**INTRODUZIONE AI SISTEMI DISTRIBUITI**

**Legge di Metcalfe**Il valore di una rete di telecomunicazioni è proporzionale al quadrato del numero degli utenti connessi al sistema.

**Definizioni di sistema distribuito**1) Van Steen & Tanenbaum: un sistema distribuito è una collezione di elementi di computazione autonomi che appare agli utenti come un singolo sistema coerente.  
2) Coulouris & Dollimore: un sistema distribuito è un sistema i cui componenti si trovano in computer interconnessi in rete e comunicano tra loro solo scambiandosi messaggi.  
3) Lamport: un sistema distribuito è un sistema in cui la failure di un componente di cui neanche si sapeva l’esistenza può rendere il proprio computer inutilizzabile.

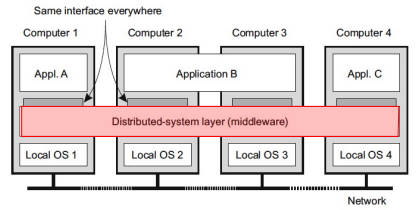
**Vantaggi di un sistema distribuito**- Permette di condividere risorse.  
- Riduce i costi.  
- Permette di risolvere problemi di dimensioni maggiori.  
- Supporta la Quality of Service (QoS).  
- Migliora le prestazioni, la sicurezza, la disponibilità e l’affidabilità.

NB: non confondere la disponibilità con l’affidabilità.  
-> **Disponibilità**: è la probabilità di trovare un server attivo (disponibile) nel momento in cui viene contattato per usufruire un servizio:  
-> **Affidabilità**: è la probabilità che, se il servizio è operante al tempo 0 (ovvero quando viene invocato), continua a essere funzionante anche al tempo t.

**Caratteristiche di un sistema distribuito**- Concorrenza, che, a differenza dei sistemi entralizzati, non si può fare a meno di gestire.  
- Assenza di un clock globale: piuttosto si hanno tanti clock fisici non necessariamente sincronizzati tra loro.  
- Fallimenti parziali e indipendenti ad esempio di qualche nodo o di qualche cavo di rete.

**Challenge per un sistema distribuito**1) Eterogeneità: i sistemi possono essere eterogenei:  
- Per la connessione di rete.  
- Per la capacità computazionale.  
- Per il sistema operativo.  
- Per i linguaggi di programmazione usati.  
- A causa di molteplici implementazioni effettuate da parte di più sviluppatori differenti.

Soluzione: introduzione di un **middleware**.  
Il middleware è uno strato software al di sopra dei sistemi operativi che fornisce un’astrazione atta a mascherare l’eterogeneità dei servizi sottostanti. Contiene componenti comuni alle diverse applicazioni.



Il middleware inoltre offre diversi servizi:  
- Comunicazione.  
- Gestione delle transazioni.  
- Composizione dell’applicazione.  
- Affidabilità.

2) Trasparenza alla distribuzione: è quella proprietà per cui un sistema distribuito appare come un tutt’uno agli utenti. Esistono diversi tipi di trasparenza:  
- **Trasparenza all’accesso**: il sistema nasconde le differenze nella rappresentazione dei dati e la modalità con cui le risorse vengono accedute.  
- **Trasparenza alla locazione**: il sistema nasconde dove le risorse sono localizzate. Insieme alla trasparenza all’accesso costituisce la network transparency.  
- **Trasparenza alla migrazione**: il sistema nasconde il fatto che le risorse possono spostarsi verso locazioni differenti (anche a runtime) senza inficiare sul funzionamento del servizio. Un esempio pratico è la redirezione in http.  
- **Trasparenza alla replicazione**: il sistema nasconde l’esistenza di molteplici repliche della medesima risorsa. In tal modo le repliche devono avere tutte lo stesso nome.  
- **Trasparenza alla concorrenza**: il sistema nasconde il fatto che le risorse possono essere condivise da più utenti indipendenti.  
- **Trasparenza ai fallimenti**: il sistema nasconde i fallimenti e gli eventuali ripristini.

Ambire a una trasparenza completa (soddisfacendo tutti e sei i tipi di trasparenza) spesso risulta eccessivo: ad esempio, le latenze di comunicazione non possono essere sempre nascoste, così come i fallimenti non sono sempre copribili (è difficile distinguere un nodo non funzionante da uno molto lento); inoltre, mantenere i dati replicati perfettamente consistenti e identici tra loro è molto costoso per cui, nella realtà, si effettua piuttosto un trade-off tra consistenza e performance del sistema.

3) Apertura: un sistema è aperto se è in grado di interagire con servizi offerti da altri sistemi indipendentemente dall’implementazione. Perciò, il sistema dovrebbe essere conforme a delle interfacce difinite tramite IDL (Interface Definition Language).  
Per implementare un sistema distribuito aperto è opportuno separare le politiche dai meccanismi.  
- **Esempio di meccanismo**: caching dei dati.  
- **Esempio di politica**: per quanto tempo una risorsa può rimanere in cache?  
In questo modo però, devono essere configurati molti parametri. A tal proposito, la soluzione consiste nel ricorrere ai sistemi auto-configurabili.

4) Scalabilità: è quella proprietà per cui un sistema distribuito mantiene un livello adeguato di prestazioni anche a fronte di un aumento di:  
- Numero di utenti e/o di processi (scalabilità rispetto alla dimensione).  
- Distanza massima tra i nodi (scalabilità geografica).  
- Numero di domini amministrativi (scalabilità amministrativa).

La scalabilità rispetto alla dimensione (la più sfruttata nella pratica) può essere attuata in due direzioni:  
-> **Scalabilità verticale (scale up)**: utilizzo di risorse più potenti (è una soluzione con costo esponenziale).  
-> **Scalabilità orizzontale (scale out)**: aggiunta di risorse della stessa capacità di quelle già presenti (è una soluzione con costo lineare).

Vediamo alcune tecniche per scalare un sistema:  
- Nascondere la latenza della comunicazione (utilizzando una comunicazione asincrona, che però non è sempre possibile).  
- Spostare parte della computazione sul client.  
- Partizionare i dati e la computazione su molteplici nodi (principio del divide et impera). Il DNS è un esempio di questo approccio.  
- Replicare le risorse e i dati.

**Tipi di sistemi distribuiti**1) Sistemi di computing distribuito ad alte prestazioni: si classificano a loro volta in:  
-> Cluster computing  
-> Cloud computing  
-> Fog/edge computing

Cluster = insieme di server connessi in una rete ad alte prestazioni, i cui obiettivi sono l’High Performance Computing (HPC) e l’High Availability (HA).  
L’architettura tipica del **cluster computing** è quella del master-worker: il nodo master suddivide il carico di lavoro e lo distribuisce tra i nodi worker.

Il cluster computing è una pietra miliare per il **cloud computing**. Il cloud, però, è disponibile a chiunque e su una scala geografica più ampia.

Il trend attuale consiste nello spostare alcune delle funzionalità meno onerose ai bordi della rete (**fog/edge computing**).

2) Distributed information system: tra questi figurano i **transaction processing system**, che effettuano le operazioni di read e write incapsulate in delle transazioni distribuite (con una politica all-or-nothing), che sono composte da sotto-transazioni eseguite su differenti database server.  
La gestione delle transazioni distribuite avviene con un’architettura master-worker, dove il master è il Transaction Processing (TP) Monitor, che è responsabile del coordinamento dell’esecuzione delle transazioni.

3) Distributed pervasive system: sono composti da molteplici nodi piccoli, mobili, alimentati a batteria e spesso incorporati in un sistema più grande.  
Un esempio di questo tipo di sistemi sono le **reti di sensori**, che possono avere:  
- Uno schema centralizzato, in cui i singoli sensori inviano i dati a una locazione centralizzata per processarli.  
- Uno schema distribuito, in cui i singoli sensori possono effettuare storage e processamento di dati, seppur in modo limitato.

**INTRODUZIONE AL CLOUD COMPUTING**

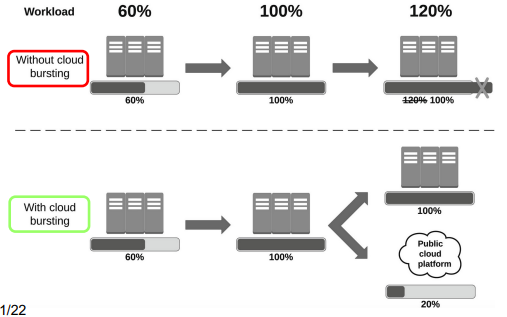
**Paradigmi di computing**Come si può realizzare un servizio che gestisca milioni di richieste al giorno, faccia fronte a picchi improvvisi di carico e memorizzi exabyte (1018 B) di dati?  
Una risposta vecchia e parziale a questa domanda risiede nei primi quattro paradigmi di computing:  
- **Utility computing** (forma di servizio informatico in base al quale la società che fornisce il servizio addebita la quantità di utilizzo).  
- **Grid computing** (infrastruttura di calcolo distribuito utilizzata per l’elaborazione di grandi quantità di dati mediante l’uso di una vasta quantità di risorse).  
- **Autonomic computing** (sistema in grado di autogestirsi senza l’intervento umano).  
- **Software as a Service** – SaaS (modello di servizio del software dove il fornitore sviluppa e gestisce un’applicazione web, mettendola a disposizione dei clienti via Internet previo abbonamento).

Nel 2006 però nacque il **cloud computing**, che rappresenta un passo avanti verso la vera soluzione del problema della scalabilità. In particolare, il cloud computing rappresenta lo spostamento di computazione, storage e coordinazione dal singolo server o data center all’Internet.

**Caratteristiche del cloud computing**Esiste una miriade di definizioni di cloud computing, dalle quali è possibile estrarre alcune caratteristiche essenziali:  
- **On-demand & self-service**: le risorse cloud possono essere fornite on-demand dagli utenti, senza dover richiedere interazioni col cloud service provider.  
- **Ampio accesso alla rete**: le risorse cloud sono in Internet e possono essere accedute indipendentemente dalla piattaforma.  
- **Elasticità**: è la capacità di ottenere e rilasciare le risorse cloud al momento del bisogno. A tal proposito, le risorse cloud devono poter essere allocate (scale-out) o deallocate (scale-in) velocemente.  
- **Resource pooling**: molti utenti possono usare le stesse risorse hardware (multi-tenancy).  
- **Virtualizzazione** delle risorse.  
- **Prezzo calibrato all’utilizzo** (modello pay-per-use).  
- **Servizio misurato**: molto spesso viene utilizzato il Service Level Agreement (SLA) come metrica di riferimento.

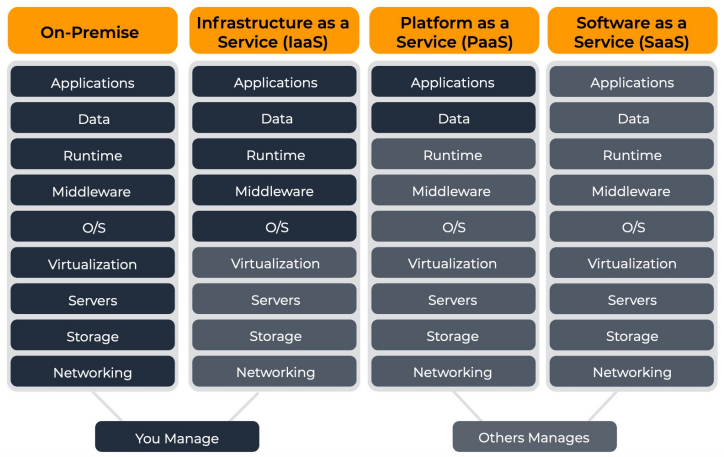
**Modelli di deployment nel cloud**1) Public cloud: l’infrastruttura cloud è resa disponibile da un cloud provider ed è utilizzabile da chiunque. È gestita da un’azienda o un’organizzazione e i suoi servizi possono essere gratuiti oppure a pagamento.

2) Private cloud: l’infrastruttura è ad uso esclusivo di un’organizzazione ed è gestita dall’organizzazione stessa oppure da una terza parte. Offre una maggiore sicurezza e una maggiore facilità di personalizzazione dei servizi. Tuttavia, il suo costo è più elevato e la sua scalabilità è più laboriosa perché le risorse utilizzabili sono limitate a quelle messe a disposizione per l’azienda.

3) Hybrid cloud: l’infrastruttura è data dalla composizione di più infrastrutture (private o pubbliche) distinte. Ha diversi vantaggi:  
- Permette il bilanciamento di risorse e costi.  
- Permette la differenziazione della privacy, con i dati normali mantenuti nel cloud pubblico e i dati sensibili mantenuti nel cloud privato.  
- Migliora la disponibilità del servizio: se va in crash il cloud privato, rimane possibile utilizzare quello pubblico.  
- Migliora la scalabilità, mediante l’uso congiunto di cloud privato e pubblico per gestire carichi di lavoro variabili: i picchi insostenibili per il cloud privato vengono gestiti tirando in ballo il cloud pubblico (cloud bursting).  
  
  
  
**Modelli di servizio**  
1) Infrastracture as a Service (IaaS): le risorse fisiche (capacità di processamento, capacità di storage, ecc.) sono gestite dal provider e sono esposte come servizi.  
Il customer (ovvero l’utente) non ha controllo sull’infrastruttura cloud sottostante, ma può scegliere l’area geografica, il sitema operativo, la dimensione della macchina virtuale e così via.



2) Platform as a Service (PaaS): consiste in delle piattaforme per sviluppare, eseguire e gestire applicazioni scalabili senza dover preoccuparsi dell’infrastruttura sottostante.  
Il customer si occupa solo dello sviluppo e del testing delle applicazioni ed è vincolato ai framework, linguaggi di programmazione e tool offerti dal provider.

3) Software as a Service (SaaS): le applicazioni sono rese disponibili agli utenti tramite Internet.  
Il customer non ha alcun controllo sull’infrastruttura e sulle impostazioni dell’applicazione (è tutto controllato dal provider).  


**Modelli di tariffazione**1) Pay-per-use: il prezzo dipende dal livello di utilizzo del servizio.  
2) Tariffazione fissa: gli utenti devono pagare periodicamente una quota fissa per l’usufruizione del servizio.  
3) Spot: il prezzo dipende dalla domanda e dall’offerta che il provider sta avendo. Il servizio è interrompibile dal provider in qualunque momento nel caso in cui le risorse disponibili scarseggiano. Perciò, la modalità spot è possibile da utilizzare solo se si devono portare a termine dei compiti non critici (senza vincoli temporali).

**Multi-cloud**Consiste nell’uso concorrente di molteplici ambienti cloud. Presenta diversi vantaggi:  
- Riduce i rischi di vendor lock-in (situazione di dipendenza tra customer e provider in cui il customer non può richiedere il servizio a provider differenti).  
- Migliora i costi.  
- Fornisce una maggiore flessibilità.  
- Migliora la disponibilità.  
- Differenzia la privacy tra dati sensibili e dati normali.  
- Migliora la distribuzione geografica.

Tuttavia, il multi-cloud è difficile da gestire perché:  
-> Richiede strumenti che automatizzino la divisione del carico di lavoro.  
-> Il monitoraggio delle prestazioni deve riguardare molteplici cloud provider.  
-> L’utilizzo e i costi delle risorse devono essere riassunti e predetti.

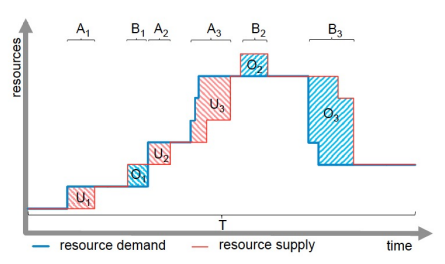
Comunque sia, il multi-cloud può utilizzare **l’Infrastructure as Code** **(IaC)** per automatizzare la gestione dell’infrastruttura cloud.

**Capacity planning ed elasticità**Le applicazioni multi-tier possono essere soggette a variazioni repentine del carico di lavoro richiesto (trattasi di un carico di tipo bursty). Perciò, è necessario dimensionare le capacità delle risorse.  
**Capacity planning** = determinazione della dimensione corretta di ogni tier dell’applicazione in termini di:  
- Numero di risorse.  
- Capacità e dimensioni di ciascuna risorsa.  
- Storage, memoria e risorse di rete.

Primo approccio (over-provisioning): consiste nel dimensionamento delle risorse solo in base ai picchi di carico, per cui causa un sottoutilizzo delle risorse e un costo maggiore del necessario.

Secondo approccio (under-provisioning): consiste nel dimensionamento delle risorse solo in base al carico medio, per cui causa un sovraccarico delle risorse e una perdita degli utenti.

Approccio elastico: consiste nell’allocazione e deallocazione delle risorse in modo autonomico (non richiede cioè l’intervento umano) in modo tale che, in ogni istante di tempo, la quantità di risorse allocata corrisponda il più possibile alla reale domanda.



Vediamo due possibili metriche per misurare l’elasticità di un sistema:  
1) **Accuracy** = somma delle aree di over-provisioning (O) e di under-provisioning (U) durante il periodo di tempo T.  
2) **Timing** = tempo totale speso in over-provisioning (B) e in under-provisioning (A) durante il periodo di tempo T.

Nella realtà è normale avere dei piccoli momenti di over-provisioning e under-provisioning poiché allocare e deallocare risorse richiede un certo delay (minuti in caso di macchine virtuali, secondi in caso di container).

**Load balancer**Ma come si gestisce la distribuzione delle richieste tra molteplici repliche di server?  
Tramite il load balancer, che può essere:  
- Centralizzato (single point of failure, collo di bottiglia).  
- Distribuito.

Il load balancer può distribuire il carico di lavoro tra più server seguendo più politiche differenti:  
-> Scegliendo via via il server in modo randomico.  
-> Con uno schema Round Robin.  
-> Con uno schema con livelli di priorità.  
-> In base alle capacità dei server.  
-> In base al livello di carico dei server (problema: molteplici richieste possono essere inoltrate tutte contemporaneamente allo stesso server, che diventa così sovraccarico, dando luogo al cosiddetto effetto gregge; è più indicato individuare un insieme di k server meno carichi e selezionarne randomicamente uno tra questi k).

**Service Level Agreement (SLA)**È l’accordo formale tra il consumatore e il fornitore di un servizio. È composto da:  
- Scopo del contratto.  
- Scopo del servizio.  
- Restrizioni (ovvero casi di non applicabilità).  
- Servizi opzionali.  
- Tempo di validità del contratto.  
- Contraenti.  
- SLO (Service Level Objective), che sono indicazioni quantitative sui parametri di qualità di servizio che il fornitore di impegna a garantire (e.g. disponibilità, tempo di ritardo).  
- Penalità o compensazioni nel caso in cui qualche SLO non venga rispettato.

Il fatto che un SLO venga rispettato o meno tipicamente deve essere stabilito dall’utente mediante un’attività di monitoraggio, che può incentrarsi su delle metriche orientate al sistema (e.g. utilizzo della CPU, utilizzo del disco, memoria usata) oppure su delle metriche orientate all’applicazione (e.g. tempo di risposta, disponibilità dei componenti).

**Fog computing vs edge computing**-> Fog computing: è una piattaforma virtualizzata che fornisce computazione, storage e servizi network tra gli end system e i tradizionali server del cloud. Le risorse si trovano tipicamente (ma non esclusivamente) ai bordi della rete.  
-> Edge computing: è simile al fog computing ma è meno integrato al cloud e le risorse sono solo ai bordi della rete.

**Problemi per i cloud customer**- Privacy e sicurezza: dove sono localizzati i dati? Chi può accedervi? Vengono protetti crittograficamente anche in transito?  
- Latenze di comunicazione: è possibile anche avere millisecondi di latenza che, tra l’altro, è possibile dipendano dalla rete e non dal cloud provider. Una possibile soluzione sta nella content delivery network (la più famosa è Akamai), che è un’infrastruttura di reti (con server in tutto il globo) con lo scopo di distribuire i servizi e le risorse.  
- Portabilità: capacità di adattare velocemente i dati per passare da un servizio a un altro. Un modo per migliorare la portabilità sta nell’utilizzare container o tool di automazione per il deployment.  
- Interoperabilità: capacità di usare più servizi contemporaneamente.  
- Supporto misero per la gestione e la negoziazione dei SLA.  
- Scalabilità & elasticità.  
- Outage dell’infrastruttura: non è così raro avere un importante disservizio.

**Problemi per i cloud provider**- Variabilità del livello di servizio richiesto: una possibile soluzione sta nell’utilizzare il modello di tariffazione spot.  
- Gestione dei SLA: a volte i cloud provider sono a loro volta utenti per altri servizi; se loro sono vittime di una qualche violazione, si potrebbe generare un effetto a cascata.  
- Gestione dei consumi energetici.  
- Interoperabilità tra cloud.

**ARCHITETTURA DEI SISTEMI DISTRIBUITI**

**Architettura software vs architettura di sistema**- Architettura software = organizzazione logica e interazione dei componenti software.  
- Architettura di sistema = instanziazione finale di un’architettura software.

**Stili architetturali**Concentriamoci dapprima sull’architettura software, che può essere definita mediante uno stile architetturale.  
Uno stile architetturale è un insieme coerente di decisioni di design riguardanti l’architettura software, ed è formulato in termini di componenti e connettori:  
- **Componente** = unità modulare con un’interfaccia definita.  
- **Connettore** = meccanismo per collegare più componenti tra loro.

Noi analizzeremo 6 stili architetturali principali.

1) Layered style: i componenti sono organizzati in layer (livelli), col componente al livello i che invoca tramite messaggi il componente al livello j < i. I layer tipicamente sono:  
- Layer di presentazione (interfaccia).  
- Layer di business (logica applicativa).  
- Layer di persistenza (talvolta accorpato col layer di business).  
- Layer di database.

2) Object-based style: i componenti sono oggetti che incapsulano una struttura dati e offrono API per accedere e modificare i dati. La comunicazione tra le componenti avviene tramite chiamata a procedura remota (RPC) o remote method invocation (RMI).

3) RESTful style: il sistema distribuito è visto come un insieme di risorse che sono gestite in modo individuale dai componenti e possono essere aggiunte, rimosse, recuperate e modificate rispettivamente tramite i metodi PUT, DELETE, GET e POST di http. L’interazione è stateless, per cui lo stato, nel caso in cui ce ne fosse bisogno, viene trasferito dai client verso i server. Le risorse sono identificate da un URI (Uniform Resource Identifier).  
Esempio: AWS S3, dove gli oggetti sono memorizzati in dei bucket.  
- REST = Representational State Transfer.

Questi tre stili presentano tutti un alto grado di accoppiamento tra i componenti, il che può introdurre delle limitazioni.  
Soluzione: far comunicare i componenti indirettamente aggiungendo un intermediario (e.g. un ulteriore livello di indirezione).  
Esistono tre diverse proprietà di disaccoppiamento:  
-> **Disaccoppiamento spaziale**: i componenti non devono necessariamente conoscersi per comunicare e cooperare.  
-> **Disaccoppiamento temporale**: i componenti non devono necessariamente essere presenti in contemporanea durante la comunicazione.  
-> **Disaccoppiamento rispetto alla sincronizzazione**: i componenti non hanno bisogno di aspettarsi a vicenda e non entrano in uno stato di blocco in attesa di una risposta.

4) Event-driven style: si ha un bus di eventi; nel momento in cui un componente pubblica un evento, quest’ultimo viene notificato ai componenti interessati. Si tratta di un meccanismo che offre disaccoppiamento spaziale; quello temporale, invece, viene soddisfatto se e solo se il bus è in grado di memorizzare i messaggi.  
Esempio: Java Swing.

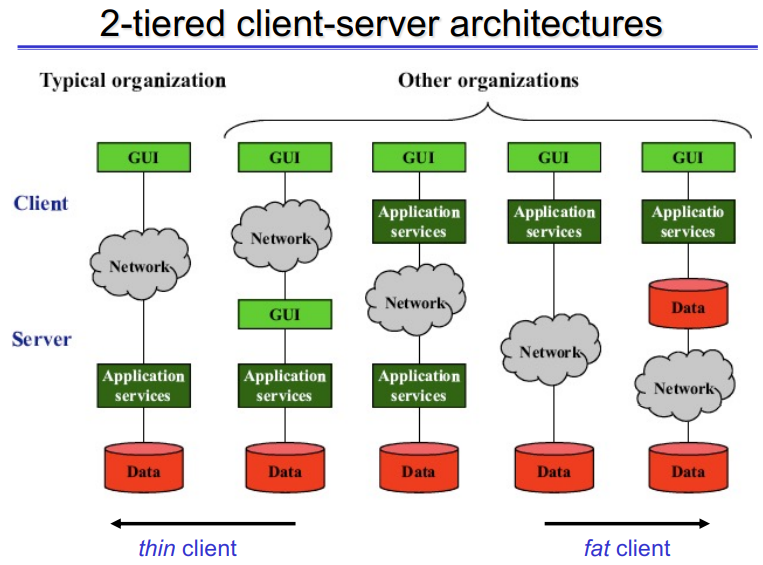
5) Data-oriented style: la comunicazione è mediata da un’area di memoria condivisa, che può essere attiva o passiva. Si tratta di un meccanismo che offre disaccoppiamento temporale; per quanto riguarda quello spaziale, invece, dipende dall’implementazione.  
Le API per lo spazio condiviso sono:  
- Write.  
- Read.  
- Take (= leggi ed elimina).  
- Push / notify (API che previene il polling, ma è disponibile solo nel caso in cui la memoria condivisa è attiva).  
Esempio: Linda tuple space.

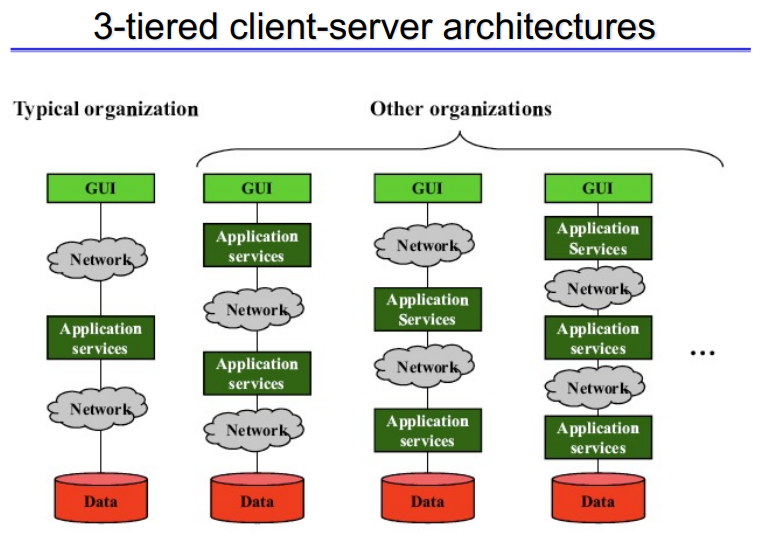
6) Publish-subscribe style: la comunicazione è mediata da un middleware. I componenti publisher generano eventi, mentre i subscriber si sottoscrivono al middleware in modo da essere notificati in caso di pubblicazione degli eventi a cui sono interessati. Si tratta di un meccanismo che offre tutte e tre le proprietà di disaccoppiamento.  
Esistono più varianti di questo schema di sottoscrizione:  
-> **Topic-based**: gli eventi sono marcati da un topic, per cui i subscriber possono interessarsi agli eventi di un determinato topic. Tuttavia, poiché i topic sono identificati da delle keyword, l’espressività è limitata.  
-> **Content-based**: gli eventi sono classificati rispetto al proprio contenuto (e.g. i metadati). Perciò i subscriber possono sottoscriversi agli eventi utilizzando dei filtri anziché i topic. Tuttavia, si tratta di una variante più complessa da implementare.  
D’altra parte, è possibile ricorrere a un’implementazione centralizzata (🡪 single point of failure, collo di bottiglia) oppure decentralizzata (con un’architettura master-worker, come ad esempio Apache Kafka).

**Tipologie di architetture di sistema**Le architetture di sistema possono essere di tre tipi: centralizzato, decentralizzato e ibrido. Analizziamole nel dettaglio.

**Architettura centralizzata**È basata sul modello client-server. La comunicazione tra client e server è basata su scambio di messaggi (il che può rappresentare un collo di bottiglia) ed è spesso sincrona e bloccante. Inoltre, un’operazione nello scambio di messaggi può essere idempotente o meno.  
Un servizio idempotente prevede che, a parità di messaggi di richiesta da parte del client, vengano inviati gli stessi messaggi di risposta da parte del server.

Supponiamo ora di avere un’architettura composta da tre layer (presentation, business e data storage).  
Come si possono mappare i livelli logici (layer) sui livelli fisici (tier)?





Il sistema può essere composto anche da N tier (N > 3): in tal caso si avrebbe maggiore flessibilità e disaccoppiamento, ma potrebbe esserci un degrado delle prestazioni dovuto a un maggiore costo di gestione e manutenzione.

Inoltre, un sistema multi-tier può avere una distribuzione:  
- **Verticale** se ciascun layer si trova su un determinato server fisico.  
- **Orizzontale** se, per ciascun layer, si ha una replicazione dei nodi; in tal caso deve esistere un load balancer che distribuisca le richieste. Questa soluzione contribuisce a migliorare l’elasticità e la disponibilità del sistema.

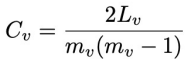
**Architettura decentralizzata**È basata sul modello P2P. I sistemi usano risorse distribuite per fornire servizi in modo decentralizzato.  
I nodi hanno capacità computazionali simili tra loro e possono ricoprire sia il ruolo di client che il ruolo di server; inoltre, non sono fissi ma possono entrare e uscire improvvisamente e/o frequentemente.  
Problemi da affrontare nell’ambito P2P:  
- **Eterogeneità** dei peer.  
- **Scalabilità**: avere tanti nodi non è una condizione sufficiente per avere un sistema scalabile.  
- **Locazione dei dati** e distanza tra i nodi.  
- **Tolleranza ai guasti** e agli abbandoni improvvisi dei nodi.  
- **Prestazioni**: ad esempio, come si può bilanciare in modo efficiente il carico dei nodi?  
- **Free-rider**: è un nodo egoista che richiede delle risorse agli altri nodi ma non mette a disposizione le proprie.  
- **Anonimato e privacy**: è possibile introdurre l’onion routing per far sì che la comunicazione sia anonima.  
- **Gestione della trustness** e della reputazione dei nodi.  
- **Necessità di difendersi** dagli attacchi.  
- **Resilienza ai churn**: è necessario gestire aggiunte e abbandoni continui dei nodi.

Un nodo, quando accede a una rete P2P, esegue tre operazioni principali:  
-> Bootstrap: è la fase iniziale in cui il nodo conosce la rete e gli altri nodi da contattare.  
-> Resource lookup: è la fase che si occupa di come vengono localizzate le risorse.  
-> Retrieval: ottenimento delle risorse.

**Overlay network**È una rete logica che interconnette i nodi tra loro; tuttavia, i collegamenti di un’overlay network non corrisponde ai collegamenti fisici tra i nodi.  
Qui viene effettuato l’**overlay routing**, che è un meccanismo di instradamento che, al contrario del routing tradizionale, non si incentra sugli indirizzi dei nodi, bensì sulle risorse da recuperare (file, CPU, banda, ecc.).  
Ciascuna risorsa viene identificata tramite un GUID (Globally Unique Identifier), che in genere deriva da una funzione hash sicura; in realtà, anche i peer hanno un identificatore calcolato mediante una funzione hash.  
Le overlay network si suddividono in strutturate e non strutturate.

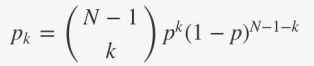
**Overlay network non strutturate**Sono overlay network rappresentabili con un grafo qualsiasi: per un nodo che entra in un’overlay network non strutturata è sufficiente contattare un sottoinsieme di nodi già presenti per diventare un loro vicino.  
Le overlay network non strutturate sono molto utili quando si ha un elevato tasso di churn (connessioni / disconnessioni) e sono altamente resilienti ai guasti; d’altra parte, però, sono caratterizzate da un alto costo di lookup.

Le overlay network sono rappresentabili con dei grafi random di cui ci interessano le seguenti caratteristiche:  
- **Coefficiente di clustering Cv di un vertice v** = rapporto tra il numero Lv di archi tra i mv vicini di v e il massimo numero di archi possibili tra loro.



- **Coefficiente di clustering del grafo** = media dei coefficienti di clustering di tutti i vertici.  
- **Lunghezza media del percorso minimo** = media aritmetica tra le lunghezze dei percorsi minimi tra tutte le coppie di vertici del grafo.  
- **Diametro del grafo** = lunghezza del percorso minimo più lungo all’interno del grafo.

Noi analizzeremo tre modelli di grafi random.

1) Erdos-Renyi: fissati il numero N di vertici e la probabilità p che una coppia di vertici sia connessa con un arco, abbiamo che:  
- Il grado di ciascun vertice segue una distribuzione binomiale; in particolare, se pk è la probabilità che un vertice v abbia grado k:  


- Coefficiente di clustering del grafo = p (è un valore basso, il che è atipico per le reti reali).  
- Lunghezza media del percorso minimo = log(N) / log((N-1)p) (è anch’esso un valore basso ma, stavolta, è tipico per le reti reali).  
- Il diametro del grafo ha la dimensione di log(N).

2) Watts-Strogatz: è un modello di grafo con un alto coefficiente di clustering e con una lunghezza media del percorso minimo proporzionale a log(N). Quest’ultima proprietà fa sì che i grafi di Watts-Strogatz godano della proprietà dello small world.  
In particolare, lo small-world phenomenon (noto anche come 6 degrees of separation) determina che, se scegliamo due persone casuali sulla Terra, troveremo un path di al massimo 6 persone che le legano (dove un arco tra due persone è relativo alla relazione di “conoscenza”).  
Tuttavia, a differenza delle reti reali, una rete di Watts-Strogatz non è caratterizzata dalla presenza di hub, che sono nodi con grado molto elevato.

3) Albert-Barabasi: è un modello di grafo noto anche come **scale-free network**. Il grado di ciascun vertice segue una **legge di potenza**; in particolare la probabilità che un certo vertice abbia grado k è esprimibile nel seguente modo:  


Un esempio pratico di legge di potenza è dato dalla legge di Pareto: “L’80% degli effetti deriva dal 20% delle cause”.

La legge di potenza gode della proprietà di **invarianza alla scala**:  
  
In altre parole, f(lambda\*x) ha la stessa identica forma di f(x) ma con una scala diversa.

Inoltre, le distribuzioni che seguono una legge di potenza sono **heavy-tailed**: tendono a 0 molto più lentamente rispetto alle distribuzioni esponenziali. Di conseguenza, i valori che si trovano nella coda hanno una probabilità non trascurabile di verificarsi.  
Le scale-free network sono anche caratterizzate dall’**attacco preferenziale** (preferential attachment): quando un nuovo nodo si connette alla rete, sceglie di collegarsi con probabilità maggiore ai nodi con grado più elevato. In tal modo, si forma un numero limitato di hub e il diametro del grafo cresce come ln(ln(N)): lentissimamente!  
Infine, le reti scale-free sono altamente tolleranti ai guasti.

Tornando al discorso sulle overlay network non strutturate, queste possono essere classificate in:  
- **Centralizzate**: vi è un directory server che tiene traccia di quali peer posseggono ciascuna risorsa. Sono reti molto semplici ma sono caratterizzate da collo di bottiglia e single point of failure. Un esempio è Napster.  
- **Decentralizzate**: non vi è alcun server, per cui le informazioni sono completamente distribuite. Un esempio è Gnutella.  
- **Semi-decentralizzate**: vi sono dei super-peer che detengono le informazioni su quali peer posseggono ciascuna risorsa.

Per quanto riguarda le overlay network non strutturate decentralizzate, i peer, per trovare le risorse di cui hanno bisogno, possono seguire più approcci possibili. Per ora vediamone due.

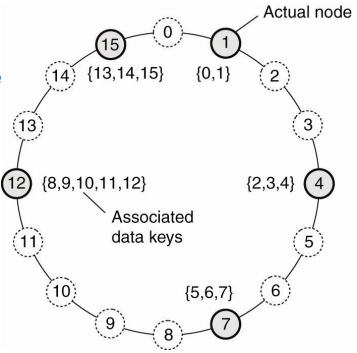
1) Query flooding: il peer sorgente invia una richiesta di lookup a tutti i suoi vicini i quali, a loro volta, inoltrano la richiesta ai loro vicini e così via, finché non si trova il possessore della risorsa cercata.  
- Nelle richieste viene aggiunto un TTL (time to live) che proibisce alle richieste stesse di essere inoltrate all’infinito.  
- Nelle richieste viene aggiunto anche un query ID in modo tale che, se un nodo riceve più volte una stessa richiesta, non deve processarla nuovamente.  
- Per inviare un messaggio di risposta contenente la risorsa al peer sorgente, è possibile avviare una comunicazione diretta oppure ricorrere al backward routing (che consiste nel far viaggiare la risposta sullo stesso percorso attraversato dalla richiesta); quest’ultimo approccio è vantaggioso perché non richiede di specificare il peer sorgente nel messaggio di richiesta e le informazioni sulla disponibilità delle risorse si propagano tra i vari peer.  
Il query flooding presenta i seguenti svantaggi:  
-> Vi è una crescita esponenziale del numero di messaggi.  
-> Sono possibili attacchi DoS.  
-> Il costo di lookup è elevato: O(N), dove N = numero di nodi nella rete.  
-> Non è detto che si riescano a trovare le risorse cercate, anche se esiste un qualche peer che le possiede.  
-> Non è detto che i nodi vicini nell’overlay network siano anche vicini fisicamente.

2) Random walk: il peer sorgente invia una richiesta di lookup a un solo vicino scelto randomicamente il quale, a sua volta, inoltra la richiesta a un proprio vicino e così via, finché non si trova il possessore della risorsa cercata.  
Il problema principale di questo meccanismo è il costo di lookup che cresce ulteriormente: per ovviare a ciò, il peer sorgente (e solo il peer sorgente) può inviare la richiesta a k suoi vicini, avviando k random walk simultaneamente.

**Overlay network strutturate**Sono overlay network con una topologia ben definita. Il loro obiettivo è diminuire il costo di lookup, a discapito però del costo di mantenimento della struttura a seguito di ingressi e uscite di peer.

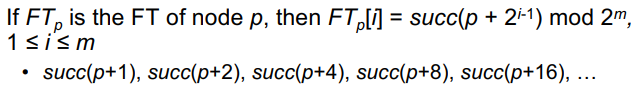
Il routing nelle overlay network strutturate avviene con l’utilizzo di una **Distributed Hash Table (DHT)**, in cui le informazioni sulle coppie (chiave K, valore V) sono distribuite tra i peer. K è la chiave che identifica la risorsa V, corrisponde a un determinato GUID ed è il risultato di una funzione hash (che riceve in input un qualcosa come i metadati di V). Tipicamente anche i nodi sono identificati da un GUID.  
Noi vogliamo che le chiavi siano equamente distribuite tra i peer e che, a seguito di un inserimento o abbandono di un nodo, non venga stravolto tutto in modo da dover ridistribuire da capo le chiavi tra i peer. A tale scopo, si fa uso del **consistent hashing**, che è una speciale tecnica di hashing in cui:  
- Sia le risorse che i nodi (i bucket) sono mappati uniformemente sullo stesso spazio di indirizzamento mediante una funzione hash.  
- Ciascun nodo gestisce un intervallo di chiavi hash consecutive, non un insieme di chiavi sparse.

Noi analizzeremo due protocolli per le overlay network strutturate: Chord e Pastry.

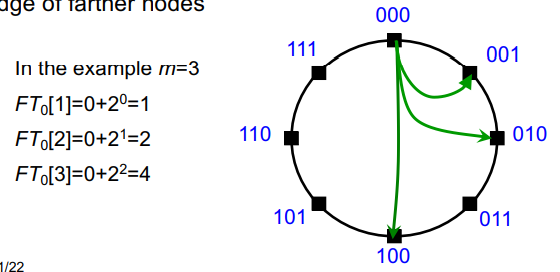
**Chord**I peer sono organizzati in una struttura ad anello. Ciascuno di essi è responsabile delle chiavi piazzate tra se stesso (incluso) e il nodo precedente (escluso): ciò implica che la risorsa con chiave K è gestita dal peer il cui identificatore è il più piccolo tra tutti quelli maggiori o uguali a K; tale peer è detto successore della chiave K (succ(K)).  




- Metrica di distanza tra i nodi = differenza lineare tra i GUID.

Operazione di get (routing):  
Esistono più soluzioni possibili per effettuare il lookup di una risorsa:  
1) Ogni nodo conosce solo il proprio successore: se è in cerca di una risorsa, la richiede a lui (al successore), il quale, eventualmente, inoltra la richiesta al proprio successore e così via. In tal modo, il costo di lookup è O(N) (piuttosto elevato).  
2) Ogni nodo conosce tutti gli altri peer, il che consente di avere un costo di lookup di O(1). D’altra parte, però, le operazioni di join e leave richiedono che il nodo che si aggiunge o abbandona la rete informi tutti gli altri.  
3) Ciascun nodo conosce bene i nodi a lui vicini e la conoscenza decresce via via che ci si allontana.  
In particolare, ogni nodo ha una finger table (FT) in cui sono registrati i peer che conosce. La FT ha m righe (dove m = numero di bit del GUID).  








Supponiamo che il nodo p debba trovare il peer responsabile della chiave k (ovvero succ(k)).  
- Se k appartiene alla porzione dell’anello gestita da p, il lookup termina immediatamente.  
- Se p < k <= FTp [1] allora p invia la richiesta al suo successore.  
- Altrimenti p invia la richiesta al nodo q più lontano da lui tra quelli il cui GUID è minore o uguale a k:  
  
dove FTp[j] = q.  
In tal modo, il costo di lookup risulta essere O(log N).

Operazione di join:  
Con l’ipotesi per cui ogni nodo mantenga le informazioni anche sul proprio predecessore, quando un peer p si inserisce all’interno dell’overlay network, compie i seguenti passi:  
1) Cerca il nodo succ(p+1) in modo da posizionarsi subito prima di lui.  
2) Avverte il suo successore della sua presenza.  
3) Inizializza la propria FT.  
4) Avverte il suo predecessore della sua presenza (in modo da fargli aggiornare la FT).  
5) Trasferisce su di sé le chiavi di sua competenza (prendendole dal suo successore).

Operazione di leave:  
Quando un peer p lascia volontariamente l’overlay network, compie i seguenti passi:  
1) Trasferisce le proprie chiavi sul successore.  
2) Avverte il suo successore del suo abbandono (in modo da fargli aggiornare il puntatore “predecessore”).  
3) Avverte il suo predecessore del suo abbandono (in modo da fargli aggiornare la FT).

In realtà, a seguito di un’operazione di join o di leave da parte di un peer, dovrebbero essere aggiornate le FT di tutti i nodi della rete; quello che si fa è effettuare il refresh di tutte le FT periodicamente, in modo da tener conto anche degli abbandoni dovuti a guasti improvvisi.

Le operazioni di join e di leave hanno una complessità di O(log2 N).

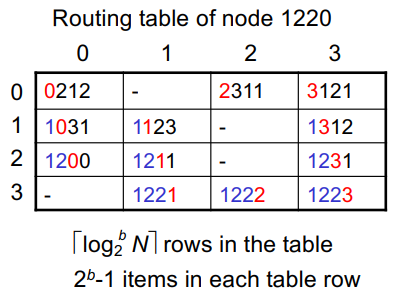
È rimasto ancora un punto in sospeso: quando un nodo crasha, non c’è più nessuno che gestisca le chiavi di cui quel peer era responsabile. Per ovviare a questo problema, si affida un certo numero R di copie delle chiavi ad altrettanti successori del peer; di conseguenza, ogni nodo deve conoscere R successori e non più soltanto uno.

Vantaggi di Chord:  
- È un protocollo semplice ed elegante.  
- Load balancing: le chiavi sono distribuite uniformemente tra i peer.  
- Scalabilità: è efficiente nelle operazioni di lookup.  
- Robustezza: prevede aggiornamenti periodici delle FT affinché siano consistenti coi cambiamenti che avvengono nell’overlay network.

Svantaggi di Chord:  
- La vicinanza fisica tra i nodi non viene considerata.  
- Le query per intervalli di chiavi sono del tutto inefficienti.

**Pastry**Anche qui i peer sono organizzati in una struttura ad anello. Ciascuna chiave è rappresentata con d cifre, dove ogni cifra è composta da b bit (solitamente b=4).  
Pastry è un protocollo basato sul **Plaxton routing**, secondo cui una risorsa V viene memorizzata dal peer il cui GUID ha il prefisso più lungo che matcha col GUID di V. Ciascun nodo p detiene una routing table che viene costruita secondo i seguenti criteri:  
-> I GUID dei peer sull’n-esima riga condividono le prime n cifre col GUID del nodo p.  
-> L’(n+1)–esima cifra dei GUID dei peer sull’n-esima riga è pari all’indice della colonna della routing table.  
-> Esistono entry della routing table che possono essere associate a più di un nodo: ne viene selezionato uno in base a qualche metrica di prossimità (e.g. RTT).

NB: la numerazione delle righe e delle colonne della routing table parte da 0 e non da 1.



In pastry l’operazione di lookup ha una complessità di O(logb N).

**Architettura ibrida**Un’architettura ibrida cerca di combinare i benefici delle architetture centralizzata e decentralizzata.  
Due esempi di architetture ibride sono:  
1) Sistema P2P ibrido con dei super-peer che hanno maggiori funzionalità degli altri peer; il routing avviene solo tra questi super-peer.  
2) BitTorrent, dove un nuovo utente ottiene il file torrent contenente i puntatori ai tracker (che sono i server che conoscono i peer attivi che forniscono i chunk del file desiserato dall’utente); l’utente poi parla con un tracker che gli restituisce la lista dei peer che posseggono i chunk del file desiderato; infine, si unisce a uno swarm di peer da cui effettua il download dei chunk dei file e, nel frattempo, mette a disposizione i chunk che ha già scaricato.

**Definizioni di middleware**1) Bernstein: il middleware è un insieme di strumenti situati tra le applicazioni e il supporto di basso livello (hardware, sistema operativo, ecc).  
2) Coulouris & Dollimore: il middleware è uno strato software che fornisce un’astrazione di programmazione che maschera l’eterogeneità delle reti, dell’hardware, dei sistemi operativi e dei linguaggi di programmazione sottostanti.  
3) Anthony: il middleware è uno strato virtuale tra le applicazioni e le piattaforme sottostanti che fornisce una trasparenza significativa.

**Tipi di middleware**  
1) RPC/RMI middleware: offre una funzionalità di comunicazione in remoto e fornisce un Interface Definition Langauge per definire il contratto di cominucazione. Alcuni esempi sono SUN RPC, Java RMI e gRPC.  
2) Message-oriented middleware (MOM): fornisce un servizio di comunicazione persistente e asincrona tramite un sistema a code di messaggi. Alcuni esempi sono ActiveMQ, Kafka e RabbitMQ.  
3) Component-based middleware: qui i componenti vengono eseguiti all’interno di container. Alcuni esempi sono .NET e Java EE.  
4) Service-oriented middleware: è un middleware per le architetture basate su microservizi.

**Middleware auto-adattativo**Utilizzare un middleware che segue uno stile architetturale fisso è indice di poca flessibilità. È per questo che esistono i reflective middleware (che utilizzano il meccanismo di reflection per adattarsi ai cambiamenti del contesto ambientale) e i self-adaptive middleware. Noi ci concentreremo su questi ultimi.

I **sistemi software auto-adattativi** (o autonomici) sono sistemi in grado di adattare le proprie operazioni a run-time in base a sé stessi e all’ambiente (gli interventi umani devono essere assenti o fortemente limitati).  
Gli obiettivi di un sistema auto-adattativo sono:  
- **Self-optimizing** = capacità di un sistema di ottimizzare le sue performance o l’uso delle sue risorse.  
- **Self-healing** = capacità di un sistema di diagnosticare e adottare misure correttive a problemi o guasti.  
- **Self-protecting** = capacità di un sistema di mettere in atto strategie di difesa contro possibili attacchi anche in modo pro-attivo.  
- **Self-configuring** = capacità del sistema di integrare nuovi elementi senza alterare il suo corrente funzionamento e di impostare determinati parametri di configurazione.

Come si raggiungono questi 4 obiettivi?  
-> Il sistema deve essere consapevole del suo stato interno (**self-awareness**) e dell’ambiente in cui si trova (**self-situation**), dove:  
Ambiente = software non controllabile dal sistema + hardware + rete + contesto fisico + utenti.  
-> Il sistema deve poter identificare dei cambiamenti (**self-monitoring**) e adattarsi di conseguenza (**self-adjustment**).

**MAPE**È un’architettura di riferimento per i sistemi auto-adattativi che consiste di 4 fasi: Monitor, Analyze, Plan ed Execute; talvolta, viene considerata anche una quinta componente, Knowledge, per cui l’architettura viene spesso denotata con MAPE-K.  
- **Monitor**: riceve dei dati in ingresso, li filtra e li correla per la fase successiva.  
- **Analyze**: osserva i risultati del monitoraggio e determina se è necessario attuare delle operazioni di adattamento; se sì, dà il via alla fase successiva.  
- **Plan**: pianifica le azioni di adattamento da mettere in atto.  
- **Execute**: esegue ciò che è stato stabilito nella fase precedente.  
- **Knowledge**: i dati usati dal sistema autonomico vengono memorizzati come conoscenza condivisa tra le 4 fasi appena descritte.

Vediamo ora come avviene la progettazione di alcune di queste fasi.

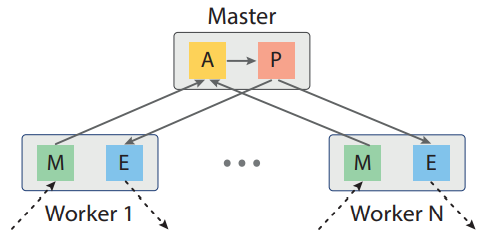
Monitor:  
1) Il monitoraggio deve essere effettuato periodicamente o on-demand?  
2) Cosa deve essere monitorato (risorse, carico di lavoro, prestazioni)?  
3) L’architettura del monitoraggio deve essere centralizzata o decentralizzata?  
4) Il monitoraggio deve essere passivo (mera osservazione) o attivo (scatenando le operazioni da monitorare)?  
5) Dove e come devono essere memorizzati i dati monitorati?

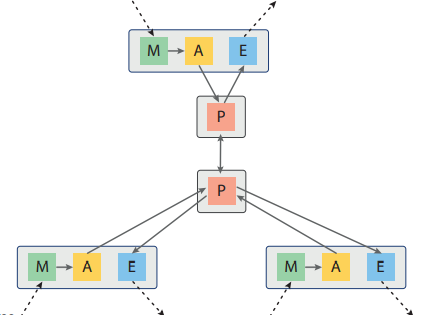
Analisi:  
1) L’analisi deve avvenire periodicamente o a seguito di un determinato evento?  
2) L’analisi deve essere reattiva o pro-attiva?

Plan:  
Esiste una gran varietà di metodologie che possono essere adottate nella fase di plan:  
1) Formulazione di problemi di ottimizzazione (spesso NP-hard).  
2) Utilizzo di euristiche che portano a una soluzione sub-ottima ma in modo efficiente.  
3) Utilizzo di machine learning e reinforcement learning.  
4) Utilizzo della teoria del controllo.  
5) Utilizzo della teoria delle code.  
E così via.

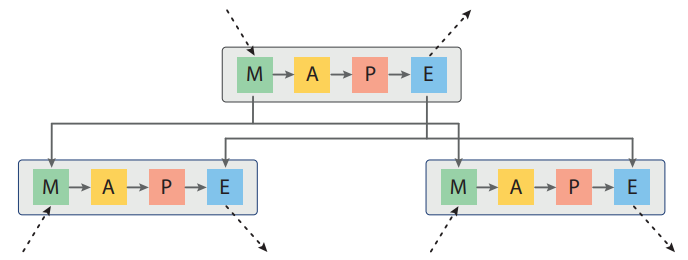
Il sistema MAPE può essere **centralizzato** (tutte le componenti sono nello stesso nodo) oppure **distribuito**. In particolare, un’architettura distribuita può essere a sua volta **gerarchica** oppure **flat**.

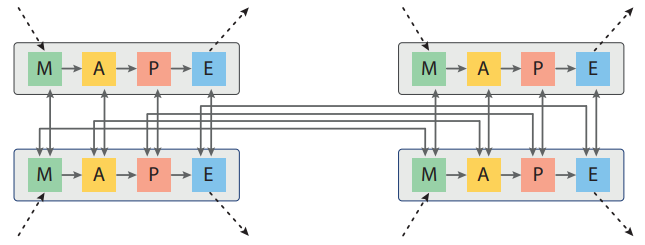
**Pattern MAPE gerarchici**1) Master-worker pattern: decentralizza Monitor ed Execute su molteplici nodi worker e mantiene Analyze e Plan centralizzate su un unico nodo master.  
- PRO: il master sa tutto di tutti, per cui può prendere facilmente delle decisioni globali.  
- CONTRO: c’è molto overhead di comunicazione dato che il master comunica con tutti i worker; inoltre, il master potrebbe rappresentare un collo di bottiglia e un single point of failure.

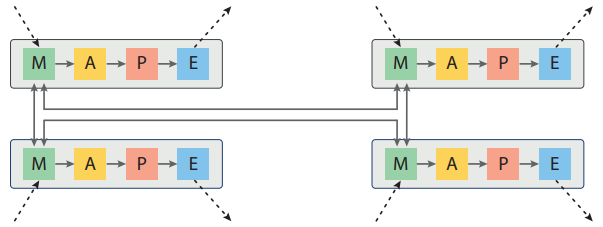


2) Regional pattern: è composto da molteplici regioni debolmente accoppiate, ciascuna delle quali ha una struttura gerarchica a due livelli: il livello superiore comprende Plan, mentre quello inferiore comprende Monitor, Analyze ed Execute.  
PRO: offre una maggiore flessibilità (le regioni possono stare sotto amministrazioni differenti).  
CONTRO: è difficile conseguire gli obiettivi globali.  




3) Hierarchical control pattern: consiste in tanti nodi che comunicano tra loro (e hanno una visione locale dei dati) e un nodo su un livello superiore che ha una visione più globale. In tutti i nodi vengono effettuate tutte e 4 le fasi.  
PRO: il nodo di livello superiore può conseguire facilmente gli obiettivi globali.  
CONTRO: potrebbe essere farraginoso decidere cosa controllare a livello locale e cosa a livello globale.  


**Pattern MAPE flat**1) Coordinated control pattern: prevede molteplici cicli MAPE situati su altrettanti nodi; ciascuna fase di ogni nodo comunica con la corrispondente fase di tutti gli altri nodi.  
PRO: è scalabile.  
CONTRO: c’è moltissimo overhead di comunicazione.  


2) Information sharing pattern: è un caso particolare del coordinated control pattern, in cui solo i componenti Monitor comunicano tra loro.  
PRO: è scalabile.  
CONTRO: c’è poca coordinazione a causa della mancanza di comunicazione nella fase Plan.  


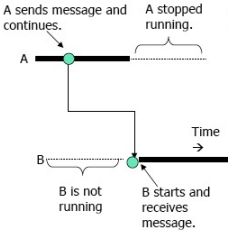
**COMUNICAZIONE NEI SISTEMI DISTRIBUITI**

**Tipi di comunicazione**Ricordandoci che i servizi per la comunicazione nei sistemi distribuiti vengono forniti dal middleware, distinguiamo i tipi di comunicazione rispetto a tre assi:  
- Persistenza (comunicazione persistente vs transiente).  
- Sincronizzazione (comunicazione sincrona vs asincrona).  
- Dipendenza dal tempo (comunicazione discreta vs streaming).

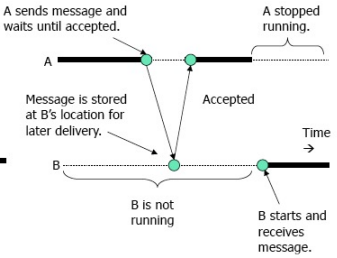
Comunicazione persistente:  
Il messaggio viene memorizzato dal middleware finché non viene ricevuto dal destinatario. Ciò consente di avere disaccoppiamento temporale.  
Comunicazione transiente:  
Il messaggio viene memorizzato dal middleware solo finché sia il mittente sia il destinatario sono in esecuzione; se la consegna non è possibile, il messaggio viene scartato. Ciò proibisce di avere disaccoppiamento temporale.

Comunicazione sincrona:  
Le operazioni di invio e di ricezione di un messaggio sono bloccanti. Per quanto riguarda il mittente, esistono tre alternative per stabilire fino a quando viene bloccato:  
1) Finché il middleware destinatario non prende in carico il messaggio.  
2) Finché il destinatario non riceve il messaggio a livello applicativo.  
3) Finché il messaggio non viene processato completamente dal destinatario.  
Comunicazione asincrona:  
L’operazione di invio di un messaggio è non bloccante, mentre l’operazione di ricezione può essere sia bloccante che non bloccante.

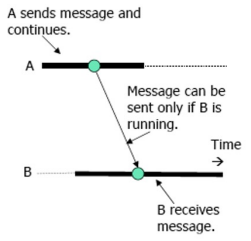
Comunicazione discreta:  
Ciascun messaggio costituisce un’unità completa di informazione.  
Comunicazione basata su streaming:  
I messaggi hanno una relazione temporale, sono caratterizzati da un ordine di invio e costituiscono uno stream.

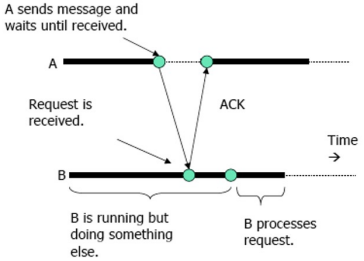
Combinando persistenza e sincronia, possiamo ottemere i seguenti scenari di comunicazione:  
a) **Comunicazione persistente e asincrona**: è tipica delle e-mail.  


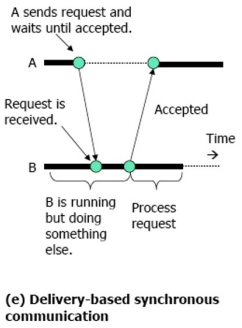
b) **Comunicazione persistente e sincrona**: il mittente rimane bloccato finché non è sicuro che il messaggio sia stato memorizzato nel middleware lato destinatario.

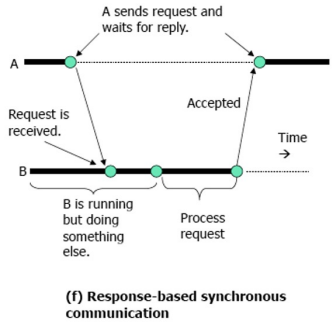




c) **Comunicazione transiente e asincrona**: il mittente non rimane mai in stato di blocco ma il messaggio può andare perso se il destinatario è irraggiungibile.  


d) **Comunicazione transiente e “receipt-based synchronous”**: il mittente rimane in stato di blocco finché il middleware destinatario non prende in carico il messaggio.  


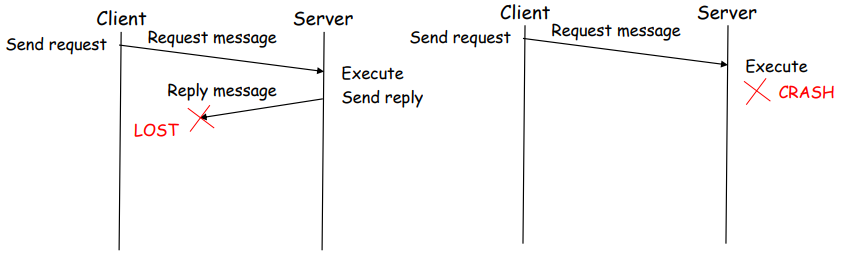
e) **Comunicazione transiente e “delivery-based synchronous”**: il mittente rimane in stato di blocco finché il destinatario non riceve il messaggio a livello applicativo.  


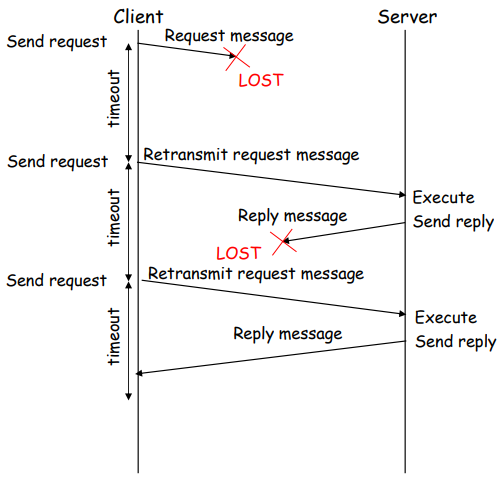
f) **Comunicazione transiente e “response-based synchronous”**: il mittente rimane in stato di blocco finché il messaggio non viene processato completamente dal destinatario (e quest’ultimo non invia un messaggio di risposta).  


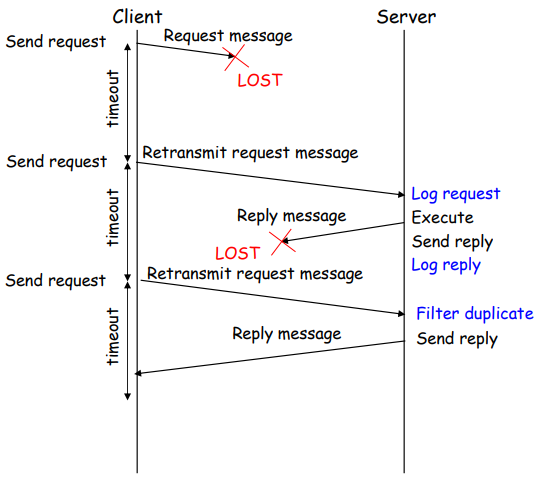
**Semantica degli errori di comunicazione**Durante la comunicazione tra client e server possono verificarsi diversi errori:  
-> Un messaggio può essere perso o ritardato, o la connessione può essere resettata.  
-> Il server può crashare (sia prima che dopo aver processato il servizio richiesto).  
-> Il client può crashare.

È possibile gestire tali situazioni tramite le cosiddette semantiche di errore, che dipendono dalla combinazione dei seguenti tre meccanismi di base:  
- **Request Retry (RR1)**: il client riprova a inviare il messaggio di richiesta finché non riceve una risposta oppure finché non ha effettuato un numero sufficiente di tentativi per cui può assumere che il server sia andato in crash.  
- **Duplicate Filtering (DF)**: il server scarta i messaggi di richiesta duplicati.  
- **Result Retransmit (RR2)**: il server memorizza la risposta per ciascun messaggio di richiesta in modo da ritrasmetterla senza calcolarla nuovamente nel caso in cui ricevesse un messaggio di richiesta duplicato.

Analizziamo ora le 4 possibili semantiche di errore.

1) Semantica May-be: il client invia il messaggio di richiesta una sola volta e non è sicuro se il server ha eseguito l’operazione o meno. Non viene utilizzato alcun meccanismo tra RR1, DF e RR2.  


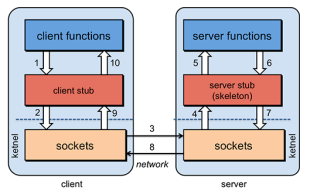
2) Semantica At-least-once: se il server è funzionante, processa ciascuna richiesta **almeno** una volta. Di fatto, il client mette in atto il meccanismo RR1, mentre il server non utilizza alcun meccanismo. Si tratta di una semantica compatibile solo coi servizi idempotenti: per un servizio non idempotente, invece, il server rischia di inviare risposte diverse per duplicati del medesimo messaggio di richiesta.  


3) Semantica At-most-once: se il server è funzionante, processa ciascuna richiesta **al più** una volta; tuttavia, in caso di fallimento, il client e il server non hanno alcuna informazione sullo stato della controparte (e.g. il client non ha idea se il server ha processato la richiesta). I meccanismi di base RR1, DF e RR2 vengono utilizzati tutti e tre. Si tratta di una semantica compatibile sia coi servizi idempotenti sia con quelli non idempotenti.  
Ma come fa il server a riconoscere le richieste duplicate? Basta che il client includa nei messaggi un ID univoco da riutilizzare quando deve inviare una richiesta duplicata.  


4) Semantica Exactly-once: se il server è funzionante, processa ciascuna richiesta **esattamente** una volta; ciò vuol dire che, in ogni caso, sia il client che il server conoscono lo stato della controparte. Di conseguenza, i tre meccanismi di base (RR1, DF, RR2) non sono sufficienti: sono richiesti dei meccanismi addizionali per far fronte ai guasti lato server:  
- Transparent server replication.  
- Write-ahead logging (WAL).  
- Recovery.

Per quanto riguarda il WAL, è un pattern che prevede che i cambiamenti di stato del server, prima di manifestarsi, debbano essere scritti nel log.  
Il log è un file che si trova su un dispositivo di memoria persistente e viene scritto tramite un’operazione di append (nel nostro caso specifico, viene scritto un gruppo di entry per volta). Nel caso in cui il server subisce un crash, è possibile recuperare il suo stato originale riproducendo le operazioni scritte nel log (recovery).

**RPC (Remote Procedure Call)**Idea: usare il modello client / server per chiamare procedure in esecuzione su macchine remote, in modo tale che lo scambio dei messaggi venga gestito dal middleware, in modo trasparente al programmatore (il quale deve programmare come se stesse implementando una chiamata a procedura locale, senza ulteriori complicazioni).  
Questa illusione si ottiene aggiungendo due entità intermedie dette **stub** (una lato client e una lato server) che si occupano dei dettagli implementativi relativi allo scambio dei messaggi.

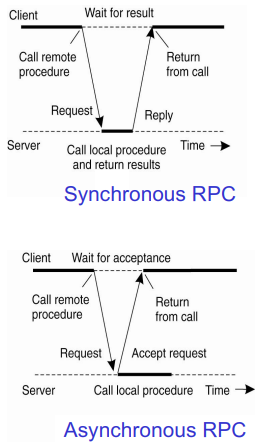
Step di base di RPC:  
1) Il client invoca una procedura locale chiamata **client stub**.  
2) Il client stub impacchetta il messaggio di richiesta mediante il **marshaling** dei parametri (dove gli argomenti vengono convertiti dal formato locale al formato comune) e chiama il sistema operativo locale.  
3) Il sistema operativo del client invia il messaggio di richiesta al sistema operativo del server.  
4) Il sistema operativo del server passa il messaggio di richiesta al **server stub**.  
5) Il server stub spacchetta il messaggio di richiesta mediante l’**unmarshaling** dei parametri (dove gli argomenti vengono convertiti dal formato comune al formato locale) e invoca la procedura.  
6) Il server esegue la procedura e restituisce i valori di ritorno al server stub.  
7) Il server stub impacchetta il messaggio di risposta mediante il marshaling dei valori di ritorno e chiama il sistema operativo.  
8) Il sistema operativo del server invia il messaggio di risposta al sistema operativo del client.  
9) Il sistema operativo del client passa il messaggio di risposta al client stub.  
10) Il client stub spacchetta il messaggio di risposta mediante l’unmarshaling dei valori di ritorno e restituisce il risultato al client.  


**Problemi legati a RPC**1) Eterogeneità nella rappresentazione dei dati: client e server possono utilizzare rappresentazioni di dati differenti; per gestire questo, esistono quattro soluzioni alternative:  
-> Specificare la codifica nel messaggio.  
-> Fare in modo che il mittente converta i dati nella codifica del destinatario: la conversione è veloce (solo il mittente deve effettuarla) ma ciascun componente deve conoscere le codifiche di tutti gli altri componenti.  
-> Fare in modo che il mittente converta i dati nella codifica standard: la conversione è lenta (sia il mittente che il destinatario devono effettuarla) ma ciascun componente deve conoscere solo la codifica standard.  
Questo è il meccanismo adottato in RPC e viene realizzato mediante l’utilizzo di **proxy** (che in RPC sono gli stub).  
-> Introdurre un intermediario (detto **broker**) che si occupi della conversione dei dati.

2) Passaggio dei parametri: può avvenire secondo una delle seguenti tre modalità:  
- **Call by value**: il valore dei parametri viene copiato sullo stack della funzione chiamata. Eventuali modifiche sul valore dei parametri non si riflettono nel chiamante.  
- **Call by reference**: il riferimento (l’indirizzo) ai parametri viene copiato sullo stack della funzione chiamata, la quale agisce direttamente sui dati del chiamante.  
- **Call by copy-restore**: il chiamante passa alla funzione chiamata una copia dei parametri. Quando la funzione chiamata restituisce il controllo al chiamante, quest’ultimo riporta le eventuali modifiche dei parametri sui dati originali. Tale meccanismo simula il passaggio dei parametri per riferimento (call by reference).

3) Semantica di errore: RPC utilizza una semantica **At-least-once** oppure **At-most-once**.

4) Server binding: è un problema in cui ci si chiede come deve fare il client per localizzare il server endpoint e la procedura remota da invocare su di esso. Di base, il binding può essere **statico** (l’indirizzo del server e altre informazioni sono incorporate nel codice, il che non permette trasparenza e flessibilità) oppure **dinamico** (viene effettuato a run-time, il che porta a un maggiore overhead).  
Il binding dinamico si articola in due fasi:  
-> **Naming**: è una fase statica, che avviene prima dell’esecuzione. Qui il client specifica quali sono il servizio e il server a cui vuole far riferimento.  
-> **Addressing**: è una fase dinamica, che avviene durante l’esecuzione. Qui si ha il binding vero e proprio tra client e server. L’addressing può essere **esplicito** (il client invia una richiesta broadcast o multicast e si connette al primo server che gli risponde) oppure **implicito** (c’è un **name server** che registra i servizi offerti dai server, gestisce una binding table e si occupa dei collegamenti tra i client e i server).

5) Sincronia: RPC può essere sia sincrono che asincrono.  
  
  
6) Trasparenza: RPC in realtà non è davvero trasparente rispetto all’accesso e alla locazione. Infatti le chiamate a procedura remota su LAN richiedono da 0,1 a 1 ms, per cui sono circa 100.000 volte più lente delle chiamate locali che richiedono 3 ns (per quanto riguarda le reti WAN le cose peggiorano ulteriormente). Inoltre, possono verificarsi dei fallimenti che non si presentano mai in locale; ad esempio:  
- Il client potrebbe non riuscire a localizzare il server.  
- Il messaggio di richiesta o di risposta potrebbe andare perso.  
- Il server e il client potrebbero crashare.

7) Sicurezza: poiché abbiamo a che fare con una comunicazione che avviene in rete, dobbiamo fare i conti con delle problematiche relative alla sicurezza: In particolare, potrebbe essere necessario:  
-> Autenticare il server e/o il client.  
-> Cifrare i messaggi, che altrimenti sarebbero visibili in rete.  
-> Autenticare i messaggi, che altrimenti potrebbero essere modificati.  
-> Far fronte ai replay attack.

**IDL (Interface Definition Language)**È un linguaggio che consente al programmatore di descrivere un’interfaccia per la chiamata a procedura remota. In particolare, risolve sia il problema per cui alcuni linguaggi (come C e C++) non concepiscono le chiamate a procedure remote, sia il problema per cui client e server possono girare su ambienti di esecuzione differenti (e.g. client Java che comunica con un servizio Python).  
Esiste inoltre il compilatore IDL (detto **pre-compilatore**) che genera automaticamente il client stub e il server stub.

**SunRPC**Detto anche Open Network Computing (ONC), è un esempio di RPC di prima generazione ed è una suite di prodotti tra cui:  
- eXternal Data Representation (XDR) come IDL.  
- Remote Procedure Call GENerator (RPCGEN) come compilatore IDL.  
- Port mapper come name server.  
- Network File System (NFS) come file system distribuito.

Step di base per programmare con SunRPC:1) Definire l’interfaccia RPC definendo un file con formato .x che è scritto col linguaggio XDR ed è suddiviso in tre porzioni:  
- Definizione delle costanti e dei parametri di input e output delle procedure.  
- Definizione delle procedure stesse con i relativi identificatore (= numero di procedura), numero di versione e numero di programma (tutti questi valori, che sono identificatori, vanno scritti interamente in UPPERCASE).  
- Definizione di eventuali tipi di dato non predefiniti in XDR.  
NB: XDR è un formato binario che usa implicit typing, per cui vengono trasmessi soltanto i valori, non i tipi di dato.

2) Compilare il file *nomeProgramma.x* con RPCGEN, che genera:  
-> Il client stub (*nomeProgramma\_clnt.c*).  
-> Il server stub (*nomeProgramma\_svc.c*).  
-> Le routine XDR per convertire i dati dal formato locale al formato comune XDR (*nomeProgramma\_xdr.c*).  
-> Il file header che contiene le strutture dati (i tipi di dato) necessarie (*nomeProgramma.h*).

3) Scrivere il file *server.c*, che implementa le procedure remote fornite dal server RPC. Osserviamo che:  
- Non c’è alcuna funzione main: piuttosto si trova nel server stub (di fatto è lui a invocare in ultima istanza le procedure).  
- Ciascuna procedura ha esattamente due parametri di input e un parametro di output; tutti e tre questi parametri sono dei puntatori. In particolare, il secondo parametro di input è un puntatore al client transport manager, che è utilizzato dal client stub per gestire la comunicazione col server. Inoltre, il parametro di output dev’essere un puntatore a una variabile *static*, cosicché l’area di memoria puntata continui a esistere dopo la terminazione della procedura.  
- Il nome delle procedure deve essere del tipo *nomeProcedura\_1\_svc*, dove 1 (o qualunque altro intero positivo) indica il numero di versione.

4) Scrivere il file *client.c*, che implementa il main e la logica necessaria per trovare e invocare la procedura remota desiderata. Osserviamo che:  
- Preliminarmente il client deve invocare la seguente funzione:  
*clnt\_create (host, NUM\_PROG, NUM\_VERS, PROTOCOL)* dove PROTOCOL = “tcp” o “udp”.  
In particolare, *host* è l’hostname del server da contattare.  
- In base anche al punto precedente, il client deve conoscere l’hostname del server, il numero di programma, il numero di versione e il nome della procedura remota.  
- Il client deve gestire eventuali errori che possono verificarsi durante la chiamata remota, e lo fa tramite le chiamate a clnt\_pcreateerror() (in caso di errore nella creazione del client transport manager) e a clnt\_perror() (in caso di errore di comunicazione col server).

5) Compilare tutti i file sorgente (client, server, client stub, server stub e routine di conversione) e linkare i file oggetto.

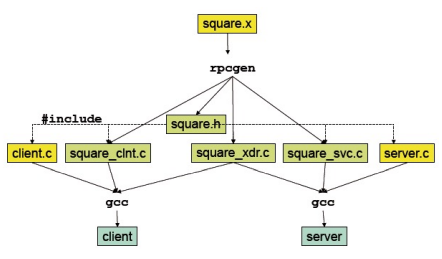
6) Quando il server viene avviato, pubblica il servizio (ovvero registra le procedure remote col name server, detto anche port mapper o RPCBIND).

7) Quando il client viene avviato, trova l’endpoint del servizio (il server) tramite il port mapper.

SunRPC features:-> Un programma tipicamente contiene diverse procedure remote e, per ognuna di esse, esistono molteplici versioni.  
-> Il server, di default, è sequenziale: può essere eseguita una sola call per volta, per cui viene garantita la mutua esclusione.  
-> Il client, quando contatta il server, entra in stato di blocco in attesa di risposta (comunicazione sincrona).  
-> La semantica di errore adottata è At-least-once.  
-> La comunicazione tra client e server è transiente (non persistente).

Server binding:Prima della comunicazione, il server RPC registra i programmi RPC nella **port map**, che è una tabella contenente i servizi RPC offerti in quella macchina e gestita da un singolo processo (il port mapper).  
Ciascuna entry della port map contiene:  
- Il numero di programma.  
- Il numero di versione.  
- Il protocollo di livello di trasporto (TCP o UDP).  
- Il numero di porta su cui contattare il server.  
Il numero di procedura non viene specificato, poiché tutte le procedure all’interno di un programma condividono il medesimo transport manager.

Il port mapper è in ascolto sulla porta 111 del server e supporta le seguenti operazioni:  
-> Inserimento di un servizio.  
-> Eliminazione di un servizio.  
-> Ricerca del numero di porta associato a un determinato servizio.  
-> Listing dei servizi registrati.

Development process:

**Lato server:** nella funzione main, il server stub crea una socket e vi collega qualunque porta disponibile. Dopodiché, invoca *svc\_register(),* una funzione della libreria RPC che serve a registrare le procedure col port mapper. Infine, attende le richieste chiamando *svc\_run()*, un’altra funzione della libreria RPC che ha il compito di invocare le procedure in risposta delle richieste provenienti dai client.

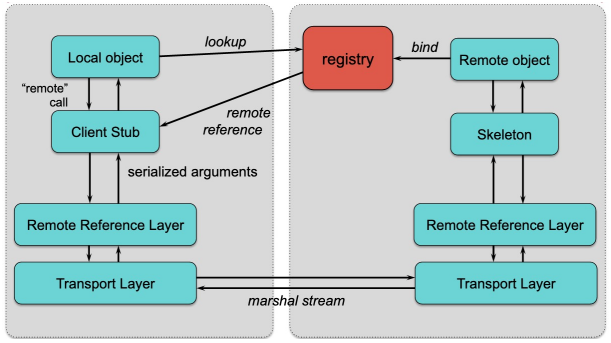
**Lato client:** quando viene avviato il programma del client, la funzione *clnt\_create()* contatta il port mapper lato server per trovare il numero di porta per l’interfaccia usata. Questa operazione è detta early binding e viene eseguita una sola volta (non a ogni chiamata a procedura). Dopodiché, sarà il client stub a gestire la comunicazione col server: è lui a implementare il timeout per le richieste e le operazioni di marshaling e unmarshaling.

Tipi di trasparenza (non) offerti da SunRPC:- SunRPC è trasparente all’accesso solo parzialmente: è vero che determinati meccanismi per l’invocazione a procedura remota (instanziazione degli stub, marshaling, unmarshaling e così via) non sono oggetto di preoccupazione per il programmatore, ma nel codice ci sono comunque delle differenze rispetto alle chiamate locali: c’è una convenzione sui nomi delle procedure ben precisa, ci sono dei vincoli sul numero dei parametri delle procedure, è necessario gestire molti più errori e il client deve conoscere l’host name del server.  
- SunRPC non è trasparente alla concorrenza: abbiamo infatti osservato che il server lavora in maniera sequenziale.  
- SunRPC non è trasparente alla replicazione: il port mapper è locale al server che offre il servizio, per cui dovrebbe essere replicato nel caso in cui si abbiano più server.

**Java RMI (Remote Method Invocation)**È un esempio di RPC di seconda generazione ed è un insieme di tool, politiche e meccanismi che consentono a un’applicazione Java di invocare i metodi di un oggetto remoto (instanziato su un host differente, tipicamente un server), rendendo possibile la comunicazione tra JVM diverse.  
Benché l’oggetto utilizzato sia in esecuzione su un server remoto, il client invoca i metodi remoti dell’oggetto mediante un riferimento locale. Ciò richiede una separazione prima di tutto logica tra l’interfaccia e l’implementazione dell’oggetto (ovvero la classe).  
In particolare, nel momento in cui il client si connette con l’oggetto nel server remoto, all’interno del client space viene caricato lo **stub**, che è la copia dell’interfaccia server e svolge un ruolo analogo al client stub di RPC. D’altra parte, le richieste che giungono all’oggetto nel server remoto vengono gestite dallo **skeleton**, che è un client agent ubicato nel server e svolge un ruolo analogo al server stub di RPC.  
Lo stub e lo skeleton vengono generati automaticamente.

Serializzazione e deserializzazione:Poiché, grazie al bytecode e alla portabilità del codice Java, gli ambienti di esecuzione di client e server sono simili tra loro, con Java RMI non è necessario eseguire le operazioni di marshaling e unmarshaling dei dati, bensì è sufficiente ricorrere alla serializzazione e deserializzazione, che sono operazioni direttamente supportate da Java.  
- **Serializzazione**: trasforma un oggetto in una sequenza di byte; per realizzarla, bisogna invocare il metodo *writeObject* su uno stream di output.  
- **Deserializzazione**: decodifica una sequenza di byte e ricostruisce il relativo oggetto; per realizzarla, bisogna invocare il metodo *readObject* da uno stream di input.  
Possiamo notare che la serializzazione è solo uno dei passaggi del marshaling.

Interazione tra stub e skeleton:1) Il client ottiene un’istanza dello stub.  
2) Il client invoca i metodi sullo stub.  
3) Lo stub serializza le informazioni necessarie per l’invocazione (ID del metodo e parametri) e le invia allo skeleton in un messaggio.  
4) Lo skeleton riceve e deserializza il messaggio, invoca il metodo richiesto, serializza il valore di ritorno e lo invia allo stub in un messaggio.  
5) Lo stub deserializza il valore di ritorno e lo restituisce al client.

In realtà, sotto a stub e skeleton esistono altri due layer intermedi:  


- **Remote Reference Layer**: gestisce i riferimenti remoti e le connessioni stream-oriented.  
- **Transport Layer**: gestisce le connessioni tra JVM differenti e può utilizzare diversi protocolli di livello di trasporto, purché siano connection-oriented (come TCP).

Dalla figura notiamo anche la presenza dell’**RMI registry**, che è l’equivalente del port mapper di SunRPC: permette al server di pubblicare i servizi e consente al client di ottenere una copia dello stub necessaria a invocare i metodi remoti. Per motivi di sicurezza, può essere eseguito solo sulla stessa macchina del server, per cui viene meno la trasparenza alla locazione.

Programmazione con Java RMI**:  
Server-side:**  
1) Definire un’interfaccia pubblica che estenda *java.rmi.Remote* cosicché i suoi metodi possano essere invocati da altre JVM. Ciascun metodo remoto:  
-> Deve dichiarare di essere in grado di lanciare l’eccezione *java.rmi.RemoteException* per indicare l’occorrenza di un errore di comunicazione o di protocollo.  
-> Accetta un qualunque numero di parametri in input ma può restituire solo un risultato.  
-> Ha una signature indistinguibile dai metodi locali (ad esempio il suo nome non deve seguire un pattern ben preciso, come invece accade con SunRPC).

2) Definire la classe che implementi l’interfaccia ed estenda *java.rmi.UnicastRemoteObject*; il termine **unicast** indica che l’oggetto non può essere duplicato. Ciascuna classe dovrebbe:  
-> Dichiarare l’interfaccia che sta implementando.  
-> Definire un costruttore.  
-> Fornire un’implementazione per ciascun metodo dichiarato nell’interfaccia.

3) Definire il codice del server, in cui:  
-> Venga creata un’istanza dell’oggetto.  
-> Venga registrato l’oggetto tramite l’RMI registry, utilizzando il metodo *bind()* o *rebind()* della classe *java.rmi.Naming* (*rebind()* rimpiazza un’associazione già esistente).  
Da qui si può notare come la registrazione dei servizi, a differenza di SunRPC, sia a cura non dello stub ma dello sviluppatore.

**Client-side:**  
1) Ottenere il riferimento all’oggetto remoto invocando sull’RMI registry il metodo *lookup()* della classe *java.rmi.Naming*.  
2) Assegnare il riferimento appena ottenuto a una variabile che ha l’interfaccia remota come tipo.  
3) Invocare i metodi remoti desiderati utilizzando la variabile di cui sopra.

Java RMI features**:**-> Il passaggio dei parametri avviene per valore per i tipi di dato primitivi e gli oggetti serializzabili, mentre avviene per riferimento per gli oggetti remoti.  
-> Java RMI offre supporto alla concorrenza; a tal proposito, per proteggere un metodo remoto da accessi concorrenti, esso deve essere definito come “synchronized”.  
-> Il client, quando contatta il server, entra in stato di blocco in attesa di risposta (comunicazione sincrona).  
-> La semantica di errore adottata è At-most-once.  
-> La comunicazione tra client e server è transiente (non persistente).

Garbage collector distribuito:  
Quando all’interno del server si ha un oggetto non più usato dai client, è bene deallocarlo. Affinché ciò sia possibile, l’oggetto deve tener traccia del numero dei client che lo stanno utilizzando. Tuttavia, si tratta di un requisito non semplice a causa di possibili crush dei client e fallimenti della rete, per cui si può affermare che Java RMI non è adatto alle applicazioni su larga scala.  
Quel che fa Java RMI per far funzionare correttamente la garbage collection è sfruttare un meccanismo basato sul **lease**. In particolare, la JVM del client:  
- Invia periodicamente un messaggio di **dirty** alla JVM del server mentre l’oggetto remoto è in uso.  
- Invia un messaggio di **clean** alla JVM del server quando l’oggetto non serve più.  
L’intervallo di tempo ogni cui la JVM del client deve inviare il messaggio di dirty è detto tempo di lease, scaduto il quale il server può assumere che il client sia andato in crash e, quindi, non stia più referenziando l’oggetto in questione.  
Nel momento in cui il server non riceve più alcun messaggio di dirty, può deallocare l’oggetto mediante il garbage collector.

**Confronto tra SunRPC e Java RMI**

|  |  |
| --- | --- |
| **SunRPC** | **Java RMI** |
| È process oriented. | È object oriented. |
| Offre una trasparenza incompleta all’accesso. | Offre una trasparenza migliore all’accesso. |
| Le entità che possono essere richieste sono le operazioni e le funzioni. | Le entità che possono essere richieste sono i metodi degli oggetti remoti. |
| Supporta la comunicazione sincrona (quella asincrona è disponibile solo in alcune estensioni di SunRPC). | Supporta solo la comunicazione sincrona. |
| La semantica di comunicazione è At-least-once. | La semantica di comunicazione è At-most-once. |
| Il server binding avviene tramite il port mapper. | Il server binding avviene mediante l’RMI registry. |
| Utilizza XDR come IDL per la rappresentazione dei dati. | Utilizza Java come IDL per la rappresentazione dei dati. |
| Il passaggio dei parametri avviene tramite il meccanismo di copy-restore. | Il passaggio dei parametri avviene per valore per i tipi di dato primitivi e gli oggetti serializzabili, mentre avviene per riferimento per gli oggetti remoti. |
| Non supporta né disaccoppiamento spaziale, né disaccoppiamento temporale, e nemmeno disaccoppiamento rispetto alla sincronia. | Non supporta né disaccoppiamento spaziale, né disaccoppiamento temporale, e nemmeno disaccoppiamento rispetto alla sincronia. |

**RPC in Go**Go supporta nativamente RPC, tant’è vero che comprende il package *net/rpc*.  
Vengono imposti dei vincoli sui metodi RPC:  
- Accettano solo due argomenti, di cui il secondo deve essere un puntatore a una struttura che memorizza la risposta da parte del server.  
- Devono restituire sempre un errore (che varrà nil se l’invocazione del metodo va a buon fine).

Anche marshaling e unmarshaling sono supportati da un package di Go, *encoding/gob* (gestisce gli stream come valori binari), che tuttavia richiede che sia il client che il server siano scritti in Go. Se non si vuole sottostare a questo vincolo, si possono considerare due alternative:  
-> Package *net/rpc/jsonrpc*  
-> gRPC  
In conclusione, poiché la conversione dati è supportata da un apposito package, non viene utilizzato un IDL specifico.

Server:  
Lato server è richiesta l’invocazione alle seguenti funzioni:  
- **Register** (o **RegisterName**), che serve a pubblicare i metodi che costituiscono una particolare interfaccia: a proposito di ciò, non è necessario fare un uso esplicito di un name server.  
- **Listen**, che serve a mettersi in ascolto delle richieste da parte dei client.  
- **Accept**, che serve ad accettare le connessioni sul listener restituito dalla Listen. È una funzione bloccante ma, se il server desidera portare avanti altro lavoro subito dopo la sua invocazione, può effettuarne la chiamata in una goroutine (keyword **go**).

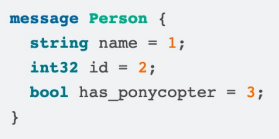
Client:  
Lato client è richiesta l’invocazione alle seguenti funzioni:  
- **Dial** (o **DialHTTP**), che serve a connettersi al server RPC relativo a uno specifico indirizzo IP e a uno specifico numero di porta.  
- **Call**, che serve a effettuare una chiamarta RPC sincrona, oppure **Go**, che serve a effettuare una chiamata RPC asincrona (in questo secondo caso la risposta del server viene inviata su un apposito canale).

Caratteristiche di RPC in Go**:**-> Fornisce una trasparenza all’accesso non completa a causa dei vincoli imposti sui parametri dei metodi.  
-> Non fornisce trasparenza alla locazione: il client, per connettersi con il server, deve conoscerne indirizzo IP e numero di porta.  
-> Viene supportata sia la comunicazione sincrona (funzione Call) sia la comunicazione asincrona (funzione Go).  
-> La semantica di errore adottata è At-most-once.  
-> La comunicazione tra client e server è transiente (non persistente).  
-> Si può avere disaccoppiamento rispetto alla sincronia, ma non quelli spaziale e temporale.

**gRPC**È un framework open-source multi-piattaforma, multi-linguaggio e nativamente utilizzabile per i servizi cloud e le applicazioni a microservizi.  
Un’architettura a microservizi è costituita da servizi indipendenti, autonomi, debolmente accoppiati e sviluppati da team diversi.

gRPC utilizza il protocollo **http/2** per il trasporto: in particolare, la sua idea di base è trattare le Remote Procedure Call come riferimenti agli oggetti http.  
Le principali caratteristiche di http/2 sono:  
- Presenza di un **binary framing layer** al di sopra degli strati di TCP e TLS che si occupa di suddividere le richieste / risposte in messaggi più piccoli e convertirle in un formato binario, in modo tale che la trasmissione sia particolarmente efficiente.  
- Possibilità di trasmettere stream di dati; a proposito di ciò, distinguiamo gli stream, i messaggi e i frame nel seguente modo:  
 -> **Stream** = flusso di byte bidirezionale in una connessione; può trasportare uno o più messaggi.  
 -> **Messaggio** = sequenza completa di frame che mappa in una richiesta o in una risposta.  
 -> **Frame** = la più piccola unità di comunicazione in http/2.  
- Supporto nativo dello streaming bidirezionale.  
- Compressione dell’header http per ridurre l’overhead del protocollo.

gRPC utilizza i **protocol buffer** sia come IDL sia per definire l’interfaccia del servizio. Inoltre, è basato sul solito proxy pattern, con stub (= client stub) e server (=server stub) generati automaticamente; a tal proposito, il protocol buffer fa anche da tramite per la comunicazione tra stub e server.

Step di base per programmare con gRPC:1) Utilizzare il protocol buffer come IDL per definire in un file .proto il servizio (= collezione di metodi remoti) e i tipi di messaggio che vengono scambiati tra client e server. In particolare, i messaggi definiscono i parametri di input e di output di ciascun metodo, sono fortemente tipati e hanno la seguente sintassi:  
  
dove 1, 2, 3 sono gli indici dei singoli campi del messaggio e sono univoci all’interno dello stesso messaggio.

2) Usare il **protocol buffer compiler** (protoc) per generare client e server scritti con linguaggi di programmazione arbitrari (anche diversi tra loro) a partire dal file .proto.  
Più specificatamente, vengono generati due file:  
- *NomeServizio.pb.go*, che contiene il codice per la gestione dei messaggi.  
- *NomeServizio\_grpc.pb.go*, che contiene gli stub di client e server, che servono rispettivamente a invocare i metodi definiti nel servizio e a implementare i metodi definiti nel servizio.

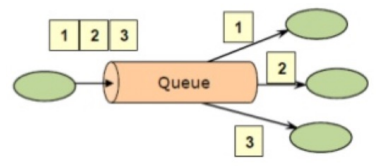
3) Implementare la logica del server e del client usando le API gRPC. In particolare:  
-> Il codice del server è composto da due parti: da un lato si ha l’implementazione dei metodi, mentre dall’altro si ha l’invocazione alle seguenti 4 API:  
 - **Listen**, che serve a mettersi in ascolto delle richieste da parte dei client.  
 - **NewServer**, che crea un nuovo server stub.  
 - **RegisterGreeterServer**, che serve a registrare il servizio.  
 - **Serve**, che attiva il server.  
-> Anche il codice del client è composto da due parti: all’inizio si ha l’invocazione delle seguenti 2 API:  
 - **Dial**, che serve a creare un canale di comunicazione su cui inviare le richieste al server.  
 - **NewGreeterClient**, che serve a creare lo stub.  
Infine, è possibile invocare sul client stub i metodi che compongono il servizio. Notiamo che il primo parametro è un oggetto *context* che ci permette di cambiare il comportamento della Remote Procedure Call, se necessario.

Tipi di metodi di gRPC**:**- **Simple RPC**: il client invia una richiesta al server e riceve un’unica risposta (è il caso considerato finora).  
- **Server-side streaming RPC**: il client invia una richiesta al server e riceve un flusso di risposte.  
- **Client-side streaming RPC**: il client invia un flusso di richieste e riceve un’unica risposta.  
- **Bidirectional streaming RPC**: sia il client che il server comunicano con flussi di messaggi.

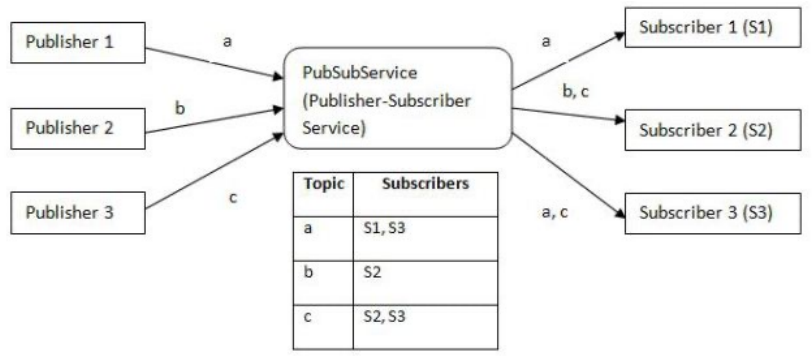
Debolezze di gRPC:-> Poiché utilizza solo il protocollo http/2 per il trasporto, non supporta le chiamate ai servizi gRPC tramite browser. Per mitigare tale problema, è possibile usare gRPC-Web, che però fornisce un supporto ai browser soltanto di alcune funzionalità gRPC.  
-> Il formato dei dati per i protocol buffer è binario, per cui non è human-readable; di conseguenza, l’attività di debugging deve essere supportata da particolari tool aggiuntivi.

**Comunicazione orientata ai messaggi**Abbiamo visto come RPC provi a fornire trasparenza alla distribuzione per la comunicazione tra componenti remoti; tuttavia, fa in modo che i componenti abbiano un certo grado di accoppiamento. Per questo motivo, introduciamo dei modelli di comunicazione message-oriented, che migliorano il disaccoppiamento e la flessibilità:  
- Message Passing Interface (MPI), che offre comunicazione transiente.  
- Message Oriented Middleware (MOM), che offre comunicazione persistente.

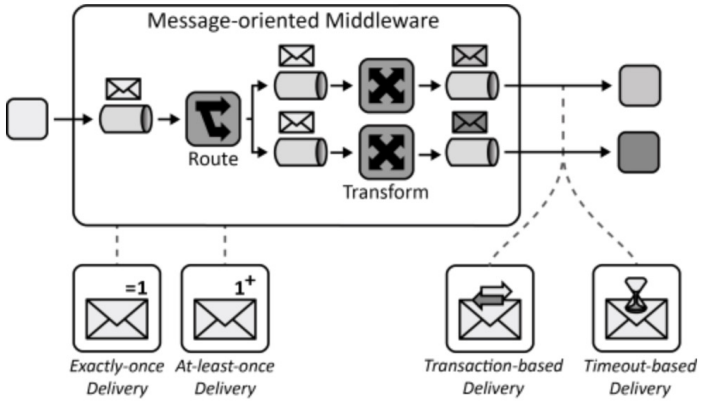
**Message Oriented Middleware (MOM)**È un middleware di comunicazione che offre disaccoppiamento spaziale e temporale, ma può anche supportare il disaccoppiamento rispetto alla sincronia. Tipicamente viene usato nelle architetture serverless e a microservizi.  
Esistono due pattern architetturali per il MOM: **message queue** e **publish-subscribe**.

Message queue pattern:  
Il mittente invia un messaggio alla coda di messaggi, dove viene memorizzato finché non viene ricevuto dal destinatario. Molteplici consumer (lettori) possono utilizzare la coda di messaggi ma ciascun messaggio può essere consegnato una sola volta a un solo consumer.  


I sistemi a code di messaggi mettono a disposizione le seguenti API:  
- **Put**: invio non bloccante di un messaggio alla coda (avviene tramite un’operazione di append).  
- **Get**: ricezione bloccante di un messaggio (tipicamente del primo messaggio della coda specificata, ma esistono varianti in cui si può fare ricerca di un messaggio specifico).  
- **Poll**: ricezione non bloccante di un messaggio (fallisce nel caso in cui non ci sia alcun messaggio).  
- **Notify**: ricezione non bloccante di un messaggio in cui viene installato un handler (una funzione di callback) che verrà chiamato automaticamente quando verrà inserito un messaggio nella coda specificata.

Publish-subscribe pattern:  
I componenti dell’applicazione possono pubblicare messaggi (i.e. eventi) in modo asincrono e/o dichiarare il loro interesse in determinati topic effettuando una sottoscrizione. Qui, a differenza del message queue pattern, ciascun messaggio può essere inviato a molteplici subscriber (= consumer), dato che più consumer possono sottoscriversi a uno stesso topic.  
Le sottoscrizioni sono gestite da un componente chiamato **event dispatcher**, che ha la responsabilità di effettuare il routing dei messaggi a tutti i subscriber interessati. Per motivi di scalabilità, l’implementazione dell’event dispatcher è distribuita.  


I sistemi publish-subscribe mettono a disposizione le seguenti API:  
- **Publish**: pubblicazione di un evento.  
- **Subscribe**: sottoscrizione agli eventi di interesse.  
- **Unsubscribe**: annullamento di una particolare sottoscrizione.  
- **Notify\_cb**: invio di un evento ai subscriber interessati (questa API viene utilizzata dal sistema publish-subscribe).

**Funzionalità del MOM**

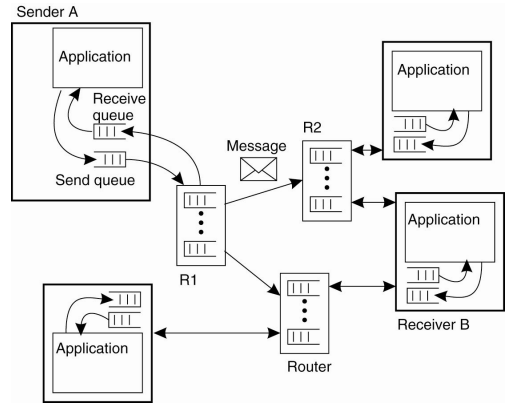
Semantiche di consegna:  
1) **Consegna At-least-once**: il consumer riceve il messaggio almeno una volta; in particolare, viene implementato il meccanismo di Request Retry. Ciò richiede che l’applicazione sia idempotente.

2) **Consegna Exactly-once**: il consumer riceve il messaggio almeno una volta; stavolta, oltre al meccanismo di Request Retry, viene implementato anche il Duplicate Filtering.  
Immagine che contiene testo, clipart

Descrizione generata automaticamente

3) **Consegna transaction-based**: affinché si abbia la garanzia che i messaggi vengano eliminati dalla message queue solo dopo essere stati ricevuti correttamente, le operazioni di *read* e di *delete* costituiscono una transazione che assume un comportamento ACID.

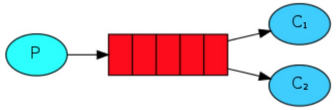
4) **Consegna timeout-based**: quando un messaggio viene ricevuto dal destinatario, non viene eliminato subito, bensì viene portato dallo stato **visible** allo stato **invisible**, in modo tale che sia ancora presente nella coda ma non possa essere ricevuto da nessuno; nel frattempo, viene avviato il cosiddetto **timer di visibilità**. A questo punto si possono avere due scenari:  
- Il mittente riceve un ack da parte del destinatario 🡪 il messaggio viene eliminato dalla coda.  
- Scatta il timeout di visibilità 🡪 il messaggio viene reso nuovamente visibile.

Routing di messaggi:  
Ciascuna coda di messaggi viene gestita da un **queue manager** (**QM**) e, in particolare:  
-> Un’applicazione può inserire messaggi solo in una coda locale.  
-> Un’applicazione può ricevere messaggi solo estraendoli da una coda locale.  
Di conseguenza, i QM hanno bisogno di effettuare il routing di messaggi e tutti insieme formano un’overlay network che può essere configurata staticamente oppure dinamicamente.  


Trasformazione di messaggi:  
Nel contesto dei MOM, l’eterogeneità nella rappresentazione dei dati viene affrontata mediante l’utilizzo di un **message broker**, il quale:  
- Converte i messaggi in ingresso nel formato target fornendo trasparenza all’accesso.  
- Gestisce un repository di programmi e di regole di conversione che supportano la trasformazione dei messaggi.  
- Può essere implementato in modo distribuito per incentivare la scalabilità e l’affidabilità.

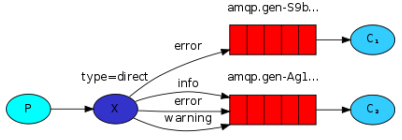
**RabbitMQ**È un famoso framework open-source a coda di messaggi che supporta diversi casi d’uso:  
1) Store & forward di messaggi che vengono inviati da un producer e ricevuti da un consumer (**message queue pattern**).  
Immagine che contiene testo, edificio

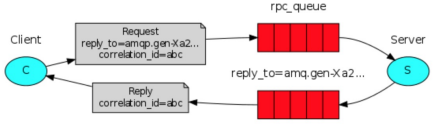
Descrizione generata automaticamente

2) Distribuzione di task tra molteplici worker (**competing consumers pattern**).  


3) Consegna di messaggi a più consumatori per volta utilizzando un message exchange (**publish-subscribe pattern**).  
Immagine che contiene testo, orologio, clipart

Descrizione generata automaticamente

4) Invio di messaggi a un message exchange, il quale seleziona le code a cui inoltrarli (**ricezione selettiva dei messaggi**).  


5) Esecuzione di una funzione su un nodo remoto + attesa del risultato (**request-reply pattern**).  


**IBM MQ**Storicamente è la prima tecnologia enterprise per le code di messaggi, e presenta le seguenti caratteristiche:  
- Le code risiedono sotto il regime di un queue manager (QM).  
- I processi possono inserire i messaggi in code locali oppure in code remote tramite un meccanismo RPC.  
- I messaggi vengono trasferiti da una coda all’altra, e il trasferimento dei messaggi richiede un canale.  
- In ciascun endpoint del canale vi è un **message channel agent** (**MCA**), che è responsabile dell’istanziazione dei canali e di invio, ricezione e cifratura dei messaggi.  
- I canali di comunicazione sono unidirezionali.  
- È possibile definire un’overlay network in cui la configurazione del routing è manuale.

**Amazon SQS (Simple Queue Service)**È un servizio cloud di code di messaggi basato su polling. Il suo obiettivo è disaccoppiare i componenti di un’applicazione cloud-native, tant’è vero che questi possono essere eseguiti indipendentemente, possono comunicare in maniera asincrona e possono essere sviluppati con tecnologie differenti.  
Amazon SQS implementa la consegna timeout-based e può essere combinato con Amazon SNS (Simple Notification Service) per inviare messaggi a molteplici code SQS in parallelo.

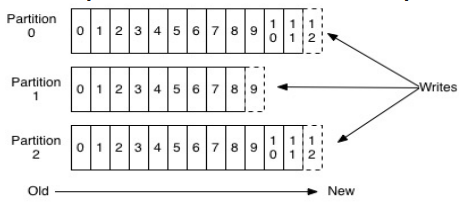
Le API offerte da Amazon SQS sono:  
-> CreateQueue, ListQueues, DeleteQueue.  
-> SendMessage, ReceiveMessage.  
-> DeleteMessage.  
-> ChangeMessageVisibility.  
-> SetQueueAttributes, GetQueueAttributes.  
Notiamo che la SendMessage può inserire in una determinata coda solo messaggi di dimensione non superiore ai 256 KB: per messaggi più grandi, si aggiunge nella coda un riferimento (un link) al payload del messaggio il quale invece viene memorizzato in un data storage (Amazon S3).

**Apache Kafka**È un sistema publish-subscribe distribuito scritto nel linguaggio Scala e con un throughput di ingestion (velocità nel caricare messaggi) molto elevato.

I messaggi sono suddivisi in categorie chiamati topic. Gli attori in gioco sono i **producer**, che pubblicano i messaggi, i **consumer**, che si sottoscrivono a determinati topic, e il **cluster di Kafka**, che per ogni topic mantiene un log di dati.

Il log è una struttura dati suddivisa in un certo numero fissato di partizioni; ciascuna partizione rappresenta un’unità di parallelismo del topic (infatti il partizionamento consente gli accessi paralleli) e consiste in una sequenza ordinata, numerata e immutabile di record a cui vengono effettuate le operazioni di append. In particolare, ogni record è associato a un sequence number via via crescente chiamato offset.



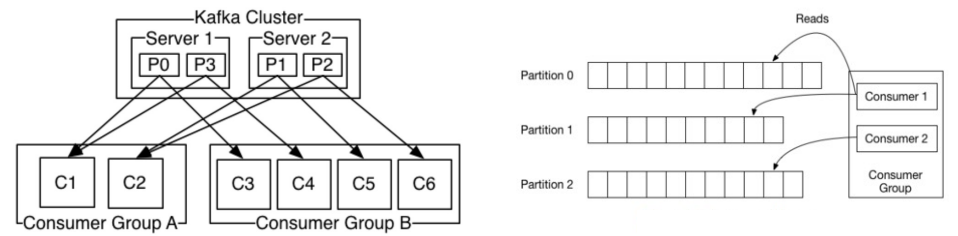
Si ha una forte garanzia sull’ordinamento dei record soltanto all’interno delle singole partizioni: per avere un ordinamento totale a livello di topic, il producer dovrebbe attuare dei particolari accorgimenti.  




Per motivi di scalabilità, le partizioni di ciascun log sono distribuite e gestite da un insieme di server detti broker.  
Inoltre, per incentivare la tolleranza ai guasti, le partizioni vengono replicate su un numero configurabile di broker. In particolare, ogni partizione ha un broker **leader** e 0+ broker **follower**; le operazioni di read e write sono gestite dal leader, mentre i follower fungono unicamente da backup. Per aumentare ulteriormente la tolleranza ai guasti, si fa in modo che, a ogni scrittura, il messaggio non venga reso disponibile ai consumer finché tutti i follower della partizione interessata non hanno a loro volta completato l’operazione di write e hanno inviato un ack al leader.  
Per distribuire al meglio il carico di lavoro tra i broker, si fa in modo che ciascuno di essi sia leader per alcune delle sue partizioni e sia follower per le altre.

Producer:  
Pubblicano i record su una partizione di un topic a loro scelta (la selezione della partizione può avvenire secondo uno schema round-robin oppure sull’utilizzo di una chiave) e, nel farlo, inviano i dati direttamente (i.e. senza alcun intermediario) al broker che funge da leader per quella partizione.  
Più producer differenti possono scrivere sulla medesima partizione.

Consumer:  
Possono essere implementati secondo due modelli diversi:  
- **Push model**: il broker effettua la push dei messaggi verso i consumer. Il problema è che si potrebbero avere dei consumer che girano a velocità differenti, e il broker deve adattarsi di conseguenza. Inoltre, il broker deve decidere se inviare ciascun messaggio immediatamente oppure inviare i dati solo dopo averne accumulati un po’.  
- **Pull model**: il consumer ha la responsabilità di recuperare i messaggi dal broker e, per farlo, deve mantenere un offset che identifica il prossimo messaggio che dovrà processare. Qui si ha maggiore scalabilità (i broker sono meno carichi) e maggiore flessibilità (è più facile gestire consumer che hanno necessità e capacità computazionali diverse). D’altra parte, però, i consumer possono essere soggetti al busy waiting. Comunque sia, questo è il modello utilizzato da Kafka.

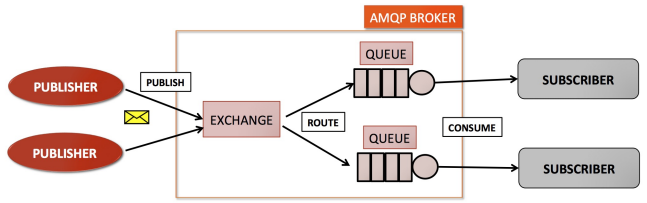
Per migliorare la scalabilità e la tolleranza ai guasti, i consumer possono essere raggruppati e condividere un group ID comune, formando così un **consumer group**, il quale è relativo a un subscriber logico.  


Semantiche di comunicazione supportate da Kafka:  
-> **At-least-once** (default).  
-> **Exactly-once** (non è però un’Exactly-once completa e in più utilizza un protocollo oneroso – il 2 phase commit - per la gestione delle transazioni).  
-> L’utente può anche implementare la sematica **At-most-once** disabilitando le ritrasmissioni sul producer e facendo effettuare il commit al consumer prima di processare un messaggio.

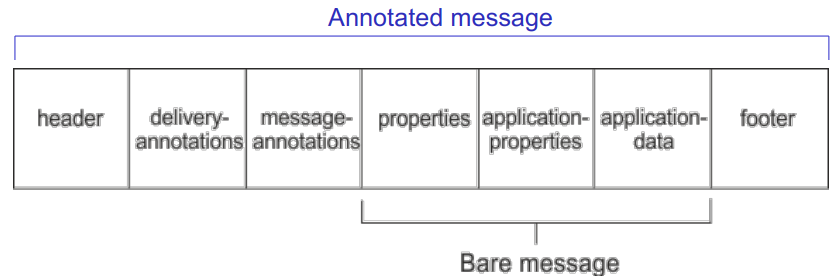
Kafka e ZooKeeper:  
ZooKeeper è un datastore di tipo {chiave, valore} e ha un’architettura distribuita.  
Kafka usa ZooKeeper per coordinare i producer, i consumer e i broker. In particolare, ZooKeeper memorizza i metadati di Kafka, che comprendono:  
- La lista di broker.  
- La lista di consumer e i relativi offset da cui riprendere a leggere.  
- La lista di producer.

API di Kafka:  
-> **Producer API**: servono a pubblicare nuovi record su qualche topic.  
-> **Consumer API**: servono a leggere dei record da qualche topic.  
-> **Connect API**: servono a implementare connettori riutilizzabili (per connettere i topic di Kafka con applicazioni esistenti in modo tale da poter spostare grandi insiemi di dati da e verso Kafka).  
-> **Stream API**: servono a processare i dati che arrivano (e.g. filtrarli).

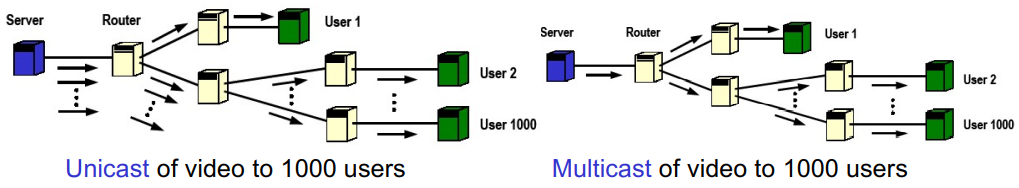
**Protocolli per MOM**Sono protocolli per sistemi a code di messaggi che garantiscono disaccoppiamento, resilienza ai guasti e gestione dei picchi di carico. Sono agnostici rispetto alla piattaforma utilizzata e sono spesso utilizzati nell’ambito dell’Internet of Things. Tra questi abbiamo:  
- AMQP (Advanced Message Queueing Protocol).  
- MQTT (Message Queue Telemetry Transport).  
- STOMP (Simple Text Oriented Messaging Protocol).  
Tra questi introdurremo AMQP.

**AMQP**È un protocollo applicativo di tipo binario che si appoggia su TCP ed è programmabile. Coinvolge tre attori principali: i **publisher**, i **subscriber** e i **broker**. All’interno di un broker si hanno tre entità: le **code**, gli **exchange** e i **binding**.  
-> I publisher pubblicano i messaggi sugli exchange (che possiamo vederli come delle mailbox).  
-> Gli exchange distribuiscono le copie dei messaggi alle code utilizzando delle particolari regole chiamate binding.  
-> È possibile sia che i broker effettuino la push dei messaggi, sia che i subscriber richiedano i messaggi tramite una pull.  


Esistono vari binding disponibili:  
1) **Direct exchange**: ciascun messaggio viene inoltrato alle code associate a una particolare routing key.  
2) **Fanout exchange**: ciascun messaggio viene inoltrato a tutte le code collegate all’exchange.  
3) **Topic exchange**: ciascun messaggio viene inoltrato alle code associate a un particolare topic.  
4) **Headers exchange**: ciascun messaggio viene inoltrato alle code associate a determinati attributi nell’header.

AMQP, infine, definisce due tipi di messaggi:  
- Bare messagges, che vengono forniti dal server.  
- Annotated messages, a cui vengono aggiunti un header e un footer da AMQP durante il transito. In particolare, l’header specifica i parametri di consegna.  


**Comunicazione multicast**È un pattern di comunicazione in cui i dati vengono inviati a più destinatari per volta. Un caso speciale di multicast è il broadcast, in cui i dati vengono inviati a tutti i destinatari possibili.  
La comunicazione multicast può essere:  
- **One-to-many** (e.g. distribuzione di file).  
- **Many-to-many** (e.g. tool di conferenza).

La comunicazione unicast, a differenza della multicast, non scala poiché prevede la generazione di n flussi di dati da parte del nodo sorgente.  


Il multicast può essere realizzato a livello di rete oppure a livello applicativo.

Network-level multicast:  
La replicazione dei pacchetti e il routing sono gestiti dai router.  
Il protocollo **IPMC** (IP Multicast) generalizza UDP mediante un comportamento one-to-many ed è basato sui gruppi. Un gruppo è un insieme di host interessati nella stessa applicazione multicast e identificati dal medesimo indirizzo IP (compreso tra 224.0.0.0 e 239.255.255.255). Per inserirsi all’interno di un gruppo si utilizza il protocollo **IGMP** (Internet Group Management Protocol).

IPMC non è supportato su larga scala. Infatti, presenta problemi nel tenere traccia dell’appartenenza al gruppo e viene tendenzialmente disabilitato dalle piattaforme cloud a causa del **broadcast storm problem** (crescita esponenziale del traffico di rete con possibile saturazione).

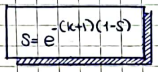
Application-level multicast:  
La replicazione dei pacchetti e il routing sono gestiti dagli end host.  
Qui i nodi sono organizzati in un’overlay network, che può essere:  
- **Strutturata**, con dei path di comunicazione espliciti, che possono essere dei Tree (alberi) o Mesh (maglie).  
- **Non strutturata**, con la possibilità di utilizzare un meccanismo di comunicazione basato su flooding, random walk oppure gossiping.

Consideriamo un albero di multicast strutturato in Scribe (dove Scribe è un sistema publish-subscribe con architettura decentralizzata basata sulla DHT Pastry):  
-> Il nodo che inizializza la sessione genera un identificatore multicast (**mid**).  
-> Il nodo inizializzatore, usando Pastry, cerca **succ(mid)**, che è il nodo responsabile per il multicast mid.  
-> Il succ(mid) diventa il root dell’albero multicast.  
-> Se il nodo P vuole aggiungersi al multicast, gli invia una richiesta di join.  
-> Quando la richiesta arriva al nodo Q, P diventa child di Q; se Q non aveva mai visto la richiesta di join, diventa forwarder, altrimenti non c’è bisogno di inoltrare ulteriormente la richiesta.

**Gossiping**I protocolli basati sul gossiping sono probabilistici e sono anche detti epidemici. L’idea è che il mittente invia un messaggio a un sottoinsieme dei suoi vicini scelti casualmente; dopodiché ciascuno di essi inoltra a sua volta il messaggio a un sottoinsieme dei suoi vicini scelti randomicamente, e così via. Tale meccanismo consente la disseminazione delle informazioni nelle reti a larga scala; la propagazione è definita lazy (non immediata).

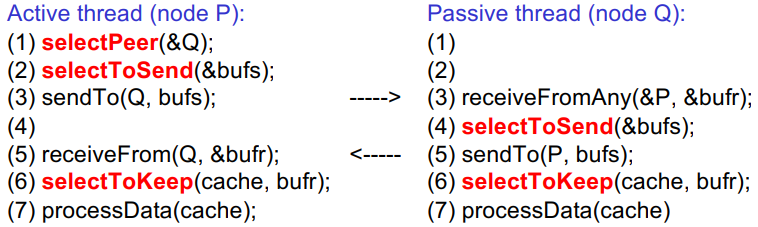
Il gossiping è molto utilizzato nei sistemi distribuiti su larga scala poiché presentano i seguenti vantaggi:  
- **Semplicità** degli algoritmi di gossiping.  
- **Assenza di un controllo centralizzato** (coi relativi bottleneck e single point of failure).  
- **Scalabilità**: ciascun nodo invia solo un numero limitato di messaggi indipendentemente dalla dimensione complessiva del sistema.  
- **Affidabilità e robustezza** grazie alla ridondanza dei messaggi.

Il gossiping è molto utilizzato da AWS S3, Amazon Dynamo, BitTorrent e Cassandra.

Modelli di propagazione:  
-> **Pure gossiping** (aka **rumor spreading**): un nodo (P) che ha appena ricevuto un aggiornamento (i.e. è stato appena contaminato) lo inoltra a un certo numero di altri nodi, contaminandoli a loro volta.  
Supponiamo che tra questi nodi rientri Q. Se Q conosceva già l’aggiornamento, P può perdere interesse nel propagare il gossip e smette di contattare altir nodi con probabilità 1/k (dove k è un parametro che indica il grado di diffusione).  
Se denotiamo con s la frazione dei nodi che non sono mai stati aggiornati, è possibile mostrare che:  
  
Notiamo che, all’aumentare di k, anche il valore di s cresce.  
Comunque sia, se vogliamo davvero assicurarci che tutti i nodi vengano aggiornati, il pure gossiping da solo non è sufficiente ma dovremmo combinarlo con un altro modello di propagazione: l’anti-entropia.



-> **Anti-entropy**: il suo obiettivo è incrementare la similarità tra gli stati dei nodi, in modo tale da diminuire il disordine (da qui deriva il nome “anti-entropy”). Un nodo (P) seleziona un altro nodo Q randomicamente; l’aggiornamento può avvenire in tre modi differenti:  
1) Push: P è l’unico a inviare i dati a Q.  
2) Pull: P è l’unico a richiedere gli aggiornamenti da Q.  
3) Push-pull: P, Q si inviano gli aggiornamenti a vicenda. Questa è la strategia più veloce e richiede O(log(N)) round per disseminare i messaggi a tutti gli N i nodi, dove un round (= gossip cycle) è l’intervallo di tempo in cui tutti i nodi hanno preso l’iniziativa di iniziare uno scambio di informazioni.

Algoritmo di base del gossiping:  


- selectPeer: seleziona un vicino randomicamente.  
- selectToSend: seleziona dalla cache locale alcune entry da inviare al destinatario.  
- selectToKeep: seleziona quali entry ricevute memorizzare nella cache locale; in particolare, rimuove le entry replicate.

Ora introdurremo due esempi di protocolli di gossiping: **Blind counter rumor mongering** e **Bimodal multicast**.

**Blind counter rumor mongering**Da dove viene fuori il nome di questo protocollo?  
-> Rumor mongering = atto del diffondere un rumore = gossip.  
-> Blind = perdita di interesse nel gossip a prescindere da quale sia il nodo destinatario.  
-> Counter = perdita di interesse nel gossip dopo un certo numero di contatti.

Per questo protocollo sono definiti due parametri:  
- **B**: numero di vicini che ciascun nodo contatta a ogni iterazione.  
- **F**: numero di volte in cui ogni nodo riceve un determinato aggiornamento prima di smettere di propagarlo.

Il meccanismo è il seguente:  
-> Il nodo sorgente inizia la comunicazione inviando il messaggio m a B vicini scelti randomicamente.  
-> Quando il nodo p riceve m dal nodo q, controlla se, complessivamente, non ha ricevuto m più di F volte. Se il check passa, allora inoltra m a B vicini tra quelli che teoricamente non hanno mai visto il messaggio.  
Almeno per quel che sa p, i nodi che hanno già ricevuto almeno una volta il messaggio sono quelli che lo hanno inviato direttamente a p e quelli che lo hanno ricevuto direttamente da parte di p.

**Bimodal multicast**Detto anche **pbcast** (probabilistic multicast), è un protocollo composto da due fasi:  
1) Message distribution: un processo invia un messaggio in multicast senza particolari garanzie di affidabilità.  
2) Gossip repair: a intervalli regolari, ciascun processo contatta un sottoinsieme di peer per confrontare i loro stati ed eliminare i gap (le differenze) nelle sequenze di messaggi che hanno ricevuto nella prima fase.  
In particolare, le informazioni scambiate tra i processi in questa seconda fase sono i **digest**, che identificano i messaggi ricevuti in precendenza (senza però includerli). Nel momento in cui un peer realizza di aver ricevuto qualche messaggio in meno del dovuto, invia una sollecitazione al processo con cui ha scambiato i digest affinché esso risponda coi messaggi mancanti originali.

Il termine bimodale deriva da due particolari caratteristiche del protocollo:  
- Nella stragrande maggioranza dei casi, le informazioni vengono consegnate o a quasi tutti i processi o a quasi nessun processo.  
- Nella stragrande maggioranza dei casi, le informazioni vengono consegnate o con ritardi estremamente piccoli, o con ritardi estremamente grandi.

**Event matching**Detto anche **notification filtering**, è una funzionalità centrale dei sistemi publish-subscribe che, ogni qualvolta viene pubblicato un evento, si occupa di:  
- Verificare le sottoscrizioni a quel tipo di evento.  
- Notificare i subscriber che hanno effettuato tali sottoscrizioni.

Per implementare l’event matching esistono diverse soluzioni architetturali:  
-> Architettura centralizzata: è semplice da realizzare ma non scala.  
-> Architettura master-worker (o gerarchica): si ha un unico nodo master che suddivide il lavoro tra più nodi worker; ciascun worker si occupa di un sottoinsieme delle sottoscrizioni. Il lavoro può essere ad esempio partizionato utilizzando l’hashing dei nomi dei topic per mappare le sottoscrizioni e gli eventi sui worker (a patto che il sistema publish-subscribe sia topic-based).  
-> Architetttura flat: è simile all’architettura master-worker ma prevede l’uso di più server distribuiti (chiamati broker) tra cui il lavoro viene suddiviso, in modo tale da fare a meno del singolo nodo master.  
-> P2P unstructured overlay: usa il flooding o il gossiping per propagare le informazioni.  
-> P2P structured overlay.

**VIRTUALIZZAZIONE**

**Introduzione**La virtualizzazione è un’astrazione delle risorse computazionali che consente all’utilizzatore di avere una vista logica del sistema diversa dalla vista fisica. Ciò avviene disaccoppiando il comportamento e l’architettura delle risorse hardware e software percepiti dall’utente dalla loro realizzazione fisica.

Abbiamo tre componenti fondamentali:  
1) **Guest** = componente del sistema che interagisce col layer di virtualizzazione (anziché con l’host) e ha una vista virtuale del sistema.  
2) **Host** = ambiente originale sottostante al layer di virtualizzazione.  
3) **Layer di virtualizzazione** = strato che ha la responsabilità di ricreare un ambiente virtuale possibilmente diverso da quello reale dove il guest andrà a operare.

Ma cosa è possibile virtualizzare?  
- L’ambiente di esecuzione con le relative risorse hardware e software (noi ci focalizzeremo su questo).  
- Lo storage (e.g. Storage Area Network).  
- La rete (e.g. VLAN, VPN).  
- Il data center.

**Vantaggi della virtualizzazione**-> Facilita la compatibilità, la portabilità, l’interoperabilità e la migrazione delle applicazioni e degli ambienti di esecuzione da una macchina all’altra (politica **create once, run anywhere**).  
-> Permette il **consolidamento dei server**: è possibile avere dei server software (e.g. http server, application server e DB server) su macchine virtuali diverse ma all’interno dello stesso host fisico, necessitando così di molti meno host. Ciò porta a una riduzione dei costi, dei consumi energetici, dello spazio occupato e dei tempi di downtime (poiché è possibile migrare una VM da un server a un altro mentre è in funzionamento).  
-> Permette di isolare componenti malfunzionanti o soggetti ad attacchi di sicurezza, incrementando così l’affidabilità e la sicurezza delle applicazioni.  
-> Permette di isolare le prestazioni di diverse VM in esecuzione su uno stesso host tramite uno scheduling delle risorse fisiche tra le VM.  
-> Consente di bilanciare il carico sui server mediante la migrazione di una VM da un server a un altro.

**Interfacce di sistema**- User-level ISA: comprende le istruzioni macchina non privilegiate invocate da qualunque programma.  
- System ISA: comprende le istruzioni macchina privilegiate, usate per lo più per la gestione delle risorse del sistema.  
- ABI (Application Binary Interface): comprende le system call.  
- API (Application Programming Interface): è una funzione di libreria.

**Livelli di virtualizzazione**1) Livello ISA: il suo obiettivo è emulare una data ISA (e.g. MIPS) a partire dall’ISA della macchina host (e.g. x86); l’emulazione può avvenire tramite **code interpretation** (più lenta poiché lavora istruzione per istruzione) oppure tramite **dynamic binary translation** (più veloce poiché lavora a blocchi di istruzioni).  
2) Livello hardware (aka virtualizzazione di sistema): il suo obiettivo è virtualizzare le risorse di un computer come i processori, la memoria e i dispositivi I/O. È basato sul **Virtual Machine Monitor** (**VMM**, detto anche **hypervisor**), il quale gestisce le risorse hardware e le condivide tra molteplici VM, garantendo isolamento e protezione delle VM.  
3) Livello di sistema operativo (aka container): il suo obiettivo è creare molteplici container isolati.  
4) Livello di libreria: il suo obiettivo è creare un ambiente di esecuzione per far girare delle applicazioni che altrimenti non sarebbero compatibili con l’ambiente host.  
5) Livello di applicazione utente (aka process VM): il suo obiettivo è compilare un’applicazione in un codice intermediario e portabile (e.g. Java bytecode) ed eseguirla in una piattaforma virtuale (e.g. JVM); tale piattaforma può eseguire un singolo processo.

**Virtualizzazione di sistema**Qui l’**host** è la piattaforma di base sulla quale vengono eseguite le VM, mentre il **guest** rappresenta tutto ciò che si trova all’interno della singola VM.

La virtualizzazione di sistema viene classificata in base a due assi:  
1) Dove viene posto il VMM?  
- **VMM di sistema** (o **type-1 hypervisor**): viene eseguito direttamente sull’hardware e offre funzionalità di virtualizzazione integrate in un sistema operativo semplificato. Può avere un’architettura a micro-kernel oppure monolitica.  
- **VMM ospitato** (o **type-2 hypervisor**): viene eseguito sul sistema operativo host, accede alle risorse hardware tramite le chiamate di sistema del sistema operativo host ed emula l’ISA dell’hardware virtuale per i sistemi operativi guest. Poiché il sistema operativo host rappresenta un layer di indirezione aggiuntivo, questa soluzione porta a un degrado delle prestazioni (anche se non occorre modificare il sistema operativo guest).

2) Come viene gestita l’esecuzione delle istruzioni privilegiate?  
- **Virtualizzazione completa**: il VMM espone a ogni VM interfacce hardware simulate funzionalmente identiche a quelle della macchina fisica sottostante; in particolare, intercetta le richieste di accesso privilegiato all’hardware e ne emula il comportamento atteso.  
- **Paravirtualizzazione**: il VMM espone a ogni VM interfacce hardware simulate funzionalmente simili (ma non identiche) a quelle della macchina fisica sottostante; non viene emulato l’hardware, bensì viene creato uno strato minimale di software (Virtual Hardware API) per assicurare la gestione delle VM e il loro isolamento.

Teorema di Popek e Goldberg:  
Introduciamo le seguenti definizioni:  
-> **Istruzione privilegiata** = istruzione che invia un trap (un’eccezione) e trasferisce il controllo al VMM se viene eseguita in user mode.  
-> **Stato privilegiato** = stato che determina l’allocazione delle risorse.  
-> **Istruzione sensibile** = istruzione che modifica lo stato privilegiato (control sensitive instruction) oppure lo espone (behavior sensitive instruction).

Il teorema afferma che è possibile costruire un VMM se l’insieme delle istruzioni sensibili è incluso nell’insieme delle istruzioni privilegiate.  
In altre parole, è sufficiente che tutte le istruzioni che possono coinvolgere il funzionamento del VMM (ovvero le istruzioni sensibili) inviino un trap e passino il controllo al VMM stesso, il quale verifica la correttezza dell’operazione richiesta e ne emula il comportamento (meccanismo **trap-and-emulate**).

Tuttavia, l’implementazione del trap-and-emulate può essere complessa: infatti, le architetture comuni (come x86, MIPS, ecc.) non sono virtualizzabili secondo il teorema di Popek e Goldberg.

Problemi nel realizzare la virtualizzazione di sistema:  
L’architettura del processore opera secondo almeno due livelli (**ring**) di protezione:  
- **Ring 0**: privilegi massimi (supervisor mode).  
- **Ring 3**: privilegi minimi (user mode).  
Con la virtualizzazione, il VMM (ring 0) opera in supervisor mode, mentre il sistema operativo guest (ring 1) e le applicazioni (ring 3) operano in user mode. Da qui sorgono due problemi:  
-> **Ring deprivileging**: poiché il sistema operativo guest opera in un ring che non gli è proprio, non può eseguire le istruzioni privilegiate.  
-> **Ring compression**: poiché le applicazioni e il sistema operativo guest eseguono allo stesso livello, occorre proteggere lo spazio del sistema operativo.

Virtualizzazione completa:  
Risolve il problema del ring deprivileging mediante il meccanismo trap-and-emulate, che può essere realizzato in due modi:  
- **Hardware-assisted CPU virtualization**: è la soluzione adottata se il processore fornisce supporto alla virtualizzazione; fornisce alla CPU due modalità di operare, chiamate **root mode** e **non-root mode**. Il VMM gira in root mode (Root-Ring 0), mentre il sistema operativo guest esegue in non-root mode nel suo ring originale (Non-Root Ring 0): in tal modo vengono risolti sia il problema del ring deprivileging che il problema del ring compression.  
- **Fast binary translation**: è la soluzione adottata se il processore non fornisce supporto alla virtualizzazione. Qui il VMM scansiona il codice prima della sua esecuzione per sostituire i blocchi contenenti istruzioni privilegiate con blocchi funzionalmente equivalenti e contenenti istruzioni per la notifica di eccezioni al VMM. Ciò comporta una maggiore complessità del VMM e minori prestazioni.

-> **Vantaggi** della virtualizzazione completa: non occorre modificare il sistema operativo guest e si ha un isolamento completo tra le istanze di VM.  
-> **Svantaggi** della virtualizzazione completa: richiede o una collaborazione del processore o un VMM molto più complesso.

Paravirtualizzazione:  
Si tratta di una soluzione alla virtualizzazione non trasparente, in cui il kernel del sistema operativo guest deve essere modificato affinché invochi le API virtuali esposte dall’hypervisor. In particolare, le istruzioni non virtualizzabili vengono rimpiazzate dalle **hypercall**, che sono trap software diretti verso l’hypervisor.  
  
Anche qui il problema del ring deprivileging viene risolto: il sistema operativo guest opera nel ring 0.

-> **Vantaggi** della paravirtualizzazione: l’implementazione è semplice e non richiede una collaborazione del processore.  
-> **Svantaggi** della paravirtualizzazione: richiede che il codice sorgente dei sistemi operativi guest sia disponibile; se non lo fosse (come accade in Windows), è necessario ricorrere a dei driver ad-hoc che mappano l’esecuzione delle istruzioni non virtualizzabili sulle API virtuali esposte dal VMM.

Architettura dei VMM:  
Un VMM è composto da tre moduli principali:  
- **Dispatcher**: è l’entry point del VMM che inoltra le istruzioni privilegiate invocate dalla VM verso uno degli altri due moduli.  
- **Allocator** (o **scheduler**): decide come distribuire le CPU virtuali tra le VM in esecuzione.  
- **Interpreter**: esegue una routine appropriata quando la VM invoca un’istruzione privilegiata.

**Virtualizzazione della memoria**Noi sappiamo che in un ambiente non virtualizzato si ha l’**one-level memory mapping**, in cui le tabelle delle pagine contengono le conversioni dagli indirizzi di memoria virtuali a quelli fisici.

In un ambiente virtualizzato, invece, tutte le VM condividono la memoria di uno stesso host e il VMM ha bisogno di partizionare la memoria tra le VM. Per questo motivo, si ha il **two-level memory mapping**, in cui si passa dai **guest virtual address** (**GVA**) ai **guest physical address** (**GPA**) agli **host physical address** (**HPA**).  
In particolare:  
- Guest virtual memory = memoria visibile alle applicazioni, che consiste in un address space virtuale composto da indirizzi contigui.  
- Guest physical memory = memoria visibile al sistema operativo guest.  
- Host physical memory = memoria hardware reale visibile al VMM.

Immagine che contiene testo, antenna

Descrizione generata automaticamente

Esistono due soluzioni per evitare un crollo delle prestazioni dovute al two-level memory mapping: la **shadow page table** (**SPT**) e la **second level address translation** (**SLAT**).

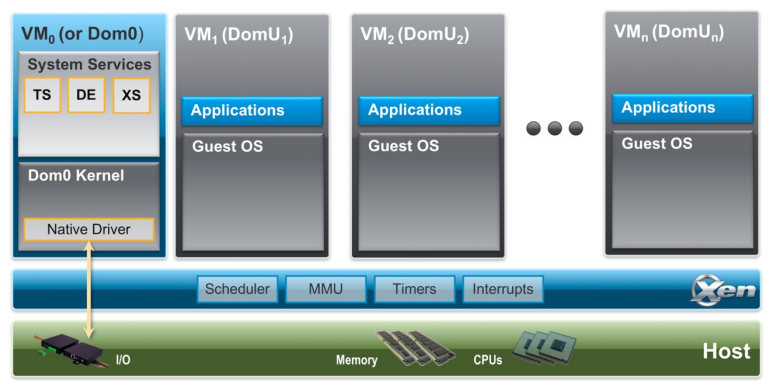
Shadow page table:  
È una tabella mantenuta dall’hypervisor e caricata all’interno del MMU (unità di gestione di memoria) che fa il mapping direttamente da GVA a HPA (senza passare per il GPA). Quando il sistema operativo guest modifica la sua tabella delle pagine, le modifiche vanno riportate nella SPT a seguito di un trap.  
Tale soluzione presenta le seguenti problematiche:  
- Il sistema operativo guest si aspetta uno spazio di memoria contiguo che inizia dall’indirizzo 0, ma la memoria della macchina sottostante potrebbe non essere contigua: il VMM deve preservare questa illusione.  
- L’implementazione della SPT è complessa.  
- Le SPT richiedono parecchio spazio di memoria e causano un overhead sul tempo di esecuzione poiché, per essere mantenute, richiedono che le operazioni del sistema operativo guest vengano intercettate.

Second level address translation:  
È una soluzione hardware-assisted per la virtualizzazione della memoria e, in particolare, per la traduzione dei GVA in HPA. Rispetto all’utilizzo delle SPT, porta a un guadagno delle prestazioni di circa il 50%.

**Xen**È un VMM di sistema open-source con un design a microkernel. Nonostante sia l’esempio più famoso di paravirtualizzazione, supporta anche la virtualizzazione hardware-assisted. Inoltre offre ai sistemi operativi guest un’interfaccia virtuale (hypercall API) utile per accedere alle risorse della macchina fisica.  
Gli elementi che possono essere paravirtualizzati sono:  
- Disco e dispositivi di rete.  
- Interrupt e timer.  
- Scheda madre simulata.  
- Istruzioni privilegiate e tabelle delle pagine.

Vantaggi:  
-> È un hypervisor scritto con poche righe di codice (circa 300.000), per cui occupa poco spazio, è scalabile ed è più robusto e sicuro rispetto ad altri hypervisor.  
-> Ha un’ampia comunità di sviluppatori ed è in continuo miglioramento.  
-> Può essere configurato.  
-> Presenta un overhead contenuto.  
-> Supporta la migrazione live delle macchine virtuali.

Svantaggi:  
-> L’I/O non è particolarmente efficiente.

Architettura:  


Si hanno diversi domini che distinguiamo in:  
- **DomU** (domini non privilegiati): sono le VM guest, che sono totalmente isolate dall’hardware (ad esempio non hanno accesso privilegiato all’hardware).  
- **Dom0** (dominio di controllo): è un dominio speciale che si occupa dell’esecuzione delle funzioni di controllo di Xen e delle istruzioni privilegiate. È un dominio obbligatorio, contiene i driver nativi per l’accesso alle risorse, può accedere direttamente all’hardware e può interagire con le VM.

Tra i componenti del dominio di controllo troviamo XenStore e Toolstack.  
-> **XenStore**: è uno storage condiviso tra i vari domini e gestito dal demone di sistema Xenstored. Memorizza le informazioni sullo stato e sulla configurazione delle VM. È implementato con un meccanismo a coppie chiave-valore: quando un valore viene modificato all’interno dello store, una particolare funzione notifica i listener (e.g. i driver) interessati alla relativa chiave.  
-> **Toolstack**: consente all’utente di configurare e di gestire il ciclo di vita (creazione, shutdown, pausa, migrazione) di una VM. Per creare una nuova VM, l’utente fornisce un file di configurazione dove vengono descritte l’allocazione di memoria e di CPU e la configurazione dei dispositivi; dopodiché Toolstack effettua il parse di tale file e scrive le varie informazioni nello XenStore.

Scheduling delle CPU:  
Uno scheduler dell’hypervisor decide quali CPU virtuali devono essere eseguite su quali CPU fisiche.  
Xen permette di selezionare uno scheduler tra tutti quelli disponibili. Comunque sia, in generale, gli scheduler concorrono tutti ai seguenti obiettivi:  
- Fornire un algoritmo **proportional share**: per ogni CPU virtuale viene fornita una quantità di CPU fisica proporzionale al suo numero di share (ovvero al suo peso); ciò serve a garantire una distribuzione fair delle CPU fisiche.  
- Fornire un algoritmo **work-conserving**: le CPU fisiche non devono essere lasciate idle se c’è del lavoro da fare.  
- Fornire uno schedulo con **bassa latenza**.

Lo scheduler di default per Xen è il **Credit scheduler**, in cui a ciascun dominio vengono assegnati un weight e un cap:  
-> Weight = numero di CPU relative allocate per dominio (default: 256).  
-> Cap = limite massimo di CPU fisiche che possono essere assegnate a una CPU virtuale (default: 0, che corrisponde all’illimitatezza).  
Lo scheduler converte il weight in credito, che viene consumato dalle CPU virtuali nel momento in cui sono in esecuzione.  
Per ciascuna CPU fisica viene mantenuta una coda di CPU virtuali, dove in testa si ha la CPU virtuale con meno credito. Inoltre, viene attuato un meccanismo di load balancing automatico per cui, se una CPU fisica rimane senza CPU virtuali in coda, controlla lo stato delle altre CPU fisiche ed eventualmente preleva da una di loro una CPU virtuale in coda (in tal modo viene anche soddisfatta la proprietà del work-conserving).

**Confronto tra performance di diversi hypervisor**A seguito di alcuni studi effettuati nel 2013, è emerso che:  
- Per applicazioni CPU-intensive non ci sono particolari differenze tra le prestazioni di diversi hypervisor.  
- Per applicazioni I/O-intensive, invece, si hanno delle variazioni significative nelle prestazioni. In ogni caso, la scelta del VMM migliore dipende dal carico di lavoro.  
In conclusione, non esiste un hypervisor migliore di altri in assoluto, ma la sua scelta dipende molto dall’applicazione in esame.