Sistemi distribuiti e cloud computing

October 30, 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.

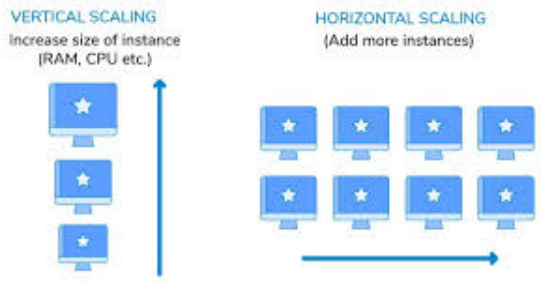
2.3 Apertura

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.

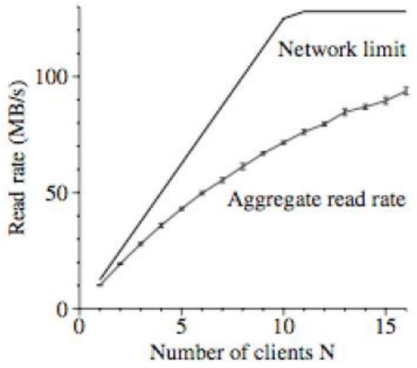
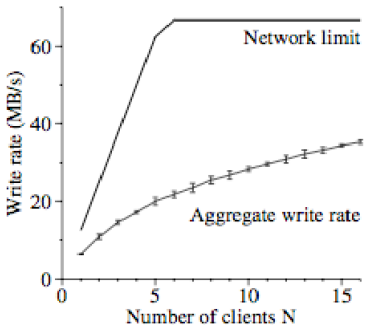


Fig 1 Read Performance

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:

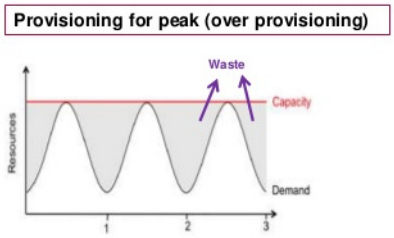
- pay-per-use: prezzo in base al tempo di utilizzo
- fixed: prezzo mensile fisso
- spot: prezzo variabile per le risorse cloud, guidato dalla domanda di mercato. Il provider può togliere la risorsa improvvisamente.

3.5 Uso del cloud

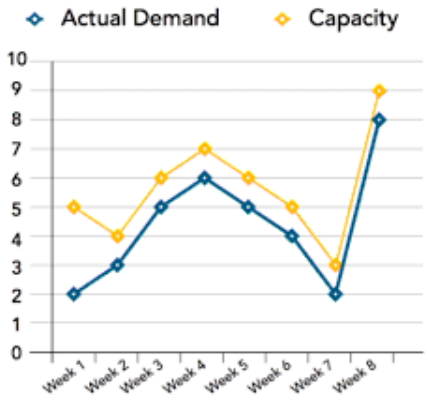
Un report annuale mostra come per le organizzazioni risulta difficile gestire i costiper le risorse cloud. Capure bene come usarlo è cruciale, la maggiora parte delle organizzazioni opta per una soluzione multi-cloud: i 3 principali provider sono Amazon, Google e Azure, i PaaS più usati sono DBaaS e 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:

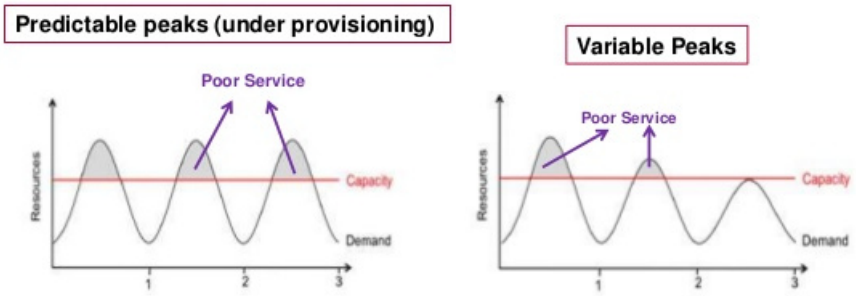


In questo caso, ho una parte delle risorse sprecate



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:



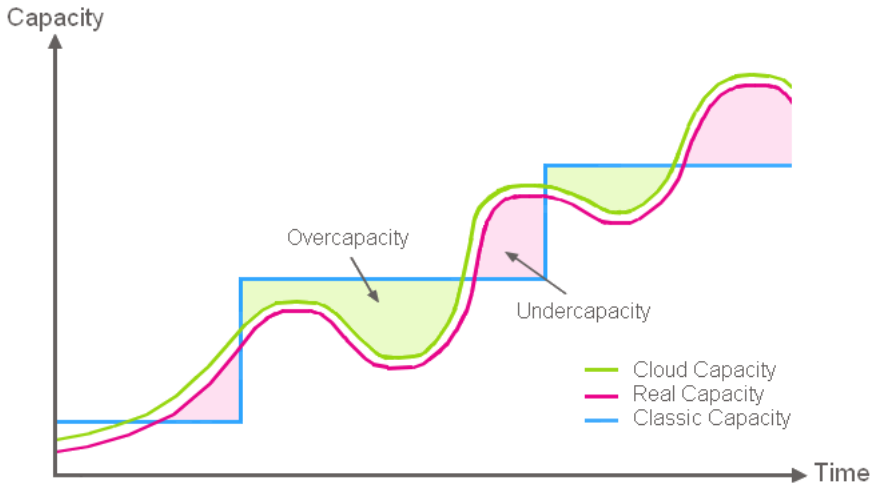
Qui, in caso di sovraccarico, perdo rispettivamente revenue e clienti.
Con il cloud voglio sfruttare al meglio l'elasticità, ovvero il grado con cui il sistema sa adattarsi ai cambiamenti di carico, andando a de/allocare risorse in maniera automatica in modo che le risorse disponibili siano quanto più 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:

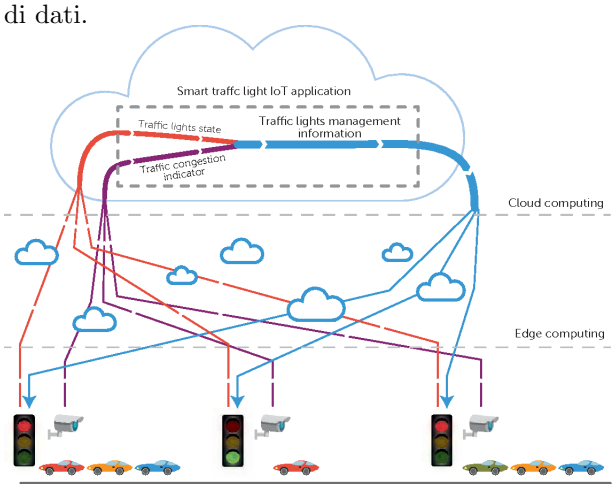
1. design: tipo e capacità delle risorse cloud
2. valutazione performance: verifica se l'applicativo è in accordo con i requirement di performance, mediante applicazioni di monitoring del workload e dei parametri di performance
3. refinement: considerazione alternative per scaling, interconnessione dei 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 q tale che $FT_p[j] \leq k \leq FT_p[j+1]$, ovvero cerca di identificare chi è il nodo che conosce che è più vicino alla 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^bN). 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 esplicito 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 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.