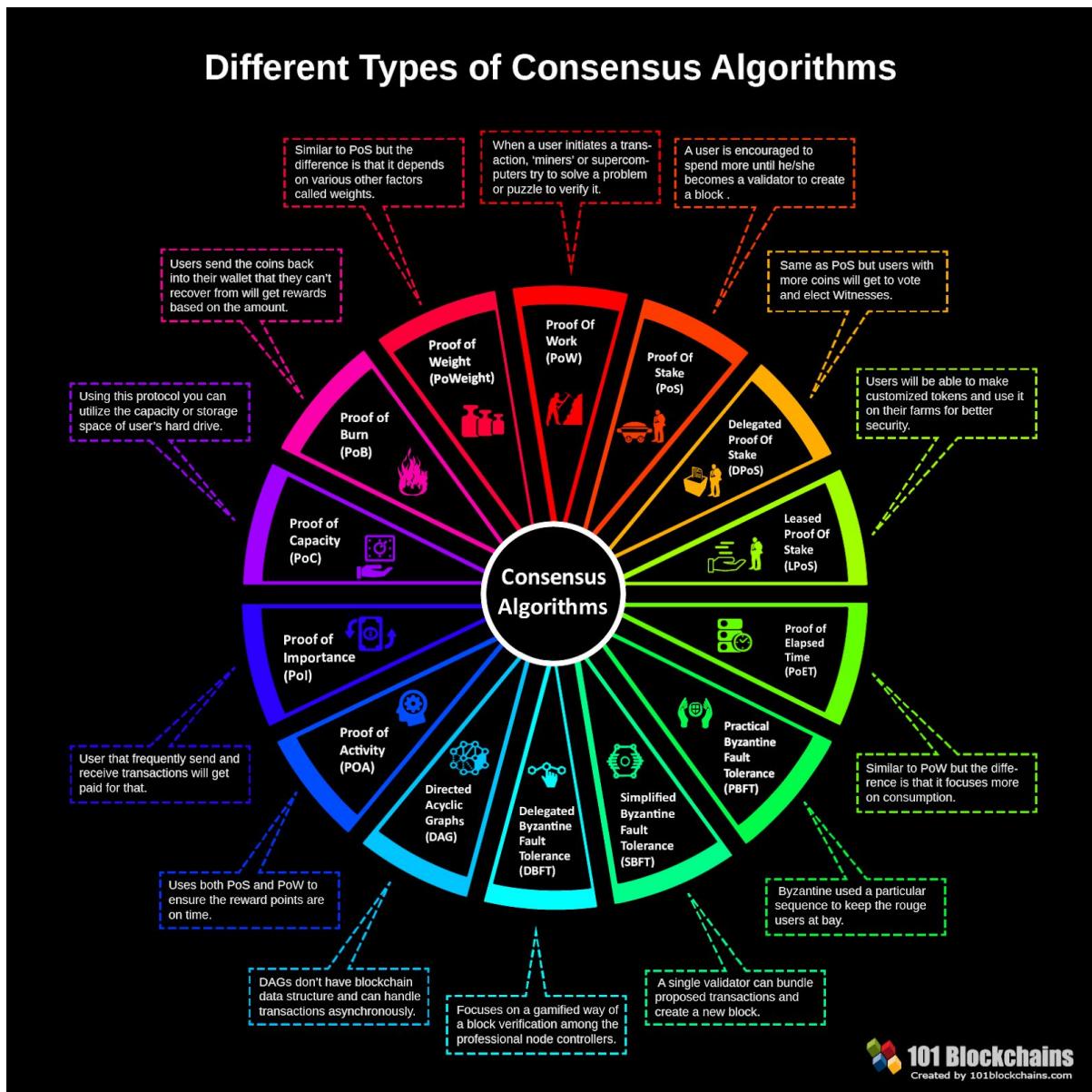
Un blocco verificato viene inviato agli altri nodi, e alla ricezione viene verificata la soluzione. Se è corretta allora viene aggiunto alla copia locale della catena.



Il PoW ha le seguenti proprietà:

- La catena che cresce più velocemente diventa quella più affidabile e sarà quella che cresce di più;
- Se un nodo malevolo vuole cambiare la transazione di un blocco intermedio nella catena, deve ricalcolare l'hash del blocco stesso e di tutti i suoi successivi, e prevalere sul resto dei nodi nella rete (deve essere quindi il più veloce);
- La blockchain è considerata sicura finché almeno il 50% dei nodi non è malevole.

Il PoW non è l'unico algoritmo per il consenso esistente, infatti ve ne sono molti altri che si basano su altri approcci.



Il consenso deve essere nella storia della catena, che deve contenere tutte le transazioni a partire da 6 blocchi indietro, quelli che vengono considerati affidabili.

Smart contract

Grazie alla popolarità dei bitcoin, i DLT sono stati utilizzati anche in altre forme. L'idea è di estendere le transazioni non solo usandole in ambito finanziario, ma andando a condividere pezzi di software che definiscono un "contratto" che si esegue da solo, detti smart contract.

Gli smart contracts sono una serie di condizioni memorizzate nella blockchain stessa (diventando quindi immutabili). Ogni nodo può quindi verificare se le condizioni del contratto sono rispettate, ed effettuare le azioni conseguenti (l'output è "validato distributivamente"). Alla fine tutti i nodi devono accordarsi sullo stato risultante, richiedendo quindi un consenso.

Gli smart contract hanno anche fatto disastri dal momento che eventuali errori al loro interno non possono essere risolti.

Tipi di DL

Un DL può essere essenzialmente di due tipi:

- Permissionless DL: tutti i nodi sono alla pari e svolgono lo stesso lavoro, l'accesso al registro delle transazioni è senza condizioni, le transazioni sono immutabili e vengono validate dai nodi stessi e vi è un principio di pseudo-anonimato tra i partecipanti.
- Permissioned DL: è un DL decentralizzato che traccia tutte le transazioni, fatta da uno o più sistemi di terze parti fidato (ad es. una banca). Per entrare all'interno della chain si ha bisogno di un consenso, ciò vuol dire che i nodi non svolgono tutti lo stesso ruolo che non tutti possono accedere al registro, e inoltre lo pseudo-anonimato viene meno (il third party sa chi siamo). Le transazioni rimangono immutabili.

L'idea dello smart contract è rivoluzionaria perché permette di eseguire codice in maniera distribuita: ad esempio nella blockchain passerà l'associazione creatore del SW: SW creato, così la ownership di un programma è stabilita oggettivamente e non con millemila giri. Ciò rivoluziona fortemente l'informatica. Se vuoi ridere, guardati CryptoKitties (<https://www.cryptokitties.co/>) è un ottimo sistema usato per testare la ownership (si sa che tu possiedi un certo gattino). Il solito problema dell'affidabilità rimane, un registro con meno del 50% di nodi sicuri è inaffidabile e bisogna quindi saper distinguere i ledger affidabili da quelli che non lo sono.

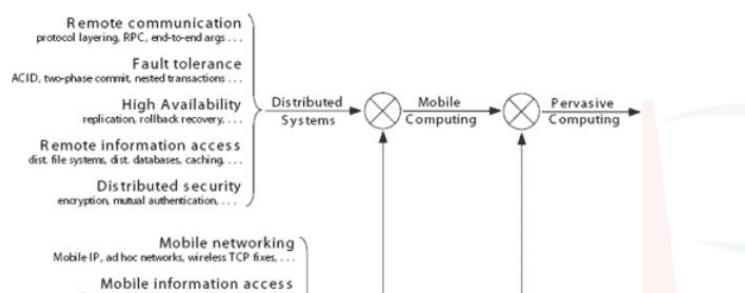
Introduzione al computing pervasivo

Un sistema pervasivo è un sistema distribuito particolare, che presenta le seguenti caratteristiche:

- Include al suo interno nodi non convenzionali come oggetti con abilità di computazione come smartphones, smart objects, ecc;
- Adattività in base al contesto: la logica di sistema riesce a conoscere il contesto corrente in cui si trova e a modificare il suo comportamento per ottimizzare l'obiettivo di sistema in base al contesto.

L'idea alla base dei sistemi pervasivi è quindi quella di avere sistemi portatili in cui il calcolo è diffuso su nodi non convenzionali e in maniera decentralizzata. Questi sistemi sono molto volatili data la natura dei nodi (sensori con batterie limitate, ecc), quindi è possibile che vi sia un alto numero di failure che comportano cambiamenti frequenti nella comunicazione tra i nodi. Di conseguenza la creazione/distruzione di associazioni tra componenti software risiede nel device stesso: tutto questo non è così banale e deve essere gestito in maniera dinamica.

Questo tipo di computing è mobile dal momento che le risorse sono limitate, vi sono



differenti tipi di interfacce, c'è un'alta varianza nella connettività e la locazione dei dispositivi è variabile (pensa a un cellulare). Inoltre a livello di rete bisogna utilizzare più tipi di networking (IP, mobile, ad hoc, ecc.), e persino l'accesso ai dati è mobile così come la loro gestione, la quale può essere spazio-temporale, Location Based, ecc.

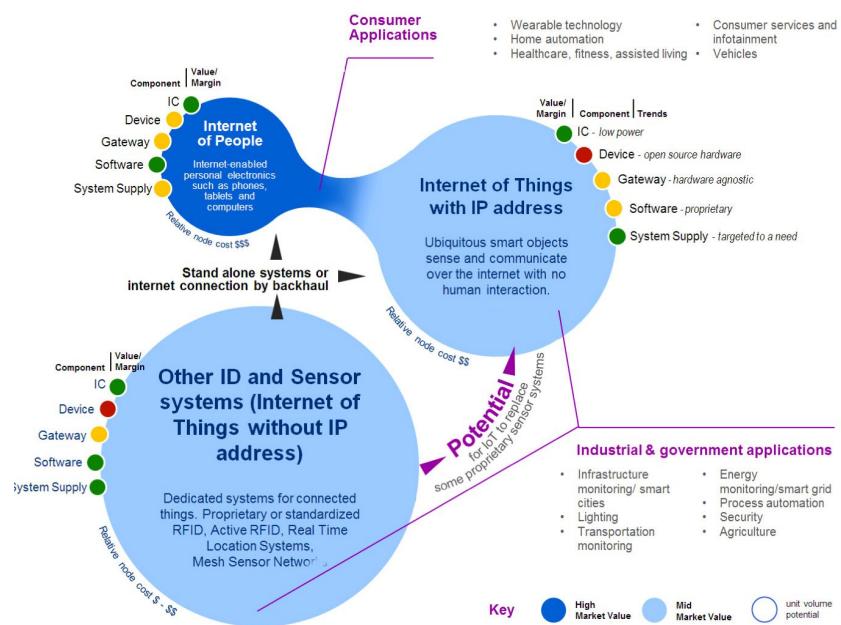
Alcuni esempi di computing pervasivo riguardano gli ambienti smart, in cui l'idea è inserire in essi dei sistemi che raccolgono e condividono informazioni. Questi sistemi sono in grado di agire sui dati raccolti quando necessario: un esempio è la gestione dell'energia al fine di minimizzare costi e consumi, i trasporti smart per diminuire il traffico ed evitare congestioni, ecc.

IoT

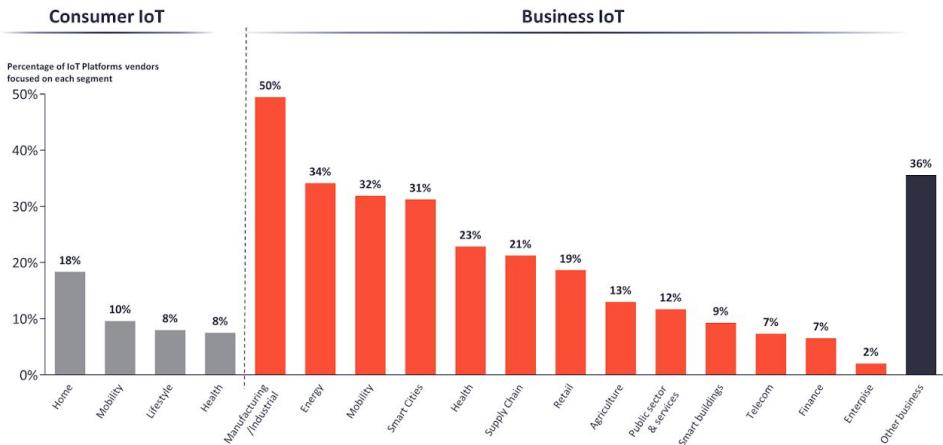
L'Internet of Things (IoT) è un particolare tipo di computing pervasivo in cui i dispositivi sono divisi in tre aree:

- device IoP(Internet of People): dispositivi che permettono alle persone di accedere a internet come smartphone, ecc.
- device IoT con IP: device che accedono a internet inviando i dati misurati;
- device senza IP: device solitamente usati in ambito IoT ma che non possono accedere a Internet, per farlo hanno bisogno di un'interfaccia esterna;

La dimensione di queste aree equivale al numero di oggetti di dispositivi utilizzati, si può inoltre notare che le componenti sono comuni a tutte le aree, quindi può esserci comunicazione tra essi.



La parte più estesa dell'IoT è quello Business per quanto riguarda gli ambiti di utilizzo, precisamente il più popolare è quello di manufacturing industriale, seguito dalla gestione dell'energia, delle mobilità e delle smart cities.



Perché “Smart”?

Con smart si indicano quei device connessi a Internet che permettono l'esecuzione di algoritmi in locale o in remoto al fine di interpretare dati. Ciò permette di capire il contesto in cui operano e quindi offrire un servizio personalizzato in base ad esso: la guida autonoma ad esempio valuta il contesto con dei sensori e risponde in tempo reale per avanti, fermarsi a un semaforo, ecc. Dal momento che tutti questi sensori sono connessi in una rete, bisogna trovare un modo per risolvere le problematiche che questi portano (o comunque di ridurne l'impatto). Si possono utilizzare ad esempio delle reti ad albero anziché a stella per evitare la saturazione del gateway.

In ambito smart bisogna sfruttare il più possibile gli spazi smart cercando di ottenere una buona adattività, context-awareness e anticipazione dei bisogni dell'utente, oltre a dover garantire invisibilità interagendo il meno possibile con l'utente mentre le risorse dovrebbero essere scoperte in maniera dinamica.

Sistemi Pervasivi: sensori e acquisizione dati

In questa sezione vedremo principalmente cosa sono i sensori di un sistema pervasivo e quali dati vengono gestiti e raccolti da essi.

Parliamo prima di cos'è un sensore, dei dati che deve gestire e poi definiremo in che modo lo fa.

Introduzione ai sensori

Trasduttori:

Dispositivi che trasformano una forma di energia in un'altra. Si dividono in :

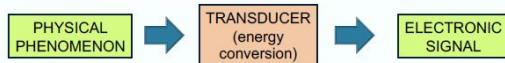
- Trasduttori Input: sono i sensori veri e propri, che acquisiscono il dato che "sentono";
- Trasduttori Output: sono detti attuatori, e sono quelli che trasformano l'impulso elettrico in un'altra forma, come ad es in comandi specifici dettati per un sistema di controllo.

Un esempio: il sensore è un microfono nella videochiamata che registra le onde sonore e le converte in impulsi elettrici, mentre le nostre casse che convertono il segnale in altre onde sonore sono gli attuatori.

I sensori possono essere fisici o virtuali. I primi misurano una quantità fisica e la trasformano in un segnale, come ad es. l'accelerometro dello smartphone. I secondi invece sono servizi e applicazioni che offrono dati contestualizzati a client remoti, ad es. un weather web service che pesca dati da sensori non proprietari usando dei servizi tipo REST.

Trasduttori: I sensori

I sensori seguono un processo di sensing:



I sensori captano i cambiamenti fisici (es. oscillazione massa) e la convertono in un segnale elettrico. L'output risulta essere un data stream.

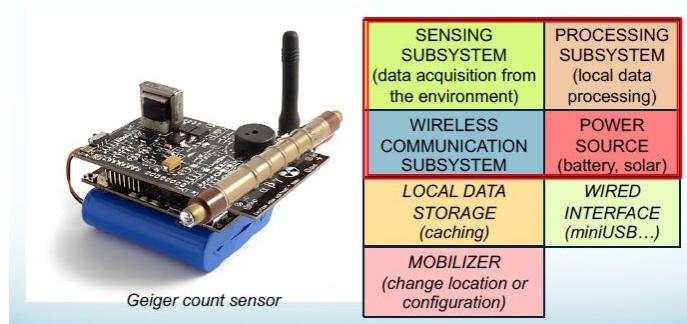
Esempio giroscopio

[How MEMS Accelerometer Gyroscope Magnetometer Work & Arduino Tutorial](#)

Trasduttori: attuatori

Sono anch'essi un tipo di trasduttori, ma che trasformano l'energia elettrica in un'azione (movimento, interruttore...) quando ricevono un comando dal sistema di controllo. Es. luci smart che quando glielo dici abbassano la luce, ecc. Normalmente sono collegati alla rete elettrica o ad una batteria. Nei dispositivi attuali, lo spazio maggiore è occupato proprio da quest'ultima.

Componenti di un sensore



Parti necessarie:

- Sensing subsystem: sistema che si occupa di raccogliere i dati dall'ambiente;
- Processing subsystem: sistema che si occupa di processare i dati e convertirli in un segnale elettrico. Può essere primitivo o avere un'alta capacità di calcolo.
- Wireless communication subsystem: sistema per collegarsi alla rete.
- Power source: la batteria che dà energia al sistema, la parte più ingombrante;

Parti extra:

- Local Data Storage: utile per fare caching dei dati nel caso servano di nuovo a distanza di poco tempo;
- Wired Interface: un'interfaccia per collegarci USB, ecc per estrarre i dati/altre operazioni;

- Mobilizer: un sistema per spostarlo dallo slot in caso ci sia bisogno di cambiare posizione o configurazione;

Ogni sensore di questo tipo è un nodo del nostro sistema distribuito.

Networking con sensori e attuatori

Si possono scegliere diversi protocolli di rete per decidere la comunicazione dei sensori. Vediamone alcuni:

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 4.x (BLE)	6/100 m	1 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

C'è una netta differenza tra consumo del WiFi rispetto a quello delle altre reti, cosa per cui di solito si usa per sistemi attaccati alla corrente e non a batteria (consumo elevato=batteria+capiente=spazio in più), inoltre la struttura della rete è completamente diversa dalle altre (star).

Piccola digressione su ZWave: è un protocollo di rete che usa una particolarità: i nodi della rete si svegliano solo quando richiesti, e non sono sempre attivi! (va saputo!)

Il problema della gestione di energia nei sistemi pervasivi è un punto importantissimo, studiato ancora oggi per migliorarla sempre di più.

Il passaggio di consumo maggiore è quello della comunicazione: vediamo come il WiFi in comunicazione sia più veloce, ma consuma un botto.

Il protocollo Thread è un protocollo che ha la particolarità di essere IPv6 addressable, cioè i nodi sono contrassegnati da un IP.

Stazioni base / Router di confine / Controller

Sono dispositivi che girano di solito su Arduino, Raspberry PI, ecc. Essi comunicano coi sensori e fanno anche da gateway tra le reti, permettendo lo scambio di dati inoltrandoli da un router a un altro fino alla destinazione: un'unità di processing (cloud/server/locale) dove verranno gestiti e processati. Possono anche avere un Database embedded al loro interno, dove memorizzare i dati nel caso si perda il messaggio a metà.

Tipi di sensori

Sensori ambientali	Sono tantissimi: misuratori di luce, temperatura, umidità, campi magnetici, inquinamento, pressione, campi elettrici, suono, elementi chimici...
Biosensori e Biosegnali	Sono usati per monitorare i cosiddetti biosegnali: frequenza cardiaca, saturazione O ² , pressione sanguigna, ecc.
Wearables	Sensori che possono essere indossati: vestiti, orologi, smartbands, microchips, occhiali, ciondoli, ecc. L'insieme di tutti i dispositivi indossati da una persona forma la cosiddetta Personal Body Network.
Sensori per le smart homes	Sono sensori che servono ad automatizzare delle operazioni in casa (es. alle 19 chiudi tutte le tapparelle di casa, abbassa il termostato se T>30°C...)
Beacon BLE	Sono dispositivi creati per effettuare la microlocalizzazione dei dispositivi tramite invio di segnali Bluetooth. Di solito li faceva passare in classe ma nada.
Smartphone-based sensing	I sensori sono anche presenti sugli smartphone, e usati da apposite app per monitorare il meteo, le attività dell'utente guardandone la posizione, ecc. Esistono delle app che monitorano e misurano la nostra qualità della vita, come MI Fit che dice quanto tempo dormi ogni notte, che esercizi fare, ecc.

Gestione e utilizzo dei dati dei sensori

Discovery e pairing del dispositivo

I sensori possono apparire e sparire molto spesso in uno spazio smart: ad es uno smartwatch non sta sempre in casa ma va anche fuori. Abbiamo quindi bisogno di collegarci alla rete esterna ed essere sempre identificabili attraverso un indirizzo unico (network bootstrapping). L'associazione del device con l'applicazione che lo usa è basata sui limiti dello smart space (i boundaries) e i servizi del device: prendiamo come esempio la serratura della porta di una camera d'albergo. Devo fare in modo che il segnale di associazione resti all'interno delle quattro mura della stanza, che sono effettivamente i boundaries del dispositivo.

Acquisizione dei dati

Esistono diversi metodi per acquisire i dati dai sensori, sia base che avanzati. Vedremo prima quelli base e poi quelli più complessi.

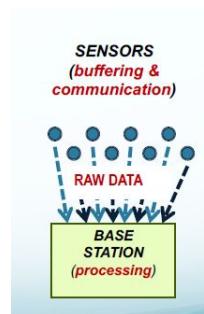
Il problema dell'acquisizione dei dati è molto diverso rispetto a quello dei DB tradizionali:

- **Continuous vs one-time query:** le query sono continue, in quanto i sensori restituiscono uno stream di dati in continuazione. Naturalmente non faccio una query per ogni dato: sarebbe super dispendioso e stupido.
- **I dati sono fortemente caratterizzati dallo spazio e dal tempo** (strong spatial-temporal characterization of data): il tempo nei data stream è fondamentale! Un dato deve arrivare il più presto possibile. Anche lo spazio è fondamentale: devo poter localizzare i dati in maniera precisa ed efficiente. Le query quindi possono essere spazio temporali.

Metodi base

Batch processing

Primo metodo base: si fa buffering nel sensore, e si processano i dati offline nella base station (o nel server remoto). In questo modo si ottiene una risposta precisa, senza processare nel sensore. Questo però comporta un grosso costo, perché si fa tanta comunicazione. Inoltre, la base station riceve tutti i dati, ma con un delay che può diventare ingestibile se i dati sono troppi.



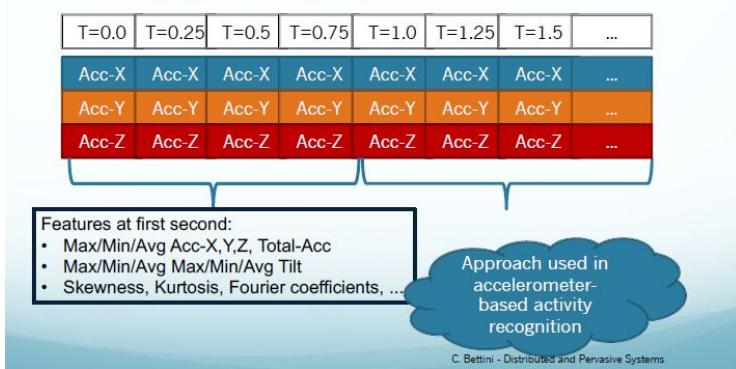
Sampling

Secondo metodo base: si campionano i dati, considerando la distribuzione di probabilità attesa dei valori. In questo modo prendo solamente i dati che caratterizzano di più il sistema. Inoltre creo dei bounds formali che mi tengono l'errore di approssimazione indotto dal campionamento sotto controllo (supero il boundary==prendo dato subito sotto/sopra).

Sliding windows

Approssimo la risposta basandomi su un gruppo di letture consecutive usando finestre temporali. Quello che faccio è una computazione sul device a runtime, e manda i dati alla base station sotto forma di stream di letture dei valori dei sensori approssimati (con un piccolo ritardo). Più è grande la finestra, maggiore il tempo che passa prima di una nuova ritrasmissione.

Example from an accelerometer:



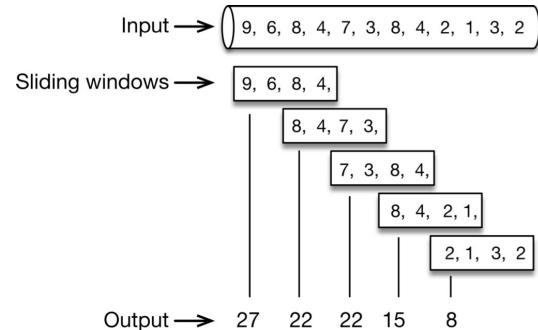
Le features sono delle funzioni di aggregazione che decidono come usare i dati. Al sensore quindi non arriva il dato singolo, ma le grandezze calcolate tramite la feature!

Un errore che si fa di solito è mettere le finestre sempre consecutive: in questo modo la frequenza di invio è bassa, e se per caso c'è un cambio repentino dei dati tra le due finestre temporali rischio di non rilevarlo. La soluzione è usare le Overlapping Sliding Windows (finestre temporali sovrapposte).

Overlapping Sliding Windows

Quando uso un algoritmo basato sulle sliding windows, faccio in modo di specificare non solo la larghezza della finestra, ma anche la percentuale di overlapping tra di esse: in questo modo avrò una finestra temporale che sarà in parte sovrapposta con quella precedente: nell'esempio, le finestre si sovrappongono per il 50% (f1: 9,6,**8,4** → f2: **8,4,7,3**).

Con l'overlapping ho una frequenza di invio molto più alta, e ho precisione migliorata in quanto replica dei dati, e dunque c'è meno delay tra un dato e l'altro. Inoltre non perdo un cambio repentino avvenuto tra due finestre: gli ultimi dati sono replicati nella successiva, perciò non perdo nulla. Non ci sono standard specifici su quanto deve essere la percentuale di overlapping: si va a esperienza.



Metodi avanzati

I metodi base a volte sono troppo semplici per essere utilizzati: i dati inviati sono di qualità bassa (errori di approssimazione, ecc), e non si ottimizza il consumo di energia.

Questi due punti sono ciò che i metodi avanzati cercano di migliorare: ridurre al minimo possibile il consumo e migliorare la qualità dei dati. Sappiamo che la trasmissione di un dato è molto più power consuming del processing (Trasmissione di 1 bit=1000 operazioni CPU!), mentre invece il costo del sensing dipende dal fenomeno che si vuole “catturare”.

I metodi avanzati lavorano quindi sulla riduzione delle trasmissioni, e sull’ottimizzazione del sensing per ottenere dati più precisi.

In-network query processing

Questo metodo si oppone al central processing che abbiamo visto prima. L’idea è di costruire una rete di overlay, spesso a forma di albero inverso, facendo l’aggregazione dei dati in nodi intermedi dell’albero: in questo modo riduco la mole di dati trasmessi alla base station, in quanto computo sulla rete di overlay presente sul sensore e non sulla control station. Si utilizza soprattutto per computare la media tra numeri, il massimo/minimo... tutte quelle operazioni che ritornano meno dati di quelli in input.

Duty Cycling

La control station viene messa a dormire per la maggior parte del tempo. Quando ci sono dei dati da processare, viene svegliata da delle operazioni di sleep/wakeup coordinate da un algoritmo di scheduling, che la sveglia e le fa processare i dati. In questo modo ho un risparmio facendo le trasmissioni solo nel caso quando effettivamente ci sono dati da computare. Negli altri casi la control station non è attiva, perciò non consuma energia. Questo metodo non si adatta molto bene con le sliding windows.

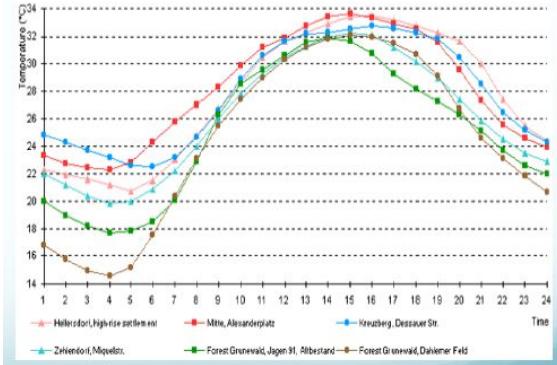
Metodi Mobility-based

Se la base station è mobile (es nodi portati da macchine, uomini, animali...) bisogna sfruttare la mobilità dove possibile, per distribuire il costo delle trasmissioni su più nodi, ad es. usando algoritmi per sfruttare la vicinanza di un nodo alla base station (meno trasmissioni intermedie tra router).

Metodi Model-based (a.k.a. “Data-driven”)

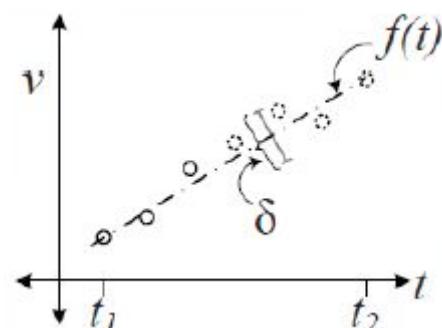
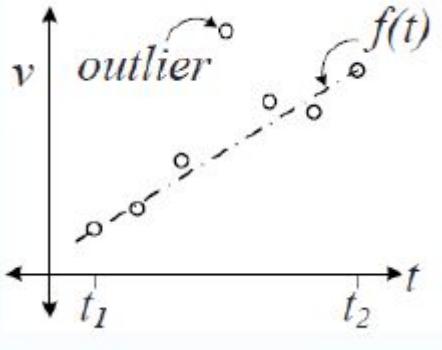
I dati vengono campionati, e presentano una forte correlazione spazio-temporale (spazio e tempo influenzano i dati): so che un sensore misura la temperatura, perciò so che la notte la temperatura cala di 10°C e risale di giorno. Quindi c’è una correlazione temporale in questo sbalzo di temperatura. Il mio obiettivo è ridurre il numero di campioni (letti o inviati) mantenendo comunque una buona qualità dei dati. Per raggiungerlo, devo sfruttare il modello costruito sul fenomeno osservato.

Quindi, si crea un modello basandosi sulla statistica (es. variazione di temperatura media in italia in primavera secondo dati istat, ecc.), e si campionano i dati su quel modello.



Ci sono diversi approcci model based:

- Data cleaning: tipicamente i dati da sensori non hanno sempre i dati giusti, a volte ci sono dati sbagliati che vanno ignorati. Per fare ciò, usiamo i modelli che ci aiutano a capire se un dato è/non è ragionevole. Il modello in questione è una funzione lineare, e i dati seguono questa funzione. Se notiamo un dato che non segue questa funzione (es. $f(x)=10$, $f(next)=12$ ma dato =35) allora so che è sporco (è un outlier) e non lo trasmetto. Il risultato è una qualità molto più alta dei dati, che sono puliti.
- Data acquisition: Se so che il modello è una funzione lineare del tempo, non trasmetto subito i dati quando li ricevo. Invece trasmetto solamente i dati che non seguono quel modello cioè i dati che superano un certo δ . In questo modo vedo solamente quando i dati variano, permettendomi di correggere eventuali situazioni che non rispettano il modello (es. temperatura sopra un certo threshold, accendo il condizionatore. Troppo freddo: li spego).



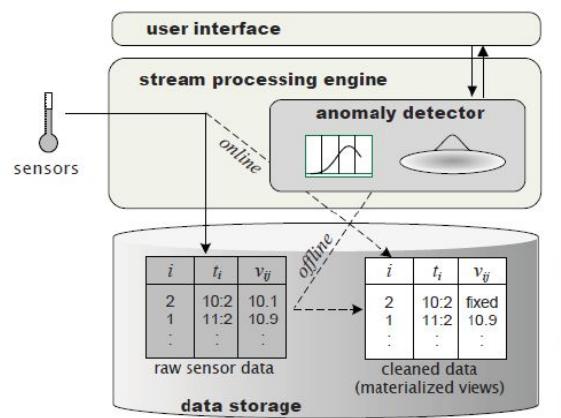
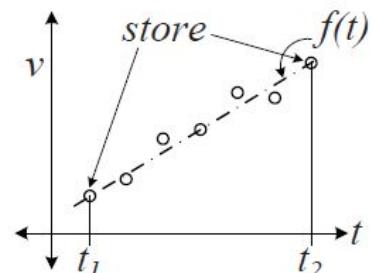
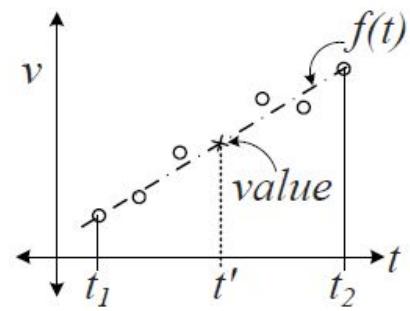
- Query processing: la base station manda i dati a un'applicazione, non li tiene mai per sé. L'app quindi interroga la base station, richiedendo i dati che le servono. Quando faccio una query alla station all'istante di tempo t' , questa è calcolata in una finestra temporale che parte dal dato trasmesso a tempo t_1 fino all'ultimo dato trasmesso t_2 . In questo modo ho non l'esatto dato che è stato trasmesso all'istante t' (quando è stata fatta la query), ma un intervallo di valori, su cui poi l'applicazione provvederà a calcolare il valore esatto. In questo modo non risparmio energia, ma spazio sul DB dell'applicazione: ci memorizzo non un solo dato, ma un intervallo di valori e nella riga a fianco associo la funzione per calcolare il risultato. (meno righe usate, non uso una riga per valore ma una riga per es 200 valori).
- Data compression: si cerca di eliminare la ridondanza dei dati. Da quel che ho capito, memorizzo solo gli estremi di un intervallo di valori, mentre gli altri li calcolo se mi servono. Così non devo memorizzare una mole eccessiva di dati.

Sensor Data Cleaning

I dati dei sensori sono incerti e proni ad errori: prendiamo ad esempio un sensore che ha la batteria scarica: esso non produrrà dati. Stessa cosa per le reti su cui comunicano: un eventuale disservizio tronca i dati. Se i sensori sono di bassa qualità si surriscalderanno/congeleranno, danneggiando il sistema di misurazione che invierà dati errati. Stessa cosa succede con l'accumulo di polvere o per atti vandalici sui sensori (vedi antenne 5G non cielodicono!).

Facciamo quindi data cleaning model based: I valori più probabili captati dai sensori vengono confrontati con un modello statistico, e le anomalie sono individuate comparando i dati raw dei sensori con i valori comparati col modello. (in inglese, anomalies are detected by comparing raw sensor values with the corresponding inferred sensor values).

I dati captati dai sensori vengono sia salvati in una tabella in raw, che puliti attraverso il confronto col modello in un anomaly detector. Dopo averli confrontati tra loro, si piazzano i dati puliti in una tabella chiamata cleaned data. I modelli di regressione possono catturare sia la continuità che la correlazione di più dati. Un modello usato di solito è la regressione polinomiale:



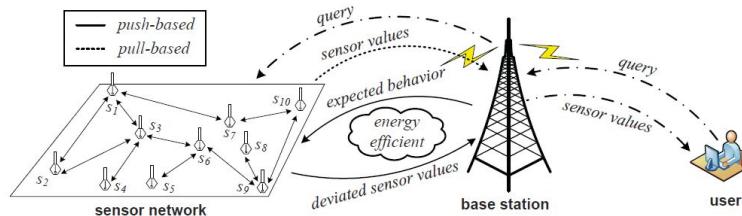
$$v_{ij} = c + \alpha_1 \cdot t_i + \dots + \alpha_d \cdot t_i^d$$

Ma esistono molti altri modelli che possono essere utilizzati.

Model-based Data Acquisition - Acquisizione dei dati

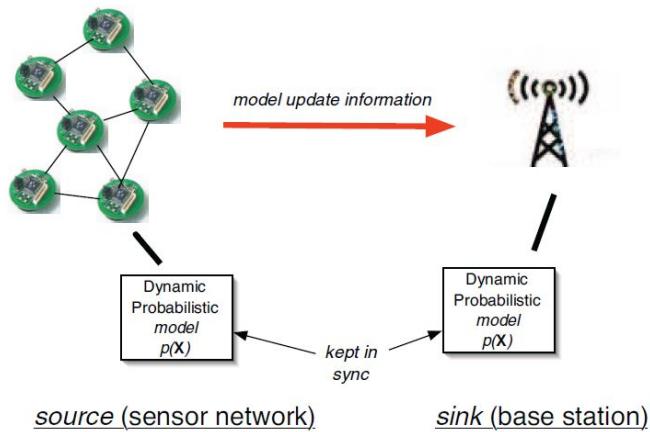
Penso farla push-based o pull based. si basano entrambi su una rete di sensori che comunicano verso una base station, la quale comunica con l'utente tramite applicazione.

- Push-based: La base station (chiamata anche SINK, perché i dati finiscono tutti su di essa) si accorda con i sensori su quale comportamento si pensi che i dati seguiranno (il modello), ad esempio la normale fluttuazione della temperatura durante il giorno. I dati che riceve dai sensori saranno esclusivamente quelli che NON seguono il modello (deviated sensor values). Questo metodo è molto energy efficient, perché i sensori trasmettono solo i dati che variano dal modello, perciò pochi rispetto al totale.
- Pull-based: L'utente richiede i dati alla base station attraverso una query fatta dall'applicazione, e decide l'intervallo e la frequenza dell'acquisizione ("ogni quanto voglio fare la query, e per quanto tempo?"). La base station quindi inoltra la query sulla rete di sensori, che le restituisce ciò che ha chiesto. La base station inoltra poi i valori dei sensori trovati all'utente.



Per quanto riguarda l'acquisizione push-based, i modelli scelti possono essere:

- Modelli probabilistici:
 - Modello di predizione lineare:
 - $X_i^{t+1} = \alpha_i X_i^t + \beta_i$
 - Considera solamente i fattori temporali e tratta ogni sensore indipendentemente.
 - Modelli Markoviani
 - Considerano le dipendenze spazio-temporali tra i sensori.
 - Lo stato iniziale è definito dalla funzione $p(X_1^{t=0}, \dots, X_n^{t=0})$
 - La probabilità di transizione è invece: $p(X_1^{t+1}, \dots, X_n^{t+1} | X_1^t, \dots, X_n^t)$



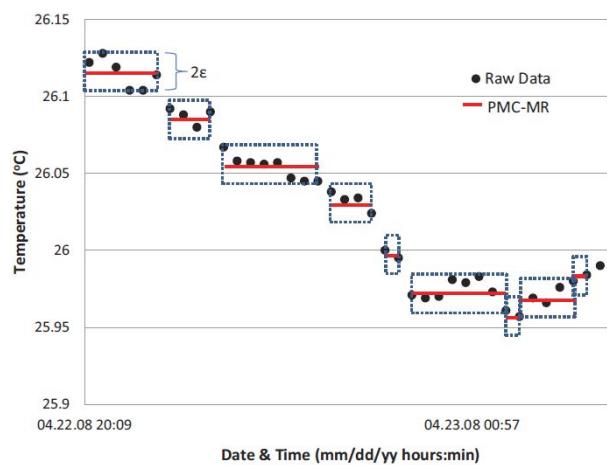
Compressione model-based

I sensori possono produrre grandi quantità di dati. L'obiettivo è quindi approssimare il data stream di un sensore in un set di funzioni. Per fare ciò si possono usare metodi di regressione, trasformazione e filtraggio, sfruttando le correlazioni spazio-temporali dei data stream. Usare metodi di trasformazioni ortogonali (trasformazioni di Fourier o wavelet) riducono la dimensionalità dei dati.

Vediamo un esempio di compressione model based.

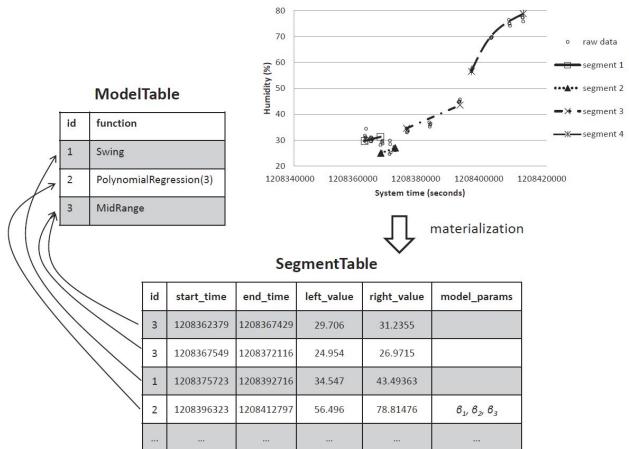
Poor Man's Compression MidRange (PMC-MR)

E' usato molto per la memorizzazione veloce nel DB di dati complessi. Quello che fa è approssimare un segmento di dati con un valore costante $((\text{Max}-\text{Min})/2$ o simili). In un certo intervallo di tempo quindi ho un solo valore da memorizzare: memorizzo quindi l'inizio dell'intervallo, la fine dell'intervallo, e il valore costante calcolato. Quando notiamo che lo stream ha superato un certo valore di limite dell'errore (bounded error) ϵ , modifico l'intervallo scelto, scendendo/salendo fino a che i valori stanno dentro al nuovo ϵ del nuovo intervallo. L'intervallo è sempre 2ϵ in ampiezza!



Approssimazione Multimodello

Uso un certo modello in un periodo di tempo, e un altro in un altro. In un DB memorizzo il segmento temporale associato al modello usato (es. da t1 a t5 ho usato il modello con funzione di swing).



Context awareness

Cos'è il contesto?

Le applicazioni convenzionali non capiscono il contesto di una richiesta: l'utente deve specificare in maniera esplicita tutti i parametri da cercare. Un'app normale ad es. non riesce a risolvere la seguente domanda: "Come vado da X a Y, sapendo che sono in macchina e che è ora di punta?".

Comunicare tenendo conto del contesto non è facile, anche perché ci sono poche interfacce (pensiamo a Google che mostra solo i risultati più pertinenti al contesto di chi scrive, che ha poco spazio su mobile per mettere i risultati su una pagina) e abbiamo bisogno di meccanismi automatici per acquisire e trasmettere il contesto della nostra richiesta, senza doverne specificare i parametri.

Definizione di Contesto

Il contesto è il set di circostanze o fatti che circondano un evento o situazione particolare; nel nostro caso l'evento generato è una chiamata di servizio, sia mobile che PC.

Esistono moltissimi statement che hanno senso solo se all'interno di un contesto: si pensi alla frase "Come arrivo là?"; Come fa un'applicazione a capire che sto chiedendo di andare in un certo posto, senza considerare il contesto? Non può. C'è quindi bisogno che l'applicazione si derivi da sola cosa fare in base al contesto trovato (es. so che il mio utente va sempre a piedi alle 15 da Mario, perciò se mi chiede quanto tempo ci metto ad arrivare là so che là=Mario e a piedi ci metto 20 min). Le prime definizioni di contesto erano basate su enumerazione:

- [F1]: location, time, surrounding people, season, temperature...
- [F2]: location, environment, identity and time
- [F3]: mood, level of attention, location, time, surrounding objects and people

... Ma non sono molto buone perché con il passare del tempo è cambiata la tecnologia che sta dietro.

Un collega di Bettini è riuscito a dare una definizione di contesto ancora migliore:

"Il contesto è qualsiasi informazione che può essere usata per caratterizzare la situazione di un'entità. Un'entità è una persona/luogo/oggetto considerato rilevante all'interazione tra utente e un'applicazione, inclusi gli utenti e l'applicazione stessi".

Semplificando: Il contesto sono tutti i dati utili per adattare un servizio.

Cosa contiene un contesto?

Nella tabella sottostante c'è una tassonomia (indicativa, non sono tutte qui le cose) che divide le informazioni presenti nel contesto in base a ciò che caratterizzano di più (utente, luogo, dispositivi...). Le sorgenti nella tabella sono molto importanti perché indicano da cosa il contesto prende i dati.

Dimensione primaria	Dimensione secondaria	Sorgenti
Utente	Identità Fisiologica Emozionale Interessi Preferenze Social Organizzazionale Attività	U S S U,O U,O U,S,O U,O U,S,O
Ambiente	Meteo Temperatura Umidità Luce Rumore Inquinamento Aria Persone/Devices circostanti	S S S S S S S,O
Localizzazione	Fisica Simbolica	S S,O
Tempo	Assoluto Granularity-based	D,S D,S,O
Dispositivo	Capacità Stato	D D
Connettività		D,O

U = user S = sensors D = device O = other

Spazio e tempo sono fondamentali all'interno del contesto: il sistema lo sa che la posizione geografica che ho dove sono io adesso è l'università, e che sono al decimo piano e non di fianco all'entrata, e che voglio sapere quanto tempo ci metto a scendere?

Il contesto è importante anche nei dispositivi: Serve un sacco a mobile, soprattutto all'inizio quando non c'era HTML5 e si dovevano adattare le pagine web secondo il device usato dall'utente.

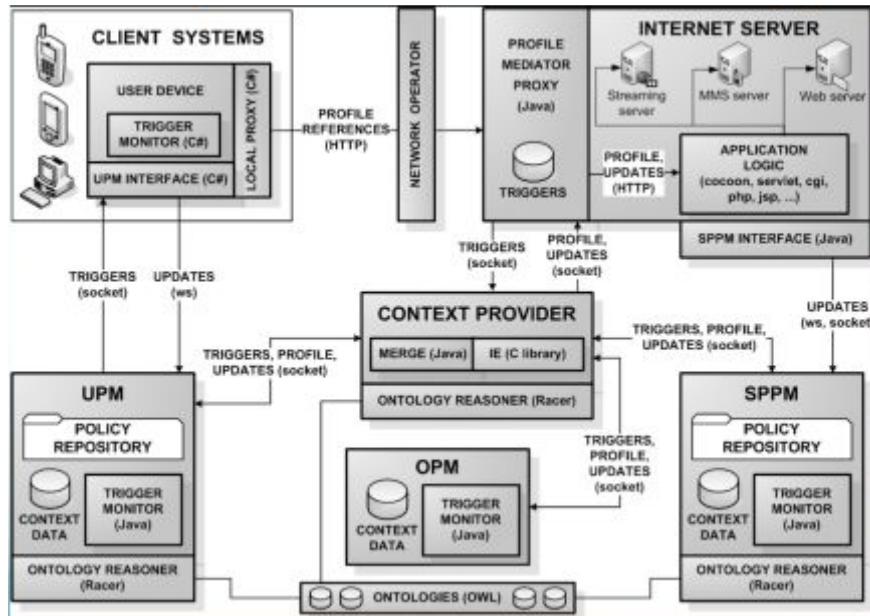
Il contesto temporale non corrisponde solamente al mese/giorno/anno, ma anche alla storia (history) del contesto stesso: possiamo usare la storia per ricostruire non delle posizioni dell'utente, ma la traiettoria seguita da esso, o le sue abitudini (es. dalle 9 alle 10 si sposta velocemente per il quartiere-->Jogging) e non solo ricostruire, ma predire le sue azioni, soddisfando eventuali bisogni anticipandoli senza richiesta dell'utente (ad es. dopo le 12 gli suggeriamo un chiosco buono dove mangiare in pausa pranzo).

Adattività

L'adattività misura quanto un sistema riesce ad adattarsi al contesto. E' particolarmente importante per il mobile, ma anche per i SP, per i seguenti motivi:

- La rete cambia spesso;
- Se cambia l'erogazione di energia ai devices (es. batteria scarica), cambia il contesto!
- Le interfacce sono limitate e non riesco a specificare gli stessi contenuti specificabili su un pc. Inoltre il metodo di uso delle interfacce cambia (es. se salgo in macchina non posso più usare il touch per comunicare col cell).
- Variazioni nella situazione dell'utente (es. vado in riunione).
- Cambio dell'ambiente circostante (luce, temperatura, folla...).

Nell'immagine sotto vediamo un esempio di architettura adattiva. Si tratta di un sistema di video streaming adattivo che aggiusta la qualità dell'immagine in base a diversi fattori (larghezza di banda, batteria, ecc.) senza interrompere il flusso video. L'idea generale mostrata in questa architettura è che il contesto può avere più sorgenti che vanno messe assieme (nell'esempio i nodi UPM, OPM, e SPPM) in un device e poi usati per riportare un contesto unico. Un esempio sarebbe effettuare una media dei contesti pesata per attendibilità (1 provider, 0.5 i operator, 0.2 l'user).



Tipi di adattività

Si cerca di adattare:

- Le Funzionalità: il sistema cambia le sue funzionalità in base al contesto (es. touch bar del mac, che nasconde i tasti funzione quando sono in un'app tipo Photoshop).
 - Usato per es. quando si aumenta il caching (vediamo che X accede alla risorsa Y 5 volte ogni 5 minuti, porto la risorsa in cache), spostare la computazione a server side (come in Edge Computing), o cambiare l'interfaccia (da touch a vocale quando entro in macchina).
- I Dati: fornisco dati diversi in base al contesto.
 - Ad es. fornisco dati di posizione di precisione diversa, usare qualità maggiore/minore nello streaming, ecc.

Ottenere i dati dal contesto

Vi sono due classificazioni diverse del contesto, definite ad alto e basso livello.

- **Contesto di basso livello** (low level context): abbiamo due tipi di dati
 1. Dati acquisiti direttamente dalle sorgenti (sensori, ecc.):
 - a. Alcuni esempi: le accelerazioni sui tre assi, la luminosità o informazioni esplicitamente indicate dall'utente;
 2. Dati ottenuti da un processing semplice e/o fusione di dati raw;
 - a. Alcuni esempi: stima della bandwidth facendo la media di un set di samples nella time window (le sliding windows della scorsa lezione), o la classificazione dei dati basata su threshold;
- **Contesto di alto livello**: sono dati che richiedono l'applicazione di inferenze sul contesto di basso livello acquisito, oppure ragionando sul contesto ad alto livello e sul common sense (il buonsenso che capiamo durante il corso della nostra vita, come "attraversare i binari è pericoloso");

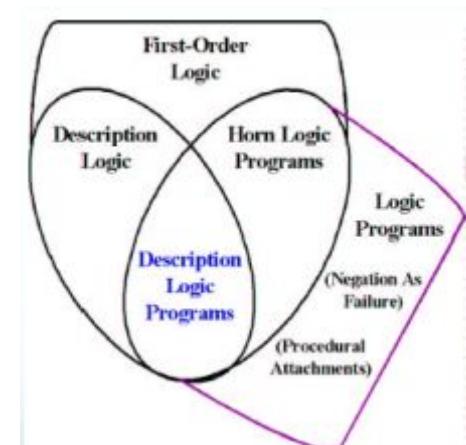
- Esempio: lo stato d'animo di un utente si ottiene combinando i dati dalle attività durante il gg, dal GSR (Galvanic Skin Response) misurati dal polso, e dalla face recognition (se disponibile).
- Altro esempio: l'attività di "cucinare un pasto" è un'inferenza sulla posizione e i dati dei sensori, usando un algoritmo di machine learning;

Osservazione: la parte di ragionamento sul contesto high-level indica che l'inferenza è applicabile ricorsivamente. Posso infatti applicare un'inferenza sui dati del contesto di basso livello per ottenere i dati di alto livello, e poi riapplicare l'inferenza sui dati ad alto livello appena trovati.

Metodi di Inferenza

Vi sono diversi metodi per realizzare un'inferenza sul contesto:

- Approcci simbolici: basati sulla logica, prendono dei simboli come base per poi sostituirli con la casistica reale;
 - Programmi basati sulla logica di Horn
 - $(p \wedge q \wedge \dots \wedge t) \rightarrow u$ (se user in bagno, ed è nudo, e il termometro segna 45°C ed è estate, allora apro l'acqua della doccia sul freddo siberiano per raffreddare l'utente);
 - Programmi basati sulla logica generica;
 - Si aggiunge la negazione, che rende le inferenze molto più complesse da creare e gestire;
 - logiche descrittive;
 - logiche descrittive + regole;
- Approcci statistici: si basano su modelli statistici costruiti nel corso del tempo, supervisionati e costruiti da un algoritmo di machine learning. Cercano di risolvere due tipi diversi di problemi:
 - Classificazione: decido delle classi generali dove inserire le varie situazioni (es. corsa, camminata, seduto, in piedi, sdraiato) e faccio un'inferenza per sapere se la situazione corrente ricade in una di queste classi.
 - Clustering: non definisco delle classi a priori, ma le ricavo mettendo nella stessa classe proprietà con dati simili tra loro (es se sto correndo a 10km/h, metto nello stesso cluster tutti i dati che hanno "corsa a 10 km/h" o simile, magari 9,11 km/h).
- Approcci Ibridi: sono un'unione tra approcci statistici che si fanno per primi, e poi si arricchiscono i dati con approcci simbolici. In questo modo ottengo dei dati più precisi di quelli approssimati con la statistica.



Activity Recognition

Per spiegare come riconoscere l'attività, si fa il seguente esempio: si fa un po' di preprocessing applicando i dati grezzi a dei sensori filtri con lo scopo di ridurre il loro rumore; Poi si suddividono i segnali in finestre temporali e infine si aggregano i segnali di diversi sensori acquisiti nella stessa finestra temporale.

Da questa aggregazione estraggo delle caratteristiche statistiche da ogni finestra temporale, e tramite un modello scelto cerco di classificare i dati in 5 categorie (nell'es. run, walk, stairs, standing e sit), ottenendo una distribuzione di probabilità plausibile che ci indica quale attività è la più frequente (nell'es. è la corsa al 55%).

Infine si passa alla fase di predizione: si predice cosa farà l'utente associando l'attività svolta al vettore di features creato.

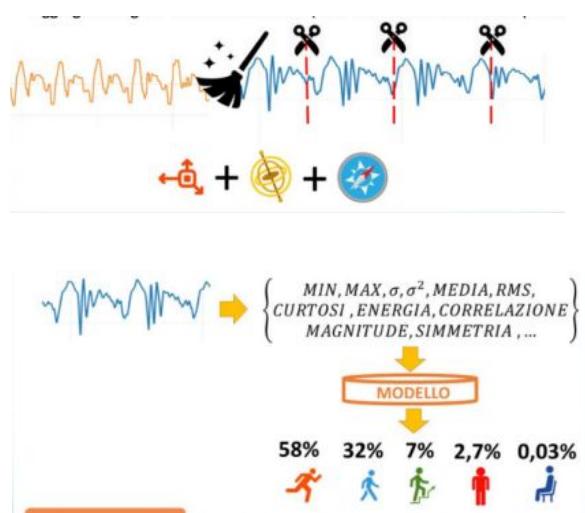
Come si inizializza questo tipo di activity recognition?

Usando un algoritmo di machine learning, possiamo definire un training set di comportamenti tipici esplicitamente definiti (es. quando corro, segnalo che "sto correndo..." per farlo capire all'algoritmo). A un certo punto avrò dei dati abbastanza generali, e quindi rimuovo il training set, lasciando l'algoritmo di machine learning con un bagaglio di valori tipici da confrontare con l'attività dell'utente che sta facendo qualcosa in quel momento.

Ragionamento Ibrido nell'activity recognition

Spesso i modelli statistici non sono perfetti: i dati sono approssimati, e quindi poco precisi. Per ridurre l'approssimazione, uso approcci ibridi dove si arricchisce con un ulteriore ragionamento simbolico

la predizione ottenuta usando il ragionamento statistico. Unire le informazioni semantiche (es. sono in aula e non fuori dall'uni) con quelle raw dei sensori porta una riduzione ulteriore dell'approssimazione. Esempio: per il machine learning, lavarsi i denti produce gli stessi dati raw di scrivere alla lavagna; capiamo che le due cose sono diversissime, ma i dati raw non bastano. Aggiungiamo dunque un dato semantico, e cioè la posizione dell'utente: se è a scuola, probabilmente sta scrivendo alla lavagna, mentre se è a casa, si starà lavando i denti. I dati risultano quindi molto più precisi.





Ad esempio nella figura sono a lavoro: quindi so per certo che non sto correndo, riducendo l'imprecisione dei dati raw che mi avrebbero magari rilevato come "in corsa".

Rappresentazione del contesto

E' molto importante rappresentare il contesto, per ragioni differenti: i dati sono ottenuti da sorgenti eterogenee (non è detto che tutti i dati vengano solamente da sensori). Ciò comporta che ci sono sorgenti che usano diversi linguaggi, e non utilizzano una semantica comune.

Inoltre, gli stessi dati di contesto possono essere usati da molteplici applicazioni e perfino condivisi tra di esse: ci serve dunque un linguaggio di rappresentazione universale;

E' necessario inoltre processare automaticamente questi dati, per evitare di farli processare all'utente, creando una rappresentazione formale di essi.

I modelli di rappresentazione di contesto devono seguire le seguenti caratteristiche:

- Gestire eterogeneità e relazioni dei dati, e rappresentarne l'incertezza e l'imprecisione;
- Mantenere la storia del contesto;
- Effettuare un ragionamento sui dati;
- Usabilità del sistema;
- Efficienza del modello;

Possiamo capire come questi diventino attributi con cui vaultare un buon sistema di rappresentazione del contesto.

Modelli di tipo flat

Sono modelli generati ad hoc per una singola applicazione. Sono basati su:

- coppie chiave-valore: i dati non sono strutturati, usati di solito dai servers delle applicazioni commerciali.
- Linguaggi di markup (XML): hanno una struttura leggermente articolata, offrono gerarchie di attributi e sono supportati da tecnologie solide (XML parsers, XPath, XSLT);

I modelli flat sono quindi molto efficienti e facili da usare poiché leggeri e con poca (o nulla) struttura, e anche abbastanza buoni dal punto di vista della gestione dell'eterogeneità, ma questa mancanza di struttura porta a problemi per quanto riguarda la rappresentazione di relazioni fra i dati, il supporto alla storia del contesto e all'incertezza dei dati, nonché il ragionamento.

Il problema della mancanza di relazione tra i dati può essere risolto usando i database relazionali.

Modelli DB-based

Modello le relazioni fra i concetti usando un DB relazionale. Ha come obiettivo l'uso di modelli formali basati sullo schema E-R per effettuare querying e ragionamenti. Un esempio di questi è CML (Context Modelling Language).

CML usa notazioni grafiche per semplificare la specificazione dei requisiti del contesto fatte dalle applicazioni. Costruisce quindi dei modelli per le classi e sorgenti dei dati del contesto, l'Incertezza, le dipendenze e la storia dei dati del contesto;

Inoltre, pur con qualche limitazione, è possibile integrare direttamente i modelli CML in un DB relazionale.

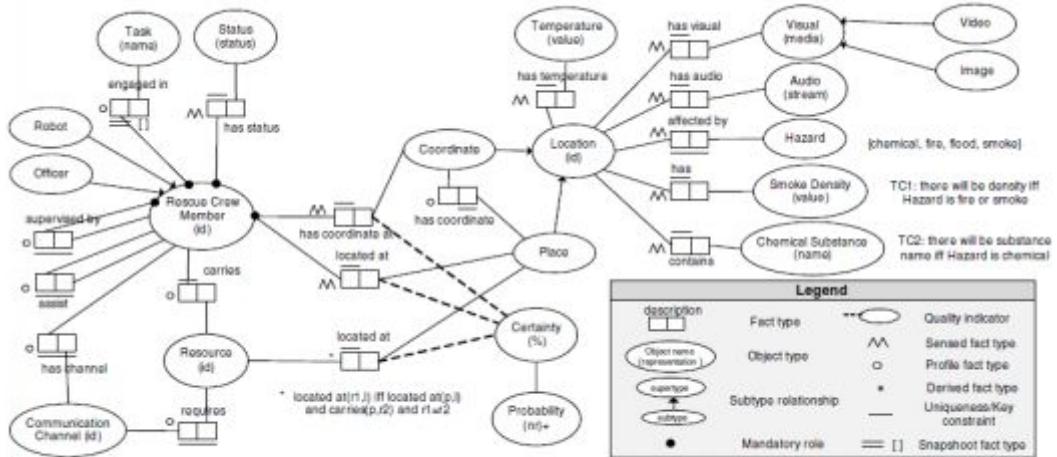


Fig. 1. Command Centre Application Context Model (Simplified)

Il pro dei modelli DB-based è che possiamo catturare tutte queste cose viste sopra, mentre i contro sono che l'eterogeneità non è perfettamente rappresentata, la storia dei dati è terribile perché dovremmo percorrere la struttura del DB al contrario, così come il ragionamento.

L'ultimo tipo di modelli che vedremo sono i modelli ontologici.

Modelli ontologici

Un'ontologia è una specificazione formale di una concettualizzazione condivisa. Il dominio di interesse è descritto in termini di classi, relazioni, individui e proprietà; Utilizza come linguaggio la logica descrittiva, e quindi useranno una famiglia di subset della logica del primo ordine. I modelli permettono la modellazione di domini molto complessi, utilizzando la semantica formale e permettono di effettuare in modo automatico ragionamento e consistency checking;

Un linguaggio per definire le ontologie è il linguaggio OWL2, che permette di ottenere descrizioni complesse dalla composizione di descrizioni più semplici: ad esempio

$$\begin{aligned} \text{CarnivalParty} &\sqsubseteq \text{FriendlyMeeting} \sqcap \forall \text{hasActor}.(\text{Person} \sqcap \exists \text{isWearing}.\text{Mask}) \\ \text{FriendlyMeeting} &\sqsubseteq \text{Activity} \sqcap \geq 2 \text{ hasActor}.\text{Friend} \end{aligned}$$

Ci permette di definire una festa di carnevale come un incontro amichevole tra persone mascherate, definendo anche l'incontro amichevole come un'attività fatta da almeno due persone che siano amiche tra loro. Un altro esempio è quello del Tea Party:

$$\begin{aligned} \text{TeaParty} &\sqsubseteq \text{SocialActivity} \sqcap \forall \text{hasTimeExtent}.\text{Afternoon} \sqcap \forall \text{hasActor}.(\text{Person} \sqcap \\ &\exists \text{hasCurrentPosture}.\text{Seated} \sqcap \exists \text{hasCurrentActivity}.\text{Sipping} \sqcap \\ &\exists \text{hasCurrentLocation}.(\text{LivingRoom}) \sqcap \geq 2 \text{ contains}.\text{Teacup} \sqcap \\ &\forall \text{hasSoundSensor}.(\text{MeasuredSoundDb} \leq 35[\text{int}])) \end{aligned}$$

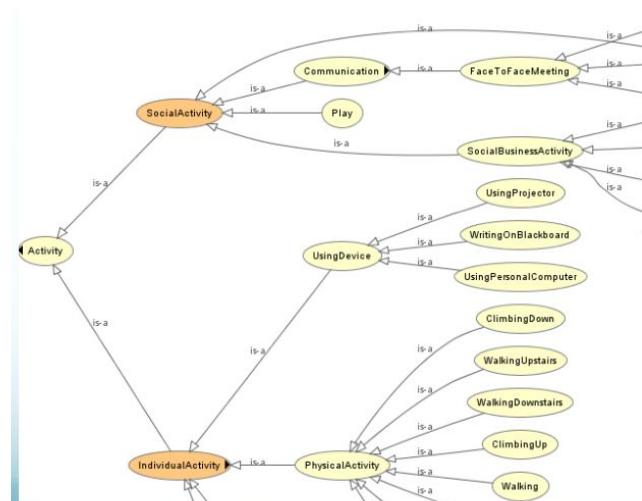
Cosa sono quindi i modelli ontologici?

Sono sistemi context-aware basati su ontologie, e quindi hanno un'ontologia del contesto (normalmente multilivello) e un'architettura SW per l'acquisizione, il management e il processing dei dati del contesto.

I modelli ontologici offrono inoltre supporto per il ragionamento automatico, con meccanismi di:

- Sussunzione (subsunction): capiscono se un concetto è un subconcetto di un altro;
- Realizzazione: capiscono se un oggetto cattura un set di osservazioni;
- Consistenza: individuano eventuali contraddizioni nelle definizioni dei concetti progettate dal programmatore (che è umano e si contraddice spesso);

Un esempio di ontologia è quella del COSAR:



Come possiamo vedere, modelliamo dei concetti all'interno di concetti più generali ($\text{Activity} \rightarrow \text{IndividualActivity} \rightarrow \text{PhysicalActivity}$...). Un editor usato per la progettazione delle ontologie è Protégé.

Come vediamo nella slide, usare modelli basati su ontologie porta dei vantaggi e svantaggi: permette di automatizzare il ragionamento, il sistema può rispondere ai quesiti di Sussunzione, Realizzazione e Consistenza. Fornisce inoltre modi per esprimere le relazioni e le dipendenze tra i dati. E' altamente usabile, si devono solamente definire le regole logiche.

Però a livello di incertezza siamo molto svantaggiati: essendo regole logiche non è facile rappresentare l'incertezza (anche se è possibile introdurla), l'efficienza è ridotta, in quanto i modelli sono grandi e devo memorizzare molte relazioni e vi è un brutto supporto per la history;

	Flat	DB-based	Ontological
Heterogeneity and extensibility	+/-	+/-	+
Expression of relations and dependencies	-	+	+
Support for historical data	-	-	-
Support for uncertainty	-	+/-	+/-
Reasoning	-	-	+
Usability	+	+	+
Efficiency	+	+	-

Modelli ibridi

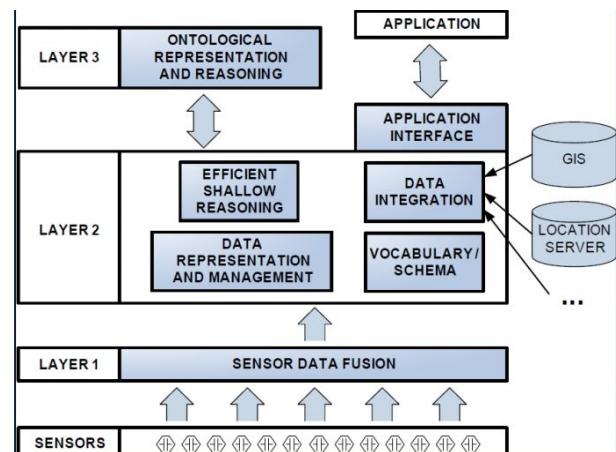
E' possibile combinare modelli di questi tre tipi per ottenere strutture di questo tipo, e avere "the best of both worlds": prendo dati da sensori e li fondo con flat, rappresento le loro relazioni con i modelli DB based e infine definisco le ontologie per ragionarci su e controllarne la consistenza.

Incertezza

Tutte le percezioni del nostro mondo, comprese le misurazioni che facciamo, sono soggette ad incertezza: sensori differenti possono dare valori differenti per lo stesso dato; Anche l'inferenza sui dati è prona ad incertezza, e perciò è importante gestire l'incertezza e i conflitti che si formano tra i dati;

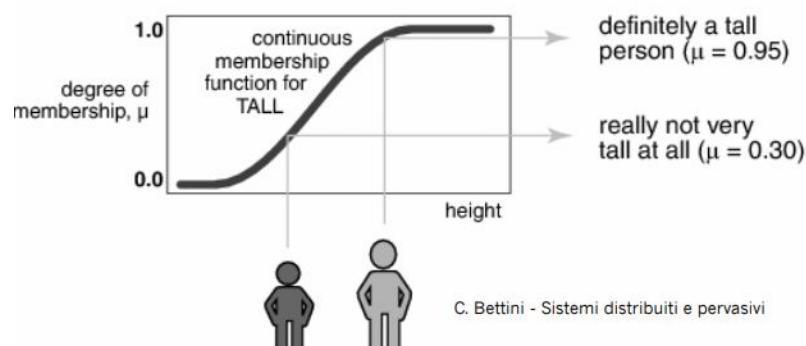
L'incertezza è caratterizzata da:

- Non completezza dei dati: Il dato che ci serve manca, ed è difficile da gestire: non usare i valori ricevuti ma usare i valori di default;



- Inaccuratezza dei dati: i dati sono imprecisi, perciò sfruttiamo l'aggregazione dei dati per aumentarne la precisione (se ho sorgenti multiple a disposizione) e uso modelli basati su logica probabilistica;
 - es. "if the probability that the user has fallen is more than 30%, call the medical center"
- Incertezza temporale: i vecchi dati potrebbero non essere più validi;
 - risolvo con TTL e data prediction, riducendo piano piano il grado di confidenza (un dato di 2 gg fa ha grado < di un dato di oggi);
- Conflitti tra dati: sorgenti diverse offrono valori contraddittori per lo stesso dato (es. l'utente è in due posti allo stesso tempo);
 - scelgo il valore più reliable (e/o considero la maggioranza);
- Inconsistenza dei dati: alcuni dati non sono compatibili con altri (es. l'utente è in macchina e cammina allo stesso tempo);
 - Uso tool per il check della consistenza automatica (es. ontologie);

Non viene usata solo la logica probabilistica, ma esistono modelli che usano la logica fuzzy: i dati non sono veri o falsi, ma stanno in un intervallo fra vero e falso. Definisco quindi una funzione di appartenenza che stabilisce questo intervallo.



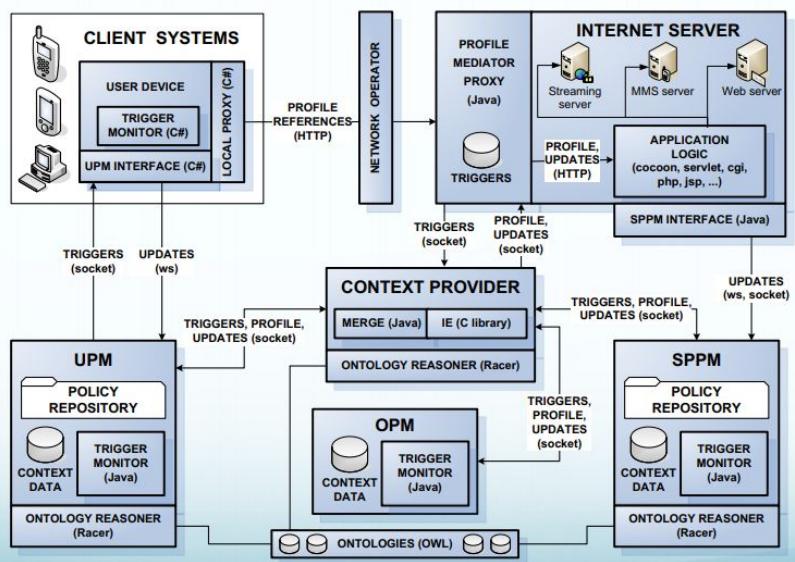
Supporto all'incertezza

- Uso coppie chiave-valore per incertezza di dati particolari (es. posizione);
- Uso CML per arricchire i dati con metadati (timestamp, accuratezza, probabilità...);
 - Alternativa: controllo la consistenza con la logica a tre vie (vero, possibilmente vero, falso);
- Uso modelli ontologici per fare il check automatico della consistenza;
 - Alcuni linguaggi supportano probabilità e fuzziness, ma sono (a volte) meno espressivi di OWL2;

Supporto al Middleware (Context Provisioning)

Il middleware è un layer di SW che sta sotto alle applicazioni che offrono servizi comuni per il contesto. Il middleware si occupa dunque di acquisire il contesto da fonti multiple, controllando e risolvendo i conflitti fra versioni discordanti e fondendo i contesti in uno solo (anche se si riferiscono allo stesso dato). Inoltre, il middleware deve effettuare il ragionamento sul contesto e preservare la privacy di quest'ultimo. Riprendendo l'esempio di CARE, notiamo come tutte queste richieste siano soddisfatte:

CARE: a context middleware developed at EWLab



Grazie a un sistema di ontologie create in OWL e inserite negli Ontology reasoner delle sorgenti UPM, OPM e SPPM, si controlla la consistenza fra i contesti calcolati in queste tre sorgenti. Il tutto finisce al context provider che si occuperà di fondere i contesti seguendo una certa proprietà (es. media pesata dei contesti), e di inviarle al profile mediator per comandare i tre server di Streaming, MMS e Web, e al UPM che aggiornerà l'interfaccia utente in base al contesto.

FINE!