Sistemi distribuiti e cloud computing

December 11, 2020

Contents

1 Capitolo 1 - Introduzione

Dal 1977, con la nascita e l'avvento dell'Internet, al 2017 il numero di nodi IPv4 a livello globale è incredibilmente aumentato, con il più del 50% del traffico criptato.
Sono nati nuovi trend, tra cui Iot, cryptocurrency ed advertisement mobile, ed una stima della CISCO prevede che nel 2023 ci saranno 6 miliardi di utenti in

Internet.
Il traffico predominante è dell'ordine dei miliardi di nodi e la mole di dati prodotta è imponente: per esempio un auto a guida autonoma produce quotidianamente 4TB di dati, che vanno raccolti ed analizzati, in parte nel cloud, ed il problema è spostare questi dati dai bordi della rete al centro. Per questo motivo nascono nuovi paradigmi di computazione come fog ed edge computing.

L'Internet ed il web sono due esempi di sistemi distribuiti, ma anche il cloud,HPC(High Performance Computing), sistemi p2p, rete di casa WNS(Wireless Sensor Network), IoT etc...
Gartner's: processo di nascita, crescita e affermazione di una nuova tecnologia viene messa su un grafico, si parte dalle prime startup, poi c'è la fase di hype, se ne parla etc..., comincia la generazione di prodotti, poi c'è il periodo di disillusione: i primi problemi e fallimenti, a seguito dei quali la tecnologia viene consolidata.
2° generazione di prodotti è raggiunta solo da meno del 5%.

2 I sistemi distribuiti

Per i sistemi distribuiti esistono varie definizioni:

1. "Un sistema distribuito è un insieme di elementi di computazione autonomi che appare all'utente come un singolo sistema coerente."
   Gli elementi di computazione sono nodi: possono essere processi software, virtual machines etc.., sono autonomi fra loro.Il sistema appare però all'esterno come un tutt'uno, ma i nodi hanno bisogno di collaborare fra loro e quindi c'è bisogno di un apposito layer software che permetta questo, ed è il middleware.

2. "Un sistema distribuito è un sistema i cui componenti, collocati su computer interconnessi fra loro che comunicano e coordinano le loro azioni scambiandosi messaggi."
   I messaggi possono essere sincroni e temporanei o asincroni e persistenti, a seconda del particolare middleware usato.

3. "Un sistema distribuito è un sistema in cui il fallimento di un nodo di cui nemmeno conoscevo l'esistenza può rendere il mio pc inutilizzabile"

In un sistema distribuito, ci sono alcuni parametri di cui bisogna tenere conto:

- Disponibilità: $\frac{\%\text{tempo disponibile}}{\text{tempo totale di running}}$
- Migliorare la sicurezza: avere più componenti vuol dire essere più tollerante ad attacchi, che devono andare a colpire molteplici nodi per avere successo.
- Assenza di clock globale: nei SD non c'è un clock globale, quindi se ad esempio due thread hanno bisogno di sincronizzare le azioni ci deve essere

un modo per farlo (il clock delle macchine potrebbero non essere sincronizzati e se si usano protocolli come NTP questi possono avere delle soglie di precisione, nel caso di NTP O(ms)).Inoltre, i componenti non sono sempre in esecuzione allo stesso tempo, quindi questo porta a dover risolvere un problema complesso.

- Concorrenza: in un sistema centralizzato è una scelta implementativa se avere o meno concorrenza, nei SD c'è per natura e va gestita.
- Fallimenti indipendenti o parziali: se ad esempio ho un fallimento nel componente dell'applicazione che mi permette di effettuare il log in⇒blocco tutta l'app, devo prestare attenzione anche a come gestisco i vari componenti per essere il più robusto possibile a guasti, in modo da nascondere fallimenti parziali ed effettuare fasi di recovery.

2.1 Eterogeneità

Nei SD ho tipicamente un numero elevato di componenti che possono differire per hardware, risorse di computazione come RAM, spazio di storage, per SO o per linguaggi di programmazione usati.
La soluzione è il middleware: un layer software aggiuntivo messo sopra il SO e che offre un'astrazione di programmazione e nasconde l'eterogeneità sottostante. Ho diversi componenti e funzionalità usate da diverse applicazioni, in questo modo non c'è bisogno di doverle replicare nelle applicazioni. ad esempio: un componente vuole conoscere gli altri nodi presenti nel SD: si rivolge al middleware, che mette a disposizione una API apposita per interrogare un repository condiviso in cui vengono registrati tutti i nodi del sistema.I tipi di middleware di comunicazione sono di 3 tipi:

1. Remote Procedure Call
2. Remote Method Invocation
3. Message Oriented Middleware

Nei primi due casi, il problema è che la comunicazione richiede che entrambe le entità siano presenti nel sistema all'atto della comunicazione, mentre nel caso di MOM ci sono componenti aggiuntive che memorizzano i messaggi.

2.2 Trasparenza

Volgio che il mio sistema distribuito appaia come un unico sistema coerente all'esterno, quindi la distribuzione delle risorse deve essere invisibile.
La trasparenza nei sistemi distribuiti può essere di vario tipo:

- Trasparenza all'accesso: nasconde le differenze che possono esistere nella rappresentazione dei dati e come le risorse vengono accedute.

- Trasparenza alla locazione: nasconde la locazione delle risorse (ad esempio l'URL nasconde l'indirizzo ip di una macchina), unita alla trasparenza di accesso costituisce la network trasaprency.
- Trasparenza alla migrazione: nasconde il fatto che le risorse possono essere spostate in un'altra locazione (anche a run-time), senza compromettere l'operatività del sistema. (esempio: codici di redirect).
- Trasparenza alla replicazione: nascone che ci sono molteplici repliche della stessa risorsa.
  Ogni replica dovrebbe avere lo stesso nome e richiedere anche trasparenza alla locazione.
- Trasparenza alla concorrenza: nasconde il fatto che le risorse sono accedute in concorrenza, ad esempio gli accessi ad una stessa tabella di un db, regolati con meccanismi di locking.
- Trasparenza ai fallimenti: cerco di nascondere i fallimenti.

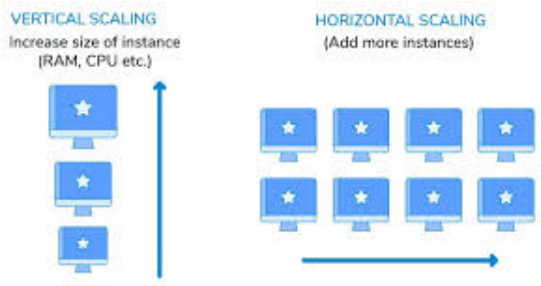
2.3 Apertura

Un sistema che è in grado di interagire con i servizi offerti da altri sistemi aperti.
Il sistema dovrebbe avere delle interfacce ben definite, la stessa idea viene applicata ai sistemi distribuiti: uso un linguaggio di IDL(Interface Definition Language).
I sistemi aperti dovrebbero poter inter-operare e garantire la portabilità.
Uso il meccanismo dei container, che mi permetto di tenervi all'interno il codice e le librerie dell'applicazione.
Principio di progettazione generale: è opportuno separare le politiche dai meccanismi.
Il sistema dovrebbe fornire i meccanismi e lasciare l'implementazione delle politiche agli utenti del sistema stesso.
Per esempio, i web browser: vengono offerti meccanismi, tra cui il caching web locale.
Il browser lascia la definizione delle politiche all'utente, che decide quali risorse memorizzare in cache, per quanto tempo salvarle etc...
Il problema di avere un sistema auto-configurabile è quello di trovare un buon compromesso $\Rightarrow$ più arte che scienza.

2.4 Scalabilità

Un'altra proprietà fondamentale per mantenere la performance adeguata nonostante l'incremento del numero di utenti, della massima distanza fra i nodi etc...
Posso averla in due dimensioni:

- Scalabilità verticale: uso una risorsa di storage/hw più potente
- Scalabilità orizzontale: aggiungo più risorse, tipicamente con la stessa capacità



esempio: Google file system
File system distribuito, che offre le stesse API di un file system classico.
I file sono replicati su più macchine e ciascuno di essi è diviso in pezzi, che possono essere replicati su più nodi.
la lettura va quindi effettuata leggendo i vari chunks dai server, andando ad effettuare più letture in parallelo ⇒ aumento il throughput.

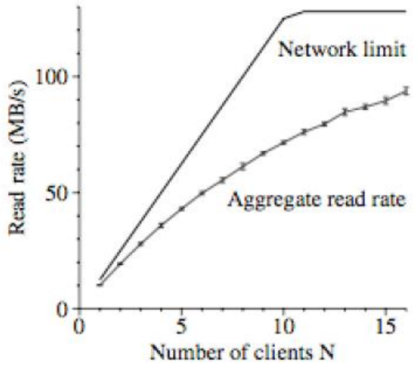
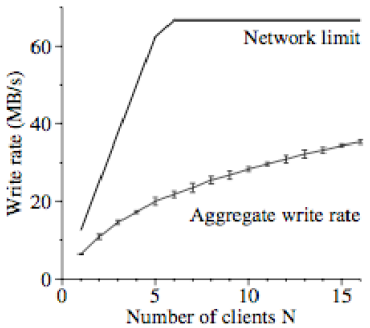


Fig 1 Read Performance

Più la curva si avvicina al limite teorico, tanto più il throughput aumenta, che è quello che mi aspetto.
La scrittura è più complessa da gestire, in quanto le modifiche vanno propagate su tutte le repliche, inoltre va gestito l'accesso condiviso.



Ci sono poi alcune tecniche per migliorare la scalabilità:

- Nascondere la latenza di comunicazione, mediante l'uso della comuni-

cazione asincrona. Per questo si possono usare degli handler per la gestione, ma non tutte le applicazioni si adattano a questo modello.

- Suddividere i dati e replicarli su più risorse, usando il principio del divide et impera.In questo modo, scambio in parallelo più chunks (solitamente dell'O(64MB)).
  Applicabile a server e db distribuiti, web server replicati, cache web etc...

esempio: One drive e Dropbox tengono una copia dei file in locale (sulla macchina utente), Dropbox in particolare usa un servizio cloud di Amazon, mentre i metadati sono gestiti direttamente da Dropbox.
Il problema è che bisogna gestire bene le repliche, mantenendo consistenza nei dati, tollerando un certo grado di inconsistenza.

2.5 Errori comuni nell'implementazione di sistemi distribuiti

Alcuni SD sono complicati per errori dovuti all'implementazione ed al design, che hanno portato a delle patch, che a loro volta hanno aggiunto complessità al sistema.
Si prendono decisioni fallaci o si fanno assunzioni errate:

1. La rete è affidabile: assunzione errata, bisogna tener conto dei fallimenti, vale anche per i componenti del sistema.
2. Assumere latenza 0: la latenza è più problematica del bandwidth, in quanto non può essere evitata, è legata a limiti fisici.
3. Bandwidth "infinito": assunzione meno forte della 2, ad oggi ci sono notevoli miglioramenti.
4. LA rete è sicura
5. La topologia della rete non cambia: avviene solo in piccoli ambienti controllati.
6. Avere un unico sys admin: non è vero per i SD a larga scala, alcuni bug sono dovuti infatti ad errate configurazioni del sistema.
7. Costo di trasporto pari a 0: passare dal livello 5 al 4 e viceversa non ha tempo nullo, anche una chiamata a procedura remota che non fa nulla ha un suo overhead non indifferente.
8. L'ambiente di rete è omogeneo.

2.6 Cluster computing

Gruppi di server che comunicano connessi con LAN ad alte prestazioni, oltre al cluster di server posso avere gruppi di storage.
Obiettivi del cluster computing sono ad esempio High Performance Computing,

High Aviability etc..
I nodi nel cluster hanno un'architettura di tipo master/worker, con il master replicato per motivi di scalabilità e robustezza.
Si usano software specifici (come MPI) per il passaggio di messaggi dai worker ai master, ed i cluster sono gestiti con specifici tool sw che considerano tutto come un singolo sistema.
esempio: Mosix, sistema che automatizza l'aggiunta/rimozione di nodi e bilancia il workload usando la migrazione per renderlo eguale su tutti i nodi; ovviamente, si vuole che questo sia trasparente all'utente.

2.7 Cloud computing

Differisce dal cluster computing in quanto quest'ultimo è un componente fondamentale del cloud.
Il cloud computing è però disponibile per chiunque (a pagamento o gratuitamente) ed inoltre cambia la scala: nel cluster computing i nodi sono inuna LAN, mentre il cloud è connesso ad Internet ed usa molteplici livelli di virtualizzazione.
Prevede la dislocazione delle risorse di computazione, network, storage etc... verso i bordi della rete, avvicinandoli agli utenti.

2.7.1 Distributed Information Systems

Sistemi per il processamento di transazioni, ovvero operazioni atomiche di tipo "all-or-nothing".
Una transazione è un'unità di lavoro trattata in modo coerente e indipendente dalle altre, per cui valgono le così dette proprietà ACID:

- Atomic : la transazione deve essere atomica, tutto o nulla
- Consistent: la transazione non deve violare lo stato del sistema, che deve rimanere consistente.
- Isolated: tutte le transazioni eseguono come se fossero isolate fra di loro
- Durable: quando avviene il commit i cambiamenti devono essere mantenuti permanentemente.

Transazioni distribuite: transazioni costituite da più sotto-transazioni.ì, eseguibili in modo distribuito su molteplici server.
TP monitor: componente responsabile di coordinare l'esecuzione della transazione.

2.8 Sistemi distribuiti pervasivi

Sistemi i cui nodi sono di piccola dimensione, mobili, spesso a batteria e che spesso fanno parte di un sistema più grande.
Un sistema di mobile computing, in cui i nodi che lo compongono sono mobili, come ad esempio le sensor network: vi sono una serie di dispositivi che ascoltano

l'ambiente circostante e si occupano dell'attuazione di comandi sull'ambiente.
I fallimenti nelle sensor network sono molto frequenti, quindi vanno gestite.
Ci sono due possibili situazioni estreme:

- Unico nodo centralizzato verso cui vengono inviate le informazioni, che le processa.
- Architettura completamente decentralizzate: ogni sensore può comunicare con sotto un insieme di sensori.
  Ogni sensore memorizza e processa le informazioni, ci sono protocolli di gossiping o epidemici, per far si che l'informazione si diffonda nella rete distribuita.

3 Cloud computing

Ormai quasi tutte le applicazioni di rete vanno pensate per il cloud, prendo ad esempio un playback video: voglio che scali rispetto al n° di utenti, la soluzione classica è progettare un'applicazione multithreaded andando a sfruttare il multicore.
Ho della scalabilità, ma è comunque limitata dal n° dei thread che è allocabile nello stesso momento ed inoltre ho un single point of failure; inoltre più thread condividono lo stesso stato, quindi vanno sincronizzati fra loro.
Penso ad una soluzione nativa per il cloud: un applicazione single-threaded che allochi un'istanza di VM ogni volta che l'utente richiede di utilizzare l'applicazione.
In questo modo isolo le diverse applicazioni concorrenti, sono più tollerabile ai guasti. L'applicazione così risultante scala (basta aggiungere un container), è robusta e non devo gestire la sincronizzazione fra thread, inoltre il design dell'applicazione è semplificata.
Problema reale: nella realizzazione di sistemi e servizi che sono in grado di servire milioni di richieste al giorno, devo far fronte ad un n° di richieste variabile e che deve memorizzare Exabyte di dati.
Il cloud computing non nasce dal nulla, ci sono soluzioni anche parziali studiati e realizzati:

- Utility computing: computazione come utilità
- Grid computing: evoluzione del cluster computing
- Autonomic computing: sistemi adattativi
- Software as a Service: uno dei layer del cloud.

SI comincia a parlare del cloud nel 2006,una delle idee fondamentali nasce dall'utility computing: la computazione è utile al pari dell'acqua, del gas etc...;se ne parlava già nel 1961.
Nel 2006: Jeff Bezos, Amazon. La constatazione fu che Amazon aveva un'infrastruttura molto potente per garantire un'elevata QoS,l'infrastruttura era però sotto-utilizzata e sovra-dimensionata per i picchi di carico.

L'idea fu quella di fare profitto sulle risorse non utilizzate, offrendole come servizio al pubblico.
Vi fu il lancio della beta dei primi due servizi, ovvero EC2 ed S3, che corrispose alla nascita del cloud computing. Cluster computing:

- Eterogeneità
- Unica immagine
- Fortemente accoppiata

Distributed computing:

- Meno accoppiamento
- Eterogeneità
- Singola amministrazione

Grid computing:

- Scala più ampia
- Diverse organizzazioni
- Gestione distribuita

Cloud computing: risorse "infinite" dietro richiesta, prevede di considerare requisiti di QoS, che devono essere rispettati dal cloud provider oppure c'è una penalità.
Ha alla base la virtualizzazione e si basa sull'evoluzione del web e di http.

3.1 Definizioni di cloud computing

1. Il cloud computing si riferisce sia alle applicazioni come servizi, ma anche all'hardware ed al software nei data center. Il cloud computing da l'illusione di avere a disposizione risorse infinite, ottenute con un uso efficiente della virtualizzazione. Viene eliminato l'impegno dell'utente: non ho costi di installazione, è tutto a carico del fornitore dei servizi cloud. È un sistema pay-per-use: la granularità è fine, ma non finissima (ad esempio: con EC2 pago il tempo per cui istanzio la VM, non l'effettivo tempo di utilizzo). Con il serverless computing avrò una granularità ancora più fine.

2. Il cloud computing è un modello per abilitare un accesso di rete conveniente, ubiquo, on-demand (non necessita di pre-allocazione). Permette l'accesso ad un insieme condiviso di risorse configurabili (network, storage, applicazioni e servizi) ed elastiche $\Rightarrow$ allocazione e de-allocazione rapida con il minimo sforzo d'interazione con il fornitore.

3. Clouds sono grandi pool di risorse virtuali, facili da usare ed accessibili. Queste risorse possono essere riconfigurate dinamicamente per adattarsi ad un carico di lavoro che cambi.

3.2 Caratteristiche essenziali del cloud computing

- On-demand self-service: le risorse e i servizi sono ottenibili quando necessarie.
- Network access: le risorse cloud sono accedute via Internet, usando meccanismi standard di accesso, ho dei servizi di interfaccia/API. L'accesso è inoltre platform indipendent.
- Rapida elasticità: abilità per un utente cloud di richiedere risorse rapidamente, ricevendone o rilasciandone tante quante sono necessarie. Le risorse sono ottenibili rapidamente ed è facile fare scale in/out in base alla domanda.
- Resource pooling: le risorse vengono accedute da molteplici utenti che tipicamente non si accorgono che le stanno condividendo, multi-tenancy.
- Virtualization: si virtualizza l'ambiente di esecuzione, ma anche eventualmente i data center.
- Pay-per-use: pricing model comodo.
- Measured service: l'utilizzo delle risorse è misurato, l'utente paga in base a delle metriche.

3.3 Modelli di cloud

3.3.1 Cloud pubblica

L'infrastruttura cloud è fornita dal provider dei servizi cloud affinché possa essere utilizzata dal pubblico, tutti gli utenti condividono le risorse hardware. È gestita da organizzazioni di business, le risorse sono gestite dal provider del servizio. Possono essere gratuiti o a pagamento.

3.3.2 Cloud privata

L'infrastruttura è pensata per un uso esclusivo di una singola organizzazione. Viene gestita dall'organizzazione, da terze parti etc...
Il modello esiste con o senza permesso, non essendoci condivisione c'è una maggiore sicurezza sui dati e sulle applicazioni, inoltre i servizi sono più personalizzabili. Gli svantaggi rispetto ad un modello di cloud pubblica è che il costo è maggiore ed inoltre la scalabilità non è "infinita".

3.3.3 Cloud ibrida

Infrastruttura che deriva dalla composizione di due o più infrastrutture che possono essere pubbliche o private, usabili individualmente, ma legate fra loro da tecnologie standardizzate o proprietarie.
Sfrutta i lati positivi di entrambe le infrastrutture:

- Possibilità di bilanciare il costo
- Maggiore sicurezza e privacy per i dati, memorizzati nell'infrastruttura privata.
- Migliore disponibilità, in quanto posso usare il clou pubblico per migrarvi dati e applicazioni in caso di guasto inatteso del cloud privato.
- migliori prestazioni con il cloud bursting: l'idea è quella di mixare fra loro le due infrastrutture per gestire il workload, partendo dalla cloud privata scalo integrando la colud pubblica dinamicamente in mod da gestire l'eccesso di carico.

3.3.4 Cloud ibrida

Evoluzione recente, in cui il fornitore dell'applicazione usa in modo concorrente più ambienti cloud (ad esempio usato da Netflix).
Sta divenendo sempre più popolare, per le seguenti motivazioni:

- sfrutta la distribuzione in senso geografico dei diversi cloud provider, per avere una migliore copertura del servizio.
- l'utente dell'applicazione cloud deve soddisfare i requisiti di privacy, il fornitore dell'applicazione può usare uno dei big provider per la maggior parte dei servizi ed altri per memorizzare i dati sensibili.
- vendor lock-in: i fornitori mettono a disposizione un'interfaccia non standardizzata, quindi gli utilizzatori sono costretti ad usare l'API fornita e questo peggiora la portabilità.

3.4 Modelli di servizio

3.4.1 IaaS-Infrastructure as a Service

IL servizio è esposto in termini di risorse di calcolo, di storage, di rete etc... L'utente può istanziare VM, su cui può fare deployment ed esecuzione di software. In questo modo l'utente gestisce e controlla meglio la sua applicazione, il controllo si riduce via via salendo di livello.
In questo layer, il controllo sulle risorse non è totale, però c'è la possibilità do scegliere il SO da istanziare, la regione in cui istanziare la VM, il software di base, è anche possibile installare librerie, linguaggi di programmazione etc... L'utente può inoltre controllare alcune componenti di rete, ad esempio le porte accessibili dall'esterno.
Caratteristiche principali:

- Risorse virtualizzate pure: CPU, memoria etc...
- OS incluso
- Taglia su misura

- Effort di configurazione dei servizi,in quanto le VM da istanziare per lo scaling vanno configurate

3.4.2 IaaS Amazon: EC2

Il servizio EC2 di AWS è un servizio di tipo IaaS che permette di istanziare VM, all'avvio dell'istanza è possibile scegliere l'immagine di VM da installare e configurare il SO. È inoltre possibile scegliere la taglia: n° CPU, quantità di memoria (in Gbit), prestazioni di rete etc...
A seguito delle scelte si lancia la VM

3.4.3 PaaS-Platform as a Service

Viene offerta una piattaforma su cui poter sviluppare e gestire applicazioni cloud scalabili, senza dover badare all'infrastruttura sottostante.
L'utente non deve occuparsi di gestire e configurare gli altri servizi, deve solo sviluppare la sua applicazione web.
Caratteristiche principali:

- Risorse virtualizzate + framework per applicazioni
- Servizi addizionali come scaling etc...
- Maggior rischio in caso di vendor lock-in
- Costrizioni sulla struttura dell'applicazione e l'architettura dati

3.4.4 PaaS Amazon: Elastic Beanstalk

Servizio per lo sviluppo ed il deploy di applicazioni senza dover gestire l'architettura. Una volta caricati i sorgenti (in formato .war) e dati tutti i dettagli, ci pensa Beanstalk: istanzia le risorse, permette la gestione a runtime, inoltre effettua il balancing e gestisce lo scaling, monitorando l'applicazione.
Usando il load-balancing di Beanstalk, le risorse EC2 mal funzionanti non saranno usate per bilanciare il workload

3.4.5 SaaS-Software as a Service

Le applicazioni vengono rese disponibili come servizi cloud, l'utente le usa eseguendole nel cloud e può includerle nelle proprie applicazioni. Non c'è controllo sull'infrastruttura cloud o sui parametri di deployment sulla piattaforma di sviluppo, è solo possibile configurare i parametri dell'applicazione SaaS che viene utilizzata.
Servizi come Gmail, Zoom etc..., prevedono vari modelli di tariffazione:

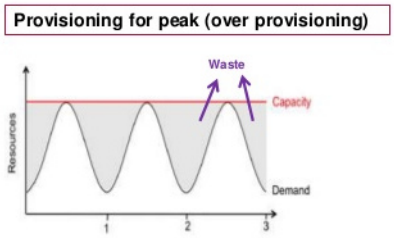
- pay-per-use: prezzo in base al tempo di utilizzo
- fixed: prezzo mensile fisso
- spot: prezzo variabile per le risorse cloud, guidato dalla domanda di mercato. Il provider può togliere la risorsa improvvisamente.

3.5 Uso del cloud

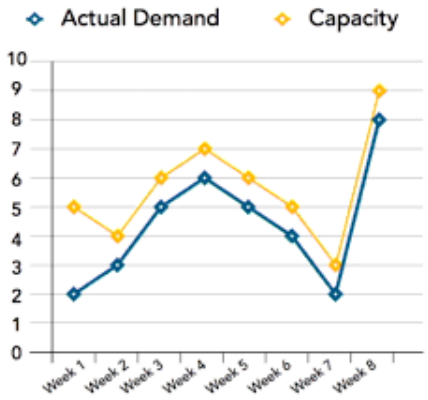
Un report annuale mostra come per le organizzazioni risulta difficile gestire i costiper le risorse cloud. Capure bene come usarlo è cruciale, la maggiora parte delle organizzazioni opta per una soluzione multi-cloud: i 3 principali provider sono Amazon, Google e Azure, i PaaS più usati sono DBaaS e IoT.
I container sono ormai largamente diffusi, più del 65% delle aziende usa Docker, ed il 58% usa come orchestratore Kubernetes.
La sfida affrontata dalle aziende è quella di capire come gestire le dipendenze all'interno dei diversi componenti dell'applicazione, in quanto le riconfigurazioni sono dispendiose, per questo motivo si fa uso di strumenti per la configurazione dell'applicazione nel cloud e per il deployment automatizzato.
Il cloud è inoltre utilizzato come infrastruttura di back-end da molte aziende, i servizi risultato più sicuri ed elastici, nonché convenienti.

3.6 Elasticità

Possibilità di incrementare o diminuire il numero di risorse velocemente, è molto importante perché le applicazioni sono soggette a rapidi cambiamenti nel traffico.
Applicazione multi-tier: bisogna capire il dimensionamento di ciascun livello in termini di risorse e di capacità delle risorse.
Per questo, si possono usare strumenti di analisi, strumenti per la previsione di carico etc..., in modo da gestire in anticipo le variazioni di carico. L'approccio tipico è fare over-provisioning:

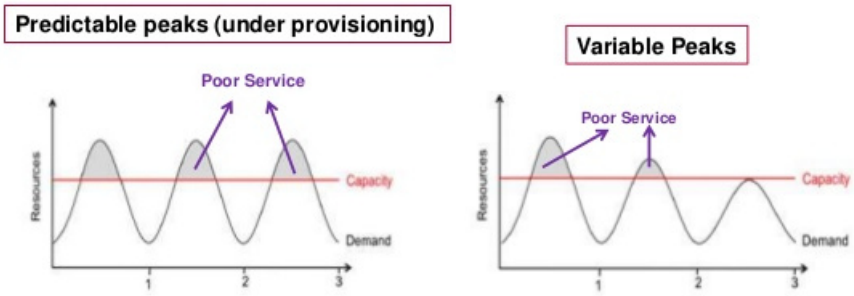


In questo caso, ho una parte delle risorse sprecate



L'obiettivo che voglio ottenere con l'elasticità nel cloud computing: posso allocare/de-allocare risorse velocemente e quindi "inseguire" la domanda

Un altro approccio classico consiste nell'under provisioning:



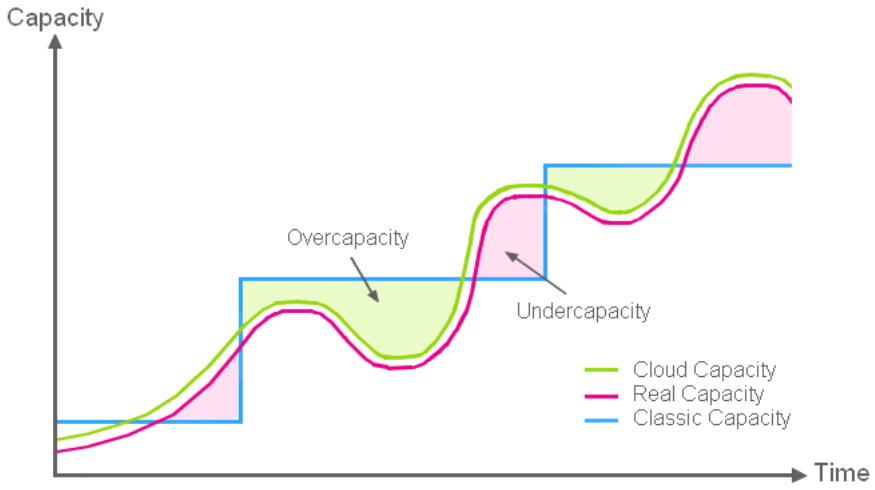
Qui, in caso di sovraccarico, perdo rispettivamente revenue e clienti.
Con il cloud voglio sfruttare al meglio l'elasticità, ovvero il grado con cui il sistema sa adattarsi ai cambiamenti di carico, andando a de/allocare risorse in maniera automatica in modo che le risorse disponibili siano quanto più vicine alla domanda dell'utente.
esempio: Animoto su Facebook.
Animoto è un'applicazione per creare video, lanciata ad Aprile del 2008. L'applicazione ebbe uno scale-up del numero di utenti da 25k a 750k solo nei primi 3 giorni.
Per la gestione dell'infrastruttura, i developer usarono il servizio EC2 di Amazon, in modo da fronteggiare l'incremento del traffico, che ebbe un picco di 20k nuovi utenti in solo un'ora.
Tra i componenti fondamentali per gestire l'elasticità c'è sicuramente il load-balancer, che si occupa di distribuire il traffico in ingresso sulle repliche.
Il load-balancer può essere centralizzato o distribuito, con i soliti vantaggi svantaggi a seconda della scelta.
I principali obiettivi sono:

- massimizzare l'utilizzo delle risorse
- massimizzare il throughput

- minimizzare i tempi di risposta

3.6.1 Misura dell'elasticità

Si possono adottare molteplici metriche, presenti in letteratura. L'approccio più comune consiste nel misurare l'under/over provisioning:



- Accuratezza: somma delle aree di under/over-provisioning nel periodo T.
- Timing: ammontare totale di tempo speso nelle aree di under/over-provisioning

3.7 SLA-Service Level Agreement

Aspetto legato alla fornitura di servizi con un certo QoS, è un accordo formale tra fornitore ed utente. Uno o più obiettivi dello SLA sono detti SLO (Service Level Objective): il SLO è una condizione di misura di una specifica QoS, ad esempio il massimo tempo di risposta.
Possono essere previste delle penalties nel caso di violazione del SLA. Ciclo di vita di un SLA:

1. Scelta del provider specifico
2. Definizione dello SLA
3. Decisione dell'accordo
4. Monitoring di eventuali violazioni
5. Terminazione SLA
6. Penalty per la violazione.

7. Si riparte dal punto 1

esempio: SLA di EC2
Vi sono due sezioni fondamentali nel documento: il commitment di Amazon nell'offerta del servizio, SLO basato sulla disponibilità, ovvero la % di tempo per cui un'istanza di VM è up&running, che nel caso del SLO di Amazon è garantito al 99.99% durante un periodo di ciclo di monitoring di 1 mese.
La penality di AWS in caso di mancato rispetto del SLA, che scatta se la 99% <t. disponibilità < 99.99%, prevede il rimborso di un credito, spendibile nei futuri pagamenti.

3.8 Applicazioni cloud

Ormai qualunque applicazione può essere realizzata nel cloud, esempio: Shazam, in cui il sample audio viene confrontato con un DB di pattern audio, quindi si risolve un problema di pattern matching.
Lato client, il pattern viene catturato dall'applicazione e trasformato in una stringa alfanumerica per poi inviarla al server, che cerca il matching migliore.L'aspetto fondamentale: conviene pre-calcolare gli indici di risposta dei finger print per fornire risposte veloci.
Ormai qualunque tipo di applicazione può essere migrata nel cloud, o essere pensata nativamente per il cloud: sono applicazioni non migrate nel cloud, ma progettate dall'inizio per il cloud. Posso inoltre prevedere:

- servizi stateless: senza stato, quindi semplici da replicare
- servizi statefull: le richieste devono fornire e mantenere uno stato,che può essere memorizzato in un database
- layer applicativo di dati: redis (caching, framework per DB in RAM), DB relazionali, AWS S3.
- framework e strumenti per il monitoring e la diagnosi dello stato delle applicazioni: attività di logging, dove i file di log sono analizzabili con appositi tool di analisi temporale per predire l'andamento futuro di traffico e richieste.

La fase di deployment richiede diverse fasi dell'ingegneria del software, il processo diviene esso stesso interattivo:

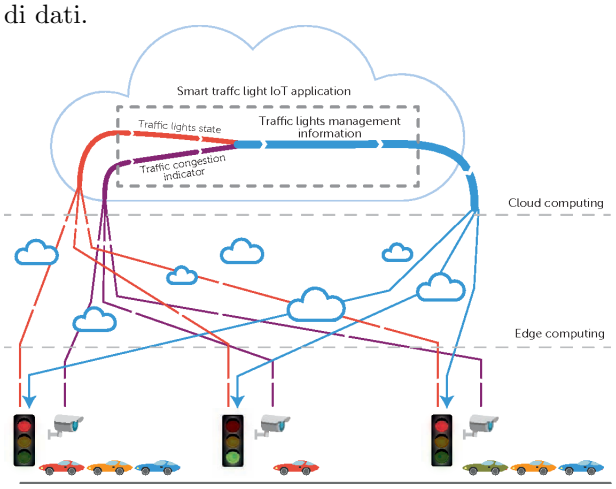
1. design: tipo e capacità delle risorse cloud
2. valutazione performance: verifica se l'applicativo è in accordo con i requirement di performance, mediante applicazioni di monitoring del workload e dei parametri di performance
3. refinement: considerazione alternative per scaling, interconnessione dei componenti,load balancing e strategie di replicazione.

Le applicazioni future prevederanno la convergenza fra più tecnologie: IoT, Big Data, AI etc..., i dati vengono processati, analizzati e memorizzati nel cloud.
Ma spesso il cloud è lontano (in senso di distanza geografica), quindi nascono nuovi paradigmi di computazione come fog ed edge computing, che prevedono di spostare le risorse di computazione e di storage più vicino agli utenti che generano i dati.
Cloudlet: localizzato in prossimità dell'access point dei dispositivi mobili, in cui viene eseguita parte della computazione.
Le cloudlet possono anche essere mobili, ad esempio i nodi di computazione delle auto a guida autonoma.

3.8.1 Fog ed edge computing

- Fog computing: introdotto nel 2012, da Bonomi. Piattaforma altamente virtualizzata che fornisce computazione, storage e servizi di rete, posta tra i device ed i tipici data center del cloud.
  2017: definita come un'architettura orizzontale che distribuisce funzioni di computing, storage, controllo e networking più vicine agli utenti per avere un continuo di computazione.

- Edge computing (ambito FoTel/5G): sinonimo di fog computing, entrambe hanno radici dal content delivery network. Derivano da un'architettura P2P decentralizzata, il fog computing è più strettamente integrato col cloud. esempio: regolazione del traffico urbano con controllo dei semafori nel cloud, prevede un'organizzazione gerarchica con i nodi che sono divisi in layer.
  Posso avere un livello più basso fatto da sistemi IoT che fanno pre-processamento di dati.



3.9 Sommario del cloud

I benefit del cloud riguardano molteplici utenti:

- IT:
  - "infinita" quantità di risorse, quindi scalabilità ed elasticità
  - risorse sempre accessibili, da qualunque servizio
  - sw per il management
- Business:
  - costi capitali ⇒ costi operativi
  - costi scalabili con l'utilizzo
- Impatto ambientale minore, quindi più sostenibile

Va però gestita la privacy e la sicurezza dei dati, ci sono aspetti legali e politici riguardo dove vengono storati i dati.
Inoltre, ci sono vari aspetti che vanno considerati:

- La latenza di comunicazione, per cui le applicazioni in tempo reale soffrono di più, per contrastarla sta divenendo predominante fog/edge computing.
- Portabilità dell'applicazione: rischio il vendor lock-in, quindi la costrizione a migrare verso un altro provider. Metodo per migliorare la portabilità: uso dei container e di tool per l'automazione.
- Interoperabilità: modello multicloud, uso diversi modelli di cloud ma anche diversi cloud provider e devo far si che possano inter-operare fra loro, OVF (Open Virtualization Format).
- SLA negotiation e management: l'utente non può in, molti casi, negoziare il SLA. Gli aspetti di gestione e monitoring del SLAche sono a carico dell'utente, così come il claim in caso di violazione.
- Guasti: un guasto può innescare altri guasti nelle applicazioni che usano il SaaS. Va inoltre considerata la variabilità e l'incertezza nella domanda di servizio, ad esempio Amazon vende istanze EC2 ma può toglierle all'improvviso, adatto ad applicazioni tolleranti ai guasti.
  Spesso fallimenti IaaS ⇒ fallimenti SaaS.

4 Architetture per sistemi distribuiti

L'architettura software definisce l'organizzazione logica e l'interazione dei vari componenti software che costituiscono il sistema distribuito.
L'architettura di sistema prevede il deployment del software, quindi si considera dove verranno istanziati i componenti sw del sistema distribuito.
Terminologia:

- Pattern: soluzione comunemente applicata ad una classe di problemi. Lo stile o pattern architetturale è un insieme di decisioni coerenti riguardanti l'architettura, in termini di componenti e connettori.

- Componente: unità modulare con dei requisiti di interfaccia ben definiti, che sono sostituibili completamente.
- Connettore: meccanismo con cui collego due o più componenti tra loro, che si occupa di mediare la comunicazione e di coordinare i componenti.esempio: coda di messaggi, chiamata a procedura remota.

4.1 Stile a livelli

I componenti del SD sono organizzati a livelli, ogni componente invoca il servizio del componente del livello sottostante. La comunicazione è basata su scambio di messaggi, le richieste scendono e le risposte salgono. Un'applicazione web è simile, se il deployment è distribuito di parla di 3-tier.
Possono esserci delle varianti, ad esempio il layer di livello N-1 evoca il livello N-3, o N-2 da cui evoca N-3.La tradizionale applicazione a 3 livelli prevede :

- application-interface layer
- processing layer
- data layer.

4.2 Stile object-based

Vi è un mapping fra componente ed oggetto. L'oggetto incapsula una struttura dati o un API per modificare i dati, in modo da garantire incapsulamento ed information hiding, come amche wrapping di componenti legacy.
La comunicazione avviene con chiamata a procedura remota o chiamata a metodo remoto

4.3 Stile RESTful

Il SD è un insieme di risorse gestite da diversi componenti e la comunicazione è tra i componenti. Le risorse possono essere aggiunte/rimosse/modificate con i classici metodi HTTP: ogni risorsa è identificata da un URI, di cui l'URL è un'istanza, l'interazione è tipicamente stateless, in modo da rendere il server più leggero.
I servizi cloud forniscono una API anche tramite REST API, come ad esempio in S3: gli oggetti sono messi in dei bucket (che rispecchiano la struttura di un filesystem, ma con delle limitazioni), ed è possibile eseguire le operazioni sui bucket mediante metodi HTTP. Nell'header della richiesta viene inserita una stringa di autorizzazione, contenente le informazioni necessarie per ottenere l'autenticazione da S3.

4.4 Disaccoppiamento

Gli stili architetturali richiedono che ci sia una comunicazione diretta fra i componenti, ma vorrei un maggior grado di flessibilità.Disaccoppiamento: ad esempio, far comunicare fra loro due layer adiacenti tramite un intermedio, introduco un livello di indirezione (strategia comune). Con il disaccoppiamento ho una maggiore flessibilità e sfrutto al meglio la distribuzione.
Tipi di disaccoppiamento:

- spaziale: diversi componenti non devono necessariamente conoscersi fra loro per poter comunicare.
- temporale: gestisco bene le applicazioni con un alto tasso di volatilità, ovvero con componenti che vanno e vengono. Posso gestire l'aspetto temporale in maniera flessibile.
- di sincronia: Non blocco un componente in attesa della risposta da parte degli altri. Lo svantaggio è un overhead prestazionale, ma anche in termini di gestione.

Posso quindi definire nuovi tipi di architetture, in cui i componenti comunicano in modo indiretto.

4.5 Stile event driven

Basato su architettura di tipo event-drvien: ho alcun componenti che si registrano come subscriber per ricevere notifiche per un evento, altri nodi fanno da publisher per l'evento; tutto è orchestrato da un event bus.
La comunicazione tra i componenti avviene mediante propagazione di eventi.
Evento: un cambiamento significativo nello satto del sistema, per cui voglio che il mio sistema compia una determinata azione.
Comunicazione basata su scambi di messaggi, asincrona, multicast ed anonima.
Permette di avere disaccoppiamento di sincronia e spaziale.

4.6 Stile data driven

Architettura di tipo data-oriented,anche in questo caso ho componenti publisher e subscriber, la comunicazione avviene tramite uno spazio di memoria condiviso e persistente. È tipicamente uno spazio passivo (non invia le notifiche ai componenti, ma devono essere loro a fare polling), ma in alcuni casi può essere attivo.
Lo spazio di dati può essere pensato come una lavagna, difatti è spesso chiamato blackboard model. La tipica API di interazione:

- read (o readIfExists): legge un dato senza cancellarlo
- take (o takeIfExists): legge e rimuove il dato
- write(writeIfExists): scrive un dato

È di tipo push se lo spazio di memoria è arrivo, pull se è passivo, tipicamente è anche prevista la mutua esclusione per interagire con l'area di memoria.
Ho disaccoppiamento temporale, di sincronia se lo spazio di memoria è attivo.
Per quello che riguarda lo spazio di memoria, l'implementazione può essere distribuita o in memoria centrale o RAM; nel secondo caso ho delle memorie associative, a cui non accedo tramite indirizzo ma in base al contenuto.esempio: Linda tuple space
I dati sono contenuti in delle tuple ordinate. Le tuple sono salvate in una memoria globale shared, con le tipiche operazioni:

- in(t): legge e rimuove una tupla che matcha il template t, la lettura è atomica
- rd(t): ritorna una copia di t
- out(t): scrive t nella shared memory.

Chiamare una out(t) due volte, stora due tuple t, lo spazio delle tuple è modellato come multiset. Sia in(t) che rd(t) sono bloccanti, ovvero il chiamante è bloccato finché non trova una tupla che matcha to che diviene disponibile.
Lo spazio delle tuple è implementato in maniera passiva.

4.7 Stile publish-subscribe

I produttori generano eventi (publish) e si disinteressano della consegna. I subscriber si registrano ad un evento/dato e vengono notificati dell'occorrenza.
Permette disaccoppiamento completo: ho un middleware che fornisce la memorizzazione del dato, presso cui vengono pubblicati/ricevuti messaggi.

4.7.1 Schema basato su topic

Variante più semplice e diffusa, i pusblisher pubblicano ed i consumer o subscriber si sottoscrivono agli specifici topic, identificati da keywords. Lo svantaggio: espressività limitata, posso solo identificare che caratterizzano determinato topic.

4.7.2 Schema basato su contenuti

Eventi classificati in base all'effettivo contenuto (ad esempio i meta-dati associati agli eventi). I subscriber precisano dei filtri per indicare eventi/dati di interesse nel sistema.Lo svantaggio è che l'implementazione più complessa.

4.8 Problemi di implementazione

Ci sono vari problemi legati all'implementazione:

- Distribuzione di eventi e dati deve essere efficace, sicuro, scalabile, affidabile e permette concorrenza.

- centralizzato vs decentralizzato: posso avere un singolo componente fa da broker di eventi in un singolo nodo che mantiene un repository delle sottoscrizioni e deve fare il matching per inviare le notifiche. È una soluzione semplice, ma è un single point of failure e non è scalabile (o meglio, solo verticalmente).
  Una soluzione distribuita prevede una rete di broker che cooperano, può essere completamente decentralizzata con implementazione p2p.

Non esiste una soluzione migliore dell'altra, posso scegliere fra i diversi stili architetturali, la decisione può dipendere da:

- Costi
- Scalabilità ed elasticità
- Performance
- Sicurezza
- Tolleranza a guasti

4.9 Architettura di sistema

L'architettura di sistema prevede l'istanziazione dell'architettura software a run-time.
In un'architettura centralizzata ho un modello client-server: client e server possono essere su macchine differenti,avere un modello su richiesta/risposta.
La richiesta può essere con o senza stato, c'è la proprietà di idempotenza: posso ripetere la stessa operazione più volte (valido se stateless).
Se ho un'applicazione con stile architetturale a livelli, come mappo i tier fisici: two-tier level o three-tier level.
Ho differenti organizzazioni dei livelli logici in una web app:

- presentation layer
- business layer
- data layer

Posso avere una versione thin client o fat client.
Architettura multi-tier: migliora in termini di distribuzione, funzionalità ma peggiorano le prestazioni: più complesse le prestazioni e più costoso.esempio: applicazione 3-tier con deploy su AWS:
1° tier in cui c'è una distribuzione delle richieste, gestita da un tier2 in cui ci sono servizi di front-ende il tier 3 è specifico per il back-end.
Il tier1 effettua la distribuzione del carico sulle diverse repliche che offrono servizi di front-end, anche il tier2 distribuisce il carico verso il tier di back-end che è replicato.
Infine, il layer dati è realizzato con un DB replicato di tipo master-worker, dove

il worker viene usato solo come backup per sola lettura.
Uso il servizio EC2, con Virtual Private Network,per fare provisioning di una sezione logicamente isolata del cloud Amazon; la rete virtuale è suddivisa in sottoreti, 2 pubbliche e 2 privata.

4.10 Architetture decentralizzate

Sistemi p2p, una classe di sistemi ed applicazioni che usano risorse distribuite per eseguire funzionalità in modo decentralizzato. I sistemi p2p condividono le risorse: cicli di CPU, spazio di storage, dati etc...
Sono noti per applicazioni di file sharing, ma le stesse considerazioni valgono anche per altre risorse.
Tutti i nodi hanno le stesse capacità e responsabilità, ogni peer è sia client che server e solitamente si trovano ai bordi della rete;in alcuni sistemi ci sono dei nodi che vengono elevati di grado di super-peer.
C'è un elevata dinamicità, in quanto i nodi entrano ed escono a piacere, quindi bisogna gestire bene queste operazioni di join/leave, le risorse vengono ridondate così che se un nodo che ha una risorsa non è presente nella rete io possa comunque reperirla. Esempi di applicazioni p2p:

- File distribution
- p2p TV
- File storage
- Condisivione di risorse di calcolo
- Telefonia (Skype....)
- Content delivery network
- Piattaforme di sviluppo per applicazioni p2p

Nell'architettura p2p ci sono inoltre molti problemi:

- Eterogeneità, sai software che di rete etc...
- Scalabilità
- Località, non solo nei dati ma anche di distanza di rete
- Tolleranza ai guasti
- Performance: voglio che l'identificazione della risorsa sia efficiente
- Free-riding: devo essere robusto a via vai di nodi della rete.
- Anonimità e privacy: onion routing, anonimizzo le connessioni.Il messaggio è racchiuso in una "cipolla" ed ogni nodo viene tolto uno strato, verso la destinazione.

- Trust e reputation dei peer: devo potermi fidare
- Network threats e defense
- Resilienza ai chunks

5 Reti P2P-File sharing

Ogni odo deve entrare nella rete (fase di bootstrap):

- configurazione statica: conosce già alcuni nodi nella rete
- ha informazioni relative ad un precedente utilizzo della rete
- usa i nodi sempre attivi con ip noti.

Una volta entrato, bisogna gestire il lookup delle risorse, questo viene gestito con una rete di overlay.
La rete virtuale che interconnette i peer è basata su una rete fisica sottostante, i collegamenti diretti nella rete logica sono canali di comunicazione, che uso come lookup di risorse.
Il routing avviene a livello 5, c'è il problema della prossimità di rete: due nodi vicini nella rete di overlay possono essere lontani nella rete TCP/IP fisica.

5.1 Overlay routing

Mi concentro sul routing, il retrieve è "semplice". L'overlay network gestisce l'instradamento delle risorse, devo poter inserire/rimuovere nodi e risorse ed identificarle.
Per le risorse uso un global unique identifier: può essere ottenuto con un algoritmo di hashing crittografico usando informazioni sulla risorsa come nome, data di creazione etc...
L'overlay network può essere con o senza strutture: se con struttura, al topologia dell'overlay network è ben strutturata.

5.2 Reti P2P non strutturate

Ho un grafo random la cui struttura emerge dal comportamento dei singoli nodi che entrano nella rete mediante regole. Il nodo entra contattando nodi in modo più o meno random.
Non c'è una topologia ben definita, inserimento/uscita di nodi e risorse è facile da gestire, ma il t. lookup è maggiore rispetto alle reti strutturate, in quanto ho prestazioni impredicibili.
Principali proprietà di un grafo random:

- coefficiente di clustering: il coefficiente di clustering di un vertice è una misura del grado di connessione dei nodi.
  $\frac{narchivicinidelnodo}{ndiarchipossibili}$,ad esempio con m vicino ho al massimo m· $\frac{(m-1)}{2}$.

Il coefficiente di clustering del grafo è la media dei coefficienti di clustering dei vertici.

- Average shortest path length: considero tutti gli shortest path fra i nodi e ne faccio la media.

5.3 Modelli per reti p2p non strutturate

Varie analisi in letteratura:

- Erdos -Renyi: numero N di vertici fissato, denoto con p: la probabilità che ci sia un arco fra due nodi.Il grado di un vertice segue una distribuzione binomiale: $p_k = \binom{N-1}{k} \cdot p^k \cdot (1-p)^{N-1-k}$
  Il coefficiente di clustering è pari a p, quindi basso. Posso creare un grafo random, ma non ha proprietà simili a quelle dei grafi reali.
  L'average shortest path è: $\frac{log(N)}{log(\frac{(N-1)}{p})}$.
  La coda del grafo decade in modo esponenziale, la rete generata è omogenea,ovvero non ha degli hub (accentratori nelle reti sociali).

- Watts-Strogartz: il coefficiente di clustering è alto, segue al proprietà dello small world: la distanza media tra due nodi dipende logaritmicamente da N.
  Non soddisfa però al presenza di hub.
  Small world: anche detto dei 6 gradi di separazione, se prendo due persone molto lontane in una rete sociale, ci sono al più 6 nodi di distanza fra loro.Esperimento di Milgram: selezione di gruppi di cittadini, che dovevano inviare delle lettere a due persone in un'altra città senza conoscere l'indirizzo di residenza, ma inoltrandole ad amici delle due persone per farla prevenire.

- Albert-Barabasi: propone rete a invarianza di scala: non viene cambiata la forma anche scalando le lunghezze. I gradi dei vertici seguono una legge potenza: $p_k \simeq c \cdot k^{-\alpha}$, con $2 < \alpha < 3$.
  La frequenza degli eventi varia come la potenza rispetto ad un attributo dell'evento.
  ad esempio: il numero di città con un certo numero di abitanti, la legge di Zipsf per la distribuzione dei file. Modello caratterizzato da un coda pesante: la decadenza della coda è molto lenta al variare di α, più α è piccolo e più e piccola la decrescenza.
  Il modello permette la creazione di reti con la proprietà di invarianza di scala: considerando il diametro del grafo generato in questo modello ho che:
  $d \simeq \ln(\ln(N))$, ovvero la crescita è molto lenta.
  Un nodo si collega alla rete ai nodi che hanno meno collegamenti, ci sono degli hub ma tendono a decrescere esponenzialmente. La maggior parte delle reti p2p usa questo modello.

5.4 Routing nelle reti p2p non strutturate

Classifico in base ad un indice di distribuzione risorse e peer: se l' indice è mantenuto in un unico nodo o in un cluster di nodi ho una rete centralizzata.
Le reti decentralizzate pure prevedono che ogni nodo abbia la conoscenza solo locale delle risorse, o al massimo dei suoi vicini.Posso anche avere soluzioni ibride, con una directory dei nodi semi-distribuita.
Nel primo caso ho un possibile bottleneck ed un single point of failure, nel secondo caso la ricerca dell'informazione è più complessa. Nel caso 3, i super-peer possiedono delle informazioni sui peer che coordinano.

5.4.1 Meccanismi di routing nelle reti decentralizzate

Ho due approcci:

- Flooding: approccio più semplice per il lookup. È un approccio distribuito, ogni peer propaga la sua richiesta di localizzazione della risorsa si suoi vicini,che la mandano ai loro vicini (se non hanno la risorsa)e così via.
  Si inonda la rete di messaggi fino a scoprire la risorsa o finché non scatta il TTL associato alla richiesta.
  La richiesta ha un ID univoco per evitare inoltri duplicati nel caso in cui la rete abbia dei cicli. Il costo del lookup è O(N), con N pari al numero di nodi nella rete.
  Una volta trovata la risorsa, questa viene rimandata indietro o con una risposta diretta o seguendo il percorso inverso. Tramite la seconda strategia, gli altri nodi presenti nel percorso e che vengono attraversati per arrivare al nodo che ha inizializzato la richiesta scoprono chi possiede la risorsa e memorizzano in una cache temporanea.L'approccio ha dei problemi:
  - la crescita esponenziale del numero dei messaggi da cui la rete viene inondata, suscettibile ad attacchi DDOS, o a nodi ni sovraccarico che non riescono a smaltire le richieste.
  - TTL: se mal dimensionato, rischia o di far inviare troppi messaggi o troppo pochi
  - Non c'è relazione tra la topologia di rete e la rete fisica.
- Random walk: soluzione che cerca di limitare il numero di messaggi che circolano nella rete, l'idea è che ogni peer inoltra la richiesta solo ad un numero scelto a caso di vicini.Il numero dei messaggi è ridotto, ma il tempo di rierca aumenta, un'evoluzione prevede di avviare più cammini random (con k= n° cammini $<$ N).

5.4.2 Reti overlay strutturate

La topologia emerge da una struttura ben definita, e va mantenuta quando le risorse vengono assegnate ai nodi. Ne esistono diversi tipi e la differenza è

legata fondamentalmente alla topologia della rete di overlay. Lo svantaggio è che l'aggiunta o rimozione è più complessa.
Il routing avviene mediante una hash table distribuita, per effettuare ricerche efficienti delle risorse: nelle reti p2p ogni risorsa ha un id univoco e nelle reti strutturate anche i peer hanno un id univoco (ottenuto mediante hashing), spesso lo spazio di identificazione è lo stesso.
Ogni peer avrà un certo n° di risorse con id vicino al suo ed il concetto di vicinanza varia in base alla topologia.
Il routing avviene cercando di mappare l'id di una risorsa all'id del peer più vicino. La hash table distribuita offre la stessa API di una normale hash table:

- look up: analogo alla tradizionale hash table, ma le entry sono distribuite sui vari peer
- retrieval
- delete

Ogni risorsa ha una coppia key-value, memorizzata nella hash table, occorre quindi mappare la chiave del nodo più vicino alla risorsa.
Le risorse sono mappate con una funzione hash (SHA-1), che viene applicata sui metadati e sui dati della risorsa.
Ogni nodo ha informazioni relative alle risorse mappate negli id da lui gestiti, che sono una porzione contigua, inoltre ogni chiave può essere mappata su molteplici nodi. Le difficoltà della hash table distribuita: ogni risorsa è identificata solo con il valore di chiave, quindi occorre conoscerlo. È facile fare query ti tipo "exact-match", ovvero conosco la risorsa ed i suoi metadati e quindi ho un matching esatto. Ma per query più complesse il supporto è difficile e costoso.

5.5 CHORD

Algoritmo o protocollo per il lookup di risorse nelle reti p2p con una topologia ad anello,utilizza funzione di consistent hashing.L'id dei nodi e delle risorse spnp mappate sull'anello con la funzione hash: ogni nodo è responsabile delle risorse con ID che va dal suo ID dall'ID del nodo che lo precede.
La risorsa con chiave k viene mappata sul nodo con il più piccolo id tale che: id $\geq$ k, tale nodo è detto il successore di k, o succ(k).
La metrica usata è basata sulla differenza lineare tra gli identificatori.

5.5.1 Consistent hashing

Cis non diverse implementazioni, sia i peer che le risorse sono mappate nello stesso spazio degli identificatori, usando la stessa funzione hash. La funzione è robusta rispetto ai cambiamenti, la rete p2p ha un charn elevato, ovvero i nodi entrano ed escono liberamente e questo ha impatto minimo sulla rete.
L'assegnazione di risorse ai peer è bilanciata.

5.5.2 Finger table in CHORD

La finger table in CHORD è una tabella di routing mantenuta da ogni nodo, con una struttura ben definita. Se m = # bit dell'ID, al dimensione della finger table sarà pari ad m. Se indico la finger table di un nodo p con $FT_p \Rightarrow FT_p[i]=succ(p+2^{i-1})mod2^m$, con $1 \leq i \leq m$.
L'idea della finger table è quella di permettere ad ogni nodo di fare lookup delle risorse avendo informazioni approssimative sulle posizioni più lontane,sapendo quali sono i suoi nodi vicini e che risorse gestiscono.
L'algoritmo di routing procede come segue:

- Ho una chiave k, che è l'id di una risorsa e voglio conoscere il succ(k).
  Se p è il nodo che sta effettuando il lookup, fa un controllo per vedere se la risorsa è nella sua zona e se non lo è controlla la prima entry della finger table: se $k \leq FT_p[1]$ allora inoltra la richiesta. Altrimenti la inoltra ad un nodo che ha l'ID più vicino all'ID della risorsa cercata.
  Il costo di lookup è O(logn), con n = n° dei peers.

esempio: 0-31 nodi, m=5, quindi la finger table ha dimensione pari a 5., l'indice $i \in (1,..,5)$.

5.5.3 Ingresso/uscita di nodi in CHORD

L'operazione di join/leave ha un costo asintotico di $O(log^2(n))$. Il problema da gestire è che i valori delle finger table devono rimanere costanti, per questo motivo ogni nodo mantiene un puntatore al successore nella prima riga della finger table ed anche un puntatore al predecessore.

- Join: il nodo deve capire dove collocarsi nella rete, se il suo id è p, il suo successore sarà p+1.
  A questo punto si inserisce nella rete, inizializza la finger table chiedendo le informazioni agli altri nodi. Ora le informazioni cambiano, in particolare i nodi successori: il nodo p deve avvisare gli altri nodi di aggiornare le finger table e deve gestire le risorse che gli vengono assegnate, che sono quelle con ID compreso fra il suo e quello del predecessore; deve quindi trasferire le risorse da gestire.

- Leave: il nodo che esce deve trasferire le sue chiavi al successore, vanno inoltre aggiornate le entry del successore; p avverte il suo predecessore per fagli aggiornare la sua finger table.
  Anche il nodo successore deve cambiare predecessore, le altre entry non vengono aggiornate, periodicamente i nodi effettueranno la ricerca dei nodi nelle finger table e refresheranno le entry.

5.5.4 Vantaggi e svantaggi

Vantaggi di CHORD:

- Distribuzione del carico, stesse key per ogni nodo

- Routing piuttosto efficiente
- Robustezza: CHORD aggiorna periodicamente le finger table

Svantaggi di CHORD:

- Manca la nozione di prossimità fisica
- Supporto costoso senza matching esatto.

5.5.5 Algoritmi di verifica formale-CHORD

Operazioni di join/leave in chord: meccanismo per poter mantenere informazioni aggiornate in una finger table.Vengono effettuate delle query di lookpup per i successori che devono essere conosciuti nelle tabelle, inoltre un'informazione fondamentale per corretto funzionamento è che il 1° elemento della finger table contenga il successore del nodo all'interno dell'anello. Ricercatrice (ora prof della Priceton University), che in un articolo del 2017 (ma il lavoro copre anche degli anni precedente) ha mostrato che sotto le ipotesi dell'algoritmo di chord non abbiamo la correttezza. Ha quindi proposto una specifica di CHORD che soddisfa requisiti di correttezza e ne ha fatto una verifica formale.Il lavoro dimostra come rendere chord corretto, ed Amazon ha deciso di usare metodi di verifica formale per i protocolli utilizzati. (Tenere a mente l'articolo se dovesse servire un implementazione di chord)

5.6 Pastry

Una sorta di middleware su cui sono state sviluppate altre applicazioni, ad esempio Scribe, SQUIRREL,PAST...
In Pastry, il routing usa una soluzione basata sul meccanismo del plaxton routing. L'idea di base è nella metrica di distanza che non è lineare ma basata sul matching dell'ID che identifica una risorsa o un peer. La risorsa viene memorizzata sul nodo ceh ha il prefisso più lungo in comune con la risorsa stessa.
La soluzione è leggermente più complessa, poiché ogni nodo mantiene anche un'insieme di foglie, che sono i nodi a lui più vicini nello spazio bidimensionale degli ID.
La topologia è ad anello ed è percorribile in tutti e due i sensi. Il routing viene effettuato col longest prefix matching (se non trovo nessun nodo corrispondente,inoltro al nodo numericamente più vicino), le chiavi sono rappresentate da simboli con un certo numero di bit, solitamente d simboli di b bit ciascuno. Ad ogni passo del lookup il nodo inoltra la query al nodo con l'id vai via più vicino a quello della risorsa cercata.
Ogni nodo ha una tabella di routing ed un leaf set, il costo della ricerca è $O(log_2^b N)$. esempio: chiave con d=4 e b=2 $\Rightarrow$ 8 bit.
La tabella di routing è costruita seguendo delle regole:

- gli id dei nodi sulla riga n-esima condividono le prime n cifre con l'ID del nodo corrente

- la (n+1)-esima cifra degli ID sulla riga n-esima è il numero di colonna.

Ad ogni elemento possono corrispondere più nodi, la metrica di scelta è per prossimità, ad esempio in base al RTT della rete TCP/IP.
Le righe della tabella di routing sono $\lceil log_2^b \mathrm{N} \rceil$, con 2^b-1 elementi per riga.

5.7 Architetture ibride

Il sistema ha dei nodi super-peer, che tipicamente hanno capacità maggiori dei semplici peer, sia di hardware che di rete etc... Il routing avviene solo fra i super-peer, a cui previene la richiesta per un risorsa; i super-peer gestiscono un certo numero di peer. esempio: BitTorrent:
ogni nodo può richiedere dei chunks, ma allo stesso tempo deve anche fornire i chunck che ha scaricato, c'è un meccanismo basato su game theory per disincentivare i nodi selfish

6 Middleware

Come un SO per un SD: mette a disposizione una serie di servizi per costruire al di sopra del middleware stesso tutte le differenze dei singoli nodi del SD.
Usa dei meccanismi come le socket, sarà il developer a dover nascondere il livello di astrazione, il middleware lo fa già di per se.
Il middleware è un sistema general-purpose che sta nel mezzo, altre definizioni:

- strato software che fornisce un'astrazione di programmazione e che nasconde l'eterogeneità dell'hardware sottostante.
- layer virtuale tra applicazione e piattaforma che fornisce un grado significativo di trasparenza.

6.1 Tipi di middleware

Ci sono diversi tipi di middleware,a seconda della specifica funzione:

- Middleware object-oriented: i componenti sono visti come oggetti con un'identità propria ed un'interfaccia esposta con dei metodi pubblici, la comunicazione è tipicamente sincrona.
- Message Oriented Middleware: comunicazione asincrona, può offrire affidabilità e flessibilità. Molte implementazioni sono basate su code di messaggi.
- Middleware per componenti: evoluzione del MOM, supporta sia comunicazione sincrona che asincrona.
- Middleware orientato ai servizi: enfasi sulla comunicazione ed integrazione di componenti eterogenei sulla base dei protocolli aperti. La comunicazione è sincrona o asincrona e persistente.

Nello sviluppo dell'applicazione distribuita si sceglie il middleware che ha lo stesso stile architetturale del sistema progettato.

7 Sistemi auto-adattativi

Sistemi che adattano il loro comportamento in base a modifiche sul sistema stesso o sull'ambiente circostante.
La definizione autonomic computing deriva dal sistema nervoso autonomo, che controlla alcune funzioni vitali.
I sistemi self adaptive:

- richiedono la minima interazione umana
- sono capaci di adattarsi in maniera reattiva o proattiva:
  - reattiva: reagiscono ad eventi già accaduti
  - proattiva: predicono avvenimenti in modo da pianificare in anticipo le azioni di adattamento.

Gli obiettivi di un sistema self-adaptive:

- self-configuring: il sistema fa tooning automatico dei parametri più adatti rispetto ai cambiamenti ambientali
- self-healing: il sistema scopre i guasti e reagisce
- self-optimizing: il sistema cerca di ottimizzare le prestazioni, con azioni di adattamento per migliorare la QoS
- self-protecting: il sistema si protegge da attacchi esterni, scegliendo le soluzioni di difesa più valide

Il sistema deve conoscere il suo stato interno e le sue attuali condizioni operative, quindi deve effettuare self-monitoring ed aggiustare di conseguenza: self-adjustment. Il modello è molto simile ad un sistema di controllo a retro-azione.

7.1 Architettura MAPE

Architettura MAPE, sistemi in grado di adattarsi a cambiamenti nell'ambiente circostante. 4 fasi principali:

- monitoraggio: sys monitora ambiente in cui opera con dei sensori.
- analisi: prende output della fase di monitoraggio e durante questa fase si valuta se occorre attivare la fase di planning, che andrà a decidere il cambiamento da attivare all'interno del sistema.
- planning:decide quali azioni effettuare nel sistema.

- execute:rende effettive le modifiche sul sistema.
- knowledge:base di conoscenza comune alle varie fasi del ciclo, per questo anche detto MAPE-K.

La fase di planning è quella più challenging, per cui si possono adottare molteplici strade:

- Teoria dell'ottimizzazione
- Algoritmi euristici
- Machine Learning
- Teoria dei controlli
- Teoria delle code

7.2 Esempi di sistemi auto-adattativi

Esempi di sistemi self-adaptive: in questi sistemi l'adattamento viene effettuato per soddisfare requisiti non funzionali(prestazionali) del sistema, in particolare l'obiettivo è soddisfare requisiti riguardanti la QoS, in quanto questi possono essere specificati in un SLA in particolare nei SLO.Tra i diversi requisiti ci possono essere tempo di risposta dell'app, la sua disponibilità, costo pagato dall'utente etc...
esempio di Amazon EC2 Auto Scaling:
il servizio fornisce capacità autonomia di scalare aumentando o diminuendo il numero di istanze EC2 in base a opzioni utente e controlli sullo stato di salute delle istanze utilizzate. Consente anche di escludere un'istanza EC2 non funzionante, ad es. che non risponde a dei ping per controllare lo stato di funzionamento dell'istanza. I ping sono anche chiamati heartbeat monitoring: lo scopo è capire se le istanze sono up& running, se non riceve risposta il servizio considera la VM non disponibile e la esclude.
Considero in termini di scalabilità orizzontale:
la configurazione prevede di definire la capacità desiderata, es 2 istanze ovvero l'utente si aspetta che il servizio funzioni bene con queste due istanze, poi si specifica il valore massimo delle istanze allocabili ed il valore minimo. Il numero di istanze $\in$[max,min]. Il servizio di EC2 autoscaling determinerà il numero di istanze realmente utilizzate. Amazon dallo scorso anno mette a disposizione una politica di scaling che funziona anche in modo pro attivo: c'è anche politica reattiva, nella nuova politica il n° istanze varia con una tecnica che predice quella che sarà la metrica che farà aumentare/decrementare n° istanze attive.
La politica di EC2 è la politica di riferimento in tutti quei sistemi che offrono elasticità. Idea: politica basata su threshold, in cui l'utente del servizio specifica una metrica ed una soglia sulla metrica, al superamento della soglia su questa metrica vengono aggiunte istanze, invece al superamento di una soglia bassa vengono tolte istanze dal sistema.

Esempio: soglia e metrica di interesse è a discrezione dell'utente, nella gran parte dei framework la soglia è configurata sull'utilizzazione della cpu (es con il comando top in Linux).
E' possibile configurare la politica di scaling in modo che se l'utilizzazione della VM supera il valore del 70% per 1 min, viene aggiunta nuova VM. Posso anche impostare una soglia di scale out : se utilizzazione scende sotto il 30% per 1 min, rimuovo istanza. Ovviamente ho delle soglie definite come sopra sul n di VM. Politica semplice ed intuitiva: definisco metrica di utilizzo e soglia. Ma è non banale sapere come impostare la soglia di utilizzazione: saper dire quanta è la soglia conveniente non è banale da configurare. La teoria delle code MM1 suggerisce soglie del 30 %, ma rimane comunque difficoltoso per l'utente andare ad impostare le soglie: se eseguo componente CPU-intensive, in quel caso ha senso usare questo tipo di metrica, ma se app. è memory intensive, devo (per l'app. o per il componente specifico) considerare l'uso della memoria. Posso anche combinarle, aggiungere una metrica sull'I/O (throughput discho), la banda di rete usata, rimane comuqnue la difficoltà di definire le metriche da usare ed i valori di threshold.
Inoltre, l'utente deve fissare valori su delle metriche non proprie del SLA: voglio usare servizio di autoscaling per applicazione che do a terzi, garantendo un tempo di risposta di 1 secondo. Come lo traduco in termini di valori soglia?La politica presenta quindi degli svantaggi, ampia ricerca sull'argomento. Per usare autoscaling occorre usare servizio di monitoring. Un'altra politica di Amazon è pro-attiva, basata su Machine Learning e cerca di predire carico di lavoro futuro e utilizzazione di risorse EC2, su base di un modello di ML di cui viene fatto training, usa le informazioni per determinare n° istanze di VM da usare.
Il planning è fatto su misure acquisite dal monitoring, limitazione è che richiede training del modello, mi aspetto che sia rete neurale opportunamente definita per predire serie storiche.(Amazon non ne mostra l'implementazione esatta)
Questo è un esempio di ciclo MAPE: ci sono tutte le fasi:

- si monitora uso della cpu
- si analizza threshold, se supera 70% si da trigger a fase di planning
- planning in cui si controlla che valore di utilizzazione monitorato sia $>$ o $<$ 70% e se è $>$ viene dato trigger a fase di execute per aggiungere una nuova istanza.
- execute, in cui si mettono in atto le decisioni prese

Stesso avviene nel modello pro-attivo, usando a che dati predetti dal modello di ML. Altro esempio, che è legato allo stesso obiettivo, la struttura del ciclo MAPE è la stessa:
soluzione differisce nella fase di planning, che vuole identificare n° ottimo da stanziare in modo da soddisfare SLA basato sul t. risposta dell'app.
In questa proposta, la fase di planning usa come metodologia la formulazione di un problema di PLI, problema NP-hard quindi risolvibile rapidamente solo su problema di piccole-medie dimensioni. Se $>$ 50 VM, il tempo di risoluzione crese

esponenzialmente, quindi diviene soluzione poco efficiente in quanto lenta.Possono essere usate politiche di planning euristico.
Un altro esempio di applicazione, esaminato nell'ambito di appl. orientate a servizi (che precedono architetture a microservice): ho un app., composta da più componente e ciascuno di questi può avere diverse istanze che differiscono in base a parametri di QoS, perché fornite da diversi fornitori.Problema era fornire appl. che soddisfacesse QoS globale del sistema. Sistema MOSES(Grassi x Cardellini x Lo Presti), metodologia usata è formulazione di problema di PL. Queste soluzioni sono caratterizzate da un design del sistema adattativo come sistema centralizzato, ovvero gran parte dei componenti del sistema MAPE, o meglio tutti, sono eseguiti sullo stesso nodo.Ha evidente limite di scalabilità se usato in un contesto esecutivo, esempio ambiente di fog o edge computing.
Da un punto di vista architetturale posso realizzare in modo distribuito il sistema MAPE, distribuendo le varie fasi. MAPE decentralizzato, possibile con vari patterns la cui scelta dipende dal sistema e dai requisiti dell'app. Come decentralizzare:

7.3 MAPE Gerarchico

- pattern master-worker: il nodo master si occupa delle fasi più delicate, ovvero di analisi e planning, i nodi worker effettuano fase di monitoring ed execute. Vantaggi: master ottiene dati dai worker, ha visione globale del sistema e prende decisioni globali.
  Contro: master può essere collo di bottiglia e single point of failure.Decisione del planning va consegnata a tutti i worker, che dovranno adattarsi.

- regional pattern: se ho sd con estensione di tipo geografica, posso avere diverse regioni in cui eseguo componenti del sistema, magari fra loro lascamente accoppiate. Idea: ho nodi planner per una o più regioni che si occupano dell'attività di planning per determinare regioni.
  Vantaggio: regioni possono avere diverse amministrazioni, quindi ho maggiore flessibilità. Svantaggio: difficile raggiungere scopi globali per il controllo dell'app.

- Gerarchia per il controllo: a differenza della master-worker ho cicli mape a lvl locale e cliclo mape globale.
  Vantaggio: cicli locali controllano porzione dell'app, quello globale determina dando indicazioni ai singoli cicli per capire come procedere per l'adattamento, sono più tolleranti ai guasti Svantaggio: più difficile da realizzare perché non è semplice identificare lvl di controllo locale o globale.

7.4 Flat mape

- coordinate control pattern: ho molteplici cicli di controllo di cui ognuno controlla delle prati del sistema e le diverse parti di controllo di coordinano fra di loro.

Vantaggio : migliora la scalabilità, cicli di controllo sono largamente distribuiti.
Svantaggio: definire strategia di planning in modo che decisori prendano decisioni d adattamento.Possibile farlo con teoria dei giochi o tecniche ml.

- information sharing pattern: coordinamento è solo sulla fase di monitoring. Posso considerarlo come caso particolare dello schema di controllo coordinato in cui comunicazione si limita a questi componenti.
  Vantaggio: migliora scalabilità Svantaggio: manca coordinamento in fase di planninng: può accadere che ciascun planner prende decisioni che possono essere in contato con le altre decisioni prese dagli altri planner.

esempio: schemi di controllo su elasticità di applicazioni a microservice eseguite su Kubernetes.Applicazione dei diversi cicli di controllo.

8 Comunicazione nei sistemi distribuiti

Prettamente basata sullo scambio di messaggi, soluzione più nota è quella di suddividere il problema in livelli, così che a livello logico ciascun livello di un sistema comunichi con il livello corrispondente dell'altro sys.
Aggiunta del middleware, che è il collante dei sd, posto fra il resto del sistema e le applicazioni.A livello del middleware sono forniti servizi comuni e protocolli general purpose, con l'obiettivo di nascondere l'eterogeneità dei sys sottostanti.Nell'ambito dei protocolli middleware ci sono:

- protocolli di comunicazione
- protocolli di naming: condivisone di risorse tra applicazione.
- protocolli di sicurezza
- protocolli per consenso distribuito: algoritmi per raggiungere il consenso in modo distribuito.
- protocolli locking distribuito: servizio di locking che si ritrova in numerosi framework open source.
- protocolli per consistenza dei dati.

8.1 Protocolli di comunicazione

Tipi di comunicazione in sys o appl. distribuita:

- persistenza
- sincronizzazione
- dipendenza dal tempo

Prima differenza per la persistenza è tra comunicazione persistente o transiente:

- comunicazione persistente: msg viene memorizzato dal middleware di comunicazione per tutto il tempo dopo la consegna.Mittente non deve essere sync col destinatario, e non c'è bisogno che destinatario sia attivo quando viene inviato il msg. Ho quindi il disaccoppiamento temporale.
- comunicazione transiente: msg memorizzato dal middleware solo nel tempo in cui mittente e dest. sono in esecuzione.
- Comunicazione sincrona:quando msg è sottoposto al middleware di comunicazione mittente si blocca fintanto che la comunicazione non è completata. Esistono diversi tipi di comunicazione sincrona in base al tempo in cui mittente attende:
  - mittente bloccato finché middleware di comunicazione non prende il controllo della trasmissione.
  - bloccato finché middleware lato destinatario(sto parlando di middleware nella comunicazione a livelli)non prende in gestione la richiesta.
  - bloccato finché destinatario non ha elaborato il messaggio.
- Comunicazione async: una volta che mittente ha inviato msg, riprende elaborazione. Msg è memorizzato temporaneamente finché non viene trasmesso. Ricezione può essere bloccante o non.
- Comunicazione discreta: ogni msg inviato è unità di informazione completa a se stante, indipendente dagli altri.
- Comunicazione a streaming: prevede invio di altri messaggi, che sono in relazione temporale fra loro.(es: appl. di video-streaming).

Combinazioni possibili tra persistenza e sincronizzazione:

- Comunicazione persistente asincrona: esempio, nella chat di teams o posta elettronica. Mittente invia il messaggio ed il destinatario può non essere attivo. Il messaggio è memorizzato e nel momento in cui destinatario accede, lo riceve (riceve la notifica) e quando ciò avviene può accadere che il mittente non sia attivo.
- Comunicazione persistente sincrona: posso memorizzare info ma mittente rimane bloccato finché lato destinatario non viene accettato il msg.Middleware di comunicazione dest. invia ack relativo alla ricezione del messaggio, può accadere che il mittente non sia presente nel sys quando riceve ack.
- Comunicazione transiente asincrona: mittente invia msg e continua nella sua elaborazione, dest. riceve il msg. Mittente e dest. devono essere compresenti temporalmente.
- Comunicazione transiente e sincrona:

- Comunicazione sincrona basata su ack: mittente invia msg, dest lo riceve non lo elabora subito, ma viene inviato ack al mittente. Mittente sa che msg è stato ricevuto, non sa se questo sarà elaborato.
- Comunicazione sincrona basata su delivery: dest invia ack quando inizia processamento della richiesta, distinguerò tra ricezione e consegna del msg a lvl applicativo.Qui ack è inviato dopo consegna a lvl applicativo, quindi mittente sa che è stato consegnato.
- Comunicazione basata su risposta: mittente rimane bloccato finché non riceve risposta, attende consegna, elaborazione e risposta.

8.2 Fallimento nella comunicazione

Assumo di avere un appl distribuita con due componenti, di cui uno fa da client ed uno da server.

- Errore di comunicazione
- Crash di client o server. In particolare un crash del server può avvenire in istanti diversi:
  - prima di servire la richiesta
  - dopo aver ricevuto e processato la richiesta, ma prima di inviare la risposta.

  Il client non può distinguere tra le due situazioni, perché comunque non riceve risposta.

Ci sono diverse semantiche della comunicazione in SD quando possono avvenire degli errori.

1. Semantica may-be: il client non sa se il server ha eseguito o meno il processamento richiesto. Se riceve risposta lo sa, altrimenti non sa nulla.Semantica best-effort, quindi la più debole.
2. Semantica at-least once: server ha processato la richiesta del client almeno una volta.
3. Semantica at-most-once: al più una volta, client sa che server ha processato richiesta al più una volta.
4. Semantica exactly once: client sa che server ha processato una sola volta.Semantica più forte

Importante capire il tipo di semantica di comunicazione supportata dal middleware.Ritrovata sia quando si parla di processamento di servizi, sia nel caso di sys a code di messaggi.
Ora considero il processamento dei servizi, ma lo stesso processo si applica al delivery di messaggi.

8.2.1 Maccanismo di base

Meccanismo che prevede di ri-inoltrare la richiesta di servizio.Lato client: finché non ottiene risposta, o diviene certo che server è guasto, continua a provare l'invio della richiesta.Request Retry(RR1).
Lato server: meccanismo che server usa per filtrare duplicati,se le richieste provengono dallo stesso client per lo stesso servizio.È utile per gestire richieste di servizio non idem-potenti: richiesta idem-potente è tale per cui se esguo servizio 1 o N volte, il risultato è sempre lo stesso.(es operazione read-only su db, se assumo non ci siano scritture).
Operazione non idem-potente altera lo stato del servizio(es: contatore).
Non basta il filtraggio dei duplicati, ma server deve anche avere meccanismo Retransmission Result(RR2), ovvero memorizza risultato della computazione, in modo da poterlo ritrasmettere successivamente senza doverlo ricalcolare, nel caso riceva una richiesta duplicata.
La combinazione dei due meccanismi è necessaria nel caso in cui l'operazione è non idem-potente.

- Semantica maybe: non attuo nessuno meccanismo per garantire affidabilità della comunicazione, client invia richiesta: se riceve risposta bene, sennò pace. Semantica di comunicazione usata da UDP

- Servizio può essere stato eseguito più volte in caso di retx.Semantica at-least-once: Per implementare questa semantica, usato solo RR1.Server non usa nessun meccanismo, non può sapere se messaggio è duplicato.
  Semantica adatta a servizi stateless, ovvero idem-potenti.Server molto più semplice da realizzare, la scalabilità è più immediata: se avessi stato dovrei riportarlo anche sulle repliche.Anche in caso di tolleranza ai guasti è più semplice.
  Client comunque non sa quante volte è stata processata la richiesta dal server, o meglio sa che è stato fatto una sola volta.

- Semantica at most once: se il servizio viene eseguito è fatto al più una volta. Client sa che se riceve una risposta, questa è stata processata una sola volta.In caso di insuccesso, non ho informazioni.
  Usati lato client e lato server tutti e tre i meccanismi di base esaminati: retx lato client allo scadere di un TO dall'invio della richiesta, filtraggio duplicati e retx risultato lato server.
  Server deve essere stateless: deve tenere traccia dello stato di servizio del client e poter memorizzare risultato della computazione.Adatta a qualunque tipo di servizio, sai idem-potente che non.Semantica non ha coordinamento tra client e server: se client non riceve risposta, client non sa se server ha eseguito o meno il servizio... Implementazione lato server: devo avere duplicate filtering e memorizzazione risposta, invece di rieseguire ogni volta l'handler().
  Per identificare risposta duplicata: client inserisce unique ID di richiesta,

quando la invia duplicata usa lo stesso ID.Tenendo traccia degli ID, server può identificare il duplicato.
Ma come assicurare ID univoco? Si può usare digest che tenga conto di meta-info relative al client o alla richiesta, in modo da diminuire la prob. che due digest siano uguali e generati da due diversi client.
Server:
if seen[xid]
r = old[xid]
else
r = holder()
old[xid] = r
seen[xid] = true Server non potrà mantenere per tempo indefinito traccia di id e computazioni, quando è safe cancellare vecchi valori?
Posso usare finestre scorrevoli e n° seq, oppure assumere che le info abbiano determinato tempo di vita, in modo da rimuovere quelle vecchie.
Altro problema da gestire è cosa accade se server è sovraccarico o se TO client è < del tempo processamento.
Quello che può avvenire è che il client ritrasmetta richiesta, mentre server sta ancora effettuando computazione.Questo è un altro aspetto da gestire, ovvero come trattare richieste duplicate mentre sto computando la prima.

- Semantica exactly once: permette di offrire garanzie migliori di tutte, ma è la puù complessa da realizzare in SD. Richiede accordo completo nell'iterazione fra client e server.
  Semantica all-or-nothing: o il servizio viene eseguito per intero, o nulla. Altrimenti viene eseguito una sola volta, stando attenti ai duplicati. Semantica è poco praticabile in sys reale a larga scala, in cui ci sono aspetti di sincronia che diventano preponderanti.
  Maggior parte dei SD che offrono semantica di comunicazione la offrono at-least o at-most once.
  Semantica complessa in quanto i 3 meccanismi di base non bastano, ma sono richiesti ulteriori meccanismi per tollerare guasti lato server:

  1. Trasparenza della replicazione del server: se ho servizio stateless non è difficile, load balancer che va messo davanti repliche del server non tiene conto dello stato. Se servizio è statefull, distribuzione delle richieste sulle repliche deve tener conto dello stato.Es: ho dei tweet e voglio contare i tweet, per classificarli: devo usare contatori, per conteggiare n° occorrenze per ciascun # identificato. Se uso i contatori su ciascuna replica, non ho problemi. Ma se n° richieste è elevato, non basta più una sola replica: per replicare il servizio devo partizionare lo stato, quindi i diversi contatori tra le repliche. Devo però distribuire richieste tenendo conto del valore del' # , in modo da indirizzare correttamente il tweet.
  2. Write-ahead-logging: cambiamenti nel sistema devono essere resi effettivi solo dopo che sono stati registrati nel log. Il log deve essere

memorizzato in modo persistente su un dispositivo di storage.

3. Recovery: meccanismi per recuperare dopo fallimento del server, in modo da recuperare stato e ricominciare esecuzione del servizio da un punto sicuro.

8.3 Programmazione di applicazioni di rete

Programmazione di rete vista fin'ora è programmazione di rete esplicita: API socket, usata gestendo lo scambio esplicito tra client e server.
Usato nella maggior parte delle applicazioni di rete, es Web Server o Web browser.Distribuzione dei componenti non è trasparente e pone sulle spalle dello sviluppatore la maggior parte del peso.
Voglio innalzare il lvl di astrazione, fornisco strato di middleware.

8.3.1 Programmazione di rete implicita

Chiamata a procedura remota: meccanismo vecchio nei SD.
Invocazione di metodo remoto: trasposizione della chiamata a procedura remota.
Vedremo in C, Java RMI e Go, a seconda del metodo avrò diversi gradi di trasparenza e diversi gradi di comunicazione (maggior parte delle volte sincrona e transiente, ma in Go anche async e transiente).
Nello stack ISO/OSI, colloco layer del middleware tra layer 4 e 5.
Avrò meccanismi per gestire interazione richiesta/risposta e per gestire eterogeneità dei dati.
Middleware di comunicazione si occuperà di:

- Chiamata di metodo remoto, identificazione del metodo remoto chiamato (stesso vale per la procedura).
- Gestione eterogeneità dei dati, operazioni di marshaling/unmarshaling dei parametri.
  Nel caso di Java RMI usata serializzazione di dati.
- Gestione di errori sia durante comunicazione, sia durante chiamata del metodo remoto, anche errori lato user.

Problemi generali da dover risolvere:

1. Come gestire eterogeneità nella rappresentazione dei dati
2. In presenza di errori, qual'è semantica della chiamata a procedura remota.In caso di procedura locale: semantica exactly once, ma se procedura è remota: semantica at-least once o at-most once; trade off tra prestazioni e costo implementativo.
3. Come effettuare binding fra client e server

8.3.2 Eterogeneità dei dati

Ordinamento byte può essere little endian o big endian, ho funzioni per gestire la differente coincida dei dati.
Aspetti di eterogeneità soprattutto in caso di dati strutturati, come gestirla:

1. Codifica inserita nel messaggio stesso, ho un header con campo che specifica codifica nel msg.

2. Mittente converte i dati nel formato che si attende il destinatario.

3. Formato di dati concordato tra sender e receiver, chi invia trasforam la codifica della codifica comune, chi riceve decodifica

4. Intermediario che si interpone fra send e recv, conosce i formati di codifica di entrambi, riceve msg dal destinatario, lo converte ed invia al dest.

Nel caso di RPC è usata la 3° alternativa.
Altre soluzioni useranno 4°
Se confronto 2° e 3° alternativa in termini di prestazioni: la 2° alternativa richiede che tutti i componenti dell'appl distribuita conoscano tutte le rappresentazioni possibili per ogni dato.Il vantaggio è che conversione è immediata, ma svantaggio è che conoscenza deve essere completa: se ho N componenti, al più ciascun componente deve conoscere N · (N-1) componenti.
Nella 3° ho formato comune, e ciascuno conosce funzioni di conversione dal proprio formato a questo formato specifico. Vantaggio: poche funzioni di conversione, 2·N, ma svantaggio è che operazione di conversione è più lenta.
Da un punto di vista architetturale ho dei pattern di gestione:

- Proxy: viene aggiunto lato client e server un componente, detto proxy, che si occupa di supportare trasparenza ad accesso e locazione.
  Il proxy, lato server per esempio, controlla accesso alla specifica procedura o allo specifico metodo.
  Proxy è locale nello spazio di indirizzamento, crea replica dell'altro endpoint: su client ho proxy che fa funzioni del server e viceversa.

- Broker: Incapsulo tutti i dettagli relativi alla comunicazione, separandoli dalle funzionalità. Broker permettere di far interagire fra loro i componenti senza che si preoccupino dei dettagli.
  Premette di identificare presso chi va inoltrata la richiesta, considerando anche eterogeneità

8.3.3 Binding del server

Come faccio ad agganciare il client al server:

- Binding statico: indirizzo del server su cui viene effettuata RMI è cablato nel codice del client, non aggiungo overhead perché conosco il server da contattare, ma manca trasparenza,ad esempio rispetto alla locazione

- Binding dinamico: il collegamento effettivo tra client e componente che offre servizio avviene solo al momento dell'esecuzione, collegamento avviene con entità interposta, che fa da smistatore di richieste verso la procedura. Ho maggiore flessibilità e trasparenza, per esempio posso effettuare distribuzione della richiesta, ma pago in termini di overhead.Posso distinguere due fasi:
  - Naming: fase statica, client specifica a chi vuole essere connesso, effettuata prima dell'esecuzione.
    Associo dei nomi unici all'interno del sistema alle operazioni o alle interfacce astratte, quindi collegamento avviene con l'interfaccia specifica del servizio.
  - Adressing: fase dinamica, a run time, client deve essere realmente collegato al server. Indirizzamento può essere implicito o esplicito: nel caso esplicito mando richiesta broadcast o multicast alle repliche, attendendo la prima risposta.
    Nel caso di addressing implicito c'è componente aggiuntivo che permetterà di registrare i servizi e avendo delle tabelle di routing opportune, permette di collegare identificativo astratto del servizio a chi concretamente offre quel servizio.

8.4 Remote procedure call

Anche noto con l'acronimo RPC, primo meccanismo che ha permesso di realizzare middleware di comunicazione che astraesse dalla comunicazione di base di rete, permettendo di realizzare appl. distribuite con pattern architetturali a lvl o orientati agli oggetti.
Proposta nel 1984: utilizza modello interazione di tipo client/server con la stessa semantica della chiamata a procedura: processo in esecuzione su un client invoca procedura eseguita su un nodo server. In versione base comunicazione è sincrona: processo lato client è sospeso, parametri di input/output sono inviati via messaggi, scambiati tra client e server ma questo scambio non è visibile al programmatore.
Meccanismo usato in molti sistemi distribuiti, sviluppato ed implementato con numerose tecnologie e linguaggi di programmazione:

- C: Sun RPC
- Java: Java RMI
- Go
- Ice
- Microsoft .NET
- Remote Python Call
- Distributed Ruby

- JSON-RPC
- CORBA

Chiamata a procedura locale: main $\Longrightarrow$ procedura locale $\Longrightarrow$ ritorno al main. Come realizzo tutto questo se le due procedure sono in esecuzione su due macchine differenti: ho eterogeneità degli ambienti di esecuzione, inoltre passaggio parametri non può avvenire mediante stack.Nel momento in cui lato client effettuo RPC, viene presa in gestione dal client stab, che invocherà chiamata di basso livello. Nel server ho demone in attesa, riceve richiesta di esecuzione della procedura, con procedura di dispatching determina esatta procedura remota di cui è stata richiesta l'esecuzione l'attiverà localmente. Otterrà risposta, assemblerà msg risposta ed invierà al nodo chiamante. Il nodo chiamante convertirà msg risposta e manderà output al chiamante.

8.4.1 Architettura RPC

Introduco due componenti che realizzo quello che è il middleware di comunicazione in questo caso specifico.Uso pattern proxy lato client, detto client stub e proxy lato server che è il server stub. Client stub svolge sulla macchina del client il ruolo del server e viceversa per il server stub.

1. Client invoca client stub, a cui passa procedura chiamata con param di input. Client stub gestirà tutti gli aspetti che middleware vuole nascondere:
   - gestisce binding col server
   - gestisce eterogeneità dei dati gestendo param input output
   - gestisce msg richiesta inviato lato server e msg risposta.
2. Client stub invia msg richiesta, che sarà ricevuto dal server stub
3. Server stub gestisce param input ed invoca procedura
4. Server stub prende param output e mette in un msg risposta ed invia a client stub
5. Client stub spacchetta il msg risposta, gestendo eterogeneità dei dati, e passa risultato alla procedura chiamante

Tutto il meccanismo è trasparente al client ed al server, gli stub sono prodotti in modo automatico. Stub sono gli unici componenti che devono essere progettati dallo sviluppatore dell'appl. distribuita. Portmapper: fa da service registry, ovvero fa binding fra client e server.
Esamino i passi della RPC:

1. Lato client, invoco client stub tramite classica chiamata a procedura locale.

2. Il client stub costruisce un msg in cui identifica procedura richiesta ed inserisce param di input effettuando il marshaling dei param di input: usato per gestire l'eterogeneità dei dati il meccanismo di usare formato comune.Il client stub a questo punto chiama SO locale
3. Il SO del client invia msg al SO remoto.
4. il SO remoto passa il messaggio al server stub.
5. server stub spacchetta msg prelevando i parametri e convertendoli in formato locale ⇒ unmarshaling. A questo punto invoca la procedura del server.
6. server al termine della procedura ridà risultato al server stub
7. Server stub crea msg risposta, facendo marshaling dei param output
8. SO server manda msg a SO client
9. SO client passa al client stub
10. Client stub passa output al client.

Problemi per RPC

- Gestire eterogeneità dei dati
- Come realizzo passaggio param per riferimento: so che proc chiamante e chiamata sono in esecuzione su due nodi con diverso spazio di indirizzamento. Non posso passare memory address
- Se ci sono errori, come lo interpreto? Cosa può considerare certo il client rispetto all'esecuzione della procedura
- Come effettuo il binding alla macchina su cui viene eseguita la procedura chiamata.

8.4.2 Rappresentazione dei dati

Middleware RPC fornisce un supporto automatizzato: il codice fa marshaling/unmarshaling dei dati e lo genera il middleware, codice diviene parte degli stub.Questo avviene tramite:

- Una rappresentazione della procedura indipendente dal linguaggio e dalla piattaforma, scritta con IDL (Interface Definition Langauge)
- Un formato comune di rappresentazione dei dati usato per la comunicazione

L'IDL per RPC permette di:

- descrivere op. remote che verranno eseguite

- la signature del servizio, ovvero l'identificativo del servizio
- generazione automatica degli stub
- deve permettere di identificare in modo non ambiguo il servizio e di dare definizione astratta dei dati che verranno trasmessi.

8.4.3 Passaggio parametri

Posso avere due tipi di passaggi per parametri:

- per valore: avviene sempre nello stack e chiamato non modifica i valori
- per riferimento.

Esiste un 3° tipo, usato in pochi linguaggi di programmazione, che viene chiamato passaggio dei parametri per copia/ripristino. In questo caso, i dati vengono copiati nello stack del chiamante e ricopiati dopo la chiamata, andando a sovrascrivere valore originale del chiamante. Passaggio per riferimento: problema è che l'indirizzo di memoria è valido solo nel contesto locale in cui è utilizzato. Risolvo simulando passaggio param per riferimento usando 3° meccanismo. CLient stub copia str dati puntata nel messaggio ed invia msg al server stub. Server stub riceve msg con la copia dell'area di memoria, usando quindi sp. di indirizzamento del nodo ricevente. Se avviene modifica, il server stub inserisce nel msg risposta e lato client, il client stub la riporta nella str dati originale del client.
Se str dati contiene dei puntatori? Nesting dello stesso meccanismo.

8.4.4 RPC asincrona

Nel caso di RPC, non ci sono elementi intermedi che si occupa della persistenza dei dati ⇒ comunicazione sempre transiente, chiamata sincrona.
Alcune implementazioni offrono supporto per RPC asincrona, client aspetta solo ack dal middleware che la sua richiesta è stata presa in carico, quindi riprende l'esecuzione; supportata in Go.
RPC asincrona può essere realizzata anche se client si aspetta output, come due operazioni, che sono due RPC separate:

- Una prima RPC per avviare richiesta di procedura remota
- Seconda RPC da server a client per restituire risultato

Client nel mentre può eseguire altre attività

8.4.5 RPC e trasparenza

RPC è veramente trasparente, e rispetto a quali gradi di trasparenza? Non è completamente trasparente, nel caso di Sun RPC non rispetta nemmeno la trasparenza all'accesso, sviluppatore deve sapere che sta facendo RPC e dve

aggiungere parametro in più, che sarà gestore del protocollo di trasporto lato client; ha anche impatto in termini di prestazioni.
Stima di perfomance per procedura locale: sull'ordine di 10 cicli, quindi O(ns).
RPC: anche ammesso che procedura remota non faccia nulla, ha un costo di 5 ordini di grandezza superiore rispetto alla chiamata locale, impatto non trascurabile. Devo considerare l'overhead: context switch, copie, comunicazione tra processi che ha overhead della rete sottostante (ancora maggiore se è TCP).Se sono in WAN, la differenza di prestazioni è notevole.
Se ho failure: possono essere svariati, errori di rete, sul client, sul server etc...; anche aspetti di sicurezza, richieste concorrenti...

8.4.6 Sun RPC

Esempio di prima generazione di RPC, implementazione del middleware per RPC fornita da Sun Microsystems. Implementazione di base per C,largamente usata. Oltre a fornire:

- L'IDL XDR per gestire eterogeneità
- RPCGEN per generare client e server stub
- Binding tramite port mapper
- NFS: Network file system, un file system distribuito.

Rispetto al modello ISO/OSI, stack RPC si pone tra livello 5 e 6: XDR a livello 6 ed RPC a livello 5.
Definizione del programma RPC: file in XRD con estensione .x, avrò due parti:

- parte in cui scrivo definizione dei programmi RPC, specifiche del protocollo RPC per procedure
- Definizioni XDR, dove definisco tipi di dato dei parametri.

Do un nome alla procedura remota, che fa parte di un programma, ogni procedura ha un solo parametro di input ed un solo parametro di uscita, quindi se ho bisogno di più parametri li devo passare in struct. Identificatori sono rappresentati con lettere maiuscole, altro requisito di Sun RPC, inoltre ad ogni procedura viene assegnato un identificativo del numero di procedura e della versione.
Inoltre il programmatore deve sviluppare lato client e server: bisogna implementare logica per effettuare binding e reperire il servizio. Nel server implemento le procedure.
Nel server non c'è main, lo invoca il server stub e questo viene generato automaticamente dal middleware.
Passi di base:

- definisco file .x: tutto l'insieme delle procedure remote offerte dal server. Con comando rpcgen genero file .h, da includere nel client e nel server.

- genero tramite rpcgen gli stub di client e server ed eventuali file di conversione dei dati. Lato server non ho completa trasparenza: devo specificare nome della procedura con versione e stare attento ai parametri di ritorno.
- lato client uso funzione clnt_create(): prende hosto, nome procedura e versione procedura, protocollo di trasporto. Ho anche funzioni per la gestione degli errori che possono avvenire durante la comunicazione.

Codice per la procedura remota non è esattamente uguale a quello per la procedura locale. Non c'è completa trasparenza all'accesso: devo specificare alcuni parametri che permettono ai due proxy (lato client e lato server client e server stub). Passi base:

- Definire file .x
- usando rpcgen, genero client e server stub, ed eventuali funzioni di conversione XDR. Le funzioni XDR vengono prodotte in un file _XDR.c
- sviluppatore scrive client e server
- compila i file e fa il linking dei file oggetto
- servizi registrati presso il port mapper, offerto dal servizio rpcbind.

Output del port mapper: servizi attivi per cui ho n° di programma, n° versione e la porta e protocollo. Nel client stub ho il meccanismo di request-retransmit, perché Sun RCP offre semantica di comunicazione di tipo at least once. clnt_call è funziona di livello più basso: prende in input il gestore di trasporto, gestore per param input/output e valore di timeout per la retx. Modo in cui viene effettuata la conversione è nelle funzioni automatiche xdr_in/xdr_out.
Server stub: c'è il main generato in automatico, che dopo un serie di funzioni di inizializzazione (anche gestione delle socket) registra port mapper. Ho possibilità di chiamare NULL proc. per fare testing. Quando faccio run: client si collega ma ho passato solo host name, numero di porta reperito dal port mapper, client stub lo ottiene invocandolo. Caratteristiche di Sun RPC

- programma può contenere più procedure remote
- ho un unico argomento di input e di output
- gestione del passaggio dei parametri avviene simulando il passaggio per riferimento tramite passaggio per copia/ripristino
- di default SUN RPC non gestisce concorrenza, ho server sequenziale, in alcune implementazioni di Sun RPC posso avere server multithreaded.
- client rimane in attesa sincrona bloccante della risposta da parte del server
- semantica di comunicazione at least once, protocollo di trasporto di default è UDP

XDR supporta conversione dei dati usando formato comunuce di rappresentazione, ci sono funzioni predefinite per tipi atomici.
Binding: procedura deve essere registrata prima di poter essere invocata, registrazione al port mapper delle procedure: server stub specifica n° programma, n° versione e n° procedura.
Invocando port mapper, client stub viene a conoscenza del n° porta del server.
Port mapper permette

- inserimento di servizio
- eliminazione di servizio
- ricerca porta servizio
- ottenere lista dei servizi

Passi salienti: definisco specifica usando XDR, uso rpcgen per generare header client stub e server stub e le routin din conversione dei dati.
Sviluppo client e server e procedura remota. Ora genero con Makefile i file.o e faccio linking e posso runnare.

8.5 Seconda generazione di RPC-Java RMI

Supporto per gli oggetti remoti, ovvero distribuiti. Diversi middleware RPC che offrono supporto di metodi remoti. Analizziamo Java RMI. Java RMi estende supporto RPC al metodo remoto, posso invocare metodo di interfaccia remota a metodo remoto. Java RMI fornisce un insieme di politiche, strumenti e meccanismi per invocare metodo su un host remoto. L'obbiettivo è tenere trasparenza all'accesso in modo da far sembrare invocazione remota e invocazione locale simili; non ho trasparenza completa ma la situazione è migliore di Sun RPC. Anche qui trasparenza alla distribuzione non è completa:

- ho trasparenza alla concorrenza, ma devo specificare che metodo è syncronized
- non ho altri tipi di trasparenze che vedremo dopo

Posso invocare metodo remoto, localmente viene creato un riferimento ad un oggetto remoto che espone quel metodo, e che è attivo su una altro host. Oggetto remoto a sua volta potrà effettuare invocazioni ad altri oggetti locali o remoti.
Differenze rispetto a metodo locale

- affidailità
- durata invocazione del metodo

Voglio separare interfaccia dell'oggetto dal suo comportamento: interfaccia remota permette di specificare i metodi dell'oggetto che possono essere invocati da remoto. Distribuisco solo interfaccia remota, ne ho accesso dallo stub lato client, ma implementazione dei metodi non saranno distribuiti. Uso il proxy

pattern, che consente di ottenre un certo grado ti trasparenza nella gestione delle RPC, ho stub nel client e skeleton nel server (ruoli analoghi a Sun ROC stubs). Quando invoco metodo remoto lato client: binding del client con oggetto server remoto e copia dell'interfaccia del server caricata nello spazio del server. La richiesta di invocazione di metodo remoto arriva all'oggetto remoto e viene trattata dal proxy del client, a differenza di Sun RPC ho un unico ambiente di lavoro, usando de/serializzazione posso gestire eterogeneità nella rappresentazione dei dati: nel caso di Java RMI basterà questo.
Stub e skeleton fanno da client stub e server stub e quindi nascono a lvl applicativo la natura distribuita dell'applicazione. Stub è proxy locale sul clien, che espone la stessa interfaccia dell'oggetto remoto, lato server ho lo skeleton che riceve le invocazioni fatti dal client e le realizza effettuando chiamata del metodo.
Sono generati automaticamente e non è necessario usare comando ad hoc.
Problema di gestire eterogeneità è risolto con de/serializzazione: grazie all'uso del bytecode non serve marshaling ed unmarshaling. COn writeObject serializzo su uno stream di output, con readObject ricostruisco copia dell'oggetto originale su stream di input. Non ho codice visibile di stub e skeleton, useranno de/serializzazione per gestire parametri di I/O del metodo remoto. Limitazione nel caso di JavaRMI per definizione di oggetti remoti: posso usare solo oggetti serializzabili, ovvero che implementano interfaccia serializable. Alla deserializzazione userò .Class che deve essere accessibile. Oggetto convertito in flusso di byte che lato ricezione verrà ricostruito nell'oggetto corrispondente.Importante che stub e skeleton usino schemi di compattazione per

- ridurre banda occupata nella comunicazione
- ridurre memoria occupata
- ridurre latenza

Differenze tra marshaling e serializzazione diventano evidenti nel caso degli oggetti: la serializzazione si basa sul codebase presente a destinazione, non bisogna gestire le referenze, però c'è problema di gestione degli oggetti ricorsivi.

8.5.1 Interazione tra stub e skeleton

Passi della comunicazione

- client deve ottenere istanza di stub, ovvero copia di interfaccia remota. La ottiene con componente intermedio che è l'RMI registry.
- client invoca metodi sullo stub, sintassi invocazione remota è identica a quella locale.
- lo stub ricevuta invocazione del metodo effettua serializzazione delle info (id metodo e parametri) incapsula in un messaggio e invia messaggio allo skeleton

- skeleton riceve messaggio, deserializza ed invoca chiamata locale
- riceve param ritorno, serializza, incapsula nel messagigo ed invia allo stub
- stub riceve messaggio,effettua deserializzazione e restituisce al client.

RMI registry è un binder per Java RMI, consente al server di registrare l'oggetto remoto e al client di recuperarne lo stub. RMI registry identificato con URL che inizia con rmi, contiene hostname, n° porta e il nome dell'oggetto remoto. Non c'è trasparenza all'ubicazione (devo sapere hostname, stesso vale per Sun RPC), non c'è gestione di sicurezza.
Passi essenziali

- realizzare componenti lato server, devo definire interfaccia e la sua implementazione con classe apposita.
- lato client devo ottenere riferimento all'oggetto remoto (lo stub), tramite RMI registry con invocazione del metodo lookup
- interfaccia deve essere public, in modo che estenda Remote e deve sollevare eccezione RemoteException.
- La classe deve estendere UnicastRemoteObject, perché viene definito un solo riferimento ad un oggetto remoto, non ho trasparenza alla replicazione.
- devo scrivere codice del server, che istanzierà oggetto remoto, registrarlo presso RMI registry. rebind(): permette di sostituire associazione già esistente rispetto a bind().

Dopo aver sviluppato il codice: compilo le classi, attivo RMI registry ed avvio client e server. Per avviare RMI registry posso farlo:

- da riga di comando, comportamento standard
- nel codice del server, registry locale per motivi di sicurezza.

Interfaccia deve estendere Remote, devo gestire RemoteException in modo da lanciare eccezione. Implementazione dei metodi non completamente trasparente. Metodo può avere 1 metodo di output, 0 o più parametro di input, passaggio può avvenire per valore o per riferimento:

- Per valore se sono oggetti primitivi che implementano Serializable, serializzazione/deserializzazione ad opera di stub/skeleton
- per riferimento, se sono oggetti Remote.

Nel caso di Java RMI non serve chiamare il compilatore ad hoc(no comando rpcgen come in Sun RPC).
Sicurezza: ho appl. distribuita, si pongono diversi aspetti di sicurezza. Se client invia msg, sono sicuro che invocazione del metodo remoto sia rispetto al server

corretto o non sia un impostore?
Server accetta messaggi solo da client legittimi? Messaggi sniffati da altri processi o intercettati e modificati Protocollo RPC può essere soggetto a reply attack.

8.5.2 Passaggio di parametri

Tipi primitivi passati per valore, mentre classi passate per riferimento. Ho visto problema del passaggio per riferimento, in Java RMI:

- Per valore se tratto tipi primitivi o tipi serializzabili. In generale avviene per tutti gli oggetti la cui locazione non è necessaria per lo stato, quello che si fa è de/serializzare l'istanza dell'oggetto per creare un'istanza remota
- Per riferimento: oggetti la cui funzione è legata alla località di esecuzione (server). Viene serializzato lo stub, creato automaticamente a partire dalla classe dello stub.
  Ogni istanza di stub identifica l'oggetto remoto al quale si riferisce attraverso un id univoco rispetto alla JVM dell'oggetto remoto.

8.5.3 Concorrenza sugli oggetti remoti

Metodi remoti possono essere invocati in modo concorrente da più client e questo può porre problemi nel caso di metodi statefull.Implementazione deve essere thread-safe se c'è stato: va usata la keyword syncronized per il metodo (es: incremento di un contatore)

8.5.4 Distributed garbage collection

Ho oggetti remoti che non vengono più utilizzati dai client, quindi è inutile tenere memoria allocata. Devo effettuare operazione di garbage collection, serevr deve sapere quanto rifermenti attivi ci sono per l'oggetto remoto e quindi quanti client stub lo stanno usando. Client possono subire un crash o possono esserci problemi di rete.
Richiede coordinazione fra client e server, ho quindi limiti nella scalabilità. Funzione del meccanismo: in locale,, Java mantiene il numero di riferimenti per un oggetto e lo schedula per la de allocazione quando contatore è 0. Meccanismo distribuito simile: meccanismo basato su lease, idea è che il server delega l'operazione di mantenere i riferimenti attivi al client in modo da distribuire il carico sui client. Ho due operazioni:

- Dirty: periodicamente JVM del client manda call dirty al server se l'oggetto è in utilizzo sul client, che viene rinfrescata entro un timeout che è il lease assegnato dal server.
- Clean: nel momento in cui JVM locale al client non sta più usando oggetto remoto, usa operazione di clean per indicare che non ci sono più riferimenti attivi all'oggetto.

Cancellazione oggetto avviene se non riceve dirty o clean prima del leasing. È delegato agli stub di mantenere l'informazione che l'oggetto è in utilizzo.

8.6 Esempi Java RMI

8.6.1 Echo server

Server: ho la calsse che implementa il server, che deve estendere UnicastRemoteObject: nel costruttore ho metodo super(), che esegue le inizializzazioni necessarie affinché server venga correttamente lanciato e rimanga in attesa delle richeiste del client. Implemento il metodo remoto getEcho() che avevo definito nell'interfaccia.
Nel main: pubblico il servizio, devo esporlo esternamente affinché i client possano invocarlo, faccio bind()/rebind() del mio server sull'RMI registry (faccio la new della calsse server e ne faccio bind). Uso nome logico, che sarà usato dal client per chiamare il metodo.
Lato client: invoco metodo, passo la stringa al server mediante invocazione di metodo remoto. La chiamata che il client effettua è una chiamata singola bloccante.

8.6.2 Compute engine

Server riceve task da client per eseguirli e restituire il risultato.
Penso ad un task oneroso dal punto di vista computazionale. Task deve implementare Serializable, viene eseguito dal server che restituisce il risultato al server.
Due interfacce:

- Interfaccia che permettere di definire al client di definire il task
- Interfaccia per eseguire il task sul server

8.7 Come fornire trasparenza

Come fornire ad esempio supporto alla trasparenza a replicazione in Java RMI/Sun RPC. Load balancer: intermediari tra client e server replicati: uso oggetto/procedura remota che faccia ruolo di load balancer. Questo è sulle spalle dello sviluppatore, per questo manca trasparenza alla replicazione ma è possibile implementare distribuzione delle RPC/RMI su più server. Client si collega la proxy server, che sceglie a chi inoltrare la richiesta a seconda del carico: si comporta da server per i client e da client per i server.
In RMI registry il client che effettua metodo remoto rimane bloccato in attesa della risposta del server. Suppongo di volere chiamata async a livello di codice. Cosa devo usare per questo meccanismo di callback: lato server devo sapere quali sono i client interessati all'elaborazione asincrona, ovvero serve meccanismo di registrazione, metodo esposto. Lato client serve oggetto remoto tramite il quale il server possa comunicare il risultato della computazione effettuata.

8.8 Confronto tra Sun RPC e Java RMI

Sun RPC:

- solo 1 param di input, client nell'invocare procedura remota deve passare altro param che è gestore di trasporto.
- Client deve specificare n° versione della procedura chiamata, devo appendere lato server al nome della procedura _svc. Trasparenza all'accesso incompleta.

- Trasparenza all'ubicazione: manca, client deve conoscere hostname sul quale viene offerto il servizio remoto, avviene mediante il port mapper.
- In Sun RPC posso richiedere operazioni e funzioni
- Comunicazione è sincrona ed asincrona e la
- Semantica è in uquesto middleware at least once (o at most once)
- Implementato timeout per la ritrasmissione e gestione di errori con apposite funzioni
- Biding eseguito dal port mapper con rpcbind(), port mapper permettere al client stub di conoscere la porta del server.
- XDR come IDL, generazione automatica di client e server stub.
- Passaggio parametri per copia/ripristino.
- Ci sono varie estensioni di SunRPC

Java RMI:

- Visione ad oggetti
- Maggiore trasparenza all'accesso: metodo remoto chiamato con la stessa sintassi del metodo locale
- Non c'è trasparenza all'ubicazione
- Distribuzione non completamente trasparente: devo tener conto che passo param primitivo o oggetto remoto.
- Entità richiedibili sono metodi di oggetti mediante interfacce remote
- Comunicazione sincrona, semantica at most once.
- Gestite le eccezioni remote
- Binding del server con RMI-registry
- Stub e skeleton generati in modo automatico, non serve precompilatore (no rpcgen()).
- Passaggio parametri per valore o per riferimento per oggetti remoti.

9 Go

Linguaggio C-like (il C del XXI secolo), eredita diversi costrutti:

- sintassi
- statement per controllo di flusso
- tipi di dato base
- puntatori
- compilazione efficiente

Introduce diverse facilites

- GO introduce facilities per la concorrenza nuove ed efficienti
- Approccio flessibile per astrazione dati
- Garbage collection

Go permette di concentrasi sulla logica del SD:

- Buon supporto per RPC
- Buon supporto alla concorrenza
- Garbage-collected
- type safe

Go permette sviluppo di app colud native, libreria Go Cloud, obbiettivi:

- Permette ai developer di fare deploy rapido su diverse combinazioni di cloud provider
- Utilizzo dei vari servizi cloud, come ad esempio S3.

Aspetti nuovi: restituzione di più parametri di ritorno, if vuole sempre {} anche se c'è solo una istruzione. Defer: ritardo un istruzione e la eseguo quando è stato completato tutto il codice intorno all'istruzione. Vantaggio è che riduco i potenziali errori di programmazione legato al fatto che dimentico di chiudere canale aperto. Reference: scaricare compilatore da goolang.org (attenzione alla variabile gopath).
Sito Golang ha playgound (tour.gooland.com).
caratteristiche:
mancano ; a differenza del C ed i return. Programmi sono composti in package. Programma inizia con dichiarazione package, seguita da dichiarazione degli import, quindi package importati.
Tool per run è standard per effettuare fetch, build e istallazione programmi.
Comandi fondamentali: go run <codice sorgente>, go build per il binario.

9.1 Package

Dove definisco codice Go. I programmi sono avviati dal package main, il case determina l'import o meno: se ha lettera maiuscola può essere esportata al di fuori.
Gli import permettono di identificare le librerie utilizzate, se ho più import non devo scriverne una per ciascuna riga ma posso fattorizzare tra parentesi ().

9.2 Funzioni

Il tipo del parametro di ritorno è dichiarato alla fine della funzione, anche i tipi sono dichiarati dopo.
Funzione può restituire qualsiasi numero di parametri di ritorno.
Per assegnazione uso :=
Statement var permette di dichiarare una lista di variabili, se dichiaro variabile con un certo valore il tipo può essere desunto.

9.3 For, while, etc...

Ciclo for ha 3 condizioni, se uso solo la prima è come un while (no parentesi). for {} = while(1).
Se variabili non inizializzate vengono settate a valore di default (esempio: stringhe a stringa vuota). If-else:
sempre necessarie le graffe, cosa nuova è che nel costrutto if-else, l'else deve essere attaccato alla chiusura dell'if. È possibile combinare più statement if-else con la sequenza if-else, if-else....
Costrutto switch, che termina quando un case ha successo, quindi non ha bisogno del break (dato automaticamente); non ci sono limitazioni sulla condizione.
Defer: meccanismo nuovo, che permette l'esecuzione di una funzione quando termina il codice che la circonda. Argomento valutato immediatamente, ma la chiamata non è eseguita finché codice che circonda la funzione non termina, funzione messa in stack e le funzioni vengono estratte in ordine LIFO.

9.4 Puntatori, struct, array etc..

Soliti operatori di de/referenziazione & e *; gestione automatica dello spazio di memoria.
Struct come in C, per gli array si possono usare slices: costrutto di dimensione variabile che permette di vedere solo una porzione degli elementi di un array: array[low:hi]: l'hi viene escluso. La slice è una sezione dell'array sottostante, è una vista logica, quindi le modifiche sono effettive sull'array. Funzione len() permette di conoscere la lunghezza di un slice. Capacità della slice si definisce con cap(), ed è capacità complessiva dell'array sottostante a partire dal primo elemento. Slice è di dimensione variabile, può essere creata con make, modifiche a run time con append ma non si può eccedere la dimensione dell'array sottostante.

Mappe: altro tipo di dato composto, tradizionali mappe chiave:valore. Si dichiarano con keyword map[K]V, dove K indica tipo della chiave e V tipo del valore; make crea la mappa.

- insert update: m[key] = value
- m[key] per retrieve
- delete(m, key) cancella chiave
- elem, ok = m[key] è un test per vedere se elemento è presente (ok avrà valore di errore in caso non c'è l'elemento).

Range: costrutto che permette di iterare su un ampio insieme di elementi di strutture dati: elementi di array, di slice, mappe etc...
_ : indica che non mi interessa un indice in for, altrimenti se poi non uso l'indice dichiarato ottengo un errore di compilazione.

9.5 Aspetti di OO

Ci sono aspetti di orientamento agli oggetti, ma non c'è concetto di classe. Tuttavia supporta concetto di metodi definiti su struct.
Metodo: una funct che ha un metodo in più chiamato receiver e che è posizionato prima del nome della funzione.
Il recevier appare nella lista degli argomenti. È possibile anche definire le interfacce, un tipo di interfaccia è una signature di metodi, a cui do nomi e tipi di ritorno.

9.6 Concorrenza in Go

goroutine: thread leggero gestito dal supporto a run time di Go. Si lancia anteponendo alla funzione "go". Le goroutines sono thread leggere (implementazione leggera in termini di SO), che condividono lo spazio di indirizzamento, quindi bisogna sincronizzare l'accesso alla memoria.
Canale: meccanismo di comunicazione che permette a due goroutine di comunicare fra loro. È un canale che permette invio/ricezioni di valori. Implementato come coda thread-safe gestita da Go, strumento di comunicazione molto potente che permette di nascondere molti dettagli relativi alla comunicazione tra thread. IL canale internamente usa concetti di mutex e semaforo per gestire la concorrenza tra thread. Canale non è esclusivamente ad utilizzo di un thread mittente e di un thread ricevente, ma da molteplici routine: utile per implementare notifiche, multiplexing etc...
Ricordare che solo una goroutine può chiudere il canale e nessuno può più inviare. Per definire canale in Go si usa l'operatore di canale $< -$: ch $< -$ v (invia v sul canale ch), i dati affluiscono in direzione della freccia. v:= $< -$ ch, inizializza v col dato ricevuto da ch. Per inizializzare canale si usa costrutto make (chan <tipo dato>. Nell'utilizzo del canale, visto che è una coda thread

safe, le goroutine non devono usare meccanismi di sync. espliciti.
Canali possono avere un buffer, posso assegnare la dimensione come parametro all'atto dell'inizializzazione con make. Vantaggio di usare canale bufferizzato si blocca solo se canale è pieno (se non specifico nulla, canale ha dimensione pari ad 1).
Receivers possono testare se canale è aperto o chiuso: passo un secondo parametro di ok: v, ok := <− ch, riceverò false se il canale è chiuso, anche possibile usare range per ricevere valori continuativamente.
Possibile usare select per attenere messaggi fra molteplici canali: costrutto prevede di definire tanti case quanti sono i canali.

9.7 Gestione errori

Go usa codici di errore per determinare stati anormali. Per convenzione è l'ultimo valore restituito da una funzione. nil: nessun errore.
C'è interfaccia built-in per errore che è Error, ha metodo Erro() string: ottengo stringa corrispondente all'errore verificato.

9.8 RPC in Go

Nel package net/rpc.
TCP ed HTTP come protocolli di trasporto supportati, realizzazione di microservizi in Go che usano RPC in Go si sta diffondendo.
Ci sono dei vincoli:

- metodo rpc può avere solo due argomenti in input, il secondo serve proprio per la gestione stessa della RPC.
- errore sempre ritornato

func(t *T) MathodName(argType T1, replyType *T2) error.
per marshaling(encode)/unmarshaling(decode) c'è package gob, package jsonrpc o framework grpc.
Lato server:

- Devo registrare metodo esposto come RPC. SI usa Register o RegisterName: pubblico metodo sull'interfaccia del server RPC di default e permetterà ai client di chiamare i metodi così esporti.
- Register ha come param interfaccia (collezione di metodi), possibile associare un nome con metodo RegisterName.
- Si usa Listen per annunciare l'indirizzo locale di rete.
- Usa Accept per ricevere richieste del servizio, è bloccante quindi se non voglio che il server rimanga bloccato uso keywork go per far si che venga eseguita in un thread.

Lato client:

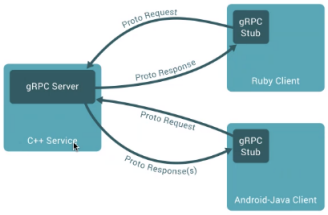
- Dial: si connette a server RPC, rida puntatore per le successive chiamate RPC o nil se c'è errore (DialHTTP per usare HTTP).
- Call è chiamata sincrona, oppure Go per chiamata asincrona: nel secondo caso arriverà un messaggio per confermare ricezione della risposta.

9.9 gRPC

Framework di nuova generazione, molto usato per realizzazione di applicazioni basate sullo stile architetturale a micro-servizi.
Tipi di utilizzo:

- connettività efficiente per servizi poliglotti, ovvero sapplicazioni in cui servizi sono scritti in diversi linguaggi di programmazione. Supporto a C++, Python, Go, Java etc...
- connessione di device mobili, browsers a servizi di back-end
- generazione efficiente di librerie lato client

Tra gli utenti più noti di gRPC c'è Netflix, primo big provider ad usare stile architetturale a micro-servizi (applicazioni composte da migliaia di micro-services).
Da un punto di vista di RPC, gRPC sfrutta il proxy pattern, quindi ci saranno stub che interagiscono fra di loro
HTTP2: è più efficiente dal punto di vista delle connessioni TCP, ogni messaggio è diviso in frame (unità minima), è un protocollo binario e c'è layer posto sopra lvl4 che fa framing del messaggio lvl5 e comprime binary stream, come anche compressione header. Questo rende trasmissione molto più efficiente.



Particolarità è nel "Proto": gRPC utilizza i protocol buffer come IDL: protocol buffer è IDL sviluppato da Google, molto più efficiente di XML e di Json.Permette di avere scambio binario di dati. Usa largamente HTTP 2.0, caratteriscihe struttate da gRPC:

- supporto a framing binario
- supporto a header compression
- supporto al multiplexing: permetto scambio di più msg da parte di client/server, in modo da portare a regime la connessione TCP.

9.9.1 Protcol buffer

Gestisce in maniera efficiente scambio di strutture dati scambiandole in formato binario: ho file .proto in cui definisco le mie strutture, avrò message nome_messaggio con i campi e i rispettivi tipi di dato.
Uso un compiler per l'IDL, che è protoc, che permette di generare classi per l'accesso ai dati nel linguaggio preferito.

10 Comunicazione orientata ai messaggi

RPC ed RMI migliroano la trasparenza della distribuzione, ma sono sincroni:

- sincronia nel tempo
- sincronia nello psazio
- funzionalità e comunicazione sono accoppiate

Considero comunicazione orientata ai messaggi: avrò che funzionalità di client e server sono disaccoppiate dalla comunicazione di messaggi.

10.0.1 MPI

Libreria per lo scambio di messaggi tra processi in esecuzione su diversi nodi (molto usata nel calcolo parallelo), specifica una serie di primitive per la comunicazione di messaggi tra processi, primite di due classi:

- punto-punto: invio ricezione tra due specifici processi
- comunicazione collettiva, ovvero verso tutti i nodi che fanno parte di un comunicatore

Principali primitive sono classiche send e receive (bloccante) ma anche comunicazione bloccante che implementa dei buffer.
Esistono diversi supporto per la comunicazione sia bloccante che non bloccante.

10.1 Sistemi publish subscribe e sistemi a code di messaggi

Fin ora visto supporto per RPC, che migliora la trasparenza rispetto alla programmazione di socket, ma non è ancora possibile avere asincronia tra le entità che interagiscono. Cerco di migliorare la flessibilità dell'applicazione.

10.1.1 Middleware orientato ai messaggi

Middleware di comunicazione che permette di realizzare comunicazione persistente. Permette di avere:

- disaccoppiamento spaziale e temporale
- componenti possono essere anonimi
- a seconda del framework può essere supportata disaccoppiamento di sincronia
- usati tipicamente in applicazioni serverless e a micro-servizi: applicazioni cloud cercano di passare da una struttura monolitica ad una struttura con vari componenti che realizzano le varie funzionalità e sono laicamente accoppiate fra loro

Due pattern principali:

- Coda di messaggi sistemi a code di messaggi
- Publish/subscribe: sistemi publish/subscribe

Alcuni sistemi soddisfano entrami i pattern, ma è importante capire le differenze fra i due.

10.1.2 Pattern a coda di messaggi

Ho a disposizione una coda che permette di memorizzare in modo persistente dei messaggi e rimangono lì finché sono prelevati ed eventualmente cancellati dai destinatari (dipende dalla semantica di comunicazione). Elemento distintivo è che un singolo messaggio viene consegnato una sola volta ad un singolo consumatore, realizzo pattern di comunicazione 1-a-1 (point-to-point), possiameo mettere in comunicazione un producer ed un consumer andandoli a disaccoppiare temporale (non necessariamente c'è disaccoppiamento spaziale o di sincronia).
Ho il producer A ed il consumer B: A manda un messaggio a B, per farlo consumerà risorse e farà locking di aree di memoria. Una volta che messaggio è stato preso in carico dalla coda producer rilascia le risorse precedentemente acquisite. Consumer prende messaggio, processa e manda risposta. Il messaggio di risposta viaggia su una coda differente rispetto a quella del messaggio da A a B.
Nel momento in cui A manda il messaggio, può anche scollegarsi dal sistema poiché il messaggio sarà comunque memorizzato in maniera persistente dalla queue.
Tipicamente l'API della coda è:

- put: appende msg ad una coda
- get: ricezione bloccante, che blocca finché coda non è vuota ed a quel punto consumer preleva messaggio che viene cancellato dalla coda. Può prelevare primo messaggio dalla coda con politica di tipo FIFO, politiche permettono altri tipi di lettura (es matching di alcuni parametri)

- poll: non bloccante, viene ricercato messaggio nella coda e viene rimosso o il primo in assoluto o il primo che matcha il pattern di ricerca.
- notify: permette di realizzare comunicazione asincrona, viene installato un handler e funzione di callback chiamata autenticamente quando viene messo in coda un messaggio che corrisponde alle specifiche indicate dal consumer.

10.1.3 Publish/subscribe pattern

Messaggio inviabile uno-a-molti, componenti possono pubblicare messaggi asincroni o dirsi interessati a ricevere dei messaggi facendo subscription.
Subscriber possono essere interessati a topic con o senza filtri, le sottoscrizioni sono raccolte da un eventi dispatcher, responsabile di mandare tutti gli eventi ai subscriber interessati. Alto grado di disaccoppiamento tra i componenti. Differenza fondamentale tra i due è che il secondo è come una generalizzazione del primo, in quanto permette di effettuare comunicazione one-to-many.
API tipica:

- publish(event): pubblica evento, può essere di ogni tipo supportato dalla specifica implementazione del sistema, può contenere anche metadati
- subscribe(filter expr, notify:cb, expiry) $\rightarrow$ sub handle: possibile specificare filtri per gli eventi a cui un subscriber è interessato. Possibile specificare una scadenza per la sottoscrizione ed un riferimento ad una funzione di notify, invocata per la gestione dell'evenot una volta che viene consegnato
- unsubscribe(sun handle): cancella sottoscrizione
- notify(sub handle, event): manda notifiche per un evento corrispondente ai filtri specificati dai subscrivere per un determinato evento.

10.1.4 Funzionalità MOM

C'è bisogno di una componente di routing, per instradare i messaggi verso i subscriber o i consumer ed una componente di trasformazione, che permetta di applicare trasformazioni ai messaggi veicolati dal middleware. Devo sempre avere in mente che i componenti che comunicano possono essere eterogenei, ad esempio nella rappresentazione dei dati.

10.1.5 Semantiche di comunicazione

Diversi pattern per la semantica di comunicazione: si distingue tra ricezione e consegna, ricezione si suppone lato recevier a livello di trasporto, consegna si intende a livello applicativo

- at least once: Almeno una consegna di messaggio. meccanismo principale che MOM deve avere per garantire successo della ricezione è retx. Sender

non si vede arrivare ack, quindi allo scadere di un TO lo rimanda. Può accadere che destinatario abbia ricevuto più volte lo stesso messaggio, quindi se uso MOM con questa semantica di comunicazione il messaggio lato recevier può essere duplicato quindi il componente deve essere tollerante alla ricezione dei messaggi duplicati.

- exactly once: esattamente una consegna. Devo avere filtro di messaggi duplicati, in modo che non siano consegnati al destinatario (messaggi hanno id univoco, usato per filtrare), ma non basta. Serve anche tolleranza ai guasti (ci sono middleware che non garantiscono tolleranza ai guasti per tutte le situazioni).

Ci sono anche altre due semantiche di comunicazione:

- Consegna basata su transazione: il sistema MOM assicura che i messaggi vengano consegnati dalla coda dei messaggi soltanto quando sono stati effettivamente ricevuti dal destinatario con successo. Incapsulo transazione ACID nel messaggio: quando recevier ha ricevuto messaggio, invia ack alla coda di messaggi e solo alla ricezione dell'ack il messaggio viene cancellato dalla coda.
  Semantica più forte dell' at least once: lì l'ack viene mandato se messaggio viene ricevuto ma non ancora processato, qui invece è cancellato dopo il processamento. Invio e ricezione fanno parte di una transazione con proprietà ACID.

- Consegna basata su timeout: messaggio cancellato dalla coda dai messaggi quando è stato correttamente ricevuto almeno una volta (diverso da sopra). Producer invia messaggio, che diviene visibile nella coda. Può essere quindi consumato da un recevier, lo preleva ed il messaggio non è subito cancellato dalla coda ma viene impostato come invisibile: è nella coda ma non può essere prelevato da altri consumer (impostato così per un timeout). Una volta che messaggio è stato processato invia ack alla coda, se timeout non è scattato e l'ack viene ricevuto allora messaggio viene cancellato. Se ack è ricevuto dopo il TO on no viene ricevuto, allo scadere del TO messaggio torna ad essere visibile e potrà essere ricevuto da un altro consumer. Introduco maggiore tolleranza ai guasti. At lest once perché l'ack potrebbe arrivare successivamente al timeout, quindi potrebbe essere letto due volte dallo stesso recevier, o da un altro recevier.

10.1.6 Routing dei messaggi

Modello generale: rounitng avviene su diverse code, ciascuna gestita da un manager delle code, supporta l'idea di overlay network formata dai gestori delle code. L'overlay network viene usata per fare il routing dei messaggi, usando routing table che sono gestite e salvate dai queue manager (a seconda della specifica implementazione del sistema di MOM). La rete di overlay deve essere manutenuta nel tempo: può esserci manutenzione manuale oppure dinamica, più complicato nel secondo caso in quanto serve mapping dinamico tra le code e la loro

locazione.
Oltre al routing, altra componente fondamentale è la capacità di trasformare i messaggi in modo da permettere eterogeneità dei dati, soluzione qui è del message broker: componente del sistema che si occupa di gestire l'aspetto di eterogeneità dei dati.
Si occupa di:

- convertite formato messaggi in input in formato atteso in output
- possibile specificare regole per la trasformazione dei messaggi
- realizzato in maniera distribuita per far si che sia scalabile e robusto

Molti framework che realizzano sistema a code di messaggi, anche servizi cloud per sistemi MOM.

10.1.7 Casi d'uso di MOM

1. Utilizzo base: producer che produce messaggio ed un consumer che lo consuma, classico uso del pattern a coda di messaggi.
2. Uso un sistema a coda di messaggi per distribuire un compito tra molteplici worker. Invece di contattarli direttamente, la fa via coda di messaggi: master produce il task che deve essere eseguito dai worker, che sono in attesa sulla coda. Uno di questi preleva il task e lo esegue.
   Possibile distribuire in modo round robin il carico di lavoro tra i worker
3. Code di messaggi per realizzare pattern publish/subscribe: nel caso di Rabbit MQ, il pattern pub/sub viene realizzato con una coda per ogni possibile consumer.Siccome messaggio va inviato presso molteplici consumer, c'è invio sus tutte le code a cui sono sottoscritti i consumer.
4. Ricezione selettiva di messaggi: producer invia messaggio ad un exchange, che seleziona la coda in base al contenuto del messaggio.
5. Uso di MQ con pattern stile RPC, ma asincrono: publisher è client, consumer è server. HO due code per gestire messaggi di richieste/risposte: client invia richiesta, server elabora e manda risposta nella coda rispettiva. Vantaggio rispetto ad RPC è che comunicazione diventa persistente.

esempio: Rabbit MQ, è sistema a code di messaggi che usa dei broker per gestire le code di messaggi, ciascuna coda può essere gestita da più broker, in AMQP (uno dei protocolli di messaggistica offerti da RabbitMP) ci sono punti di exchange che mandano i messaggi verso la giusta coda.
esempio 1:

- producer: uso Dial per chiamare la coda di messaggi, gestendo eventuali errori se server di RabbitMQ non sia up&running. Apro canale di comunicazione con Channel() e dichiaro le caratteristiche della coda:

  - nome: il nome deve corrispondere a quello usato nel recevier
  - durable: da la persistenza della coda rispetto a crash del server. Se false: se viene inviato messaggio e server crasha, messaggio viene perso; se è true accade che se il server è riuscito a fare flushing del messaggio su disco prima del crash, al riavvio è in grado di recuperare lo stato dei messaggi.
  - delete when unused: cancello messaggio se non prelevato
  - exclusive: prelevabile da un solo destinatario
  - altri param...

- invio del messaggio, API di RabbitMQ espone publish per invio del messaggio
- lato recevier: in modo speculare, recevier si collega al server di RabbitMQ con la Dial() del package AMQP.
- preleva messaggio

Possibile applicare al borker una politica di scelta round-robin, in modo che ogni task sia assegnato ad ognuno dei consumer collegati. Versione 2 del consumer: modifica al parametro di consume, in cui prima il parametro di auto-ack era settato a True. Lo stetto false in modo che l'ack ed invio un ack esplicito una volta che il consumer ha effettuato il processamento esplicito. Passo da una semantica at most once (nel caso precedente il consumer non ha completato al processamento) ad una at least once.

10.1.8 IBM MQ

I messaggi sono trasferiti tra le code, e trasferiti poi sul canale richiesto. Ad ogni end-point del canale c'è un message channel agent. Questi ultimi sono responsabili di:

- Tirare su il canale
- Mandare/ricevere messaggi
- Fare encryption dei messaggi.

Canali di comunicazione sono uni-direzionali, se serve comunicazione bidirezionale serve installare due canali. Sono gestiti dagli agenti, che vengono avviati automaticamente quando arriva un messaggio da gestire sul canale.IBM MQ mette a disposizione la possibilità di creare rete di overlay frai queue manager.Le rotte sono settate manualmente, routing avviene con nomi logici assegnati alle code.

10.1.9 Amazon SQS

I messaggi all'interno delle code hanno durata limitata (configurabile), vengono cancellati solo se sono stati ricevuti correttamente. Permettono flessibilità in quanto è possibile avere i diversi nodi dell'applicazione scritti con linguaggi diversi.Semantica di comunicazione può essere timeout based, se messaggi non viene ricevuto entro lo scadere del timeout di visibilità, il messaggio torna ad essere visibile.
L'API che viene esposta:

- Creazione, cancellazione, listato della coda
- Invio/ricezione dei messaggi: taglia max del messaggio è 256KB, per inviare messaggi più grossi, posso mettere il payload su un altro servizio (es S3) e mando solo link al bucket S3 sulla coda.
- Cancellazione del messaggio
- Controllo del valore del periodo di visibilità
- Controllo impostazioni della coda

esempio: applicazione cloud per processamento/modifica di foto. 2 macro-componenti: front-end verso l'utente, parte back-end. Due code: una che permette di inviare le foto alla parte di back-end dal font-end ed una che vada nel verso opposto (rimandando la foto trans-codificata). Siccome foto pesano, nel messaggio metto URL della foto memorizzata su S3. Posso pensare di replicare su EC2 il componente di processamento, quindi posso usare la coda per fare load balancing tra le varie istanze del server di processamento. Siccome gestisco la coda con semantica basata su time-out, posso essere tollerante ai guasti: se server subisce un crash mentre sta processando, allo scadere del periodo di visibilità la foto torna disponibile e può essere processata da un altro server.

10.2 Apache Kafka

Sistema publish-subscribe che può funzionare anche come sistema a code di messaggi, uno dei progetti più popolari dell'Apache Software Foundation.
Originariamente sviluppato da LinkedIn, usato da Netflix, Uber etc... Scritto in Scala, scalabile orizzontalmente, tollerante ai guasti con alto ingestion throughput.
Da un punto di vista architetturale, Kafka è composto da un cluster di broker, che mettono a disposizione un log distribuito per la gestione dei dati. Espone a producer/consumer die topic: un topic può avere 1 procuder che pubblica su un topic e 0-1-più consumer che si sono sottoscritti al topic e ricevono messaggi da quel topic; i broker gestiscono i topic.
Il topic è una categoria nella quale viene pubblicato il messaggio, in ogni topic il cluster Kafka contiene un log partizionato che è una struttura dati (non il

classico file log) che permette di memorizzare una sequenza di record ordinati temporalmente e le scritture avvengono solo in modalità append, ovvero solo alla fine del log. Log viene mantenuto totalmente ordinato (rel. d'ordine totale). Topic deve essere accessibile da parte di più producer e consumer contemporaneamente per favorire la scalabilità, quindi viene suddiviso in partizioni, che definisce la quantità minima per il parallelismo sul topic. Ogni partizione, per migliorare la scalabilità e la tolleranza ai guasti + replicata tra i molteplici broker del cluster.
I producer pubblicano i record sulle partizioni del topic, all'atto della pubblicazione la partizione viene scelta con politica round-robin o con partizionamento basato su chiave, ade esempio in base ad un hash calcolato sulla base del messaggio.
Consumer possono leggere le varie partizioni, ciascuna partizione è ordinata,numerata ed immutabile su cui avvengono solo operazioni di append per l'inserimento. Ciascun record è numerato con un sequence number, detto offset: nell'effettuare una lettura i consumer specificano l'offset del record da leggere.
Le partizioni sono suddivise tra molteplici broker, questo per scalabilità così che più producer/consumer in modo concorrente possano agire sullo stesso topic e sulle sue partizioni. Ciascuna partizione è replicata per tolleranza ai guasti ed aumento del throughput. Ciascuna partizione può essere replicata, per le diverse repliche della partizione viene identificato un broker (server Kafka) che svolge ruolo di leader e 0 o più follower. Leader gestisce operazioni di lettura/scrittura, mentre i follower sono delle copie di backup della partizione del leader e possono subentrare tramite meccanismo di elezione se server che svolge ruolo di leader subisce fallimento.
Kafka distribuisce tra i diversi broker il compito di leader tra le partizioni: ciascun leader sarà leader per certe partizioni e follower per altre. I broker poggiano su un servizio che è Apache ZooKepeer.
Producer: sorgenti dei dati in Kafka, pubblicano messaggi sui topic di loro scelta, invio tra producer e server Kafka è diretto, invio al leader della partizione. I producer inoltre sono responsabili di scegliere a quale partizione viene assegnato un determinato record. Sono anche responsabili di scegliere quale record assegnare a quale partizione del topic: può avvenire con scelta basata su chiave, se ad esempio lo user id è la chiave, tutti i dati dell'utente saranno mandati sulla stessa partizione. Kafka copre anche settore dei data storage, non solo offre API di messaggistica, ma anche processamento ed analisi dati, rappresenta come sistema di messagistica uno dei componenti fondamentali di un'architettura distribuita per analisi dei dati in quanto viene usato per la fase di data injection (iniezione dei dati nel componente che farà poi processamento). All'atto della creazione del tipic specifico il nome (univoco), in numero di partizioni del topic ed il grado di replicazione di ciascuna partizione del topic.
Kafka usa modello basato su pull: è il consumer che prende i dati prodotti. Il modello push è invece usato su RabbitMQ: in questo caso i broker spingono attivamente i messaggi verso i consumer.Questo complica la realizzazione del broker,deve poter gestire diversi tipi di consumer che possono avere diverse capacità computazionali e quindi ricevere i dati a ratio differente. Inoltre decidere

se mandare i dati subito o accumularli e mandare un batch.
Nel modello pull, è consumer parte attiva nel reperimento dei messaggi, deve tener traccia dell'offset del log, per sapere fino a che punto ha scaricato i messaggi e quale è il prossimo messaggio da scaricare. Vantaggio dell'approccio pull è che si pone meno carico sui broker e si ha una maggiore flessibilità, perché i singoli consumer determinano il tasso con cui fanno pull dei messaggi, non è il borker a doversi adattare alle diverse velocità dei, consumer.
Contro: nel caso in cui consumer non trova dati, perché producer non li ha ancora prodotti rimane in busy waiting attendendo i dati in arrivo.
Consumer group: consumer possono essere raggruppati, all'occhio di Kafka appare come unico subscriber logico. In questo modo miglioro scalabilità e tolleranza ai guasti: i consumer possono leggere parallelamente su diverse partizioni.
Offset ha ruolo fondamentale nell'identificazione del punto del log a partire dal quale il consumer deve leggere, quindi è tenuto dai consumer. Vantaggio del meccanismo basato su offset è che consumer riutilizzando l'offset può riottenere messaggi scritti su una partizione, che aveva richiesto in precedenza ma non aveva potuto processare ad esempio a causa di un fallimento.
Messaggi vengono inseriti con append, consumer vedono i record nell'ordine di memorizzazione nella partizione. Topic può essere suddiviso in più partizioni, garanzia sull'ordinamento viene offerta non a livello di topic, ma solo a livello di partizione; non può preservare ordinamento tra dei messaggi tra diverse partizioni (per solito principio del keep it simple). Garanzie di consegna di Kafka:

- At-least-once: default, garantisce che non ci siano perdite ma non che non ci siano duplicati (devo essere tollerante a questo)
- Exactly-once: garantisce che non ci ci siano perdite né duplicati, ma richiede utilizzo di un protocollo transazionale distribuito che impatta sulle prestazioni di Kafka; dipende anche da qual'è l'applicazione consumer

Utente può anche implementare una semantica at-most-once, facendo in modo che sul producer disabilito le ritrasmissioni e facendo commit dell'offset prima che il consumer abbia processato il messaggio.
Kafka replica le partizioni per tolleranza a guasti, il messaggio è disponibile ai consumer solo dopo che tutti i follower hanno terminato la scrittura ed hanno fatto ack al leader di partizione, quindi messaggi possono non essere subito disponibili, priorità alla consistenza piuttosto che alla disponibilità. Kafka mantiene i messaggi per un periodo di tempo configurato, così che se il consumer fallisce i messaggi possono essere riottenuti grazie a questa persistenza a tempo.
Zookepeer: sistema di storage di tipo key-value , distribuito e con architettura gerarchica. Ampiamente usato in SD open scource, Kafka lo usa per vari scopi:

- Elenco dei broker attivi nel sistema
- Elenco consumer e dei loro fofest e dove sono arrivati a leggere; i consumer salvano l'offset a cui sono arrivati

- Elenco dei producer: prodcuer ottengono indirizzo dei broker da Zookeeper.

Usato anche nel caso in cui broker leader subisce fallimento, viene eseguito un algoritmo di elezione distribuito per eleggere il nuovo leader (Paxos).
API per gli utenti:

- Producer API: permette alle applicazioni producer di pubblicare record su Kafka
- Consumer API: leggere record dalle partizioni in cui è suddiviso il topic
- Connect API: serve per collegare producer e consumer a Kafka, ad esempio se ho un DB e voglio interagire con Kafka. Posso collegare applicazioni già esistenti a dei topic su Kafka. Connettori per S3, altri sistemi message queue (pipeline di sistemi)
- Streams API: permette la trasformazione di stream di dati da topic di input su topic di output. Kafka diventa anche piattaforma di processamento di dati in tempo reale.

API di Kafka supportano solo Java e Scala, ma ci sono librerie lato client per il supporto di altri linguaggi: Sarama per Go, Kafka-Python e libreria per NodeJS.

10.3 Protocolli per MOM

Esistono diversi protocolli, implementati con standard aperti, tra questi i 3 più usati sono:

- AMPQ (presente anche in Go)
- MQTT: molto usato in ambito IoT, più leggero rispetto ad AMQP, meno configurabile ma più adatto a dispositivi IoT semplici. COme AMQP è un protocollo binario
- STOMP: è di tipo testuale.

Basati su standard aperti, obiettivo comune è fornire soluzioni di messaggistica indipendenti dalla piattaforma o dal vendor ed offrono interoperabilità fra di versi sistemi MOM.
Spesso usati nell'IoT: usati per mandare dati da sensori verso broker di messaggistica che poi li invia a framework per farli processare. Sfruttano i vantaggi dei sistemi MOM:

- disaccopiamento
- resilienza, ho uno storage di dati temporaneo
- premette di disaccoppiare producer dal consumer, quindi posso gestire picchi di carico. Se ho picco, memorizzandoli posso far si che sistema non perda i dati se non riesce a far fronte col processamento in tempo reale al carico.

10.3.1 AMQP

Basato su standard aperto, largamente supportato dalle aziende nel settore. Approvato come standard internazionale nel 2014. È protocollo di livello applicativo che si basa su TCP (motivi di affidabilità), permette di supportare tutte e 3 le semantiche di consegna. Caratteristica interessante è che è un protocollo programmabile, posso definire schemi di routing con cui instradare i messaggi tra le diverse code. Supportato da diversi framework di messaggistica, tra cui RAbbitMQ. Componenti principali:

- Publisher
- Subscriber
- Broker AMQP: diverse entità nel broker:
  - Code
  - Exchange: punti di scambio, ricevono i messaggi dai publisher
  - Binding: definisce come exchange è collegato alle code. messaggi pubblicati qui, che instrada opportunamente i messaggi alle code. Regole con cui exchange scambiano messaggi fra le code, una volta nelle code il modo in ui messaggi sono letti dipende dall'implementazione della coda (in RabbitMQ push, in Kafka pull)

Binding sono programmabili, possono essere di diverso tipo:

- Direct exchange: messaggi consegnati alle code basandosi su chiavi di routing.
- Fanout exchange: punto di exchange replica il messaggio su tutte le code a cui è collegato
- Topic exchange: scambio del topic, viene fatto matching sul topic, consegnando messaggio ad una o più code sulla base del matching. Può essere utile se bisogna realizzare applicazione che riceva dati dipendenti dal contesto, esempio relativi a specifica locazione geografica.
- Header exchange: (slides)

Permette di definire due tipi di messaggi:

- Bare: messaggi dei producer
- Annotated: aggiunte ricevute durante il transito

L'header ha diversi parametri di consegna, a seconda del tipo di messaggio avrò un certo header.

10.4 Comunicazione multicast

Schema di comunicazione in cui di dati sono inviati verso molteplici destinatari, broadcast è un caso particolare.
Vari esempi di multicast one-to-many o many-to-many: servizi di conferenza, giochi multiplayer, etc...
Comunicazione unicast non scala: per unicast ogni utente deve ricevere il contenuto, mentre con multicast replico il messaggio sollo dove necessario.
Multicast può essere a livello di rete o a livello di applicazione.
A livello di rete: nel protocollo IP è presente IPMC, basato su gruppi ovvero un insieme di host interessati alla stessa applicazione. Uso limitato perché scarsamente supportato a larga scala, pochi autonomous systems lo implementato ed è difficile tener traccia dei gruppi. Inoltre viene in alcuni casi disabilitato perché soggetto a broadcast storm.

10.5 Multicast applicativo

Multicast invia un messaggio verso un set di destinatari, one-to-many o many-to-many. Idea di base: organizzo nodi in una overlay network su cui diffondere i messaggi. Per la realizzazione della overlay network posso usare approccio strutturato o non strutturato:

- Strutturato: soluzioni p2p già viste, creazione di percorsi di comunicazione dell'overlay network
- Non strutturato: basato su flooding o su gossiping.

10.5.1 Multicast applicativo strutturato

Mi focalizzo su multicast one-to-many: prima struttura dati che mi viene in mente è l'albero, unico percorso tra coppie di nodi. Radice è mittente, foglie i destinatari.
Altra soluzione è mesh o rete a maglia, dove ho molti percorsi tra ogni coppia di nodi, quindi maggiore tolleranza ai guasti.
multicast con albero: uso sistema basato sulla distributed hash table PASTRY, costruzione in Scribe. Scribe è sistema publish/subscribe con struttura decentralizzata basato su PASTRY.

- Il nodo che inizia la sessione multicast genera l'identificatore del gruppo di multicast, sarà il nodo mid.
- Usa PASTRY per cercare nella rete overlay il nodo responsabile per il mid.
- Tale nodo diviene la radice dell'albero di multicast
- Se il nodo P vuole unirsi all'albero di multicast identificato dall'id mid: il nodo invia la richiesta di join. Suppongo che la richiesta arrivi al nodo Q:

- Q non ha mai ricevuto richieste di join per mid, quindi Q diventa forwarder, P diventa figlio di Q e Q inoltre la richiesta di join verso la radice
- Se Q è già forwarder, P diviene figlio di Q ma Q non deve inoltrare la richiesta di join in quanto fa già parte dall'albero.

10.5.2 Multicast applicativo non strutturato

Posso usare due soluzioni: flooding: già esaminato, P invia messaggio ai suoi vicini, che se non hanno già trattato il messaggio lo inoltreranno.
Soluzione con gossiping

10.5.3 Protocolli di gossiping

Approccio probabilistico: un nodo che riceve il messaggio potrà inoltrare il messaggio solo ad un sotto-insieme di nodi che conosce, scelto in modo casuale, a sua volta ogni nodo farà l'invio con lo stesso meccanismo
Permettono rapida diffusione delle informazioni in reti a larga scala.
Origini: definiti del 1987 Demers in un lavoro sulla consistenza di database replicati su vari server. Idea di base: assumendo che non vi siano conflitti di scrittura (aggiornamenti indipendenti):

- Le operazioni di aggiornamento sono eseguite inizialmente su una o più repliche
- Una replica comunica il suo stato aggiornato ad un numero limitato di vicini
- La propagazione dell'aggiornamento è lazy
- Al termine, ogni aggiornamento dovrebbe raggiungere tutte le repliche

Svariati vantaggi nell'utilizzo:

- Semplicità degli algoritmi di gossiping
- Mancanza di controllo centralizzato e di bottlenecks
- Scalabilità, in quanto ogni peer manda solo un numero limitato di messaggi, indipendentemente dalla grandezza del sistema
- Affidabilità e robustezza, per via della ridondanza di messaggi.

Usati largamente: es Amazon S3, che usa protocollo di gossiping per diffondere informazioni all'interno di S3, questo permette tolleranza ai guasti di server Amazon.
Altro esempio è servizio Dynamo di Amazon, usato per gestire il carrello. Usa gossiping per diffondere nella rete i nodi che hanno subito fallimento, quindi usato per failure detection.

10.6 Modelli di propagazione del gossiping

Modello di gossiping puro e di anti-entropia, che potenzia il precedente.
Gossiping puro: un peer che è satto appena aggiornato contatta un altro peer scelto a caso inviandogli il proprio aggiornamento.
Anti-entropia: periodicamente ciascun peer sceglie casualmente un altro peer ed i due peer si scambiano gli aggiornamenti, giungendo al termine ad uno stato simile su entrambi. Obiettivo: minimizzare entropia del sistema, ovvero differenza di informazioni possedute dai diversi nodi.
Gossiping puro: un peer P manda il suo aggiornamento ad un altro nodo Q a caso appena è stato aggiornato. Se Q ha già ricevuto l'aggiornamento, P perde interesse verso Q, quindi con probabilità $\frac{1}{k}$ smette di contattare altri peer. Se s è la frazione di peer non ancora aggiornati, si dimostra che s = $e^{-(k-1)(1-s)}$. Al crescere di km, aumenta la probabilità che l'aggiornamento si diffonda. Combino col modello di anti-entropia.

10.6.1 Anti-entropia

DUe nodi possono scambiarsi informazioni in due modi:

- Modello push: P invia soltanto i suoi aggiornamenti a Q
- Modello pull: P richiede gli aggiornamenti a Q
- push-pull: i due nodi si scambiano gli aggiornamenti reciprocamente.

Delle diverse strategie, la push-pull risulta essere la più veloce: è possibile dimostrare che, con N peers nel sistema server un ordine di O(log(N)) round per propagare un aggiornamento agli N nodi. Round: intervallo di tempo in cui ogni peer ha preso almeno una volta l'iniziativa di scambiare aggiornamenti.
Usando solo la strategia push-pull c'è comunque bisogno di vari round, quindi si pensa di combinare con anti-entropia.

10.6.2 Schema generale del protocollo di gossiping

Considero due nodi P e Q, P ha scelto a caso Q per scambiarsi l'informazione. Considero che lo pseudo-code sia eseguito ad ogni round, quindi ogni Δ d tempo:

- selectPeer(&Q)
- selectToSend(&bufs)
- sendTO(Q, bufs)
- receiveFrom(Q, &bufr)
- selectToKeep(cache, bufr)
- processData(cache)

1. receiveFromAny(&P, &bufr)

2. selectToSend(&bufs)
3. sendTo(P, bufs)
4. selectToKeep(cache, bufr)
5. processData(cache)

10.6.3 Implementazione del protocollo

Oltre ad implementare l'algoritmo, bisogna affrontare vari problemi:

- membership: come i peer possono conoscersi fra loro e quanti conoscenti avere
- consapevolezza della rete: come far riflettere l'overlay network sulla rete fisica.
- gestione del buffer: quali informazioni scartare quando la memoria è piena. Ciascun nodo memorizza le informazioni degli altri nodi in una memoria apposita, bisogna occuparsi di come gestirla nell'implementazione.
- filtraggio dei messaggi: come considerare l'interesse per il messaggio da parte dei peer e ridurre la probabilità che ricevano informazioni a cui non sono interessati.

esempio: flooding vs gossiping
nel flooding un nodo contatta i nodi che conosce e gli invia il messaggio, nel round successivo questi invieranno ai loro vicini etc... Nel caso del gossiping contatto solo alcun nodi, con prob p: posso vedere che per la stessa struttura invio meno messaggi e con meno messaggi contatti quasi tutti i nodi.
Gossiping è quindi protocollo probabilistico che prende decisioni locali, ma che hanno impatto globale sul sistema. È più leggero in quanto scambia meno messaggi ed è robusto e tollerante ai guasti. Flooding permette di avere copertura maggiore della rete e riduce la quantità di informazione scambiata, ma il numero di messaggi scambiati nella rete è molto maggiore.
Obiettivo degli algoritmi di gossiping: ridurre il numero trasmissioni, cercando di raggiungere il maggior numero di nodi; sono però algoritmi probabilistici e ci mettono più tempo a diffondere l'informazione della rete.

10.6.4 Altre applicazioni del gossiping

Diverse applicazioni nei SD:

- peer sampling: forniscono a ciascun peer una lista di peer da contattare
- per monitoraggio di risorse in SD a larga scala.

- computazioni distribuite per l'aggregazione di dati, in particolare in sensor network: per computazione di valori aggregati, ad esempio calcolo della media. Per calcolo della media, se ho due nodi con ad esempio due valori di temperatura: all'atto del round di gossiping, i due nodi usano strategia push-pull, quindi alla fine del round i due nodi avranno l'informazione aggiornata, ovvero la media. Applicando a tutti i nodi della rete, dopo un certo numero di round il valore della media risulterà diffuso a tutti i nodi.

Due esempi di protocolli:
Blind counter rumor mongering: rumor mongering è la diffusione del pettegolezzo, quindi gossiping. Blind: ad un certo punto un nodo che sta diffondendo l'informazione perde interesse nella diffusione a prescindere da quale sia l'informazione, mente il counter per via del fatto che dopo F contatti un nodo perde interesse nel diffondere un'informazione agli altri nodi (F parametro del protocollo). Cosa fa un nodo:

- inizia diffusione dell'informazione inviando un messaggio a B dei suoi vicini
- Se nodo p ha ricevuto messaggio non più di F volte lo propaga, altrimenti smette di propagare il messaggio.
- P manda il messaggio m a B nodi scelti casualmente, scegliendo quelli che non lo hanno già ricevuto, in base a quello che lui sa.
- P sa che R ha già ricevuto il messaggio solo se o P glielo ha già inviato o se P ha ricevuto il messaggio da R.

Difficile trovare un espressione analitica in forma chiusa per descrivere il comportamento dell'algoritmo, quindi si utilizzano dei prototipi ad esempio con esperimenti.
Bimodal multicast: anche detto pbcast(probabilistic broadcast), implementato in diverse librerie. proposto alla fine dei 90'. Obiettivo è quello di effettuare in una rete un invio multicast in una rete, in modo che l'invio risulti affidabile. Composto in due fasi

- distribuzione del messaggio: un processo invia in multicast un messaggio agli altri nodi senza particolari garanzie di affidabilità. Invia con IP multicast se possibile, oc on multicast livello applicativo
- Gossip repair: uso del gossiping per migliorare l'affidabilità del multicast. una volta che processo ha ricevuto un messaggio, chiede agli altri nodi cosa hanno ricevuto, confronta con cosa ha lui e se si rende conto di non avere ricevuto qualche messaggio, chiede agli altri nodi i messaggi che non ha.

Periodicamente, ogni nodo invia un digest, che descrive il suo stato, ad alcuni nodi scelti a caso. Se un nodo, confrontando il suo stato con i digest ricevuti si accorge di non avere dei messaggi, richiede una copia di ogni messaggio dal processo che ha inviato il digest. Proposte diverse ottimizzazioni dell'algoritmo,

che è comunque semplice ed elegante.
Perché il nome: multicast perché l'invio è multicast. Perché bimodale: non è dovuto alle due fasi, ma al comportamento modale dell'algoritmo: si può vedere che l'algoritmo permette di consegnare in quasi tutti i casi il messaggio o a quasi tutti o soltanto a pochi processi. Se vedo probabilità che il messaggio venga consegnato ai nodi: o non viene consegnato a quasi nessuno, o quasi a tutti (sono queste le probabilità più alte); può non essere recapitato a nessuno perché il mittente fallisce.
Un secondo motivo del comportamento è legata alle latenze sperimentate nella consegna dei messaggi: se arriva subito latenza è bassa, se nodo invece non riceve subito il messaggio servono un certo numero di gossip repair perché tutti i nodi abbiano i messaggi.

11 Virtualizzazione

Una delle realizzazioni del concetto di indirezione: consente di ottenere astrazione di risorse computazionali, presentando un vista logica diversa da quella fisica. Permettono di usare risorse hw e sw del sistema in modo differente, disaccoppiamento dell'architettura del sistema reale dal comportamento hardware e software che l'utente percepisce: viene interposto un layer di virtualizzazione che sarà collocato a diversi livelli, a seconda di cosa si vuole virtualizzare. Obiettivi diversi: ambienti di esecuzione multitenant, affidabilità, prestazioni, sicurezza etc...
Possibile virtualizzare qualsiasi tipo di risorse:

- Virtualizzazione di risorse di sistema
- Virtualizzazione di storage
- Virtualizzazione di rete (VLAN, VPN)

FOndamentale introdurre prima distinzione dei componenti dell'ambiente virtualizzato:

- Guest: ospite, componente del sistema virtualizzato che interagisce con il layer di virtualizzazione piuttosto che con l'host
- Layer virtualizzazione: livello di indirezione che permette di astrarre le risorse fisiche e software che l'host mette a disposizione
- Host: ambiente originale

11.1 Macchina virtuale

Una macchina virtuale (VM) permette di rappresentare le risorse hw e sw diversamente da quello che sono in realtà. Posso avere vista logica della CPU, schede di rete etc... diversa dalla vista fisica delle effettive risorse, posso emulare su una macchina senza GPU la presenza di un GPU, eseguire un'abbiente

di esecuzione di CUDA (principale ambiente di esecuzione delle GPU Nvidia) anche se la GPU non è presente.
Posso anche avere SO diverso dal SO presente nell'ambiente host; possibile avere molteplici VM sulla stessa macchina fisica. Virtualizzazione è concetto degli anni 60', ma in contesto diverso in quanto non c'era ancora rete Internet, contesto completamente centralizzato: soluzione per eseguire software legacy su dei nuovi mainframe e condividere le risorse fisiche tra i diversi utenti.
È una delle tecnologie abilitanti del cloud computing.

11.2 Vantaggi della virtualizzazione

Permette facilmente la compatibilità, portabilità, interoperabilità e migrazione di applicazioni ed ambienti, rende il tutto indipendente dall'hw, principio del Create One, Run Anywhere.
Permette inoltre il consolidamento dei server in un data center, con vantaggi economici, gestionali ed energetici:

- Multiplexing di molteplici VM sullo etsso server
- Obiettivo: ridurre il numero totale di server usati, usandoli in modo efficiente
- Riduce i costi, consumi energetici, spazio occupato.
- Semplice nella gestione

Virtualizzazione consente di rendere le applicazioni più sicure, in quanto è possibile isolare semplicemente componenti guasti o sotto attacco, in quanto macchine virtuali differenti non possono accedere sulle rispettive risorse.
Permette di isolare le prestazioni di diverse VM, tramite lo scheduling delle risorse fisiche che sono condivise tra molteplici VM in esecuzione sulla stessa macchina fisica. È anche possibile spostare la VM in esecuzione da un server ad un altro, permette quindi di bilanciare carico sui server.
Svariato suo:

- Ambito personale e didattico: per eseguire più SO sulla stessa macchina, semplificazione dell'installazione del software.
- Ambito professionale: debugging, testing e sviluppo di applicazioni.
- Ambito aziendale: consolidare l'infrastruttura del data center, garantire business continuity incapsulando interi sistemi in singoli file (system image) che possono essere replicati, migrati o reinstallati su qualunque server.

11.3 Livello di virtualizzazione

Possibile realizzarla a vari livelli:

- A livello di ISA

- A livello di interfaccia tra hw e sw
- A livello di system call (ABI = Application Binary Interface)
- A livello di APIs

Possiamo introdurre 5 livelli diversi di virtualizzazione:

- Emulazione del livello ISA: interpretazione delle istruzioni o interpretazione dinamica del codice binario
- Hardware level: basati su virtual machine monitor o hypervisor: identificano il layer di virtualizzazione
- A livello di SO, containers
- A livello di librerie
- A livello di applicazioni utente

11.3.1 Process Virtual Machine vs System Virtual Machine

A livello applicazioni utente astraggo risorse virtuali per un processo (es JVM), una piattaforma virtuale che esegue processi individuali.
Fornisce un ambiente ABI o API vortuale per le applicazioni utente. Qui non si pone il problema di andare a virtualizzare le chiamate privilegiate, SO è lo stesso per tutte le risorse
Con virtualizzazione a livello di sistema, posso avere molteplici SO diversi: il virtual machine monitor gestisce insieme di risorse hardware condivise con altre VM e offre isolamento rispetto alle altre. In questo caso la VMM si fa da tramite per l'esecuzione di istruzioni privilegiate o operazioni che interagiscono direttamente con l'hardware condiviso: quando VM vuole eseguirne una, VMM la prende, ne checka la correttezza.

11.3.2 System-level virtualization

Host: piattaforma sulla quale vengono eseguite le VM, include la macchina fisica, il possibile OS ed il VMM.
Guest: il SO guest e le applicazioni eseguite nella VM.
Classificazione delle possibili soluzioni di virtualizzazione:

- Dove è possibile fare deploy del VMM: System VMM o Hosted VMM
- Come virtualizzare le istruzioni privilegiate, in particolare un sotto-insieme che sono le istruzioni sensitive non virtualizzabili: Full virtualizzazion vs Para-virtualization

Dove è possibile collocare il VMM nella macchina host:

- Direttamente sull'hardware: VMM di sistema o bare metal. In questo caso ambiente host non presenta SO, è formato da hardware e VMM.

- Sopra il SO host: VMM ospitato, host è rappresentato da hardware, SO host e dal VMM

Confronto tra le due soluzioni:

- Prestazioni migliori sul VMM di sistema, eseguo hypervisor direttamente sull'hardware. È una sorta di SO semplificato, invece di effetuare scheduling delle risorse hardware tra i processi, questo effettua scheduling delle risorse tra le VM. In termini di architettura del VMM, questo può avere architettura a mirco-kernel, avrà quindi un layer thin con poche funzionalità, o monolitica. Esempi: Xen, KVM, VMware, ESX, Hyper-V

- VMM host: hypervisor si colloca al di sopra del SO. Più seplice la realizzazione: è possibile sfruttare tutte le feature offerte dal SO host, scheduling di risorse più facile: tutto sarà visto come processi da schedulare dal SO host. L'hypervisor interagisce direttamente tramite ABI col SO e devo solo emulare l'ISA di hardware per i SO guest. Può quindi sfruttare scheduling del SO per le risorse di I/O, altro vantaggio è che non è necessario modificare il SO host per sia in grado di interagire col VMM, in alcune soluzioni affinché SO guest possano essere eseguiti al di sopra dell'hypervisor (solo nel caso di para-virtualizzazione) è necessario modificare il SO guest perché dovrà essere in grado di colloquiare con l'hypervisor. Svantaggio è che c'è degrado di prestazioni, essendo VMM in un livello più alto.
  esempi: Bochs

Modalità di dialogo tra VM ed il VMM per l'accesso alle risorse fisiche, ovvero le istruzioni privilegiate:

- Virtualizzazione completa: il VMM presenta ad ogni VM un interfaccia hardware simulata che è funzionalmente identica a quella della sottostante macchina fisica. I SO guest interagiscono con l'hypervisor come se interagissero con il bare metal, quindi non devbono essere modificati perché l'interfaccia che vendono è quella che si sarebbero aspettati in un ambiente virtualizzato.
  Le istruzioni invoca dai SO guest vengono gestite dall'hypervisor, che intercetta la richiesta di accesso privilegiato all'hardware e ne emula il comportamento atteso. esempi: KVM, VMware, Microsoft Hyper-V. Vantaggi: non occorre modificae il SO guest, c'è isolamento completo tra le istanze di VM, quindi più sicurezza, facilità di emulare le diverse architetture. Svantaggio: layer di virtualizzazione (VMM) è più corposo, inoltre per avere prestazioni più efficienti bisogna necessariamente avere la soluzione assista dall'hardware. È necessaria la collaborazione del processore per un'implementazione efficace.

- Para-virtualizzazione: il VMM espone ad ogni VM un interfaccia hardware simulata che è funzionalmente simile. Questa volta il SO guest deve sapere che ciò che ha al di sotto non è più l'hardware ma l'hypervisor, quindi i SO guest devono essere modificati per poter colloquiare con l'hypervisor.

Il VMM creerà strato minimale di software per gestire le diverse VM e per tenerle isolate fra loro, layer di virtualizzazione deve svolgere meno compiti, saranno usati direttamente device driver del sistema guest. esempi: Xen, OracleVM (basato su Xen), PikeOS.

11.3.3 Problemi da affrontare

Problemi da affrontare per realizzare virtualizzazione: architettura del processore opera su almeno due livelli (ring) di protezione: supervisor e user. In x86 ci sono 4 ring:
user application: livelli minimi di privilegi.
SO: ring 0, privilegio massimo.
Con la virtualizzazione, monitor della VM deve per forza operare nel ring 0, perché diviene gestore dell'hardware sottostante. Ma in questo caso il SO guest è nel ring 3 o 1: quindi non può più eseguire istruzioni privilegiate, problema del ring deprivileging. Inoltre c'è problema del ring compression: SO guest opera nello stesso ring dell'applicazione, va protetto lo spazio del sistema operativo guest.
Soluzioni per ring deprivileging, in virtualizzazione completa:

- Trap-and-emulate: quando SO guest cerca di eseguire istruzione privilegiata (es lidt x86) interviene il VMM, notifica di eccezione al VMM (trap) e controllo viene trasferito al VMM. Questo controlla la correttezza dell'operazione e la esegue andandone ad emulare il comportamento. Può essere a livello hardware o software

- Istruzioni non privilegiate sono eseguite direttamente dal SO guest.

Meccanismo può essere realizzato sia a livello hardware che software: per realizzarlo a livello hardware la macchina fisica deve fornire un supporto per la virtualizzazione. Si usa meccanismo a livello software se processore non fornisce supporto hardware alla virtualizzazione (introdotto nel codice macchina una chiamata all hypervisor)

11.3.4 A livello hardware

Forma di virtualizzazione a livello hardware è stata introdotta circa 10 anni fa, tipicamente i processori che supportano il meccanismo hanno una V nel nome. Processori introducono due nuove forme di modi operativi della CPU: root e non root mode, ciascuno dei quali supporta tutti e 4 gli anelli di protezione. Monitor della VM si fa operare in root mode, nel ring 0, mentre SO guest opera nel non root mode nel ring 0, così che il SO sia in esecuzione nell'anello corretto, ma se deve eseguire istruzione non privilegiata, viene inviata notifica al VMM per la gestione della chiamata.
Risolvo il problema della compressione del ring ed anche del ring deprivileging; meccanismo mette anche a disposizione dell'hypervisor delle strutture dati memorizzate nella RAM. Ma come virtualizzo architettura che non supporta tale meccanismo? Soluzione a livello software

11.3.5 Fast binary translation

Introduco meccanismo di eccezione nel codice eseguito dal SO guest: il monitor della VM scansiona il codice che deve essere eseguito e sostituisce blocchi di codice che contengono esecuzione di istruzioni privilegiate con blocchi per notifica di eccezioni per la VMM. Nel momento in cui SO guest deve eseguire tale istruzione, se ne fa carico il VMM, overhead della scansione introduce degrado delle prestazioni, legato al fatto di dover scansionare codice prima della sua esecuzione.
Per cercare di ridurre overhead, blocchi tradotti vengono conservati in cache per poterli recuperare se necessario, questo però richiede gestione cache con le eventuali politiche.
In termini di hypervisor questo ne comporta una maggiore complessità, diventa layer più spesso.

11.3.6 Paravirtualizaztion

Layer di virtualizzazione espone ai SO guest un'interfaccia funzionalmente simile all'hardware sottostante, ma non uguale. Non è più soluzione trasparente ai SO guest, per potervi colloquiare SO guest deve essere modificato, per poter in particolare invocare API dell'hypervisor.
Le istruzioni privilegiate non virtualizzabili vengono sostituite da chiamate alla API virtualizzata esposta dall'hypervisor, che prendono il nome di hypercalls.
Con le hypercalls si realizza meccanismo di trap&emulate, sono usa specie di chiamata di sistema dal SO gest verso l'hypervisor.
Vantaggi rispetto a virtualizzazione completa:

- Avere hypercall all'hypervisor è implementazione relativamente più semplice del meccanismo della fast binary translation, introduce overhead più ridotto, non devo più tradurre e sostituire i blocchi.
- Non richiede supporto hardware, posso realizzare hypervisor che non richiede hardware di ultima generazione o che supporti la virtualizzazione.

Contro:

- Per realizzare l'hypercall devo modificare il SO guest, src code dei SO che vengono para-virtualizzati deve essere disponibile. SO difficilmente portati, tipo Windows possono usare meccanismo di para-virtualizzazione usando device ad hoc di per rimappare hypercall
- C'è costo di manutenzione che si aggiunge per SO para-virtualizzati e non posso più eseguirli sull'hardware.

11.4 Architetture per VMM

3 componenti fondamentali del SO che suppoprta la virtualizzazione:

- Dispatcher: si interfaccia con istante delle VM in esecuzione ed ha il compito di fare da interfaccia per le istruzioni privilegiate e di distribuirle agli altri 2 componenti del sistema
- Interprete: esegue routine apposita quando la VM chiede di eseguire una istruzione privilegiata
- Allocatore: decide come assegnare risorse hardware alle VM.

11.4.1 Scheduler

Scheduler VMM introduce layer addizionale di scheduling: nell'ambiente tradizionale non virtualizzato sono abituato a scheduler di CPU che deve schedulare i diversi processi alle CPU. COn virtualizzazione aggiungo layer in più perché bisogna assegnare le CPU virtuali viste dai SO guest sulle CPU fisiche della macchina fisica. Scheduler del SO guest si occupa di schedulare processi sulle CPU virtuali, mentre scheduler delle CPU fisiche realizza scheduling dellle CPU virtuali.

11.5 Virtualizzazione di memoria

In ambiente non virtualizzato ho distinzione fra memoria fisica e memoria virtuale, con meccanismo di traduzione (paginazione) ho mapping fra le due memorie, ed all'interno dell'architettura hardware del calcolatore ho due elementi per ottimizzare traduzioni che sono MMU e TLB, per ridurre aggravio di prestazioni legate al concetto di memoria virtuale.
In ambiente virtualizzato situazione si complica: ho molteplici VM che condividono la memoria fisica. Hypervisor deve partizionare la meoria fisica fra le diverse VM, c'è la necessità di introdurre ulteriore livello di mapping in mood che nella macchina virtuale ci sia traduzione da memoria virtuale guest a memoria fisica guest a memoria fisica host. Distinguo tra memoria virtuale del guest, memoria fisica del fuest e memoria virtuale dell'host:

- Memoria virtuale guest: memoria vista dalle applicazioni del SO guest
- Memoria fisica del guest: visibile al SO guest
- Memoria fisica host: memoria realmente a disposizione nell'host.

Il mapping avverà: guest virtual address $\rightarrow$ guest physical address $\rightarrow$ host machine address; il guest physical address è diverso dall'host machine address.

11.5.1 Shadow page table

Per evitare degrado di performance, hypervisor mantiene una sua tabella delle pagine, detta shadow page table: nel SO host, ci sono page table e TLB, hypervisor usa SPT oer accelerare mapping, in quanto permette di evitare livello di mapping in quanto contiene mapping VA guest $\rightarrow$ MA host.
Per ciascun frame fisico della memoria fisica host il VMM mappa una entry nella SPT, problema: bisogna mantenere consistente le informazioni nella SPT,

quindi quando il SO geust cambia mapping tra memoria host fisica e memoria guest fisica anche la SPT deve essere aggiornata.
Supporto della virtualizzazione di memoria non è affatto banale da realizzare nell'hypervisor, in quanto richiede si meccanismo di traduzione degli indirizzi (devo dare al SO guest l'illusione di avere spazio di memoria fisico che inizia da 0).
Shadow pgate table è complessa in quanto va gestita ed aggiornata correttamente. Vengono ovviamente introdotti overhead non trascurabili per gestione delle SPT, che sono in memoria e quindi occupano dello spazio disponibile che può essere non trascurabile.

11.5.2 Supporto hardware per virtualizzazione

Implementazione che richiede supporto hardware, realizzata con meccanismo di second level address translation: soluzione hardware introdotta ad hoc nelle architetture hardware che supportano virtualizzazione così da realizzare in hardware le SPTs. Usare questa soluzione imprime performance significative, circa 50% in più sotto benchmark intensi, ovviamente che siano memory-intensive (se lo fai I/O intensive o CPU intensive non lo vedi).

11.6 Case study: Xen

L'esempio più noto di para-virtualizzazione. Considerato in quanto:

- Non abbiamo tempo per esaminare diverse soluzioni
- È open source, caso interessante di para-virtualizzazione (forse caso più importante)

Xen è hypervisor di tipo 1: monitor di VM eseguito su bare metal, hypervisor con architettura a micro-kernel. Offre al SO guest un'interfaccia virtuale, chiamata API delle hypercall che SO guest può invocare per accedere alle risorse fisiche della macchina. Nato come supporto del meccanismo della para-virtualizzazione, ma nel corso degli anni modificato per supportare anche virtualizzazione supportata dall'hardware. Usato dai grandi provider di servizi cloud come Alibaba, Amazon, Rackspace etc... Pro e contro di Xen:

- Hypervisor thin: circa 300k loc su x86 e 65k su Arm.
- Occupa poca memoria, piccola footprint (circa 1MB)
- Scalabile fino a 4096 CPU con 16TB RAM
- Più sicuro e robusto rispetto ad altri hypervisor in quanto espone superficie di attacco minore avendo meno loc, ma comunque vulnerabile ad alcuni attacchi.
- Continuamente migliorato
- Offre possibilità di adattamento all'architettura

- Overhead limitato che non supera il 2% anche nelle situazioni peggiori
- Supporta le VM live migration: spostare VM in esecuzione da un macchina fisica all'altra
- Prestazioni in I/O rimangono comunque sfidanti, problema sotto sforzo con benchmark intenso.

Sviluppato dall'università di Cambridge, prima release pubblica nel 2003, esigenza di sviluppare nuova soluzione di virtualizzazione in quanto era necessaria soluzione di VMM scalabile per circa 100 VM running senza modifiche all'ABI (fine anni 90' c'era solo fast binary tanslation).
Architettura a mircokernel, tutto può essere para-virtualizzato con Xen:

- Dischi ed interfacce di rete
- Interrupt e timers
- Istruzioni privilegiate e tabelle delle pagine

Hypervisor di Xen si occupa dello scheduling, della gestione della memoria, degli interrupt e del device control. Inoltre tiene traccia dei domini ospitati al di sorpa dei Xen e delle CPU viste dai dominii.
Dominio è un'istanza di VM, ciascuna avrà dei processi in esecuzione ed un SO proprio, vengono eseguiti su CPU virtuali.
Ci sono domini U, quindi unpriviledged all'interno dei quali sono eseguiti SO guest. Domini completamente isolati dall'hardware.
Dominio 0 è speciale, lo scopo è eseguire istruzioni privilegiate ed istruzioni di controllo, viene avviato al boot di Xen. Dom0 contiene driver per tutti i dispositivi e contiene 3 componenti per gestione di compiti specifici. A differenza dei domini U, dom0 può accedere direttamente all'hardware host e può interagire con le altre VM. All'interno del dom0:

- XenStore: spazio di storage che è comdiviso tra di diversi dominii ed è gestito da un demone in esecuzione su dom0. Serve a mantenere informazioni sullo stato delle diverse VM U e sulla loro configurazione. Informazioni salvate in forma di key-value, se c'è cambiamento del valore viene notificato la VM che possiede la relativa chiave. Può comunicare con i domU con memoria condivisa, sfruttando i privilegi che è in esecuzione sul dom0 che ha tutti i privilegi.
- Toolstack: permette a dom0 di gestire ciclo di vita delle VM: dalla creazione, messa in pausa, migrazione su altro host e spegnimento.
  Se user vuole creare VM, può fornire file di confirguazione in cui spiega n° virtual CPU e memoria + dispositivi di I/O per la VM. Toolstack parsa le informazioni e le scrive nello XenStore, Toolstack sfrutta i privilegi del dom0 e quindi può effettuare mapping della memoria, caricare kernel e setuppare il canale di comunicazione per far comunicare XenStore con le altre VM.

Xen hypervisor usa le hypercall: domU in esecuzione nel ring1, applicazioni in ring 3 e hypervisor+ dom0 nel ring 0.
Scheduling delle CPU in Xen: lo scheduler deve scegliere tra le vCPU quali saranno eseguite sulle CPU fisiche. Stesso obiettivo dello scheduler di CPU: dare illusione ai processi di poter utilizzare sempre la CPU, introduce ulteriore livello di scheduling, Xen permette di scegliere diverse politiche di scheduling. Default è credi scheduler, assegna crediti a vCPU per usare CPU fisiche. Goal degli algoritmi di scheduling:

- Assicurarsi che tutti i dominii abbiano allocazione di quote nell'uso di CPU fair. Algoritmo del credit scheduler permette di allocare la CPU fisica in modo proporzionale alle diverse VM, alloca CPU fisica proporzionalmente alla quantità di share (peso) assegnato alla VM.
- Tenere la CPU impegnata, algoritmo work-conserving, che evita stato di idle della CPU
- Non introdurre overhead nell'operazione di scheduling

Algoritmo credit scheduler:

- Ogni VM ha un peso ed un parametro, detto cap; sono entrambi configurabili. Peso: quantità di allocazione di CPU per dominio, 256 per default. Cap è max quantità di CPU che un dominio può usare: se pari a 0 ma vCPU non ha da fare può eseguire la VM, se invece cap è diverso da 0 limita CPU ricevibile
- Scheduler trasforma il peso in credito per allocazione per ogni vCPU. Se consuma tutto e cap = 0 può avere allocazione extra, altrimenti no.
- Per ciascuna pCPU, lo scheduler mantiene vCPU e sceglie di schedulare quella con credito più basso. Bilancia automaticamente il numero di vCPU nel caso di più CPU fisiche.

11.6.1 Performance degli hypervisor

La virtualizzazione ha avuto forte sviluppo negli ultimi anni, costi ridotti della virtualizzazione a livello di sistema, ma ci sono ancora overhead non trascurabili quando vengono distanziate più VM sulla stessa macchina fisica che competono per accesso alle risorse.
Riassunto: non c'è hypervisor che è migliore per qualsiasi carico di lavoro o per qualsiasi situazione, le prestazioni differiscono al variare del carico di lavoro.
Risultati di overall in studio (vedi slides): performance variano tra il 3% ed il 140% a seconda del tipo di risorsa, ma nessun hypervisor ha performance migliori di tutti gli altri in qualunque tipo di test.
In caso di benchmark CPU intensive o memory intensive overhead risulta essere minimale, mentre per benchmark I/O intensive o network intensive ho overhead maggiori e ci sono differenze più marcate tra gli hypervisor (tra questi Xen soffre di più.

11.7 Portabilità e migrazione

Immagine di una VM: copia della VM, che contiene un SO, file dati ed applicazioni, ogni VM ne ha una associata.Come si importano / esportano le immagini di VM per evitare vendor lock-in? SI usa formato comune, l'OVF (Open Virtualization Format) che è uno standard aperto per il packaging e la distribuzione delle VMs. La configurazione della VM è specificata in XML all'interno del file. Supportato da molti, ma non tutti, i prodotti di virtualizzazione.
Migrazione di VM: ci sono due tecniche utili per deployare e gestire ambienti virtualizzati a larga scala:

- Dynamic resizing: come faccio vertical scaling (sia up che down)
- Live migration: sposto la VM su differenti macchine fisiche senza stopparla

11.7.1 Dynamic resizing

Dynamic resizing: meccanismo a grana fine per controllo delle caratteristiche della VM rispetto alla soluzione di migrare la VM su un altro server o farne il reboot. es: appl. su VM comincia a consumare risorse, VM comincia ad essere a corto di memoria o a over-usare CPU virtuali. Quindi come soluzione posso pensare di ridimensionare la VM in modo da darle più memoria o vCPU.
Vantaggi: soluzione a basso cost, di ed evita il reboot (devo spegnere, cambiare file di configurazione con param di allocazione, riavvio). Contro: non è supportato da tutti i prodotto di virtualizzazione e da tutti i SO guest.
Se supporto è offerto, le risorse hw di cui si può fare resizing dinamicamente sono n° vCPU e quantità di memoria assegnata.
Per quello che riguarda CPU, hce supportano hot-plug/unplug, posso modificare n° CPU agendo sulle informazione contenute in un file system virtuale (sysfs) che contiene file virtuale che contiene info sui processori a disposizione. Agisco sulla directory dove ci sono i file, uno per ogni CPU vista dal SO guest e posso modificare il numero di CPU viste: posso "accendere/spegnere" le vCPU viste dal SO guest.
Altra risorsa di cui si può fare resizing a run time è la memoria: tecnica basata sul memory balooning. VI è quantità massima di memoria assegnata alla singola VM, la quantità di memoria massima può essere ridotta e aumentata a run time con un "palloncino" che perette, nel caso in cui è sgonfio, di avere una certa quantità di memoria in più, o di ridurre memoria "sgonfiando" il palloncino: quando si gonfia palloncino: situazione di swap su disco, inoltre palloncino non può superare valore max assegnato come parametro per la VM (per farlo devo spegnere VM e cambiarlo). Tecnica usata da diversi hypervisors (KVM, Xen VMware etc...).

11.7.2 Migration in LAN

Vantaggi della migrazione:

- utile in cluster e data center virtuali
- Maggiore flessibilità nei failover
- Bilancio del workload

Svantaggi e problemi:

- Supporto da parte del VMM
- Overhead non trascurabile
- Migrazione in ambito WAN non banale

Per migrare VM ci sono due approcci:

- Spengo VM, trasferisco l'immagine su server di destinazione e lì riavvio la VM. Svantaggio: tempo tra spegnimento ed up&running nuovo può essere lungo (circa min per shutdown/restart). Inoltre se VM pesa molto, può essere problematico spostarla al server di destinazione.
- Live migration: sposto mentre VM è in esecuzione e voglio tenerla attiva per il maggior tempo possibile. Soluzioni cercano di ridurre tempo effettivo per cui VM non è raggiungibile.

Soluzione live è largamente utilizzata dai maggior cloud provider (Google circa 1M di migrazioni al mese nei suoi data center).
Fasi della migrazione:

- Set up: scelgo VM da migrare e host su cui la migrerò. Posso scegliere in base a soluzione di problema di ottimizzazione o su base di un'euristica, come bilancio carico di lavoro, tolleranza a guasti etc...
- Ora so quale VM migrare e dove migrarla, cosa devo spostare in migrazione live: la memoria, lo storage, e le connessioni di rete.
  Obiettivo e farlo in modo che le applicazioni che sono in esecuzione sulla VM non se ne accorgano, quindi in modo trasparente. (ci sarà downtime per le applicazioni sulla VM in cui non saranno disponibili).

Migrazione dei componenti:

- Migrazione dello storage:
  - Storage condiviso dalla scr machine alla dest machine, potrei avere una SAN (Storage Area Network), n NAS(Network Attached Server) o file system distribuito (come NSF, GlusterFS o CEPH).
  - Se non ho storage condiviso: il VMM sorgente salva tutti i dati della VM sorgente in un file immagine, che viene trasferito sull'host di destinazione.
- Per la migrazione delle connessioni di rete:

  - La VM sorgente ha un IP virtuale (eventualmente anche MAC virtuale), il VMM conosce quindi il mapping tra IP virtuale e MAC
  - Se host src e dest sono sulla stessa sotto-rete IP: non occorre fare forwarding sulla sorgente, disabilito ARP, in modo da evitare che nel momento in cui ho spostato la VM sull'host di dest risponda l'host src ad una richiesta ARP, aggiorno quindi tabelle usate dal protocollo ARP. Invio di ARP reply per avvisare che IP è satto spostato.

- Migrazione memoria: devo migrare memoria ram, contenuto di registri e cache e contenuto dei device driver. Esistono 3 soluzioni per migrare la memoria. Approccio più usato è il pre-copy:

  1. fase di pre-copy: monitor della VM in modo iterativo copia le pagine usate dalla VM alla macchina VM di destinazione e lo fa mentre VM è in esecuzione.
  2. fase di stop-and-copy: la VM sorgente viene fermata e vengono copiate pagine dirty(pagine sporcate dopo l'ultima iterazione), stato della CPU, stato dei device driver. Applicazioni in esecuzione sulla VM non sono disponibili
  3. fase di commitment e reactivation: la VM di destinazione carica lo stato e riprende l'esecuzione; la VM sorgente viene rimossa (ed eventualmente l'host sorgente viene spento)

Approccio pre-copy: memoria è copiata e migrata prima che la VM sia avviata sull'host di destinazione; soluzione tipicamente usata. Svantaggio del pre-copy è che se carichi di lavoro sono CPU/memory intensive la fase di downtime è lunga. Altre soluzioni, che cercano di ridurre il tempo di down-time durante la migrazione:

- Soluzione post-copy: fa copia posticipata rispetto all'attivazione della VM sull'host di destinazione. Sposta l'esecuzione sull'host di destinazione all'inizio del processo di migrazione e in background trasferisce le pagine di memoria usate dalle applicazioni sulla macchina sorgente, man mano che vengono richieste dalle VM.
  Vantaggio è che viene ridotto tempo di downtime, ma problema è che se devo trasferire molte pagine l'esecuzione delle applicazioni risulterà più lenta, perché devo trasferire via rete le pagine di memoria. Altro svantaggio è che non posso spegnere VM finché non ho trasferito tutte le pagine, quindi non è molto tollerante ai guasti.

- soluzione hybrid: ibrido delle prime due, attua post-copy ma è preceduta da una prima fase di pre-copy. Idea è di trasferire in pre-copy le pagine di memoria più utilizzate, far partire la VM di destinazione e trasferire via via le altre pagine di memoria.
  Si riduce tempo di downtime

Implementazioni non ufficiali di post-copy e hybrid. Migration in WAN di storage e rete:
Storage:

- Shared storage: non ho problemi, ma gli accessi possono essere lenti
- On-demnad fetching: trasferisco solo alcuni blocchi a destinazione e fetcho i restanti qudno richiesti, non tollerante a guasti
- Pre-copy/write throttling: faccio pre-copy dell'immagine di disco della VM a destinazione mentre la VM continua a girare, tiene traccia delle operazioni di scrittura sulla sorgente.. (vedi slides)

Per le connessioni di rete:

- IP tunneling: set up di un tunnel IP fra il vecchio IP address sull'host sorgente ed il nuovo IP sull'host di destinazione. Uso il tunnel per effettuare forward dei pacchetti dalla src alla dest. Appnea migrazione è stata effettuata e destr è pronta a rispondere ai pacchetti, modifico DN server auth., facendo puntare al nuovo indirizzo IP ed attenendo propagazione della modifica al DN server; a quel punto tiro giù il tunnel.
- Set up VPN
- Uso soluzioni di SDN

11.8 Virtualizzazione a livello di SO

Eseguo molteplici ambienti di esecuzione che sono i container, tra di loro isolati, tutti nello stesso SO che quindi condividono il kernel del SO. Ambienti di esecuzione detti container, ma anche jail,zone, o VE (Virtual Execution Environment). Ogni container ha il proprio insieme di processi, file system, utenti, interfacce di rete con IP, routing table etc... A differenza della virtualizzazione a livello di sistema, dove ho il SO guest,che usa diversi GB di memoria, qui ho lo stesso SO, quindi primo vantaggio è che l'immagine dei container è tipicamente più piccola dell'immagine delle VM.
Meccanismi per la realizzazione dei container:

- chroot: change root directory, nei sistemi Unix/Linux like, comando per cambiare root di riferimento dei processi in esecuzione. Comando per avere differenti root directories per i differenti container. Non basta solo questo, servono meccanismi di isolamento per isolare risorse hardware e software viste dall'insieme di processi visti
- cgroups: control groups, meccanismo di SO Linux per limitare, misurare ed isolare utilizzo di risorse hardware di un insieme di processi
- namespaces: meccanismo per isolare ciò che l'insieme di processi all'interno del container può vedere dell'ambiente operativo (processi, porte, stack di rete), sono 6 in Linux.

Vantaggi della virtualizzazione a livello di SO:

- Minore overhead legato alla virtualizzazione. Per supporto alle istruzioni privilegiate, ad esempio, ci si rivolge direttamente al SO; il kernel è sempre lo stesso.
- Tempo di avvio/spegnimento di container: nel caso in cui immagine del container sia già disponibile, in questo caso avrò tempi più rapidi.
- Densità elevata: possibile avere diverse istanze di container sulla stessa macchina fisica.
- Immagine di dimensione minore, in quanto non comprende il kernel di SO. Occupo meno spazio di storage
- Possibilità di condividere pagine di memoria tra molteplici container in esecuzione sulla stessa macchina
- Maggiore portabilità ed interoperablità per applicazioni cloud, l'applicazione nel container è indipendente dall'ambiente di esecuzione.

Ci sono alcuni svantaggi:

- Minore flessibilità, in quanto non si possono eseguire contemporaneamente kernel di SO diversi sulla stessa macchina fisica, ed inoltre si possono usare solo applicazioni native per il SO supportato
- Minore isolamento, soprattutto nelle prime versioni di Docker c'erano bug
- Maggior rischio di vulnerabilità, superficie esposta è kernel del SO (maggior linee di codice esposte rispetto all'hypervisor)

Windows e Mac OS offrono supporto per la virtualizzazione basata su container, c'è Docker desktop ed in alternativa si può installare hypervisor, installare VM con Linux come SO guest e in questa VM usare virtualizzazione con Docker, ma performance saranno disastrate.

11.8.1 Container e DevOps

I container sono diventati strumento importante nell'ambito dell'ingegneria del software: costruzione, impacchettamento condivisione e deploy di applicazioni. Transizione verso il DevOps, dove c'è continuo interazione tra sviluppo e deploy, container abilitano la CI e Cd (continous deployment). Ciclo continuo di develop, deploy e monitoring: possibile rilasciare facilmente successive versioni dell'applicazione cambiando immagine del container. Container permettono di realizzare in modo agevole l'interazione tra development e release. MOlti framework utili: Docker per i container, Kubernetes per orchestrazione di container su molteplici nodi(Docker compose per orchestrare sulla stessa macchina fisica).

11.9 Docker

Docker è un'abbiente di virtualizzazione a livello di SO, open source in cui è stata posta attenzione ad alcuni aspetti di sicurezza. Permette di creare container che introducono livello minimale di overhaed nell'applicazione eseguite su container. Ciascun container include le applicazioni e tutte le dipendenze di cui essa ha bisogno, molteplici container condividono tra di loro il kernel del SO. Eseguiti come processi isolata all'interno dello user space sul SO host, i container non sono vincolati ad una struttura: servo solo che ci sia layer di containerizzazione disponibile sulla macchina sulla quale si vuole istanziare il container (cambio dei comandi a seconda del SO). Docker è sviluppato in Go, Docker al suo iinterno sfrutta meccanismi offerti dal kernel Linux per realizzare container: control groups e namespaces, prime versioni di Docker usavano Linux Container, poi sviluppato ambiente di esecuzione detto libcontainer.
Oltre ad usare cgroups e namespaces, vengono usati altri meccanismi per gestire sicurezza e controllo degli accessi.
Componenti fondamentali: è architettura di tipo client/server:

- Demone di sistema che riceve comandi
- Builder: definire il contenuto dell'immagine del container tramite docker file. Immagine definibile tramite riga di comando, specificando istruzioni, oppure (se struttura è più complicata) usando Dockerfile.
- Engine di docker per istanziare il container corrispondente ad una immagine.
- Registry esterno, che permette di registrare container, in particolare di ottenere da repo esterni immagini di container, quindi di realizzare la fare di share.

Analisi dei vari componenti: Docker engine: applicazione client/server, composto da:

- Server, detto coker deamon
- REST API che specifica le interfacce che i programmi possono usare per controllare ed interagire con il deamon
- client CLI

Possibile gestire immagini e container, creare rete interna al Docker engine per collegare molteplici container e farli comunicare fra di loro (stessa PM). Inoltre è possibile gestire volumi esterni per avere persistenza di dati, in quanto i container sono stateless. Docker deamon permette di costruire container a partire o da comandi da CLI o tramite Dockerfile, è anche possibile gestire immagini e gestire ciclo di vita del container. Permette anche di interagire con registry esterni per la distribuzione dei container.Client e deamon comunicano tramite socket o REST API.

11.9.1 Docker image

Strumento template read-only che permette di distribuire l'applicazione ,incorpora tutte le dipendenze e le configurazioni necessarie per runnare applicazioni, eliminando la necessità di scaricare ed installare package etc...
Unico requisito è avere la macchina su cui installo l'immagine con Docker engine installato. Docker può costruire immagini automaticamente leggendo istruzioni da un DOckerfile, che è file con sintassi semplice e ben definita.
Usando Docker pull posso ottenere container da repository remota pubblica, push è upload; nome di un'immagine ha sintassi [registry/][user]name[:tag] (tutto opzionale tranne name), valore di default del tag è latest che indica l'ultima versione disponibile dell'immagine, però può essere utile per discriminare fra le varie versioni.
Componenti fondamentali:

- Dockerfile: istruzioni per assemblare l'immagine
- Contesto: set di file (esempio applicazioni e librerie). Solitamente, l'immagine è stratificata su più livelli, quindi immagini possono essere basate su altre immagini.

Dockerfile è file di testo, in cui ci sono una serie di operazioni da eseguire: lettere maiuscole indicano i comandi:

- FROM: qual'è l'immagine a partire dalla quale sto costruendo la mia immagine (dopo i : specifico la versione), è mandatorio.
- RUN: permette di eseguire istruzioni all'interno del container.
- EXPOSE: permette di specificare la porta di ascolto del container
- CMD: permette di indicare il comando da eseguire all'interno del container quando questo verrà avviato.
- ENV: setta le variabili d'ambiente.

Per buildare Dockerfile: docker build [OPTIONS] PATH | URL | -.
Immagine in Docker consiste in una serie di layer sovrapposti uno sull'altro. Docker usa union file system per comporre i layer e far apparire l'immagine come un tutt'uno, ciascun layer è identificato da un ID, e può avere dimensioni differenti. Union file system sono basati sul principio del copy on write: layer verrà scritto nel file system union solo se sarà modificato.
Vantaggi del layering:

- Condivisioni e riutilizzo di layer tra diverse immagini
- Risparmio spazio di memorizzazione locale
- Facilita software specializations: se uso ubuntu tra i layer del container, non devo riscaricarlo ma lo riuso direttamente, salvando spazio su disco e banda di rete.

Container tipicamente stateless, quindi scrivo tipicamente il minimo dei dati, nel momento in cui viene rimosso vengono persi. Se ho bisogno di scrivere/leggere in modo persistente i dati, posso attaccare un volume Docker, in modo da poter avere i risultati direttamente disponibili.
Lo storage driver (overlay2) controlla come le immagini ed i container sono memorizzati e gestiti nell'ambiente host, ce ne sono differenti come AuFS (che salvano a livello di file), Device Mapper (salvataggio a livello di blocco).
Docker container: istanza in esecuzione di un'immagine in Docker, possibile fare la run, stop, kill etc... da REST API o CLI. Container è parte run d Docker, (immagine è parte di build) sono stateless.
Ultimo componente è Docker registry, che permette di distribuire immagini Docker (quindi non solo Docker file ma anche eventuali file di supporto). nota: se ho più comandi dello stesso tipo (es RUN), conviene scriverli tutti in una riga (concateno con && "slash", che non posso mettere perché non so come si fa in LaTeX) perché sennò ad ogni comando singolo corrisponde un layer, quindi l'immagine pesa di più.

11.10 Applicazioni Docker multi-container

Docker compose per girare su un unico host, per orchestrare più nodi ho Swarm e Kubernetes.

11.10.1 Docker compose

Semplice specificare la composizione, si fa in file di supporto scritto in yaml: specifico i container che vanno istanziati e quali sono le relazioni fra essi.
Comandi di base per Docker-compose: indico la composizione dei container in file yaml, di default ha nome docker-compose.yaml
docker-compose up avvia la composizione, mentre per fermarla docker-compose down.
File yaml: descrive la composizione dei container, nella terminologia di swarm i container sono chiamati services. Per ciascun container specifico:

- Immagine da cui deriva
- Nome del container (facoltativo)
- Comando da eseguire, per lanciare il servizio
- le dipendenze con gli altri container
- Link con gli altri servizi
- Specifica delle porte (binding della porta del container con la porta del sistema host)

11.10.2 Vantaggi dei container

Uso dei container è quindi molto utile, posso incapsulare tutte le risorse in un unico package.
Sono tecnologia abilitante per micro-servizi e serverless computing.
Migrazione: è possibile eseguirla come nel caso delle VM: salvo lo stato del container, lo trasferisco e riavvio. Ho ovviamente un downtime, ma posso usare soluzioni per migrare memoria come nelle VM.
Migrazione è più complicata, perché non c'è supporto diretto da parte di Docker engine, ma bisogna usare dei tool appositi: due tool principali sono CRIU e P-Haul:

- CRIU permette di fare dump and restore dello spazio utente
- P.Haul è tool eseguito sopra CRIU e eprmette di gestire migrazione di memoria e di file system (usando pre o post copy).

Nel cloud, container possono essere usati come servizi, il Container as a Service (CaaS):

- Amazon Elastic Container Service
- Azure container
- Google container engine
- ed altri...

11.10.3 Hypervisors e container nel CLoud

Tipicamente provider IaaS usano soluzione basata su hypervisors, per tutti i vantaggi su sicurezza, isolamento e flessibilità.
Con l'avvento del serverless computing, serve poter eseguire le funzioni on-the-fly quindi senza server pronti sempre per eseguire le operazioni, quindi si può pensare di usare i container, ma questi danno minore sicurezza e sopratutto dove eseguirli? Sul bare metal o sul top di un hypervisor?
Nuovo trend: combinare la sicurezza e l'isolamento offerti dagli hypervisor con la velocità e flessibilità dei container. Firecracker: nuovo strumento di virtualizzazione presentato da Amazon nel 2019, open source. Light hypervisor, basato su KVM con design minimalista che esegue il workload in una VM leggera, chiamata micro-VMs.

11.10.4 Soluzione unikernel

Sono partito dalla soluzione classica, SO sull'hardware, poi ho introdotto soluzione con hypervisor di tipo 1 (bare metal) ed in ciascuna VM posso avere diversi SO.
Poi soluzione dei container, soluzione nativa in cui container engine è posto al di sopra del SO, container sono posti sopra l'engine (es Docker) e contengono tutte le librerie, dipendenze e codice di qui l'applicazione ha bisogno. Container

si può anche istanziare in una VM, combinazione delle due soluzioni precedenti. Altra soluzione è lo unikernel: stack di layer piuttosto ridotto, ha delle limitazioni per l tipologia di applicazioni istanziabili nell'unikernel.
Perché soluzioni di virtualizzazione più leggere:

- Nuovi paradigmi: micro-servizi, componenti dell'applicazione istanziati in container. Scindo applicazione monolitica in tanti micros-servizi. COn serverless avrò funzioni istanziate su necessità
- Nuovi scenari di computazione richiedono soluzioni di virtualizzazione che introducono overhead limitato. Nel caso dell'IoT ad esempio uso soluzioni di virtualizzazione perché ho ambiente eterogeneo, quindi voglio renderlo omogeneo, operò hypervisor non è installabile su piccoli device come sensori. Anche ambiente di fog ed edge computing, dove nodi sono soggetti a charn più elevati, servono soluzioni per istanziare velocemente i componenti dell'applicazione sui nodi che vanno e vengono.
- Soluzioni di load balancing tramite virtualizzazione di funzioni di rete, che devono essere eseguite in ambienti leggeri.

Problema di come avere VM minimali, in grado di essere eseguite su hypervisor, avendo alcune caratteristiche dei container. Trend del sistema di virtualizzazione unikernel, o di SO minimali per evitare overhead del classico SO, riduco inoltre superficie d'attacco.
SO leggeri vs unikernel:

- SO minimali: architettura del kernel monolitica, rappresentano ambienti minimali per poter lanciare container. esempi: container Linux, Atomic Host. Core OS Container Linux: noto come Fedora Core OS, basato su Linux sfoltendo all'osso, lasciando solo funzionalità necessarie ad effettuare deploy di container. Eseguiti bare metal
- Unikernel, anche detti SO di libreria. Idea non recentissima, avere una sorta di VM minimale che sia in grado di eseguire una singola applicazione scritta in un unico linguaggi di programmazione. SO minimale specializzato, set di librerie minimale che eseguono i costrutti dell'applicazione eseguita, tutto nello stesso spazio di indirizzamento. Direzione estremamente specializzata di virtualizzazione leggera.

Confrontando con le altre soluzioni di virtualizzazione: nell'unikernel ho VM minimali che poggiano su un hypervisor di tipo 1. Footprint unikernel è la più piccola possibile. Vantaggi degli unikernel:

- Ambienti di virtualizzazione con footprint minima
- Veloci, no contex switch
- Sicuri, piccola superficie d'attacco
- Fast boot, in ms

Contro degli unikernel:

- Funzionano solo per ambienti virtualizzati con hypervisor
- Un solo linguaggio di run-time
- Difficile fare debug

Performance a confronto, alcuni studi di performance di tipo qualitativo:

- Overhead dei container in termini di istanziazione è molto minore delle VM. Maggiore dentista dei container, footprint minore etc..
- Svantaggio dei container in termini di sicurezza

Light virtualizaztion, molti esperimenti mostrano tempi medi di startup sono migliori di VM ma peggiori di container in quanto hanno overhead dovuti ad ambiente di virtualizzazione sottostante.

11.11 Orchestrazione di container

Docker-compose permette di eseguire applicazione multi-container, ma sulla stessa macchina fisica. Orchestrazione permette di configurare, deployare, monitorare e controllare dinamicamente applicazioni containerizzate. Forniscono strumenti per gestire applicazioni basate su container, deployati su doversi nodi del sistema. Esistono diverse, soluzioni, le due più usate per applicazioni non cloud sono Docker swarm e Kubernetes, che possono anche essere messi a disposizione come servii Cloud.

11.11.1 Docker swarm

Swarm mode, incluso nativamente in Docker, permette di gestire cluster di Docker engine, andando ad istanziare container su diversi engine. S usa come terminologia il servizio, in Docker swarm i container che fanno parte dello stesso servizio sono detti task.
Docker swarm permette di

- Scalare il numero di task, ovvero il numero di container
- Gestire il fallimento dei nodi che ospitano i container, andando a istanziare i container del nodo che ha subito il fallimento su altri nodi
- Creare rete di overlay fra i diversi servizi
- Load balacing, specificando come distribuire i container sui vari nodi

Swarm: insieme di host con Docker installato, eseguiti in modaltà swarm. Nodo = singolo Docker Engine, swarm controlla l'instanziazione dei Docker engine con architettura master worker:

- Manager o master: assegna i container ai nodi worker, monitora i worker per identificarne i fallimenti
- Nodi worker: eseguono i container

Stessa architettura è presente in Kubrenetes. È la più usata in tutti i SD, anche se non è la migliore, perché segue il principio KISS: cercare di tenere il design del sistema il più semplice possibile.
Svantaggio: master è single point of failure e collo di bottiglia, servono almeno tecniche di ridondanza a freddo del master per salvare lo stato, in modo poi da eleggere un nuovo master. Servono anche tecniche che cerchino di alleviare il troppo carico di lavoro.
Swarm permette di fare load balancing, principali comandi: istanzio il master col comando docker-swarm init: definisco IP del master. Con docker-swarm join faccio partecipare i worker.

11.11.2 Kubernetes

Sviluppato da Google, primo prodotto open-source che è stato reso disponibile da Google. Aveva già progettato due sistemi, Borg ed Omega da cui è derivato Kubernetes, servivano diversi sistemi di orchestrazione perché c'è stato grande shift verso conteinerizzaizone. Ogni settimana Google lancia più di 2 Miliardi di container.
Kubernetes serve per orchestrazioni di applicazioni di cluster multi-container ospitati su più nodi. Automatizza deploy, scaling e gestione delle applicazioni, facendo in modo di scalare dinamicamente in modo che se un container subisce un crash le applicazioni ospitate lì siano portate su altro container o che il container sia riavviato.
Caratteristiche:

- Portabilità: eseguibile su cluster di server privato, su cloud ed anche su multi-cloud
- Framework estendibile: componenti sono stati progettati con l'ottica di poter essere sostituiti ed estesi
- self-healing: si fa auto-placement, auto-restart etc...

Può essere eseguito su diverse cloud private o pubbliche, offerto anche come servizio cloud di Google.
Terminologia di Kubernetes:

- Pod: unità di base schedulata in Kubernetes. Pod è collezione di container che condividono tra di loro spazio di storage, rete e le specifiche su come eseguire quei container. È possibile che due componenti dell'applicazione siano fortemente accoppiati fra loro, quindi conviene metterli nello stesso pod.
  A ciasun pod kubernetes assegna un IP ed un URL, permette di effettuare bilanciamento del carico tra i pod

- Utenti assegnano ai pod delle etichette, che sono coppie chiave-valore. esempio: role=frontend, stage=production.
  Estrema flessibilità all'utilizzatore di Kubernetes per quello che riguarda il controllo sui pod.

Architettura di Kubernetes è distribuita, di tipo master-worker: c'è nodo master e worker, chiamati Kubernetes nodes dentro cui sono ospitati i componenti di Kubernetes che sono kube-proxy e kublet.
Master è composto da:

- Kube scheduler: si occupa di assegnare lavoro ai diversi nodi worker
- kube-controller: gestione da riga di comando
- etcd: sistema di storage key-value, usato per mantenere tutti i meta-dati della gestione del cluster. (stesso ruolo svolto da ZooKeeper in Kafka).

Nodi sono composti dal kubelet, che è l'agente di Kubernetes su ciascun nodo che viene contattato dal master per verificare che sia up&running così da verificare il corretto funzionamento del nodo.
Esiste un componente che è horizontal pod auto-scaler, responsabile di controllare elasticità dei container. Kubernetes implementa come politica di elasticità quella basata su soglia (come quella dei sistemi auto-adattativi).

11.11.3 Sharing di risorse in cluster

Ho cluster di server, devo usare molteplici framework applicativi. Come faccio per condividere le risorse fra molteplici framework eterogenei che eseguono in maniera concorrente sulle stesse macchine? Soluzione classica: partizioni staticamente il server, soluzione altamente inefficiente: utilizzazione dei diversi framework può variare nel tempo e quindi partizionamento può rivelarsi sbagliato.
Apache Mesos: cluster manager open source che offre un layer di controllo delle risorse del cluster (dei diversi server) al di sopra del quale è possibile eseguire i diversi framework applicativi.
Obiettivo di Mesos è fornire astrazione delle risorse computazionali all'interno del cluster in modo da permettere la condivisione delle risorse in modo efficiente, andando a partizionare le risorse (tutto l'hardware del server) fra i diversi framework a grana fine, quindi dinamicamente. Assegnazione può variare dinamicamente nel tempo in base alla richiesta di utilizzo da parte dei framework applicativi.Ha architettura distribuita master-worker, il worker (agente) pubblica le risorse disponibili per il master. Il master manda le risorse offerte al framework, l'elezione del master avviene con Zookeeper, in cui si salva stato del master. Master ha repliche a freddo, che vengono elette in caso di fallimento.
Accortezza per usare Apache Mesos è usare framework applicativo modificato opportunamente per eseguire su Mesos.
Non confrontabile con Kubernetes, infatti possibile usare Kubernetes al di sopra di Mesos.

11.11.4 Kubernetes distributions

Ci sono diverse distribuzioni, il src code consente la massima flessibilità.
Ci sono anche delle alternative, sopratutto se si vuole installare Kubernetes su più nodi (configurazione di Kubernetes non banale):

1. Distribuzioni pure: Kubrenetes pre-build, esempio Charmed Kubernetes.
2. Distribuzioni plus: piattaforme che integrano Kubernetes con altre tecnologie
3. Kubernetes-as-a-Service: Kubernetes come servizio cloud, utente non deve preoccuparsi di manutenere, configurare il servizio.
4. Limited-purpose: specifiche per un determinato ambiente di esecuzione.

12 Microservizi e serverless computing

12.1 Microservizi

Un "nuovo" stile architetturale per applicazioni distribuite che struttura l'applicazione come una collezione di servizi lascamente accoppiati. SI passa da un'applicazione con struttura tendenzialmente monolitica ad una suddivisione con molte componenti, i micoservizi, che sono poco collegai fra di lor. Pattern di comunicazione sarà RPC (versione moderna) e sistemi publish-subscribe e sistemi a code di messaggi, perché introducono gradi di disaccoppiamento che rispecchia quello tra i servizi del sistema stesso.
Stile architetturale non è totalmente nuovo, in quanto deriva dalle SOA, che si è largamente diffuso nei primi anni 2000, ma presente delle differenze significative dai mciroservices.
Obiettivo: costruire, gestire applicazioni composte da micro-unità aut-contenute: disaccoppiamento e modularità, applicazione divisa in un insieme di servizi, che possono essere deployati in ambiente run-time, comunicazion tramite RPC o architetture publish/subscribe. Componenti facilmente istanziati e scalati, si applicano le idee di elasticità già viste.
Tecnologie di virtualizzazione usata saranno container, ci sarà forte legame fra micro-servizi e container.
Ci saranno micro-servizi stateless per favorire l'elasticità, i micro-servizi satefull fa si che è più complesso gestirla. Stato verrà salvato in strumenti, tool e framework dedicati alla persistenza della memoria, si troveranno non solo database relazionali, ma sopratutto datastore di tipo noSQL.

12.2 Service Oriented Achitecture

SOA è paradigma architetturale per progettare sistemi software distribuiti,lascamente accoppiati.

Definizione del paradigma OASIS: paradigma per l'organizzazione e l'utilizzazione di risorse distribuite che possono essere sotto il controllo di domini di proprietà differenti. Fornisce un mezzo uniforme per offrire, scoprire, interagire ed usare le capacità di produrre gli effetti voluti in modo consistente con presupposti ed aspettative misurabili.
Esiste documento di riferimento delle architetture a servizi:

- Viene fornita vista logica
- Orientamento ai messaggi e ala descrizione
- Granularità dei servizi. orientamento alla rete, neutralità della piattaforma.

In realtà, componenti delle SOA non sono così densi come quelli delle applicazioni a micro-servizi, ma nel passare ai microservices si è persa la neutralità.
3 entità interagiscono fra di loro:

- Service requestor o conusmer: chi richiede esecuzione del servizio, andrà a cercare il servizio a cui è interessato all'interno di un registro di servizi. Diversi parametri per la ricerca del servizio: basati su QoS, informazioni più relative alla semantica del servizio
- Service registry: espone descrizioni dei servizi
- Service provider: fornisce il servizio, implementandolo o rendendolo disponibile. Pubblica il servizio sul service registry

Le 3 entità sono istanziate e gestite da diversi amministratori e su diversi domini.
Quando service requestor ha scoperto chi offre un servizio, si collega al provider ed invoca i metodi del servizio esposto.

12.2.1 Web services

Implementazione più nota è quella dei web services: sistemi software che sono sviluppati per fornire interazione machine to machine su una rete. C'è interfaccia scritta in linguaggio machine-readable e processabile da un macchina, in particolare è stato fornito un linguaggio comune (alla stregua degli IDL o protocol buffer) basato su XML (con tutti gli svantaggi del caso). Inoltre, definiscono anche protocollo di comunicazione tra i servizi, SOAP, per permettere ai requestor di invocare i servizi. Protocollo di tipo applicativo, che solitamente usa HTTP come protocollo di trasporto.
Ambiente molto variegato, ci sono più di 60 standard e specifiche, le più usate:

- WSDL: linguaggio basato su XMl per descrivere i servizi
- SOAP: protocollo di comunicazione in ambito web services
- UDDI: usato per scoprire i servizi disponibili
- BPEL e BPMN: usati per definire processi business, definiscono work-flow dell'applicazione a servizi

- WSLA: standard per definire SLA nel contesto dei web services

Numerose tecnologie sviluppate nell'ambito dei web services, ad esempio ESB (Enterprise Service Bus), piattaforma di integrazione che permette iterazione e comunicazione fra i componenti dell'applicazione

12.2.2 SOA vs microservices

- Tecnologie dei SOA sono pesanti: istanziare applicazione a servizi, configurarla, usarla richiede molto tempo. ESB è middleware complesso, lento. I micro-servizi si basano su tecnologie molto più leggere, in cui istanziazione richiede molto meno tempo, comunicazione anche richiede meno tempo perché middleware usati sono più snelli
- Protocolli tutti basati su XML, XML è lentissimo ed introduce grande overhead. Microservizi si basano su protocollo più snelli, usano REST ed HTTP direttamente
- SOA spesso viste in ambiente enterprise come soluzione di integrazione per applicazioni già esistenti. Microservices tipicamente usati per realizzare nuove applicazioni software, anche se aspetto di riusabilità non è totalmente trascurato

Heavyweight vs lightweight (se i micro-servizi sono usati bene).

12.3 Microservizi

Vantaggi:

- Microservizi è piccolo, velocemente deployabile, facile da istanziare e da scalare.
- Tecnologia sottostante ai microservizi è il container, infatti approccio più usato è avere un container per ogni istanza di micro-servizio, questo per tutte le caratteristiche efficienti dei container.
- Scalabilità migliorata
- Grande autonomia

I micro-servizi sono complemento ideale della virtualizzazione basata su container, impacchetto ogni micro-servizio come un'immagine di container e deploy come istanza di un singolo container, managing a run rime come scaling e migrazione Vantaggi sono quindi:

- Scale in/out dei micro-servizi cambiando numero delle istanze dei container
- Isolamento
- È possibile limitare le risorse usate dalle istanze

- Build e start rapido

Svantaggi:

- Serve un orchestratore di container per gestire l'applicazione multi-container

12.3.1 Come decomporre

Non c'è una strategia vincente per decomporre (è tipo un'arte)

- Decompongo in micro-servizi seguendo principio della business capability: tipico case study è sito di e-commerce, esempio di business capability può essere micro-servizio per gli ordini.
- Soluzione di decomposizione guidata dal dominio: esempio e-commerce, abbiamo due micro-servizi essenziali come gestione ordini, gestione magazzino, consegna del bene acquistato etc...
- Decomposizione per caso d'uso: considero i casi d'uso e definisco servizi responsabili per i singoli casi d'uso, ad esempio definisco micro-servizio responsabile per la consegna
- Decomposizione per nomi o risorse: definisco microservice in modo che sia responsabile per tutte le entità e risorse dell'applicazione.

12.3.2 Scalabilità

Per riuscire a scalare si possono istanziare più repliche dello stesso servizio e fare load balancing, ci sono varie tecnologie per poter riuscire a scalare: framework integrabili con Kubernetes...
Altro aspetto fondamentale legato alla scalabilità è che è semplice scalare micro-services stateless. Se micro-servizio ha stato e devo replicarlo, ogni replica avrà il suo valore ma devo partizionare il flusso di dati in ingresso per poter mantenere coerente il valore dello stato. Se invece rendo lo stato un servizio, è più semplice: mi collego a servizio di memorizzazione persistente e si fa dare valore dello stato.
C'è anche necessità di fare service discovery, servono componenti per poter identificare il servizio a cui si ci vuole collegare, così da avere un binding dinamico. Ulteriori tool per fare service discovery, scoperta a run time il servizio a cui collegarsi

12.3.3 Stateless vs statefull services

Servizi stateless più facili da gestire a run-time.
Servizio stateless non ha stato nel micro-servizio, viene gestito esternamente salvandolo ad esempio su Redis (datastore esterno), in modo che sarà più semplice replicare il micro-servizio.
Servizio statefull ha stato incorporato nel servizio, in caso di scalabilità devo replicare anche lo stato e devo mantenerlo replicato. Buona norma andare a sviluppare quanto più possibile i servizi in modo stateless.

12.3.4 Integrazione

Comunicazione, dovrebbe essere sincrona o asincrona? I due pattern più usati sono RPC o code di messaggi. Ad eccezione di Go, RPC è comunicazione sincrona, mente code di messaggi asincrona. La scelta fra le due dipende

- Comunicazione sincrona: si usano richieste HTTP per interfacce REST o gRPC
- Comunicazione asincrona: integro middleware che supportano comunicazione asincrona (Kafka, RbbitMQ). Interazione può essere one-to-oneo one-to-many

Svantaggio della comunicazione sincrona: impatto sulla disponibilità dell'applicazione a micro-servizi, può avvenire solo se entrambi gli end-point sono attivi. Comunicazione asincrona può avvenire anche se altro host non è presente, messaggio viene memorizzato nella coda.
Due principali schemi per realizzare applicazione a micro-servizi in cui più componenti comunicano fra di loro:

- Orchestrazione: approccio centralizzato, in cui message broker o API gateway coordina interazione fra i differenti servizi. Vi è questa entità centralizzata che permette agli altri microservizi di interagire fra loro, tutto mediato dal broker intermedio.
- Coreografia: approccio decentralizzato, descrizione globale di come i servizi interagiscono fra di loro, ciascun servizio sa già con quali servizi interagire e come interagirci. I diversi micro-servizi scambiano direttamente i messaggi.

Paragone fra i due:

- Orchestrazione è single point of failure e bottleneck, disaccoppiamento più leggero e più traffico di rete e latenza
- Coreografia: servizi devono conoscere la coreografia, serve fare lavoro extra per monitoring e track dei servizi. Implementazione dei meccanismi è più complesso

12.4 Design pattern per micro-servizi

Esaminiamo alcuni design pattern (vedi ISPW) per micro-servizi, ne esistono numerosi

12.4.1 Circuit breaker

Problema: come evitare che una rete o servizio che fallisce (fallimento può essere del server, della VM, nella comunicazione di rete) causi fallimento in cascata di altri servizi?

Soluzione: interrompo il circuito tra il servizio che invoca ed il servizio che dovrebbe fornire la funzionalità. Introduco una sorta di micro-servizio proxy, il circuit breaker: se tutto funziona correttamente il circuit breaker fa da proxy. Se si verifica un errore, il circuito viene aperto, quindi l'interruttore è aperto e si disaccoppia chi invoca il servizio da chi lo offre. Circuit breaker tiene traccia del n° di invocazioni fatte al servizio, se invocazioni non hanno successo e numero di invocazioni consecutive che non hanno successo supera soglia, che di solito è configurata, circuit breaker apre interruttore per un certo periodo di tempo. Client saprà che servizio è inaccessibile, mentre circuit breaker proverà a ricontattare servizio fallito. Una volta che servizio torna ad essere accessibile, il circuit breaker richiude l'interruttore e quindi riprende il normale funzionamento della comunicazione tra il micro-servizio client e quello invocato. Se invece mirco-servizio invocato continua a fallire, viene nuovamente riaperto l'interruttore per un certo periodo di tempo.
Tipicamente n° tentativi che vengono effettuati dal circuit breaker è configurato staticamente, così come timeout, si può pensare a soluzioni in cui questi valori siano configurati dinamicamente con controllore adattativo.

12.4.2 Database per service

Problema: quale architettura di database usare per applicazione a micro-servizi?
Soluzione: avere un layer di persistenza realizzato dal database (può essere anche datastore noSQL) privato per singolo, accessibile solo tramite API realizzata appositamente per il singolo micro-servizio.
Suppongo di avere applicazione di e-commerce: un micro-servizio gestisce gli ordini ed uno gestisce gli utenti, ciascuno accederà in maniera privata al proprio DB, vuol dire almeno avere le proprie tabelle private a cui ciascun servizio accede (non per forza un intero DBMS a disposizione).
Vantaggi:

- Disaccoppiamento tra i servizi
- Posso usare il DB più indicato per il tipo di servizio: posso usare un DB relazionale o datastore a documenti per gestire l'ordine ed usare un datastore orientato a colonne per gestire i customer.

Svantaggi:

- È più complesso implementare le transazioni che coinvolgono più di un servizio, quindi molteplici database o tabelle. Per realizzare transazioni distribuite, c'è a disposizione altro pattern

12.4.3 Pattern Saga

Pattern più complesso e articolato, permette di gestire transazioni distribuite.
Problema: ogni servizio ha il suo DB, ma alcune transazioni coinvolgono più servizi, voglio semantica ACID, ovvero o tutti i micro-services riescono oppure c'è rollback.

Soluzione: ciascuna transazione è implementata come una saga, sequenza di transazioni locali. Ciascuna transazione locale aggiorna il DB con l'esito della transazione locale e pubblica il risultato oppure invia un trigger a quella successiva. Nel caso in cui una transazione locale fallisca: la saga fa l'undo dei cambiamenti locali.
Come coordinare la saga:

- Coreography: ogni transazione locale pubblica eventi che triggerano le transazioni locali di altri servizi
- Orchestration: un orchestratore dice ai partecipanti quale transazione locale deve eseguire

esempio di Saga con orchestrazione (usato anche pattern event sourcing): arriva richiesta di ordine per un e-commerce. Ciascun micor-servizio gestisce la propria tabella in maniera privata. L'ordine viene inoltrato dal micro-servizio degli ordini all'orchestratore del pattern Saga che avvia il coordinamento della transazione distribuita, andando a memorizzare ciò che sta facendo in un log privato. Quando l'inserimento va a buon fine, viene registrato il fatto che si arriva l'ordine, quindi servizio di ordine porta a compimento la sua transazione locale, viene salvato nella tabella ordini. Questo viene comunicato all'orchestratore, che memorizza nel suo log che il micro-servizio order e da il La al prossimo servizio. Micro-servizio di pagamento registra che il cliente deve pagare tot (c'è un event store in cui ogni servizio ha una tabella privata), quindi servizio memorizza nella tabella privata l'informazione e comunica all'orchestratore. Viene quindi attivata la 3° transazione, relativa al magazzino ed in questo caso non c'è disponibilità del prodotto richiesto. Quindi servizio fallisce, occorre fare annullamento delle transazioni. Viene registrata la transazione all'interno di un abort, ed ora bisogna fare rollback delle due transazioni precedenti, invierà ai due servizi una richiesta di compensazione per l'ordine e per il pagamento, in modo che questi cancellino dall'event store gli eventi relativi alle due transazioni. Quindi termina la Saga relativa alla gestione di questo ordine.

12.4.4 Monitoraggio di micro-servizi

Per servizi distribuiti a larga scala è difficile effettuare monitoring:

- Performance e latenza
- Monitoring di transazioni
- Root cause analysis: posso vedere effetto del ritardo su un determinato micro-servizio, ma magari probelma è su un micro-servizio che sta più un alto nella catena.
- Service dependency analysis
- Distributed context propagation

Due pattern per fare questo

12.4.5 Log aggregation

Problema: come fare a capire il comportamento di un'applicazione e fare troubleshooting?
Soluzione: introduco servizio di logging centralizzato che aggrega i log di ciascuna istanza di micro-servizio. Gli utenti (admin dell'applicazione) hanno in un repository centralizzato tutto il log, possono andare ad analizzare i singoli log e configurare sistema di alter sul log che triggera degli eventi.
Svantaggio: è tutto centralizzato su un singolo servizio e la gestione dei log può essere piuttosto complessa perché volume dei log può essere significativo.

12.4.6 Tracing distribuito

Servizio di strumentazione che

- Assegna a ciascuna richiesta esterna un unico ID esterno, in modo che ciascun micro-servizio può registrare nel suo log a quale specifico ordine era relativo un pagamento (sto sempre nell'esempio dell'e-commerce).
- L'id esterno viene passato a tutti i servizi che sono coinvolti nella gestione del servizio ...

(vedi slides)
Ci sono dei tool per il tracing distribuito, i framework principali per micro-servizi hanno un tracing già predisposto, o è possibile integrarli con dei tool ad hoc.
Tool più famoso è Dapper, tool di Google usato in production per tracing distribuito.
Ci sono poi vari tool open source.

12.4.7 Esempi di applicazioni a micro-servizi

Esempio 1: sock shop. Shop online per la vendita di calzini.
DIversi linguaggi di sviluppo, sono applicazioni poliglotte: in questo modo si sfruttano al meglio le varie capacità del linguaggio.
Alcuni servizi hanno il loro DB, inoltre cambia il tipo di datastore per i singoli servizi: avrò diverse caratteristiche adatte per la specifica funzionalità del micro-servizio: avrò ad esempio MongoDB (datastore noSQL orientato ai documenti) per ordini carrello etc..., MongoDB sava documenti, tipicamente in formato Json. In questo modo, posso ricercare ad esempio utenti per campi come l'indirizzo etc...
Tra il front-end e gli altri micro-servizi non c'è coda di messaggi intermedia, si sfrutta REST API in questo caso, che viene esposta da ogni servizio.
Per la gestione della consegna si usa una coda di messaggi.
In questo caso, costruito con SpringBoot, Go e Node.js, è applicazione cross-platoform, si può fare deploy su diversi orchestratori. Tutti i servizi comunicano con REST su HTTP.
Esempio 2: online boutique di Google, demo per Kubernetes. Composto da vari

micro-servizi: sono scritti in diversi linguaggi di programmazione, in particolare Go. Usato redis come DB di backend: datastore di tipo key-value, informazioni del carrello memorizzate in un aggregato che ha unique ID per singolo utente nella sessione di utilizzo. A differenza dell'esempio 2, in questo caso la comunicazione avviene con gRPC, che permette la comunicazione di end-point poliglotti.
Spiegato nella documentazione come effettuare il deploy su Kubernetes, altri framework usati tra cui Istio: supporta lo sviluppo di service mash, si integra con Kubernetes per ampliarne alcune feature, tra cui implementazione di soluzioni di load balancing per le repliche dello stesso servizio. OpenCensus: tracing, effettua logging per il singolo micro-servizio.

12.4.8 Timeline dei microservizi

Possiamo identificare 4 generazioni di microservizi

- 1° generazione: virtualizzazione basata su container, framework per service discovery (come Zookeeper) in modo da permettere la comunicazione con binding dinamico. Esempi di framework per service discrovery: Eureka ed etcd.
  Altra soluzione usata sono framework per il monitoraggio, tra cui ad esempio Prometeus ed InfluxDB (DB orientato alle time series). Inoltre orchestratori, per poter sviluppare applicazioni multi-container su più nodi.

- 2° generazione: Servizi di discovery e librerie per la comunicazione tolleranti ai guasti, nella 1° generazione tolleranza ai guasti è a carico dello sviluppatore.
  Esempi di framework per fault tollerance: Consul, Finagle, Hystrix (sviluppata da Netflix), implementa già dei meccanismi del pattern circuit breaker nella libreria, che permette di evitare failure in cascata.

- 3° generazione: tecnologie basate su sidecar, incapsula in un sidecar alcune feature, messi vicino ad ogni micro-servizio. In particolare, sidecar incapsula tecnologie per discovery, fault tollerance e routing del traffico, peremtte riuso di codice e evita che lo sviluppatore debba scrivere queste funzionalità

- 4° generazione: serverless computing: sviluppo solo singole funzionalità, senza dovermi occupare della gestione dell'infrastruttura sottostante. Ambiente totalmente gestito dal cloud provider, o dai framework opne source di tipo FaaS (Function as a Service).

12.5 Serverless computing e FaaS

(slides) Caratteristiche del serverless computing:

- Le risorse di computazione usate per servire le richieste sono effimere: tempo di vita delle risorse è breve, può essere breve quanto la durata della singola funzione
- Per le singole funzioni l'elasticità è gestita completamente dal cloud privider
- Utente può specificare solo dei requisiti su quelle che sono la quantità di risorse allocabili per le singole funzioni. Alcuni cloud provider permettono solo di scegliere la memoria RAM e danno in proporzione la CPU (vedi servizio Lambda di AWS)
- Vero pay-per-use: si paga solo l'effettivo tempo di computazione utilizzato.
- È event driven: funzioni attivate in seguito di un evento, che può essere generato dalla richiesta di un utente o da un trigger che arriva da un altro servizio di storage o da un sistema a code di messaggi
- Semplifica il processo di deployment in production: scalabilità, capacità di planning etc..

I maggiori cloud provider offrono FaaS: IBM, Google, Azure, Amazon etc...
Limiti attuali: serverless è un trend recente, presenta alcune limitazioni:

1. Le funzioni, al momento dell'invocazione, se non esiste container o micro-VM che andrà ad eseguire la funzione, sperimentano tempo di latenza bello greve. Bisogna istanziare container e contesto (tutti i file di librerie etc necessarie all'app) della funzione o la micro-VM (vedi Amazon firecraker). Per evitare che ci siano fenomeni d invocazione continua di funzioni,ed ognuna sperimenti il cold start, una volta che è stato effettuato deployment, il cloud provider mantiene le risorse in stand-by. Vi è però limitazione, tipicamente c'è lifetime all'ambiente di esecuzione della funzione, che varia a seconda del provider, ma è dell'O(Min), per Amazon ad esempio 15 min.
2. Scelta del linguaggio: cloud provider supportano solo un determinato sotto-insieme dei linguaggi, ho dei vincoli a seconda del cloud provider. A seconda del linguaggio, cambiano le prestazioni.
3. Limiti delle risorse: bottleneck legati alle operazioni di I/O, nel momento in cui bisogna effettuare operazioni di I/O, dipenderà da dove è stata deployata l'applicazione, all'interno del data center selezionato.
4. Vendor lock-in: problema importante del cloud, cambia l'API esposta dai singoli provider per l'invocazione delle funzioni
5. Limiti nella comunicazione tra funzioni differenti
6. Non c'è hardware specializzato: a differenza di EC2, in cui ad esempio possiamo avere GPU, nel caso del FaaS non è possibile scegliere hardware specializzato su cui effettuare deployment della funzione; per alcuni provider viene offerto slice di CPU hyperthreaded

Altro aspetto rilevante è la possibilità di comporre funzioni serverless, ovvero compongo fra loro funzioni semplici componendo pipe di funzioni, con la possibilità di usare nel workflow costrutti specifici, ad esempio invocazioni a seconda dei risultati delle funzioni precedenti, gestire pattern di esecuzione. Diversi framework per FaaS, per realizzarlo anche in ambiente cloud privata:

- Apache OpenWhisk
- OpenFaaS
- Nuclio
- Fission
- ...

Differiscono per il linguaggio supportato, per la possibilità dello scale to 0, ovvero di avere le funzioni pari a 0 come numero di repliche o avere almeno una istanza sempre attiva di ciascuna funzione invocata dagli utenti.

12.5.1 OpenWhisk

Open source, distribuito. SI basa su container Docker, che saranno orchestrati sul singolo nodo con Docker-compose o con Kubernetes, può essere eseguito su Mesos.
I developers devono scrivere la logica delle funzioni, che verranno shcedulate in modo dinamico da OpenWhisk ed attivate in corrispondenza di trigger specifici, ad esempio da eventi esterni o da richieste HTTP. Le funzioni sono componibili, c'è composer ad hoc che permette di gestire la composizione di funzioni usando una API messa a disposizione. OpenWhisk ha al suo interno un proxy, che è Nginx, per fare da dispatcher delle richieste verso gli altri componenti. OpenWhisk usa Kafka per gestire la distribuzione degli eventi e CouchDB come datastore noSQL per gestire aspetti di sicurezza delle funzioni.

12.5.2 OpenFaaS

Framework che utilizza diversi framework open source, tra cui

- Prometeus: per fare scale in/out
- NATS: sistema pub/sub
- Gateway: nodo centralizzato che fa da smistatore di richieste. Questo componente centralizzato è un po' il limite di tutte le piattaforme FaaS open source.

13 Sincronizzazione e coordinazione nei Sistemi Distribuiti

In un SD, c'è necessitò di gestire il tempo: se considero applicazione distribuita e SD ci sono aspetti di comunicazione nel SD, obiettivo è quello di far comunicare i componenti fra loro in modo da portare a termine una computazione. Comunicazione avviene con scambio di messaggi, molti degli algoritmi distribuiti o delle computazioni distribuite richiedono sincronizzazione, ovvero richiedono una nozione comune di tempo per i diversi processi in esecuzione sui vari nodi. Tempo è fattore critico nel SD: in un sistema centralizzato, è possibile stabilire l'ordinamento fra gli eventi basandosi sul clock del nodo centrale.
IN un SD, non è possibile avere un unico clock fisico comune a tutti i nodi del sistema, ma comunque l'obiettivo è vedere la computazione portata avanti come un ordine totale di eventi. Soluzioni per affrontare il probelma di gesitoen del tempo sono due

- Nozione di tempo fisico: sincronizzo i clock fisici dei diversi nodi che compongono il SD. In ciascun nodo, in particolare la aprte del middleware di sincronizzazione aggiusterà il valore del proprio clock fisico in modo coerente.
- Nozione di tempo logico: sincronizzo gli orologi logici, Lamport ha dimostrato come in un SD non sia necessaria la sincronizzazione degli orologi fisici. Tempo logico è concetto fondamentale nei SD

13.1 Modello della computazione

Componenti del SD: N nodi e canali di comunicazioni, ogni processo p_i ($1 \leq i \leq N$) genera una sequenza di eventi

- Eventi interni: eventi che determinano un cambiamento dello stato del processo
- eventi esterni: eventi con cui il processo comunica tramite scambio di messaggi con gli altri processi, send or receive di messaggio.

Notazione per gli eventi: e_i^k: k-esimo evento evento generato dal nodo p_i.
L'evoluzione della computazione viene visualizzata su un diagramma spazio-tempo: indico con la freccia ed il pedice i l'esistenza di una relazione di ordinamento tra due eventi di uno stesso processo: e $\rightarrow_i$ e' se e solo se e è avvenuto prima di e' nel processo i.
Ogni processo etichetta gli eventi con un timestamp, in modo da realizzare una storia globale del sistema.

13.1.1 SD sincroni ed asincroni

Differenze fra i due:
SD sincrono, 3 proprietà

- esistono dei vincoli sulla velocità di esecuzione di ciascun processo che fa parte del SD sincrono. Il tempo di esecuzione di ciasun passo della computazione è limitato all'interno di una finestra con upper e lower bound
- Ciascun messaggio trasmesso su un canale di comunicazione è ricevuto in un tempo limitato (es in max 200 ms)
- Ciascun processo ha un clock fisico ha un tasso di scostamento (clock fisici non sono ideali), tasso di accelerazione o decelerazione rispetto ad un clock reale si chiama drift rate, reale e conosciuto.

Sd asincrono: non ha vincoli sulla velocità di esecuzione di ciascun processo, sul ritardo di trasmissione dei messaggi e sul tasso di scostamento dei clock.

13.1.2 Soluzioni per sincronizzare i clock

Prima soluzione: tentare di sincronizzare i clock fisici con una certa approssimazione, e quindi posso usarlo per far si che un processo etichetti i suoi eventi con timestamp, che risulterà sincronizzato con gli altri eventi degli altri processi. Problema: se SD è asincrono, la certa approssimazione non si riesce più a mantenere limitata i quanto non vale il 3° vincolo che vale nei SD sincroni.

13.2 Clock fisico

All'istante di tempo reale t, il SO legge il tempo dal clock hardware $H_i(t)$ del computer, Produce quindi il clock software $C_i(t) = aH_i(t) + b$ che approssimativamente misura l'istante di tempo fisico per il determinato processo eseguito su quel nodo. Il valore $C_i(t)$ è tipicamente numero a 64 bit che fornisce il tempo in nanosecondi trascorsi fino a quel punto.
In generale il clock non è completamente accurato, può essere diverso da t, ma se si comporta abbastanza bene può essere usato come timestamp per gli eventi del nodo.
Per poterlo usare, il tempo che intercorre tra due aggiornamenti deve essere: $T_{risoluzione} < \Delta_T$ tra i due eventi.
In un SD i clock fisici posso avere diversi valori

- Skew: differenza istantanea fra il valore di due qualsiasi clock
- Drift: i clock contano il tempo con frequenze differenti, quindi nel tempo divergono rispetto al tempo reale
- Drift rate: differenza per unità di tempo di un clock rispetto ad un orologio reale. esempio: clock con drift rate di 2μsec/sec vuol dire che il clock si discosta dal valore effettivo che dovrebbe misurare di 2μsec per ogni secondo.

In un orologio al quarzo si devia di circa un secondo in 10/11 giorni. Per clock ad alta precisione abbiamo drift rate di 10^{-7} o 10^{-8} sec/sec
Anche se i differenti clock di un SD vengono sincronizzati inu n certo istante, a

causa del drift rate, questi saranno nuovamente de sincronizzati, occorre quindi ri sincronizzare.

13.2.1 UTC

Universal Coordinated Time, standard internazionale per mantenere il tempo. È basato sul tempo atomico, ma occasionalmente viene corretto usando il tempo astronomico.
1 sec = tempo impiegato dall'atomo di Cesio 133 per eseguire un bel numero di transizioni di stato.
I clock fisici che usano oscillatori atomici sono i più precisi ed accurati.
Output dell'orologio possono essere inviati in broadcast da stazioni radio su terra e da satelliti. I nodi con ricevitori possono sincronizzare i loro clock con questi segnali, a seconda della fonte del segnale ci sarà una maggiore o minore accuratezza, se arriva da satellite è più accurato.

13.2.2 Sincronizzazione del clock fisico

Come sincronizzare i clock fisici con gli orologi del mondo reale? Sincronizzazione esterna: se i clock sono sincronizzati con una sorgente di tempo S, questo in modo che $\forall$ C_i: $|S(t) - C_i(t)| \leq \alpha$, con $\alpha > 0$.
Sincronizzazione interna: i clock sono sincronizzati l'uno con l'altro. Se considero il clock C_i e C_j questi sono sincronizzati se $|C_i(t) - C_j(t)| \leq \pi$ per tutte le coppie di clock. (vedi slides)
Sincronizzazione interna ed esterna non si implicano a vicenda: se c'è sincronia esterna, allora saranno anche sincronizzati internamente, ma non vale il viceversa, perché tutti i clock che sono sincronizzati internamente potrebbero deviare rispetto alla fonte S esterna.
Se l'insieme dei processi è sincronizzato internamente con precisione α, allora lo è internamente con precisione 2α.
Clock hardware H è corretto se il suo drift rate si maneitene in un intervallo $\rho > 0$. Se clock H è corretto, l'errore che si commette nel misurare un intervallo di istanti reale [t, t'] (t' > t) è limitato (vedi slides).
Siccome i diversi clock fisici possono essere reaizzati con materiali differenti e quindi avere diverso drift rate, vanno periodicamente sincronizzati. Se due clock hanno lo stesso tasso di scostamento massimo da UTC, che sia pari a ρ, quindi dopo Δt si saranno scostati di 2ρ. Per garantire che i due clock non differiscano per più di δ, i clock vanno sync ogni (vedi slides).

13.2.3 Sincronizzazione interna in SD sincrono

Algoritmo di sincronizzazione interna tra due processi

- Processo 1 manda il suo clock t locale al processo due in un messaggio
- Processo 2 imposta il suo clock locale al tempo contenuto nel messaggio + il tempo di trasmissione. Vale il vincolo sul tempo di trasmissione, che

sarà limitato (ma non è noto) in t_{min} e t_{max}. u = (t_{max} - t_{min}), se il processo due imposta il suo clock a t + $\frac{t_{max}-t_{min}}{2}$ il lower bound ottimo sullo skew sarà pari ad $\frac{u}{2}$

Si può generalizzare per sincronizzare tra n processi e si vede che l'ottimo sullo skew sarà pari a $\frac{u}{1-\frac{1}{N}}$. IN un SD sincrono, l'algoritmo non è più utilizzabile, in questo caso $t_{trasm} = t_{min}$ + x, con x ≥ 0, ma non noto.

13.3 Sincronizzazione mediante time service

Time service può fornire l'ora con precisione, server dotato di ricevitore UTC o clock accurato.
Gruppo di rpocessi che deve sync usa il time service, che può essere realizzato o in modo centralizzato o può essere distribuito.
Time service centralizzato

- Request-driven: algoritmo di Cristian
- Broadcast-based: algoritmo di Berkeley-UNix

Time service distribuito: NTP

13.3.1 Algoritmo di Cristian

Un time server S, centralizzato e passivo riceve il segnale da una sorgente UTC. Un processo p richiede il tempo con un messaggio m_r e riceve t nel messaggio m_t da S.
P imposta il suo clock a t + $\frac{T_{round}}{2}$, dove T_{round} è l'RTT misurato da p. Osservazioni:

- Il tempo di elaborazione sul time server è prossimo sa 0, visto che imposto il tempo come t + $\frac{T_{round}}{2}$
- Algoritmo è adatto nel caso in cui valore di RTT è basso ,quindi in rete a bassa latenza (LAN) o introduco errore che può essere importante
- Un singolo server potrebbe guastarsi, come soluzione si può introdurre un gruppo di time server sincronizzati
- Se time server viene corrotto, invia valore errato del clock e l'algoritmo non ne tiene conto

È un algoritmo di sincronizzazione esterna, calcolo l'accuratezza offerta:

- Caso 1: S non può mettere t in m_t prima che sia trascorso un tempo minimo dalla trasmissione del messaggio di chiesta da parte del processo p, dove il tempo min è il tempo che trascorre nella trasmissione del messaggio da p ad S.
- Caso 2: S non può mettere il valore di t nel messaggio di risposta dopo il momento in cui m_t arriva a t meno min.

Calcolo il tempo S quando m_t arriva a p è compreso in [t + min, t + T_{round} - min], ampiezza dell'intervallo è T_{round} - 2mim; accuratezza sarà $\leq$ +-($\frac{T_{round}}{2}$), permette una certa accuratezza solo se T_{round} è piccolo

13.3.2 Algoritmo di Berkeley

Sincronizzazione interna di un gruppo di server, di cui uno è il master e gli altri sono i worker. Master fa da time server attivo e coordina la sincronizzazione. All'Avvio dell'algoritmo, master richiede in broadcast il valore del clock, una volta ricevuti i valori dei clock usa l'RTT per stimare i valori dei block dei worker, usando criterio simile all'algoritmo di Cristian: $d_i = (\frac{C_M(t_1)+C_M(t_3)}{2} - C_i(t)$, quindi manderà a tutti i nodi il valore del clock che devono impostare per essere sincronizzati fra loro.
Se il valore correttivo richiede un salto all'indietro, il nodo che lo riceve rallenterà in modo da sincronizzarsi (altrimenti i timestamp vecchi verrebbero riconsiderati): rallentare un clock vuol dire quindi mascherare una serie di interrupt, il numero di interrupt mascherati è pari al tempo di slow down diviso il periodo di interrupt del processore.
Precisione dell'algoritmo di Berkeley dipende da un RTT nominale massimo, il master non considera valori di clock associati ad RTT superiori al massimo.
Algoritmo è più tollerante ai guasti: se master crasha, possiamo usare algoritmo di elezione per eleggere nuovo master, inoltre è anche più robusto per i valori errati dei clock dei worker (che possono farlo in modo malevolo), in quanto questo può considerare un sottoinsieme dei valori dei worker scartando gli outlayer (valori che si discostano troppo dalla media)

13.3.3 NTP

Netowrk Time Protocol, usato in Internet. Offre sync esterna accurata rispetto ad una sorgente UTC. È un servizio configurato ed attivo sul computer (demone attivo). Aspetto fondamentale di NTP: il time server è distribuito, con architettura composta da molteplici time server, architettura è altamente disponibile e scalabile. Architettura gerarchica, molteplici time server ridondanti e diversi path di comunicazione.

- Time server del primo livello sono i server primari connessi a sorgenti UTC
- Livello due ci sono server secondari, sincronizzati rispetto ai primari
- Server foglia: eseguiti sulle macchine degli utenti, che si sync rispetto ai server secondari.

Le sorgenti di tempo sono autenticate, quindi NTP soddisfa requisiti di sicurezza. Architettura è altamente disponibile, perché in seguito a fallimenti si può riconfigurare la gerarchia: se server primario perde la connessione diventa server secondario, mentre se server secondario perde connessione si collega ad altro server di livello 1. È un sistema auto-adattativo che si riconfigura da solo. Supporta 3 modi di sync, usano tutti UDP perché è fondamentale evitare latenze

- Multicast: server sono all'interno di LAN ad alta velocità che invia il suo tempo in multicast agli altri, che impostano il valore ricevuto assumendo un certo ritardo di trasmissione.
- Procedure call: un server accetta richieste da altri computer (come nell'algoritmo di Cristian). L'accuratezza è maggiore rispetto al multicast, inoltre è utile se non è disponibile il multicast.
- Simmetrico: coppie di server scambiano messaggi contenenti informazioni sul timing, l'accuratezza è molto alta per i livelli alti della gerarchia

Scambio in modo simmetrico: in particolare usato nei livelli altri della gerarchia (nel livelli basi si usano gli altri due). I server si scambiano coppie di messaggi (m,m') per migliorare l'accuratezza della loro sync.
Ogni messaggio NTP contiene timestamp di eventi recenti

- Tempo locale di send (T_{i-1}) del messaggio m'
- Tempo locale di send (T_{i-3}) e di receive (T_{i-2}) del messaggio precedente m.

Il ricevente di m' registra il tempo locale T_i (conosce quindi una 4-pla di valori), il tempo tra l'arrivo di m e l'invio di m' può essere non trascurabile.
Accuratezza: Per ogni coppia di messaggi m ed m', NTP stima l'offset o_i tra i due clock. Assumendo che:

- o: offset reale del clock di B rispetto ad A
- t e t': tempi di trasmissione di m ed m' abbiamo $T_{i-2} = T_{i-3} + t + o$ e $T_i = T_{i-1} + t' + o$

(vedi slides).
Server NTP si scambiano queste coppie di messaggi e usano algoritmo di filtraggio statistico sulle 8 coppie (o_i, d_i) più recenti, scegliendo come stima di o il valore di o_j corrispondente al minimo d_j. Applicano poi un algoritmo di selezione dei peer per modificare eventuale il peed da usare per sincronizzarsi. Accuratezza è di circa 10 ms su Internet e 1 ms in LAN.