



**Politecnico
di Torino**

Distributed Systems Programming

A.A. 2024/25

Anuar Elio Magliari

Ringraziamenti.....	4
1. Architetture di sistemi distribuiti.....	5
1.1 DS: Proprietà principali.....	5
1.2 DS: Requisiti generali.....	5
1.2.1 Trasparenza della distribuzione (Distribution Transparency).....	5
1.2.2 Membri dinamici.....	6
1.2.3 Sicurezza.....	6
1.2.4 Scalabilità.....	7
1.2.5 Apertura.....	7
1.3 DS: Architetture software generali.....	8
1.3.1 Middleware: Un livello software.....	8
1.4 DS: Principali strategie di progettazione.....	9
1.4.1 Strategie di scalabilità.....	10
1.4.2 Strategie di apertura.....	10
1.5 DS: Classificazione delle architetture.....	10
1.5.1 Il modello di interazione client/server.....	11
1.5.2 Livelli applicativi e livelli fisici per le architetture C/S.....	11
Architetture C/S a 2 livelli.....	11
Architetture C/S a 3 livelli.....	12
Architetture C/S multi-tier.....	12
1.5.3 Peer-to-Peer (P2P): Un'architettura decentralizzata.....	13
1.5.4 Livelli applicativi e livelli fisici per le architetture P2P.....	13
Rete di sovrapposizione.....	13
1.5.5 Client/Server vs. P2P.....	14
1.5.6 Modelli gerarchici e ibridi.....	14
Sistemi P2P gerarchici.....	14
Sistemi P2P ibridi.....	15
1.6 DS: Astrazioni.....	15
1.6.1 Astrazioni di livello superiore.....	16
Sistemi a oggetti distribuiti (Distributed Object Systems).....	16
Servizi distribuiti.....	16
Risorse distribuite.....	17
Sistemi Publish-Subscribe.....	17
Coda di messaggi.....	18
1.6.2 Astrazioni di livello inferiore.....	19
2. Servizi Web e stile architettonico REST.....	20
2.1 SOA: Architettura orientata ai servizi.....	20
2.2 Servizi web (Web Services).....	21
2.3 Stile architettonico REST.....	22
Principi fondamentali di REST:.....	23
2.4 Risorse.....	23
2.5 HATEOAS (Hypermedia as the Engine of Application State).....	25
2.6 Richardson Maturity Model.....	26
3. Sintassi e schemi astratti.....	27
3.1 Il linguaggio JSON Schema.....	27
3.1.1 Tipi di dati dello schema JSON.....	28
3.1.2 Combinazione di schemi.....	30
4. Progettazione dell'interfaccia del servizio (Service Interface Design).....	31

4.1 Principi di suddivisione.....	31
4.2 Scelta del livello di granularità.....	31
4.3 Progettazione dell'interfaccia: Best practices.....	32
4.4 Progettazione dell'interfaccia: Approcci.....	33
4.4.1 Centrato sul metodo.....	33
4.4.2 Centralità del messaggio.....	33
4.4.3 Constrained.....	33
4.5 Problemi e buone pratiche specifiche di REST.....	34
4.6 Mappare le risorse in URI.....	34
4.7 Considerazioni sull'efficienza.....	35
5. Chiamata di procedura remota (RPC).....	38
5.1 Il middleware RPC.....	39
5.2 Gestione dei fallimenti parziali (semantica delle chiamate remote).....	40
5.3 Altri problemi RPC.....	41
5.4 Diversi tipi di meccanismi di richiesta/risposta RPC.....	41
5.4.1 RPC classico (sincrono).....	41
5.4.2 RPC a una via (asincrono).....	41
5.4.3 RPC a due vie (asincrono).....	41
5.4.4 RPC with Streaming.....	41
5.5 In cima alle API Socket o RPC?.....	42
6. HTTP/2.....	43
6.1 HTTP/2 vs. HTTP/1.1.....	43
Ecco perché HTTP/2 è preferito:.....	43
Differenze con HTTP/1.1:.....	44
Early Hints.....	44
6.2 Avvio, negoziazione e aggiornamento della connessione HTTP/2.....	44
6.3 Gestione degli errori.....	45
7. gRPC.....	46
7.1 Protocol Buffer: un meccanismo di serializzazione per gRPC.....	46
7.2 gRPC: Tipi di RPC supportati.....	47
8. Sincronizzazione.....	48
8.1 Sincronizzazione dell'orologio fisico.....	48
8.2 Orologi logici.....	48
8.2.1 Orologi Lamport.....	49
Orologi Lamport: Discussione.....	50
Limiti degli orologi di Lamport.....	51
Vantaggi:.....	51
Esempio di orologio di Lamport con Total Ordering.....	51
Applicazione: Multicast totale ordinato.....	52
Multicast a ordinamento totale: Discussione.....	53
8.2.2 Vector Clocks.....	54
Limiti degli orologi vettoriali.....	55
Definizione di relazione causale (causal relationship).....	56
9. Coordination Algorithm.....	57
9.1 Mutual Exclusion.....	57
9.1.1 Token Ring Mutual Exclusion.....	57
9.1.2 Centralized Mutual Exclusion (permission based).....	58
9.1.3 Decentralized Mutual Exclusion (permission based).....	58
9.1.4 Confronto delle prestazioni.....	58

9.2 Election.....	59
9.2.1 Election: Bully algorithm.....	59
9.2.2 Election: Ring algorithm.....	59
9.3 Consensus.....	60
Proprietà e relazioni con altri problemi.....	60
Criteri da tenere a mente quando si valuta le performance di un algoritmo di coordinazione...	
60	
10. Data Replication and Consistency.....	62
10.1 Cost of Consistency.....	62
10.2 Consistency Model.....	62
10.2.1 Data-Centric Consistency Models (Modelli di coerenza incentrati sui dati).....	62
10.2.2 Continuous Consistency.....	63
10.2.3 Unità di consistenza (Conits).....	63
10.2.4 Sequential Consistency (Lamport).....	63
Proprietà del modello.....	64
10.2.5 Casual Consistency.....	64
10.2.6 Entry Consistency.....	64
10.2.7 Eventual Consistency.....	65
10.2.8 Client-centered consistency models.....	65
10.3 Gestione delle repliche.....	67
10.3.1 Aggiornamento delle strategie di propagazione.....	67
10.3.2 Protocolli Push (basati su server) e Pull (basati su client).....	68
10.4 Protocolli per la Continous Consistency.....	68
10.5 Protocolli per la Sequential Consistency.....	69
10.5.1 Primary Based Protocols.....	69
10.5.2 Replicated Write Protocols.....	69
10.6 Cache Coherence Protocols.....	69
10.6.1 Strategia di coerenza.....	69
10.6.2 Strategia di applicazione (Enforcement Strategy).....	70
11. WebSocket.....	71
11.1 WebSockets vs TCP.....	71
11.2 URI WebSocket.....	71
11.3 Tipi di frame.....	72
11.4 Messaggi e chiusura.....	72
11.5 Utilizzo di WebSocket.....	72
12. Sockets TCP/IP.....	74
Comunicazione tramite TCP.....	75
Gestione delle connessioni in Java.....	76
Blocking Operations and Timeouts.....	76
Server sequenziali e concorrenti.....	76
Comunicazione tramite UDP.....	77
Codifica dei dati.....	77
Perchè TCP non è abbastanza adatto per rilevare i peer process crashes?.....	77
Soluzione: heartbeat a livello applicativo (richieste di ping/pong).....	78
13. Tolleranza ai guasti (Fault Tolerance).....	79
13.1 Guasti, errori e fallimenti.....	79
13.2 Tecniche di ridondanza.....	80
Tipologie di Crash.....	80
1. Crash del Processo.....	80

2. Crash dell'Host (con successivo riavvio).....	80
3. Crash dell'Host (senza riavvio successivo) o Disconnessione Permanente della Rete....	81
Gestione dei Guasti TCP con il Socket API.....	81
13.3 Rilevamento dell'arresto anomalo del processo.....	81
13.3.1 Esempio: Rilevamento dell'arresto anomalo di un processo connesso via TCP.....	81
13.4 Affidabilità attraverso la ridondanza: Gruppi di processo.....	82
Caratteristiche principali dei gruppi di processo:.....	82
Tipi di organizzazione dei gruppi di processo:.....	82
13.5 Ottenere la tolleranza ai guasti desiderata.....	83
13.6 Il teorema CAP o Brewer's Theorem.....	83
14. MQTT.....	85
14.1 Architettura MQTT.....	85
14.2 Topics MQTT.....	85
14.2.1 Topics speciali.....	85
14.2.2 Filtri per topic.....	85
14.3 Il protocollo MQTT.....	86
14.4 Operazioni.....	87
14.4.1 Pubblicare la qualità del servizio (QoS).....	87
14.4.2 Subscription e annullamento della subscription.....	88
14.5 Last Will (Testament):.....	90
14.6 MQTT over Websocket.....	90
14.7 Linee guida per la progettazione dell'interfaccia Pub/Sub.....	91
1. Livello di granularità dei messaggi e dei topics.....	91
2. Gestione degli errori.....	91
3. Operazioni idempotenti.....	91
4. Documentazione chiara e completa.....	92
14.8 Linee guida specifiche per MQTT.....	92
1. Utilizzare caratteri ASCII nei nomi dei topic.....	92
2. Utilizzare i nomi dei topics come identificatori.....	92
3. Lasciare la struttura dei topics aperta all'estensione.....	92

Ringraziamenti

Questi appunti sono relativi al **corso DSP (Distributed Systems Programming)** del **Politecnico di Torino** e sono stati scritti con l'assistenza di *ChatGPT*. Sono forniti così come sono e non sono stati revisionati da nessun professore; pertanto, potrebbero esserci errori o refusi. Non mi assumo alcuna responsabilità per il loro utilizzo.

1. Architetture di sistemi distribuiti

L'architettura dei **sistemi distribuiti** prevede la **progettazione** e l'**organizzazione** di un **sistema composto da più componenti o nodi interconnessi**, in cui questi componenti lavorano insieme per raggiungere un obiettivo comune. In un sistema distribuito, i componenti possono essere geograficamente dispersi, operare simultaneamente e comunicare tra loro attraverso una rete. L'obiettivo principale dell'architettura di un sistema distribuito è quello di consentire una comunicazione, un coordinamento e una cooperazione efficienti e affidabili tra i componenti, affrontando al contempo sfide quali la scalabilità, la tolleranza ai guasti, la coerenza e le prestazioni. L'architettura di un sistema distribuito può variare notevolmente a seconda dei requisiti e dei vincoli specifici del sistema e può coinvolgere diversi stili architettonici, paradigmi di comunicazione, protocolli e tecnologie. In generale, la progettazione dell'architettura di un sistema distribuito svolge un ruolo fondamentale nel garantire il funzionamento e le prestazioni efficaci di sistemi distribuiti complessi.

Le definizioni di sistema distribuito possono essere molteplici, poiché può essere visto come:

- Computer collegati in rete che comunicano tra loro.
- Elementi di calcolo autonomi che agiscono come un unico sistema coerente. Un termine più generico per indicare un "elemento di calcolo" è processo o nodo.

1.1 DS: Proprietà principali

Le parole chiave sono *autonomia* e *rete*, poiché entrambe introducono concetti e implicazioni importanti.

- **L'autonomia** implica:
 - *Asincrono e concorrente*
 - *Eterogeneo*: ci sono hardware, sistemi operativi, linguaggi di programmazione e sedi diverse.
 - Possibilità di *fallimento*.
- **L'interconnessione in rete** porta a:
 - Le comunicazioni richiedono *tempi lunghi o variabili e possono fallire*.
 - Impatto elevato e ampia superficie di attacco per gli intrusi malintenzionati.

1.2 DS: Requisiti generali

In questa discussione esploreremo i principali requisiti di un sistema distribuito e approfondiremo le relative implicazioni. Come già spiegato, un sistema distribuito è una rete complessa di componenti interconnessi che lavorano insieme per un obiettivo comune. La comprensione di questi requisiti e delle loro implicazioni è fondamentale per progettare e implementare efficacemente i sistemi distribuiti.

1.2.1 Trasparenza della distribuzione (Distribution Transparency)

La **trasparenza della distribuzione** è un concetto chiave nell'architettura dei sistemi distribuiti, che si riferisce alla **capacità di nascondere i dettagli interni di un sistema distribuito ai suoi utenti**. L'idea è quella di astrarre le complessità sottostanti di un sistema distribuito e fornire una visione semplificata e coerente ai suoi utenti, indipendentemente dall'effettiva distribuzione dei dati e dei calcoli tra i diversi nodi:

- **Trasparenza della posizione**: si tratta di **nascondere i dettagli relativi alla posizione dei dati o dei calcoli nel sistema distribuito**. Gli utenti devono poter accedere ai dati o eseguire operazioni senza conoscere la posizione fisica delle risorse. Ciò può essere ottenuto attraverso tecniche come la denominazione e l'indirizzamento, in cui vengono utilizzati nomi logici per identificare le risorse invece dei loro indirizzi fisici.

- **Trasparenza della replica:** si tratta di **nascondere i dettagli della replica dei dati nel sistema distribuito**. Gli utenti devono poter accedere ai dati replicati senza essere a conoscenza dell'esistenza di copie multiple. La replica può migliorare la disponibilità, la tolleranza agli errori e le prestazioni, ma la gestione delle repliche può essere complessa. La trasparenza della replica garantisce che gli utenti non debbano essere a conoscenza del meccanismo di replica.
- **Trasparenza di accesso:** si tratta di **nascondere i dettagli delle modalità di accesso ai dati o di esecuzione delle operazioni nel sistema distribuito**. Gli utenti devono essere in grado di accedere ai dati o di eseguire operazioni utilizzando un'interfaccia coerente, indipendentemente dal modo in cui i dati sono internamente memorizzati, acceduti o condivisi tra i diversi elementi del sistema distribuito. La trasparenza dell'accesso può essere ottenuta attraverso API standard o middleware che forniscono un'interfaccia uniforme per accedere a risorse diverse.
- **Trasparenza dei guasti:** si tratta di **nascondere i dettagli dei guasti nel sistema distribuito**. Gli utenti devono essere in grado di accedere ai dati o di eseguire operazioni anche in presenza di guasti, senza essere a conoscenza dei guasti o dei loro meccanismi di recupero. La trasparenza dei guasti può essere ottenuta attraverso tecniche di tolleranza ai guasti come la replica, la ridondanza e i meccanismi di recupero degli errori.

Tuttavia, non sempre è possibile raggiungere la piena trasparenza della distribuzione. Ad esempio, la posizione fisica delle risorse può influire sulla latenza e i guasti possono compromettere la trasparenza. In alcuni casi, possono essere più appropriati servizi location-aware o context-aware, in cui il sistema tiene conto della posizione o del contesto dell'utente o delle risorse per ottimizzare le prestazioni o fornire servizi specializzati. Spesso esiste un compromesso tra trasparenza e prestazioni. Se da un lato la trasparenza della distribuzione può fornire una visione semplificata e coerente del sistema distribuito ai suoi utenti, dall'altro può introdurre spese generali in termini di comunicazione, coordinamento e gestione.

La progettazione di sistemi distribuiti deve bilanciare attentamente il compromesso tra trasparenza e prestazioni in base ai requisiti e ai vincoli specifici del sistema.

1.2.2 Membri dinamici

L'appartenenza dinamica è un requisito chiave dei sistemi distribuiti, che si riferisce alla capacità di un sistema di gestire i cambiamenti nell'appartenenza dei suoi componenti durante l'esecuzione. Ciò significa che i processi all'interno del sistema distribuito devono essere in grado di unirsi o abbandonare il sistema in modo dinamico, senza interrompere la funzionalità complessiva del sistema.

Le implicazioni dell'appartenenza dinamica sono molteplici:

1. Il sistema deve fornire **meccanismi continui per aggiungere o rimuovere processi**. Ciò può comportare protocolli per rilevare quando un nuovo processo si unisce al sistema o quando un processo esistente se ne va, e aggiornare di conseguenza le informazioni di appartenenza. Mantenere la coerenza nel sistema distribuito, nonostante i cambiamenti dinamici dei membri, può essere una sfida.
2. I processi all'interno del sistema distribuito **devono essere in grado di individuare altri componenti**, compresi quelli nuovi che si uniscono al sistema dopo che questo è stato avviato. Ciò richiede meccanismi per scoprire l'appartenenza e la posizione attuale dei processi, come i servizi di denominazione, i servizi di directory o i protocolli di ricerca dinamica.
3. Inoltre, **pone delle sfide in termini di mantenimento della stabilità, della coerenza e della tolleranza ai guasti del sistema**. Ad esempio, l'aggiunta di un nuovo processo può richiedere la ridistribuzione dei dati o del carico di lavoro tra i processi esistenti per mantenere il bilanciamento del carico o la coerenza dei dati. Allo stesso modo, la rimozione di un processo può richiedere la riassegnazione delle sue responsabilità ad altri processi in modo da evitare perdite di dati o interruzioni del servizio.

In conclusione, l'appartenenza dinamica è un requisito critico nei sistemi distribuiti, in quanto consente al sistema di adattarsi alle variazioni di disponibilità, scalabilità e affidabilità dei suoi componenti, garantendo al contempo un funzionamento regolare e una funzionalità continua.

1.2.3 Sicurezza

La sicurezza è un requisito fondamentale nei sistemi distribuiti e comprende la protezione dei dati e dei servizi da accessi o interferenze non autorizzati. In genere si tratta di garantire la riservatezza, il controllo degli accessi, l'integrità e la disponibilità del sistema e dei suoi componenti.

Le principali proprietà di sicurezza di un sistema distribuito includono:

- **Riservatezza:** Garantire che i dati o i servizi non siano accessibili a soggetti non autorizzati.
- **Controllo degli accessi:** Limitazione dell'accesso ai dati o ai servizi in base a permessi o privilegi predefiniti.
- **Integrità:** Impedire la modifica o la manomissione non autorizzata dei dati o del comportamento del sistema.
- **Disponibilità:** Garantire che il sistema rimanga operativo e accessibile agli utenti legittimi.

Queste proprietà di sicurezza sono tipicamente formulate come requisiti che specificano il livello di protezione contro determinate capacità o attacchi. L'obiettivo è rendere difficile per un attaccante con capacità specifiche compromettere il sistema o i suoi componenti.

1.2.4 Scalabilità

La **scalabilità** si riferisce alla **capacità di un sistema distribuito di adattarsi e mantenere le sue prestazioni all'aumentare delle dimensioni**, come il numero di utenti, le risorse, la distanza tra utenti/componenti o le unità organizzative. La scalabilità è una considerazione importante per i sistemi distribuiti che devono gestire richieste crescenti e requisiti in evoluzione.

I diversi tipi di scalabilità nei sistemi distribuiti possono includere:

- **Scalabilità dimensionale:** Si riferisce alla capacità di un sistema di gestire un numero crescente di utenti, risorse o dati senza un significativo degrado delle prestazioni. Un sistema distribuito scalabile deve essere in grado di gestire in modo efficiente un numero crescente di utenti o un volume crescente di dati senza sacrificare le sue prestazioni.
- **Scalabilità geografica:** Si riferisce alla capacità di un sistema distribuito di spaziare tra diverse località geografiche o distanze tra utenti o componenti. Poiché i sistemi distribuiti possono operare in più località geografiche, la scalabilità geografica è importante per garantire una comunicazione e un coordinamento efficienti tra componenti distribuiti che possono essere fisicamente separati.
- **Scalabilità amministrativa:** Si riferisce alla capacità di un sistema distribuito di gestire un numero crescente di unità organizzative o domini amministrativi. Poiché i sistemi distribuiti possono coinvolgere più organizzazioni o unità amministrative, la scalabilità amministrativa è fondamentale nella gestione di ambienti distribuiti complessi.

Essendo un requisito complesso da soddisfare, la scalabilità nei sistemi distribuiti presenta diverse sfide, tra cui:

- **Limitazioni delle risorse:** I sistemi distribuiti operano spesso con vincoli di risorse, come la potenza di elaborazione, la memoria, la larghezza di banda o lo storage. Garantire un utilizzo e una gestione efficienti delle risorse è essenziale per mantenere la scalabilità ed evitare colli di bottiglia nelle prestazioni.
- **Latenza e affidabilità:** La distanza tra gli utenti o i componenti di un sistema distribuito può comportare un aumento della latenza e una riduzione dell'affidabilità. La gestione e l'attenuazione degli effetti delle limitazioni di latenza e affidabilità, che sono intrinseche ai sistemi distribuiti, è fondamentale per ottenere la scalabilità.

In sintesi, per ottenere la scalabilità nei sistemi distribuiti è necessario affrontare la scalabilità delle dimensioni, la scalabilità geografica, la scalabilità amministrativa e la gestione delle sfide legate ai limiti delle risorse, alla latenza e all'affidabilità. La scalabilità è una considerazione fondamentale nella progettazione e nell'implementazione di sistemi distribuiti, per garantire che possano gestire efficacemente l'aumento delle richieste e l'evoluzione dei requisiti.

1.2.5 Apertura

L'apertura si riferisce alla misura in cui un sistema distribuito offre componenti che possono essere facilmente

utilizzati o integrati in altri sistemi, indipendentemente dalla loro tipologia o tecnologia. Un sistema distribuito aperto è progettato per essere interoperabile e accessibile ad altri sistemi, consentendo una perfetta integrazione e interazione.

I requisiti di apertura correlati potrebbero includere:

- **Semplicità:** Un sistema aperto distribuito deve essere progettato all'insegna della semplicità, in modo da facilitare la comprensione e l'utilizzo dei suoi componenti da parte di altri sistemi. La semplicità della progettazione e dell'implementazione favorisce la facilità di integrazione e riduce la complessità dell'interfacciamento con il sistema distribuito.
- **Modificabilità:** Un sistema aperto e distribuito deve essere progettato per essere facilmente modificabile, consentendo aggiornamenti o cambiamenti ai suoi componenti senza interrompere l'integrazione con altri sistemi. La modificabilità è importante per adattarsi all'evoluzione dei requisiti o delle tecnologie e per consentire un'integrazione agevole con altri sistemi nel tempo.
- **Documentazione:** Un sistema distribuito aperto deve essere ben documentato, fornendo una documentazione chiara e completa sui suoi componenti, sulle interfacce, sui protocolli e su altre informazioni rilevanti. La documentazione è essenziale per facilitare la comprensione e l'utilizzo del sistema distribuito da parte di altri sistemi o sviluppatori.

In poche parole, l'apertura nei sistemi distribuiti implica la progettazione di sistemi interoperabili, accessibili e facilmente integrabili con altri sistemi, tenendo conto di requisiti quali la semplicità, la modificabilità e la documentazione. Un sistema distribuito aperto favorisce l'integrazione e la collaborazione con altri sistemi, consentendo l'interoperabilità ed estendendo le capacità dell'intero ecosistema distribuito.

1.3 DS: Architetture software generali

L'architettura software di un sistema distribuito è tipicamente basata sul modello a strati, come il modello OSI (Open Systems Interconnection), che consiste in diversi strati responsabili di diversi aspetti della comunicazione e dell'elaborazione dei dati. In un sistema distribuito, l'architettura software è costituita da un insieme di processi che vengono eseguiti fisicamente su host di rete diversi e che interagiscono tra loro in base a una serie di protocolli.

I livelli OSI rilevanti per il programmatore di sistemi distribuiti sono i livelli orientati alle applicazioni, che comprendono i livelli 5 (livello di sessione), 6 (livello di presentazione) e 7 (livello di applicazione). Questi livelli sono responsabili della gestione della comunicazione e dell'elaborazione a livello di applicazione, come la creazione di sessioni, la gestione della presentazione e della formattazione dei dati e l'implementazione di protocolli specifici per l'applicazione.

1.3.1 Middleware: Un livello software

Per gestire la complessità dei numerosi requisiti di un sistema distribuito, un approccio comune è quello di costruire le applicazioni sopra un livello software chiamato *middleware*. Il **middleware funge da livello intermedio tra il livello applicativo e i livelli di sistema inferiori, fornendo un insieme di servizi e astrazioni comuni che semplificano lo sviluppo di applicazioni distribuite**.

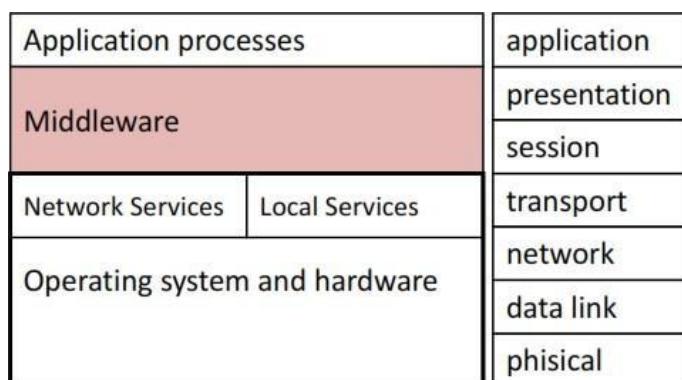


Figura 1: Posizione del middleware nel modello OSI

Inoltre, fornisce servizi business-unaware per il coordinamento e la comunicazione tra processi remoti in un sistema distribuito. Agisce come un livello intermedio che nasconde la complessità della comunicazione attraverso la rete, l'eterogeneità dei processi e degli host, i problemi di sicurezza e altri dettagli sottostanti.

Esempi di software che possono essere classificati come middleware sono i browser web e i driver di database. I browser web forniscono servizi per l'accesso alle risorse web e la visualizzazione delle pagine web, mentre i driver di database facilitano la comunicazione tra applicazioni e database per il recupero e la manipolazione dei dati.

D'altra parte, esempi di software che non possono essere classificati come middleware sono i sistemi business-aware, come un sistema di prenotazione aerea. Questi sistemi sono progettati per scopi aziendali specifici e sono consapevoli delle regole e dei requisiti specifici del dominio, a differenza dei middleware che non sono consapevoli del business e forniscono servizi generici per il coordinamento e la comunicazione.

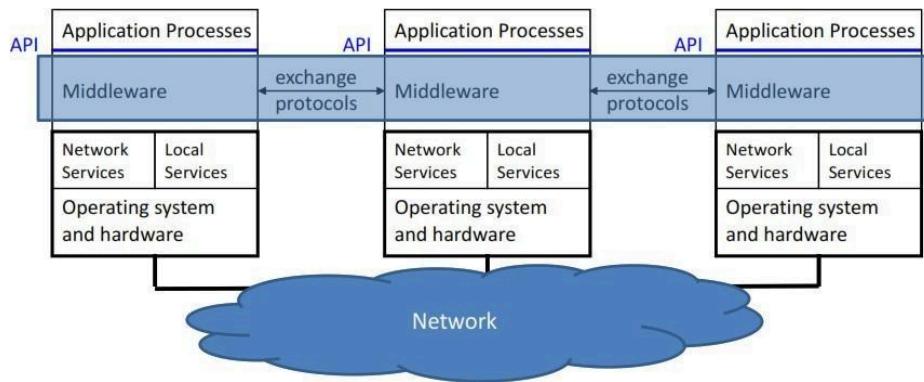


Figura 2: Funzionamento di un middleware

I servizi tipici forniti dal middleware possono essere classificati in tre categorie principali:

- Servizi di interazione:** Questi servizi comprendono lo scambio di informazioni, la gestione delle connessioni, la gestione delle sessioni, i meccanismi per evitare gli impasse e altri meccanismi che facilitano la comunicazione e il coordinamento tra i processi remoti.
- Servizi per l'accesso ad applicazioni specifiche:** Il middleware fornisce anche servizi per l'accesso ad applicazioni specifiche, come l'accesso al database, l'elaborazione delle transazioni e la gestione degli oggetti distribuiti. Questi servizi consentono alle applicazioni di interagire con le risorse e i dati sottostanti in modo trasparente.
- Servizi di gestione, controllo e amministrazione:** Il middleware offre servizi di gestione, controllo e amministrazione del sistema distribuito. Questi servizi possono includere servizi di directory per la localizzazione delle risorse, servizi di sicurezza per l'autenticazione e l'autorizzazione, servizi di monitoraggio delle prestazioni del sistema e altre funzioni amministrative.

In sintesi, il middleware fornisce servizi non consapevoli per il coordinamento e la comunicazione tra processi remoti in un sistema distribuito, nascondendo le complessità sottostanti. Esempi di middleware sono i browser web e i driver di database, mentre i sistemi business-aware non sono classificati come middleware. I servizi di middleware possono essere suddivisi in servizi di interazione, servizi di accesso ad applicazioni specifiche e servizi di gestione, controllo e amministrazione.

1.4 DS: Principali strategie di progettazione

In questa sezione esploreremo i principali obiettivi e strategie di progettazione per i DS, tra cui la limitazione delle interazioni, l'uso di soluzioni middleware e la costruzione di applicazioni che si basano solo sulle API. Questi principi mirano a promuovere la trasparenza della distribuzione, la scalabilità, il riutilizzo del codice e l'occultamento delle informazioni, che sono fondamentali per il successo dell'implementazione dei sistemi distribuiti.

1.4.1 Strategie di scalabilità

La scalabilità è un fattore critico nella progettazione di sistemi distribuiti per gestire richieste crescenti. Esistono due strategie comuni per la scalabilità:

- **Scaling Up:** Comporta il miglioramento della capacità degli elementi di calcolo, archiviazione o comunicazione, come la sostituzione delle CPU con altre più potenti. Tuttavia, lo scaling up può essere applicato solo in misura limitata a causa delle limitazioni delle risorse.
- **Scaling Out:** comporta la modifica del modo in cui il sistema è costruito o funziona, ad esempio aumentando il numero di macchine distribuite nel sistema. È tipicamente impiegato nei sistemi distribuiti a scala mondiale, in quanto consente un approccio più distribuito e parallelo per gestire carichi di lavoro maggiori.

Concentrandosi sulle tecniche di scaling out, che hanno un impatto maggiore sui sistemi distribuiti di dimensioni mondiali, vi sono diverse tecniche principali comunemente utilizzate, tra cui: il **partizionamento** e la **distribuzione del lavoro**, la **replicazione** e l'**occultamento/limitazione della latenza di comunicazione**.

- **Suddivisione e distribuzione del lavoro:** consiste nel suddividere un compito o un carico di lavoro di grandi dimensioni in parti più piccole e gestibili, assegnandole a diversi processi o nodi del sistema. Quando la domanda aumenta, si possono aggiungere altri processi per gestire il carico di lavoro. Esempi di sistemi che utilizzano questa tecnica sono il Web e il Domain Name System (DNS).
- **Replicazione:** comporta la creazione di copie di processi o dati per distribuire il carico e aumentare la ridondanza. Ad esempio, il caching è una forma di replica in cui i dati a cui si accede di frequente vengono memorizzati in più posizioni per un accesso più rapido. Un altro esempio è il clustering dei server web, in cui si utilizzano più server per distribuire il carico e migliorare la disponibilità.
- **Nascondere/limitare la latenza di comunicazione:** è un'altra tecnica utilizzata per il ridimensionamento. La comunicazione asincrona viene utilizzata al posto di quella sincrona per ridurre il tempo di attesa di una risposta. Inoltre, l'elaborazione dei dati nel luogo in cui si trovano, anziché il loro spostamento, può contribuire a ridurre la latenza di comunicazione.

1.4.2 Strategie di apertura

Le strategie di apertura nei sistemi distribuiti si concentrano sull'uso di protocolli e API standard o documentati/pubblici, lasciando alle implementazioni la libertà di aderire ai protocolli e alle API concordati. Questo approccio promuove interfacce neutrali, interoperabilità di diverse implementazioni e **portabilità** delle stesse.

L'uso di protocolli e API standard garantisce che i sistemi possano comunicare e interagire tra loro utilizzando standard ampiamente accettati, consentendo l'**interoperabilità** tra le diverse implementazioni. Ciò consente ai componenti o ai sistemi di fornitori o sviluppatori diversi di lavorare insieme senza problemi. Anche la **componibilità** e l'**estensibilità** sono strategie importanti per l'apertura. I sistemi devono essere progettati per essere composti da diversi elementi semplici e facilmente sostituibili. Ciò consente la flessibilità di aggiungere, sostituire o modificare i componenti del sistema senza interrompere la funzionalità complessiva. Questo approccio promuove la modularità e l'adattabilità, rendendo più facile estendere o modificare il sistema in futuro.

1.5 DS: Classificazione delle architetture

La classificazione delle architetture dei sistemi distribuiti può essere suddivisa a grandi linee in tre tipi principali:

1. **Client-Server:** Questa architettura prevede un'organizzazione centralizzata in cui i client richiedono servizi a un server centrale. Il server è responsabile dell'elaborazione delle richieste e della fornitura dei servizi richiesti. I client e i server hanno ruoli distinti: i client sono in genere interfacce rivolte all'utente e i server gestiscono l'elaborazione e l'archiviazione dei dati. Questa architettura è comunemente utilizzata nei sistemi in cui esiste una chiara separazione delle responsabilità tra client e server e i client si affidano ai server per le loro funzionalità.

2. **Peer-to-Peer:** In questa architettura completamente decentralizzata, tutti i nodi del sistema hanno le stesse capacità e responsabilità. Ogni nodo può agire sia come client che come server e può richiedere e fornire servizi ad altri nodi della rete. I nodi collaborano tra loro in modo peer to peer, senza un server o un'autorità centrale. Questa architettura è spesso utilizzata nei sistemi in cui non esiste un punto di controllo centrale e i nodi devono condividere risorse e servizi direttamente tra loro.
3. **Irida:** l'architettura ibrida combina elementi di entrambe le architetture, client-server e peer- to-peer. Può comportare un mix di componenti centralizzati e decentralizzati, oppure una combinazione di interazioni client-server e peer-to-peer. Questa architettura è spesso utilizzata nei sistemi in cui si desidera una combinazione di centralizzazione e decentralizzazione, in base a requisiti o vincoli specifici.

Queste diverse architetture hanno i loro punti di forza e di debolezza e la scelta dell'architettura dipende dalle esigenze, dagli obiettivi e dai vincoli specifici del sistema distribuito che si sta progettando. Questi modelli saranno discussi nei prossimi paragrafi.

1.5.1 Il modello di interazione client/server

Il modello di interazione client-server (C/S) è attualmente il modello più utilizzato nei sistemi distribuiti. In questo modello, le interazioni avvengono tra due processi, uno dei quali svolge il ruolo di client e l'altro quello di server. Ogni interazione si basa su uno scambio di messaggi, in cui il client invia una richiesta e il server restituisce una risposta.

È importante notare che il ruolo di client o di server si riferisce al modo in cui un processo svolge una singola interazione, e un singolo processo può svolgere il ruolo di client in alcune interazioni e di server in altre. In un'architettura client-server, i processi si dividono in due classi:

- **Processi client:** svolgono solo il ruolo di client.
- **Processi server:** svolgono il ruolo di server e offrono un servizio, come il calcolo o l'archiviazione, a molti client.

Il modello di interazione client-server è ampiamente utilizzato per la sua semplicità e facilità di implementazione. I client possono richiedere servizi ai server e i server possono rispondere a queste richieste in modo controllato e organizzato. Questo modello consente una gestione e un controllo centralizzati, rendendolo adatto ai sistemi in cui è necessaria una chiara separazione delle responsabilità e un'autorità centrale che coordini e gestisca le interazioni tra client e server. Tuttavia, può anche introdurre potenziali **problemi di scalabilità** e di **single-point-of-failure**, in quanto il server può diventare un collo di bottiglia o un punto di guasto se non viene progettato e gestito correttamente.

1.5.2 Livelli applicativi e livelli fisici per le architetture C/S

In un sistema client-server (C/S), ogni applicazione può essere logicamente suddivisa in tre parti principali, note anche come livelli logici: il **livello dell'interfaccia utente**, il **livello di elaborazione (processing tier)** e il **livello dei dati**. Questi livelli logici possono essere mappati su diversi processi in esecuzione su host eventualmente diversi, noti anche come livelli fisici. Esistono diverse possibilità di mappatura tra i livelli logici e fisici, a seconda dell'architettura del sistema.

Architetture C/S a 2 livelli

Le architetture C/S a due livelli sono costituite da due soli livelli fisici: un **livello client** (front end) e un **livello server** (back end). Questa è l'architettura classica, tipica della prima generazione di sistemi C/S. Sono possibili diverse configurazioni, a seconda di come i livelli logici vengono mappati sui livelli fisici.

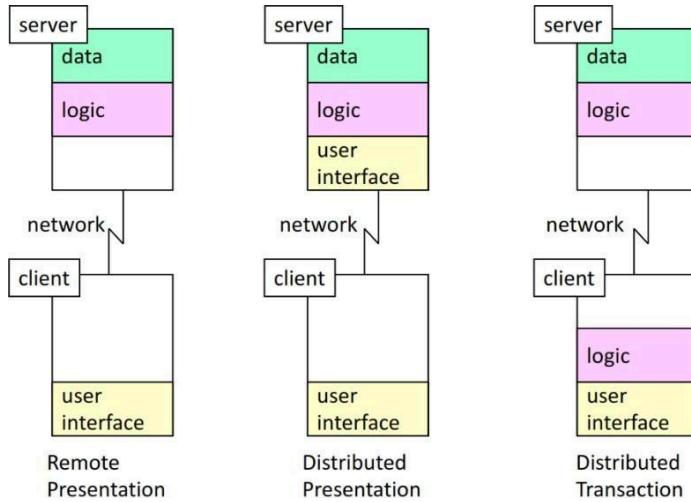


Figura 3: Architettura client/server a 2 livelli

Architetture C/S a 3 livelli

Le architetture C/S a **due livelli** possono presentare limitazioni in termini di **scalabilità** e **flessibilità**. Queste limitazioni possono essere superate introducendo un **livello fisico intermedio**, noto anche come **macchina intermedia**. Questa macchina intermedia può avere varie funzioni, come il **bilanciamento del carico di lavoro su diversi back-end**, il **filtraggio** (ad esempio, firewall), la **conversione di protocollo**, l'accesso a **sistemi legacy** (gateway) o l'accesso **simultaneo a diversi server che offrono servizi a valore aggiunto**. Un'altra possibilità è quella di aggiungere ulteriori livelli per delegare e disaccoppiare componenti specifici dell'applicazione, come i dati.

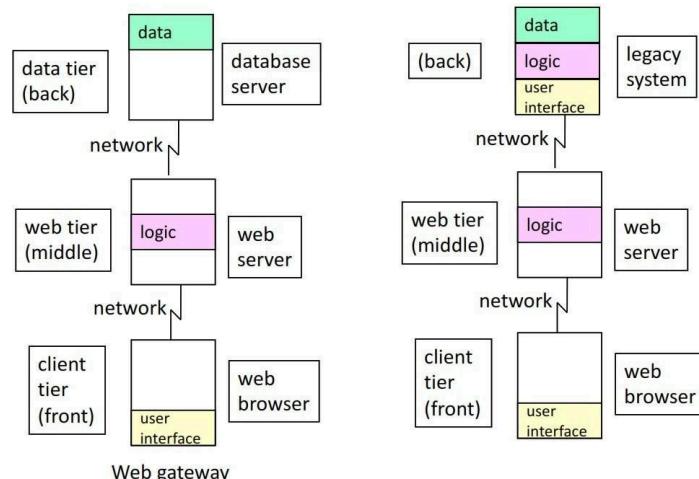


Figura 4: Architettura client/server a 3 livelli

Architetture C/S multi-tier

Per una maggiore flessibilità e complessità, il numero di livelli fisici di un sistema C/S può essere ulteriormente aumentato. Ad esempio, è possibile progettare un'architettura a 4 livelli per **raggiungere specifici requisiti di sistema e obiettivi di prestazioni**. Ciò consente un **controllo più granulare** e la **personalizzazione dell'architettura del sistema** per soddisfare le esigenze specifiche dell'applicazione e dell'ambiente.

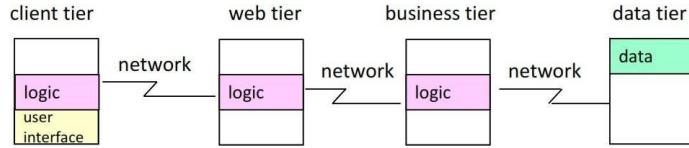


Figura 5: Architettura multilivello client/server

1.5.3 Peer-to-Peer (P2P): Un'architettura decentralizzata

In un'architettura Peer-to-Peer (P2P), tutti i processi sono considerati come **peer identici**, in grado di svolgere **sia il ruolo di client che di server**. Ogni peer ha la possibilità di avviare un **interazione client- server (C/S)** con qualsiasi altro peer che conosce, in qualsiasi momento. Ciò significa che in un'architettura P2P non esiste una distinzione tra client e server e che tutti i peer hanno le stesse capacità e responsabilità.

I peer possono avviare e rispondere alle richieste di altri peer, rendendo l'**architettura altamente decentralizzata e distribuita**, senza autorità o controllo centrale. Le architetture P2P sono comunemente utilizzate nei sistemi in cui i peer hanno bisogno di condividere risorse, collaborare o comunicare direttamente tra loro senza affidarsi a un server centrale. Esempi di sistemi P2P sono le reti di condivisione di file, le reti informatiche distribuite e le reti blockchain.

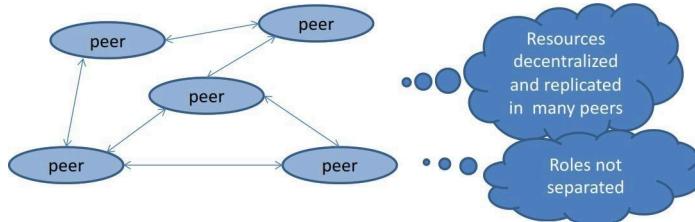


Figura 6: Architettura P2P

1.5.4 Livelli applicativi e livelli fisici per le architetture P2P

In un sistema Peer-to-Peer (P2P), **tutti i peer eseguono gli stessi compiti, ma sulla propria parte dell'insieme di dati**. L'insieme di dati può essere suddiviso tra i peer, oppure i dati possono essere replicati per aumentare l'affidabilità e le prestazioni.

Rete di sovrapposizione

Un sistema P2P è costruito dai suoi peer, che creano una **rete overlay**. **Ogni peer può comunicare solo con altri peer attraverso canali virtuali**, tipicamente implementati da connessioni TCP. Un aspetto fondamentale di ogni architettura P2P è il modo in cui la rete overlay viene creata e gestita. Esistono due tipi principali di reti overlay:

- **Sistemi strutturati:** Nei **sistemi P2P strutturati**, la **rete overlay è costruita con una topologia determinata e regolare**, come un anello (un esempio pratico P2P di questa struttura è Chord), un albero o una griglia. Gli algoritmi di routing specifici per la topologia sono utilizzati per raggiungere specifici peer nella rete overlay.

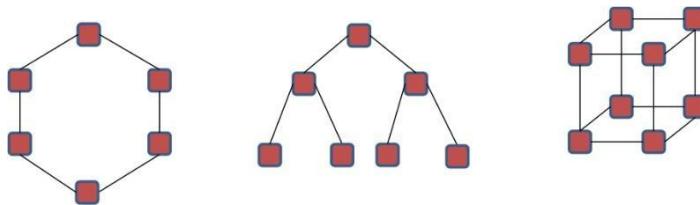


Figura 7: Rete sovrapposta P2P: Sistema strutturato

- **Sistemi non strutturati:** Nei **sistemi P2P non strutturati**, ogni peer si connette ad altri peer in

modo più o meno casuale, ad esempio attraverso la scoperta di peer vicini. Diverse strategie possono essere utilizzate per raggiungere altri peer nella rete di overlay, come il flooding, le passeggiate casuali o la ricerca basata sulle politiche.

1.5.5 Client/Server vs. P2P

Durante la fase di progettazione, nella scelta dell'architettura, è necessario un confronto tra le architetture Client-Server (C/S) e Peer-to-Peer (P2P).

Ci sono diverse differenze fondamentali da considerare:

- Scalabilità e affidabilità:** Nelle architetture C/S, i processi server possono diventare dei colli di bottiglia, poiché devono essere eseguiti su host potenti e affidabili. Possono anche essere singoli punti di guasto e possono essere congestionati dalle richieste, incidendo sulle prestazioni del sistema. D'altra parte, nelle architetture P2P, le informazioni sono distribuite e spesso replicate tra i peer, il che può fornire affidabilità e prestazioni a basso costo. Tuttavia, nelle architetture P2P è possibile esercitare un controllo minore, il che rende più difficile garantire coerenza, affidabilità e prestazioni tra tutti i peer.
- Sicurezza:** Garantire la sicurezza nelle architetture P2P può essere più impegnativo rispetto alle architetture C/S, poiché le informazioni sono distribuite tra più peer. Garantire l'integrità dei dati, la riservatezza e l'autenticazione può essere più complesso in un ambiente P2P, dove i peer possono avere diversi livelli di fiducia e misure di sicurezza.
- Controllo e organizzazione:** Le architetture C/S offrono una chiara separazione dei ruoli tra client e server, semplificando la comprensione e la gestione. I client interagiscono solo con i server e i server forniscono servizi ai client. Nelle architetture P2P, il controllo e l'organizzazione sono minori, in quanto tutti i peer hanno uguali capacità e responsabilità. I peer possono avviare e rispondere alle richieste di altri peer, il che può rendere più impegnativa la gestione e il coordinamento del sistema.
- Flessibilità e complessità:** Le architetture P2P possono offrire maggiore flessibilità e decentralizzazione rispetto alle architetture C/S, poiché non esiste un'autorità o un controllo centrale. I peer possono collaborare, condividere risorse e comunicare direttamente tra loro. Tuttavia, questo può anche aumentare la complessità della gestione e del coordinamento del sistema. Le architetture C/S, invece, sono generalmente più semplici e ben comprese, con una chiara separazione dei ruoli tra client e server.

Caratteristica	Client-Server (C/S)	Peer-to-Peer (P2P)
Controllo	Centralizzato (server)	Decentralizzato (tutti i peer sono uguali)
Scalabilità	Scalabilità verticale (server più potenti)	Scalabilità orizzontale (aumenta con i peer)
Affidabilità	Dipende dalla disponibilità del server (single point of failure)	Alta affidabilità (nessun punto di fallimento centrale)
Sicurezza	Maggiore controllo sulla sicurezza centralizzata	Sicurezza più complessa e meno centralizzata
Gestione dei dati	Centralizzata (facile da gestire e aggiornare)	Distribuita (più difficile da coordinare e gestire)
Prestazioni	Potenziale collo di bottiglia sul server	Prestazioni variabili, a seconda dei peer
Costo	Costi elevati per infrastrutture centralizzate	Più economico, ma i peer devono avere risorse adeguate
Flessibilità	Bassa (architettura fissa)	Alta (i peer possono essere aggiunti o rimossi facilmente)

In sintesi, le architetture C/S sono tipicamente utilizzate quando c'è bisogno di una chiara separazione dei ruoli, di un controllo centralizzato e di semplicità, mentre le architetture P2P sono preferite quando c'è bisogno di decentralizzazione, flessibilità e collaborazione tra pari. Ogni architettura ha i suoi punti di forza e di debolezza e la scelta tra C/S e P2P dipende dai requisiti specifici e dalle caratteristiche del sistema che si sta progettando o implementando.

1.5.6 Modelli gerarchici e ibridi

In alcuni sistemi P2P, i peer non sono tutti uguali e possono essere classificati in diversi tipi. Due approcci comuni sono i sistemi P2P gerarchici e ibridi.

Sistemi P2P gerarchici

Nei sistemi P2P gerarchici, i peer sono classificati in peer normali (deboli) e superpeer. I super-peer sono in genere più potenti e svolgono un ruolo speciale nella rete overlay. I peer deboli sono collegati direttamente ai super-peer, formando una rete overlay di super-peer. I super-peer possono gestire compiti come l'indicizzazione, la ricerca o la gestione delle risorse, mentre i weak-peer eseguono principalmente i normali compiti del P2P.

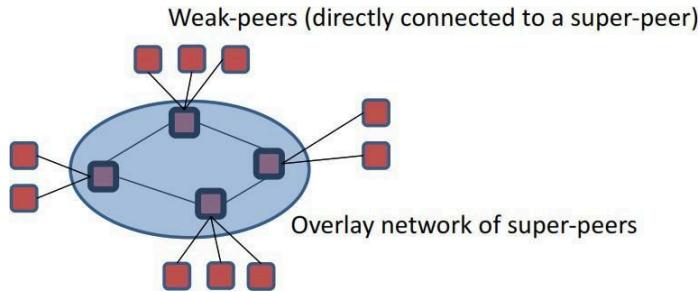


Figura 8: Sistema P2P gerarchico

Sistemi P2P ibridi

I sistemi P2P ibridi combinano insieme elementi delle architetture C/S (Client/Server) e P2P. Possono utilizzare una combinazione di approcci client-server e peer-to-peer per raggiungere i loro obiettivi. Ad esempio, alcuni peer possono agire come client, mentre altri agiscono come server, fornendo risorse o servizi ad altri peer. I sistemi P2P ibridi offrono flessibilità e scalabilità sfruttando i vantaggi delle architetture C/S e P2P.

1.6 DS: Astrazioni

Nei sistemi distribuiti le **astrazioni** forniscono un livello superiore di astrazione e incapsulamento per nascondere le **complessità** dei sistemi distribuiti, consentendo agli sviluppatori di ragionare sul comportamento e sulle interazioni del sistema in modo più intuitivo e gestibile. Le astrazioni nei sistemi distribuiti sono utilizzate per rappresentare concetti, servizi o comportamenti che aiutano a semplificare la progettazione, l'implementazione e la gestione dei sistemi distribuiti.

In questo contesto, le astrazioni più comunemente usate sono¹:

- **Chiamate di procedura remota (RPC)**
- **Passaggio di messaggi**
- **Dati persistenti condivisi**

Inoltre, nei sistemi distribuiti, la coordinazione tra i componenti o i nodi è essenziale per ottenere i comportamenti desiderati. Le diverse forme di coordinazione possono essere classificate in base all'**accoppiamento temporale** e all'**accoppiamento referenziale**.

- La coordinazione **accoppiata temporalmente** comporta una **coordinazione basata su vincoli temporali** o di tempo. I componenti sono coordinati in modo che le loro azioni o eventi siano **sincronizzati nel tempo**. Ad esempio, un sistema distribuito può richiedere che i componenti eseguano i loro compiti in un ordine specifico o a intervalli di tempo specifici per ottenere un comportamento coordinato. Il coordinamento accoppiato temporalmente può fornire un coordinamento preciso, ma può introdurre dipendenze e potenziali ritardi nel sistema.
- La coordinazione **temporalmente disaccoppiata** consente ai componenti di operare in modo **indipendente senza vincoli temporali rigidi**. I componenti possono eseguire compiti o eventi in modo indipendente senza una coordinazione basata sul tempo. Ciò può fornire una maggiore flessibilità e scalabilità del sistema, ma può richiedere meccanismi aggiuntivi per garantire il coordinamento e la coerenza tra i componenti.
- La coordinazione **ad accoppiamento referenziale** comporta una **coordinazione basata su riferimenti**

o puntatori. I componenti condividono riferimenti o puntatori alle risorse o ai servizi degli altri e la coordinazione si ottiene facendo riferimento a risorse o servizi condivisi. Ad esempio, in un sistema distribuito, i componenti possono condividere riferimenti a un database comune o a un servizio condiviso per il coordinamento. L'accoppiamento referenziale può fornire un coordinamento efficiente e diretto, ma può introdurre uno stretto accoppiamento tra i componenti.

- La coordinazione **disaccoppiata referenzialmente consente ai componenti di operare in modo indipendente senza condividere riferimenti o puntatori.** I componenti possono operare in modo indipendente senza fare riferimento diretto alle risorse o ai servizi degli altri. Il coordinamento si ottiene con altri mezzi, come il passaggio di messaggi, eventi o dati condivisi. La coordinazione disaccoppiata referenzialmente può fornire un accoppiamento lasso tra i componenti, consentendo una maggiore flessibilità e scalabilità, ma può richiedere meccanismi aggiuntivi per la coordinazione e la comunicazione.

Le diverse forme di coordinamento offrono compromessi tra precisione, flessibilità e scalabilità nei sistemi distribuiti. La scelta del meccanismo di coordinamento dipende dai requisiti, dalle caratteristiche e dai vincoli del sistema distribuito e dei suoi componenti.

1.6.1 Astrazioni di livello superiore

Le **astrazioni di livello superiore** sono livelli aggiuntivi di astrazione che vengono costruiti sopra il middleware, i protocolli o le API esistenti per fornire **funzionalità più avanzate e facilità d'uso nei sistemi distribuiti**. Queste astrazioni sono progettate per semplificare lo sviluppo e la gestione delle applicazioni distribuite, incapsulando funzionalità complesse in costrutti di livello superiore più facili da comprendere e da utilizzare. Queste astrazioni sono in genere implementate utilizzando middleware o protocolli come gRPC, Java Messaging System (JMS), JavaSpaces e altri.

Le astrazioni di livello superiore possono essere classificate in due gruppi principali: **basate su RPC** e **basate su MOM**.

Il primo include (**RPC based**):

- Sistemi a oggetti distribuiti (Distributed Object Systems)
- Servizi distribuiti e servizi web
- Risorse distribuite

Mentre quest'ultimo (**MOM based**):

- Sistemi Publish-Subscribe
- Code di messaggi

Sistemi a oggetti distribuiti (Distributed Object Systems)

I sistemi a oggetti distribuiti forniscono astrazioni per oggetti o componenti distribuiti che possono essere invocati a distanza attraverso una rete, consentendo un'interazione continua tra i componenti distribuiti come se fossero oggetti locali. Questa astrazione è spesso utilizzata nei paradigmi di programmazione orientati agli oggetti, dove gli oggetti possono essere distribuiti su più nodi di un sistema distribuito. Questa specifica astrazione può essere implementata attraverso uno scambio diretto tra client e server (utilizzando un RPC) o attraverso l'uso di un broker.

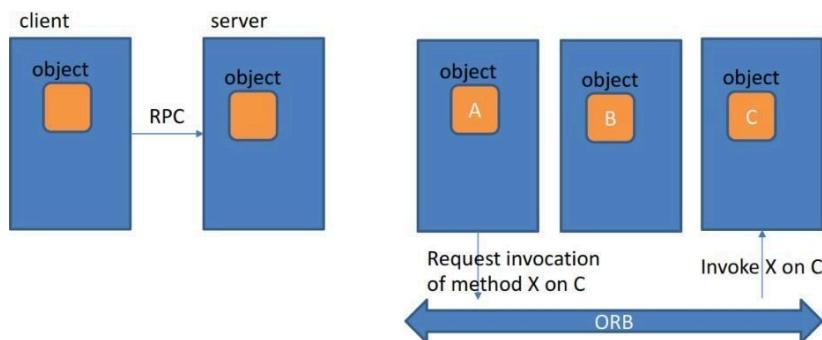


Figura 9: DOS diretto vs broker

Servizi distribuiti

I servizi distribuiti sono un'astrazione dei sistemi distribuiti simile agli oggetti distribuiti, ma con alcune differenze fondamentali. Mentre sia gli oggetti distribuiti sia i servizi distribuiti hanno alcune caratteristiche uniche che li distinguono dall'invocazione di funzionalità attraverso una rete.

Le principali differenze tra servizi distribuiti e oggetti distribuiti sono:

- **Grana più grande:** I servizi hanno in genere un'interfaccia a grana più grossa rispetto agli oggetti. Mentre gli oggetti forniscono metodi o operazioni a grana fine che possono essere invocati in remoto, i servizi spesso espongono unità di funzionalità o processi aziendali più grandi come interfaccia. Ciò consente interazioni a grana più grossa tra i componenti distribuiti, che possono essere più efficienti in certi scenari.
- **Autonomia e longevità:** I servizi sono progettati per essere entità autonome e longeve che possono operare in modo indipendente e persistente nel tempo. A differenza degli oggetti, che in genere vengono creati e distrutti secondo le necessità durante l'esecuzione di un programma, i servizi possono essere progettati per essere processi di lunga durata che forniscono funzionalità continue e sono disponibili per l'uso da parte dei clienti in qualsiasi momento.
- **Disponibilità inter-organizzativa:** I servizi possono essere resi disponibili per l'uso da parte di clienti diversi di organizzazioni diverse. I servizi possono essere progettati per essere esposti su Internet o su altre reti ad ampio raggio, consentendo ai clienti di organizzazioni o domini diversi di accedere e utilizzare le funzionalità fornite dal servizio. Ciò consente collaborazioni e partnership inter-organizzative, in cui i servizi possono essere condivisi tra più entità.
- **Composizione di servizi:** I servizi consentono la composizione dei servizi, che si riferisce alla capacità di combinare e orchestrare più servizi per creare funzionalità più complesse e di livello superiore. I servizi possono essere progettati per essere componibili, consentendo di utilizzarli come blocchi di costruzione per creare nuovi servizi o per implementare processi aziendali complessi che si estendono a più servizi. Questo garantisce flessibilità ed estensibilità nella progettazione di applicazioni distribuite.

Esempi di servizi distribuiti sono l'architettura orientata ai servizi (SOA) e i servizi web. La SOA è un approccio architettonico che utilizza i servizi come elementi fondamentali per la creazione di applicazioni distribuite, dove i servizi sono progettati per essere liberamente accoppiati e interoperabili. I servizi Web sono un'implementazione popolare della SOA, che utilizza protocolli basati sul Web come SOAP, REST e WSDL per esporre e invocare servizi su Internet. I servizi Web rappresentano un modo ampiamente adottato e standardizzato di implementare servizi distribuiti nei moderni sistemi distribuiti.

Risorse distribuite

Le risorse distribuite forniscono astrazioni per la gestione di risorse condivise in un sistema distribuito, come database distribuiti, file system distribuiti o cache distribuite. Queste astrazioni consentono un accesso efficiente e trasparente alle risorse condivise tra i nodi distribuiti, permettendo un coordinamento e una condivisione dei dati senza soluzione di continuità tra i componenti distribuiti. Sono simili ai servizi distribuiti e sono stati introdotti con l'avvento del Representational State Transfer (REST) e dell'Hypertext Transfer Protocol (HTTP). Nel contesto delle architetture RESTful, una risorsa è un'entità di dati con uno stato su cui sono possibili operazioni fisse (Create, Read, Update, Delete - CRUD).

Il punto principale delle risorse distribuite è il concetto di **risorsa**. Una risorsa è un'entità di dati con stato accessibile in rete. Le risorse possono essere qualsiasi cosa che possa essere identificata da un URI (Uniform Resource Identifier), come pagine web, documenti, immagini o qualsiasi altro tipo di dati. Nelle architetture RESTful, le risorse sono gli elementi fondamentali che vengono manipolati attraverso metodi HTTP standard, come GET, POST, PUT e DELETE, per eseguire operazioni CRUD.

Le risorse distribuite sono un concetto importante nei moderni sistemi basati sul web e sono ampiamente utilizzate nelle architetture RESTful, che sono diventate un approccio popolare per la progettazione di applicazioni distribuite scalabili e liberamente accoppiate.

Sistemi Publish-Subscribe

Publish-Subscribe è un paradigma di messaggistica utilizzato nei sistemi distribuiti che realizza il disaccoppiamento referenziale tra processi o componenti. In questo paradigma, i processi non hanno bisogno di conoscere gli altri e la comunicazione avviene attraverso il concetto di messaggi o eventi. I client possono eseguire due operazioni elementari: pubblicare, che comporta la generazione di un messaggio o di un evento, e sottoscrivere, che comporta l'espressione di interesse per una classe di messaggi o eventi.

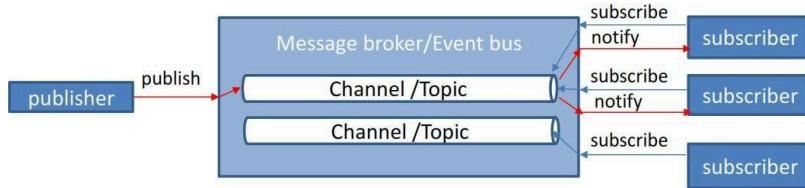


Figura 10: Sistema Publish/Subscribe

Le caratteristiche principali del meccanismo Publish-Subscribe sono:

- **Disaccoppiamento referenziale:** Publish-Subscribe consente il disaccoppiamento referenziale tra i processi o i componenti di un sistema distribuito. I processi possono operare in modo indipendente e non devono avere conoscenze dirette o dipendenze reciproche. Ciò consente un accoppiamento lasco e una flessibilità nella progettazione del sistema, in quanto i processi possono evolvere in modo indipendente senza influenzarsi a vicenda.
- **Operazione di pubblicazione:** L'operazione di pubblicazione comporta la generazione di un messaggio o di un evento da parte di un processo o di un componente. Questo messaggio o evento contiene tipicamente dati o informazioni che devono essere comunicati ad altri processi o componenti del sistema. Il messaggio o l'evento pubblicato non viene inviato direttamente a sottoscrittori specifici, ma viene trasmesso a tutti i sottoscrittori interessati.
- **Operazione di sottoscrizione:** L'operazione di sottoscrizione comporta l'espressione dell'interesse per una classe di messaggi o eventi da parte di un processo o di un componente. I sottoscrittori specificano il loro interesse per determinati tipi o categorie di messaggi o eventi e il sistema instrada i messaggi o gli eventi pertinenti ai componenti sottoscritti. Ciò consente ai componenti di ricevere selettivamente solo i messaggi o gli eventi rilevanti per la loro funzionalità o interesse.
- **Message Broker/Event Bus:** i sistemi Publish-Subscribe impiegano tipicamente un message broker o un event bus, che è responsabile della gestione dell'instradamento e della consegna di messaggi o eventi tra editori e sottoscrittori. Il message broker o event bus si occupa di notificare ogni messaggio o evento pubblicato a tutti i sottoscrittori attivi e funzionanti che hanno espresso interesse per la corrispondente categoria di messaggi o eventi.
- **Centralizzato, decentralizzato o basato su cloud:** I sistemi Publish-Subscribe possono essere implementati in modi diversi, a seconda dell'architettura e dei requisiti del sistema distribuito. Possono essere centralizzati, dove un singolo message broker o event bus gestisce tutte le operazioni di messaggio, come nel caso di MQTT (Message Queuing Telemetry Transport). Possono anche essere decentralizzati, in cui più message broker o event bus vengono utilizzati per distribuire le operazioni di messaggistica, come nel caso del Data Distribution Service (DDS). Infine, possono essere basati sul cloud, dove un fornitore di servizi cloud offre un servizio Publish-Subscribe gestito, come Amazon SNS (Simple Notification Service) nell'ambito di Amazon Web Services (AWS).

Publish-Subscribe è un paradigma di messaggistica molto diffuso e utilizzato in vari sistemi distribuiti, tra cui le applicazioni Internet of Things (IoT), le architetture event-driven e le architetture a microservizi, in quanto offre accoppiamento libero, scalabilità e flessibilità nella progettazione di sistemi distribuiti.

Coda di messaggi

Le code di messaggi sono un tipo di astrazione di livello superiore nei sistemi distribuiti che forniscono un'implementazione distribuita di una struttura di dati a coda. In una coda di messaggi, i client possono mettere in coda (o accodare) i messaggi da inviare ad altri componenti o processi, e questi messaggi rimangono memorizzati nella coda fino a quando non vengono tolti (o rimossi) da un componente destinatario. Le code di messaggi consentono l'interazione asincrona tra i componenti, che possono inviare messaggi senza attendere risposte immediate, consentendo il disaccoppiamento e la scalabilità nei sistemi.

distribuiti.

Le operazioni di base in una coda di messaggi sono enqueue e dequeue. I client possono mettere in coda i messaggi aggiungendo alla fine della coda e questi messaggi vengono memorizzati nella coda nell'ordine in cui sono stati inseriti. I client possono anche dequeueare i messaggi, rimuovendoli dalla parte anteriore della coda; questi messaggi vengono elaborati dal componente o processo destinatario nell'ordine in cui sono stati dequeueati.

Le code di messaggi offrono diversi vantaggi nei sistemi distribuiti:

- **Disaccoppiare i componenti o i processi** che inviano e ricevono messaggi, in quanto non devono conoscere i dettagli dell'altro o essere attivi nello stesso momento. Ciò consente un accoppiamento lasco tra i componenti, permettendo loro di evolvere in modo indipendente e di essere sostituiti o aggiornati senza impattare sugli altri componenti.
- **Permettere l'interazione asincrona**, in cui i componenti possono inviare messaggi senza attendere risposte immediate. Ciò può migliorare le prestazioni e la scalabilità dei sistemi distribuiti, in quanto i componenti possono continuare a elaborare altre attività mentre i messaggi sono memorizzati nella coda e i destinatari possono elaborare i messaggi al proprio ritmo.
- **Forniscono tolleranza agli errori e affidabilità** nei sistemi distribuiti. I messaggi possono essere conservati nella coda fino a quando non vengono dequotati ed elaborati con successo, garantendo che i messaggi non vadano persi a causa di guasti o crash del sistema. Inoltre, le code di messaggi sono in grado di gestire velocità variabili di messaggi e traffico intenso, consentendo una gestione fluida dell'elaborazione dei messaggi nei sistemi distribuiti.

Le code di messaggi possono essere implementate in vari modi, ad esempio utilizzando sistemi di accodamento di messaggi autonomi o come parte di middleware o framework di messaggistica. Esempi di sistemi di accodamento di messaggi molto diffusi sono RabbitMQ, Apache Kafka e ActiveMQ, mentre i framework di messaggistica come Advanced Message Queuing Protocol (AMQP) e Java Message Service (JMS) forniscono API standardizzate per lavorare con le code di messaggi nei sistemi distribuiti.

1.6.2 Astrazioni di livello inferiore

I **socket** sono un tipo di astrazione di comunicazione che **opera a un livello inferiore nei sistemi distribuiti**. Forniscono un'interfaccia generica di basso livello per stabilire canali di comunicazione tra processi o sistemi su una rete. I socket offrono primitive di comunicazione elementari, come connect, bind, listen, send e receive, che possono essere utilizzate per inviare e ricevere dati attraverso la rete.

Una caratteristica fondamentale dei socket è che i protocolli di livello superiore devono essere definiti dall'utente su di essi. I socket forniscono un'interfaccia di base per la trasmissione dei dati, ma la semantica e la struttura dei dati, così come i modelli di comunicazione, devono essere definiti dall'applicazione o dal sistema che li utilizza. Ciò consente la massima flessibilità, in quanto i socket possono essere utilizzati per implementare modalità di comunicazione personalizzate, adattate ai requisiti specifici di un sistema distribuito.

Esistono diversi esempi di socket in uso, tra cui i socket TCP/IP, che sono lo standard de-facto per l'accesso ai protocolli di livello 4-3-2 dello stack TCP/IP. I socket TCP/IP forniscono canali di comunicazione affidabili e orientati alla connessione, ampiamente utilizzati nei sistemi distribuiti per applicazioni come il trasferimento di file, la posta elettronica e la navigazione web.

Un altro esempio è WebSockets, un'interfaccia per la gestione di canali full-duplex basati su TCP attraverso una singola connessione. I WebSocket si basano su un protocollo standard compatibile con HTTP e consentono la comunicazione bidirezionale tra un client e un server in tempo reale. I WebSocket sono comunemente utilizzati nelle applicazioni web per funzioni quali la messaggistica in tempo reale, le notifiche e la modifica collaborativa.

In sintesi, i socket sono astrazioni di comunicazione di livello inferiore nei sistemi distribuiti che offrono un'interfaccia generica e flessibile per stabilire canali di comunicazione. I protocolli di livello superiore devono essere definiti sopra i socket, consentendo di implementare astrazioni di comunicazione personalizzate in base a requisiti specifici. Esempi di socket sono i socket TCP/IP e i WebSocket, ampiamente utilizzati in vari sistemi distribuiti per diversi scopi di comunicazione.

2. Servizi Web e stile architettonico REST

In questa sezione tratteremo i fondamenti dei servizi Web e approfondiremo il concetto di API REST. Esploreremo inoltre varie architetture distribuite e modelli che descrivono il livello di conformità di un servizio web ai vincoli REST.

2.1 SOA: Architettura orientata ai servizi

L'**architettura orientata ai servizi (SOA)** è un approccio alla progettazione del software in cui i sistemi software sono organizzati come un insieme di servizi. I servizi sono componenti software autonomi e contenuti che forniscono funzionalità specifiche e sono progettati per essere accoppiati in modo lasco, ovvero sono indipendenti l'uno dall'altro e possono evolvere in modo indipendente senza influenzare il sistema complessivo.

I servizi in SOA sono tipicamente forniti attraverso interfacce pubblicate e automaticamente escludibili. Queste interfacce sono leggibili dalla macchina, il che consente la scoperta, la composizione e l'invocazione automatica dei servizi. I servizi sono descritti da contratti, che sono costituiti da una o più interfacce. I contratti definiscono i messaggi di ingresso e di uscita, i formati dei dati e i protocolli di comunicazione utilizzati per interagire con i servizi.

Un servizio in SOA è implementato da una singola istanza sempre disponibile, ovvero accessibile e invocabile dai client in qualsiasi momento. I servizi sono tipicamente a grana grossa, cioè forniscono unità di funzionalità relativamente grandi e significative, in contrapposizione ai servizi a grana fine che forniscono operazioni piccole e specifiche. I servizi a grana grossa consentono un migliore incapsulamento delle funzionalità e riducono l'overhead della comunicazione tra i servizi.

Uno dei principi chiave della SOA è l'accoppiamento libero. I servizi in SOA sono progettati in modo da essere accoppiati in modo lasco, ovvero sono indipendenti l'uno dall'altro e non hanno dipendenze dirette. Ciò consente ai servizi di evolvere in modo indipendente, garantendo flessibilità e scalabilità al sistema. Le interazioni tra i servizi in SOA sono tipicamente asincrone, cioè basate su scambi di messaggi, in cui i client inviano messaggi ai servizi e ricevono risposte in modo asincrono.

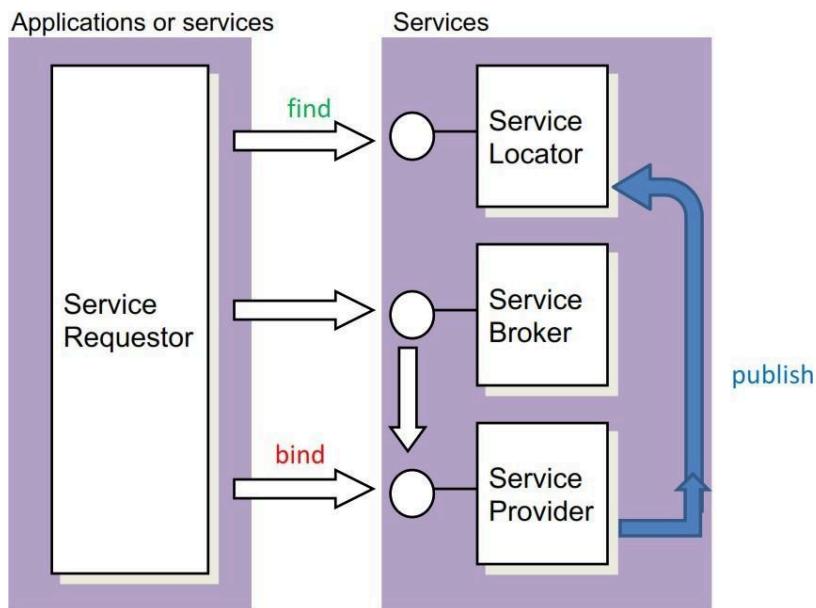


Figura 11: Architettura orientata ai servizi

In sintesi, l'architettura orientata ai servizi (SOA) è un approccio alla progettazione del software in cui i sistemi software sono organizzati come un insieme di servizi forniti attraverso interfacce, descritti da contratti e implementati da singole istanze. I servizi in SOA sono tipicamente a grana grossa, poco accoppiati e

interagiscono attraverso scambi di messaggi asincroni. La SOA fornisce flessibilità, scalabilità e autonomia ai sistemi software, consentendo ai servizi di evolvere in modo indipendente.

2.2 Servizi web (Web Services)

I servizi Web sono servizi distribuiti basati sul protocollo HTTP. Forniscono un modo standardizzato per i sistemi software di comunicare e condividere dati su Internet o intranet, indipendentemente dall'hardware, dal sistema operativo o dal linguaggio di programmazione sottostante.



Figura 12: Servizio Web

I servizi Web seguono una serie di standard e protocolli, noti collettivamente come stack di servizi Web, che comprende XML (eXtensible Markup Language), SOAP (Simple Object Access Protocol), WSDL (Web Services Description Language) e UDDI (Universal Description, Discovery, and Integration). Questi standard definiscono le modalità di descrizione, scoperta e invocazione dei servizi.

Inoltre, essi sono tipicamente progettati per essere indipendenti dalla piattaforma e magnetici rispetto al linguaggio, il che significa che possono essere implementati in qualsiasi linguaggio di programmazione e che possono essere accessibili da client in esecuzione su piattaforme diverse. Ciò rende i WS altamente interoperabili e consente una perfetta integrazione tra sistemi e tecnologie diverse.

Principalmente, è possibile classificarli in due tipi principali:

- **Servizi Web basati su SOAP:** Questi servizi utilizzano SOAP come protocollo di messaggistica per scambiare dati XML strutturati tramite HTTP o altri protocolli di trasporto. SOAP fornisce un modo standardizzato per formattare i messaggi, definire le operazioni e gestire gli errori nei servizi Web.

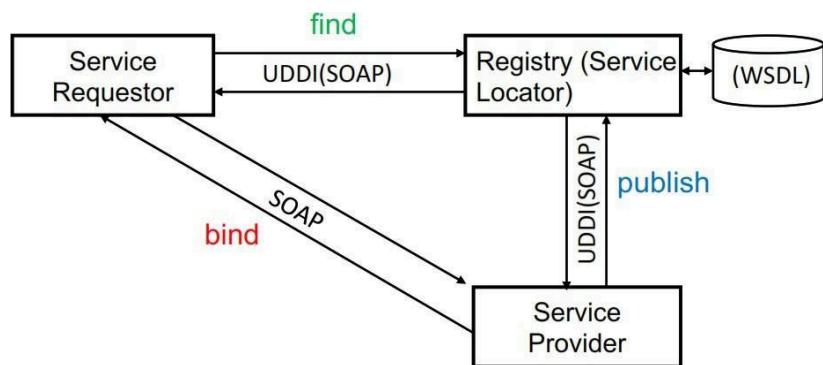


Figura 13: Servizio Web basato su SOAP

- **Servizi Web RESTful:** Questi servizi utilizzano i principi dell'architettura Representational State Transfer (REST) per progettare e implementare servizi distribuiti. I servizi Web RESTful utilizzano metodi HTTP standard come GET, POST, PUT e DELETE per eseguire operazioni sulle risorse, identificate da URL.



Figura 14: Servizio web RESTful

Ampiamente utilizzati in vari settori, tra cui l'e-commerce, la finanza, la sanità, la logistica e altri ancora, i servizi web consentono l'integrazione tra sistemi diversi, permettono la composizione e l'orchestrazione dei servizi e forniscono un approccio scalabile e flessibile alla creazione di applicazioni distribuite.

2.3 Stile architettonico REST

Representational State Transfer (REST) è uno stile architettonico per la progettazione di sistemi distribuiti, introdotto da Roy Fielding, che è stato uno dei principali progettisti del protocollo HTTP 1.1. REST è un insieme di vincoli che **definiscono come devono essere strutturati i sistemi e come devono comunicare sul web o su qualsiasi altra rete**.

L'architettura REST si basa su un **insieme di vincoli che definiscono il comportamento del sistema**. Questi vincoli forniscono linee guida per la **progettazione di sistemi distribuiti scalabili, manutenibili e interoperabili**. I principali vincoli di REST sono:

- **Client/Server:** Il sistema è diviso in due componenti separati: il client e il server. Il client è responsabile dell'interfaccia utente e dell'esperienza utente, mentre il server è responsabile dell'elaborazione delle richieste e della gestione delle risorse. Questa **separazione dei ruoli** consente un'evoluzione indipendente dei componenti client e server, mantenendoli semplici e consentendo la scalabilità e la flessibilità dell'architettura del sistema.
- **Senza stato (stateless):** Ogni richiesta da un client a un server deve contenere tutte le informazioni necessarie per comprendere ed elaborare la richiesta. Il server **non deve memorizzare alcuna informazione sullo stato del client tra una richiesta e l'altra**. Ciò consente di **semplificare il bilanciamento del carico** e la **scalabilità**, in quanto i server possono gestire le richieste di qualsiasi client senza fare affidamento sullo stato memorizzato. Inoltre, migliora la visibilità e il monitoraggio (poiché il server non deve monitorare tutte le interazioni), ma anche l'affidabilità: infatti, poiché lo stato non è conservato sul server, semplifica il ripristino da un guasto parziale.

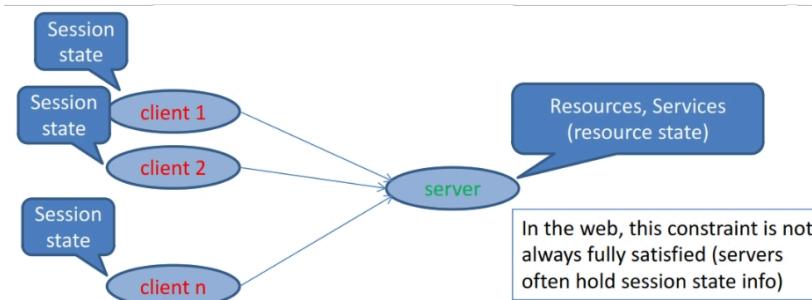


Figura 15: Protocollo di interazione stateless REST

- **Cacheable (memorizzabile nella cache):** Le risposte del server possono essere memorizzate nella cache del client, etichettandole come memorizzabili o non memorizzabili, migliorando le prestazioni e riducendo il carico sul server. La cache consente un utilizzo efficiente delle risorse e riduce la necessità di ripetere le richieste al server.
- **A strati:** L'architettura può essere composta da più livelli, dove ogni livello fornisce un insieme specifico di funzionalità. I livelli possono essere aggiunti o rimossi senza influenzare il comportamento complessivo del sistema in modo "verticale". Questo permette una flessibilità nell'architettura del sistema e promuove la separazione delle preoccupazioni, producendo di fatto un'architettura multilivello.

- **Interfaccia uniforme:** L'interfaccia tra client e server è uniforme, con un insieme fisso di metodi ben definiti e coerenti tra tutte le risorse. Questo semplifica l'architettura del sistema, ne facilita la comprensione e l'uso e consente l'evoluzione e l'estensione del sistema senza ripercussioni sui client.

Complessivamente, REST offre un approccio semplice, scalabile e non vincolato per l'architettura di sistemi distribuiti, rendendolo una scelta popolare per la costruzione di applicazioni web moderne, microservizi e altre applicazioni distribuite.

REST (REpresentational State Transfer) è un'architettura per la progettazione di servizi web che si basa su una serie di principi e vincoli per comunicare tra client e server in modo semplice, scalabile e standardizzato. È uno degli approcci più comuni per la creazione di **API web**.

Principi fondamentali di REST:

1. **Architettura client-server:**
 - Il client (ad esempio, un'applicazione frontend) e il server (che ospita i dati e la logica) sono separati, favorendo la scalabilità e l'indipendenza.
2. **Stateless:**
 - Ogni richiesta del client al server deve contenere tutte le informazioni necessarie per elaborarla, senza che il server memorizzi lo stato delle sessioni.
3. **Cacheable:**
 - Le risposte del server possono essere memorizzate nella cache per migliorare le prestazioni.
4. **Uniform Interface:**
 - L'interfaccia tra client e server deve essere standardizzata e coerente.
5. **Risorse identificate da URI:**
 - Ogni risorsa (ad esempio, utenti, prodotti) è identificata da un URL unico.
6. **Rappresentazione delle risorse:**
 - I dati delle risorse sono rappresentati in formati standard come JSON o XML.

2.4 Risorse

Nell'architettura REST, il vincolo dell'interfaccia uniforme richiede che le interfacce del server siano orientate alle risorse. Il concetto chiave di questo vincolo è la risorsa, che è un elemento informativo con stato variabile nel tempo. Una **risorsa può essere qualsiasi cosa che possa essere identificata da un URI** (Uniform Resource Identifier), come un documento, un'immagine, un profilo utente o qualsiasi altra entità con un'identità unica.

Le risorse nei sistemi RESTful possono avere zero o più rappresentazioni, che sono rappresentazioni diverse della stessa risorsa in diversi tipi di media come XML, JSON, HTML, ecc. Le rappresentazioni sono utilizzate per trasferire lo stato di una risorsa tra il client e il server. Ad esempio, un client può richiedere una risorsa in formato JSON e il server può rispondere con lo stato di tale risorsa serializzata come JSON.

In REST, quando un client richiede un'operazione su una risorsa, una rappresentazione della risorsa, che può essere il suo stato attuale, passato o futuro, può essere trasferita tra il client e il server. Tuttavia, le risorse vere e proprie rimangono sempre sul lato server e i loro URI servono come puntatori per accedervi.

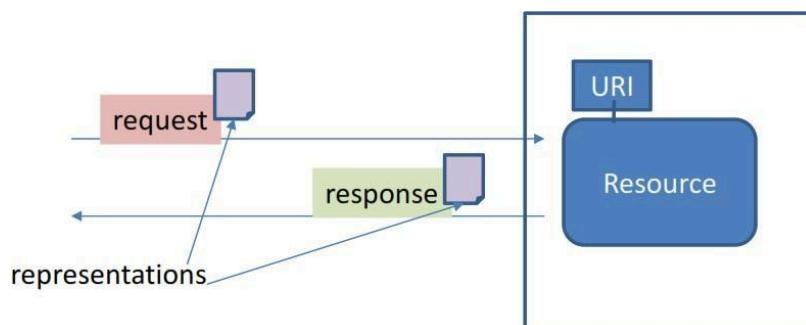


Figura 16: Richiesta e risposta di una risorsa

Il protocollo HTTP offre un insieme di interfacce fisse e predefinite per agire sulle risorse; l'operazione richiesta ha la seguente semantica per HTTP:

- **Metodo richiesto**
- **Corpo della richiesta**
- **Intestazioni della richiesta** (metadati di controllo)
- **URL di destinazione** (incluse anche le stringhe di query)

Il risultato dell'operazione viene comunicato nella risposta tramite: un *codice di stato*, un *corpo della risposta* e altre *intestazioni della risposta*. In sostanza, questo tipo di interfaccia uniforme offre le operazioni CRUD definite da REST; di seguito viene fornito uno schema che descrive la corrispondenza:

CRUD Operation	HTTP request	HTTP response (if no error)
Create a new resource under <i>res</i> (or send data to <i>res</i> for resource-specific processing)	<code>POST res</code> (resource or data representation in the body)	URL of new resource created and/or result of processing (in body)
Read resource <i>res</i>	<code>GET res</code> (no body)	representation of current state of <i>res</i> (in the body)
Update resource <i>res</i> by replacing its state with a new one (<i>res</i> can be created)	<code>PUT res</code> (new resource representation in the body)	description of executed operation (200 or 201 or 204 status code)
Delete resource <i>res</i>	<code>DELETE res</code>	

Figura 17: Operazioni CRUD nel dominio REST

Sono stati definiti altri metodi che forniscono operazioni più specifiche:

Operation	HTTP request	HTTP response (if no error)
Read information about resource <i>res</i> without transferring its representation	<code>HEAD res</code> (no body)	same as for GET but without body (only headers sent)
Read supported methods for a resource	<code>OPTIONS res</code> (no body)	list of supported methods (in the Allow header and possibly in the body)
Test the connection to resource <i>res</i> (loopback testing similar to echo)	<code>TRACE res</code>	the received request (in the body)
Partially Update the resource <i>res</i> by applying a patch	<code>PATCH res</code> (patch to be applied to <i>res</i> in the body)	description of executed operation (200 or 201 or 204 status code)

Figura 18: Altre operazioni nel dominio REST

Per tutti i metodi forniti è possibile valutare quali proprietà possiedono, in particolare in termini di: *Idempotenza*, *Sicurezza*, *Risposta memorizzabile nella cache*, *Presenza del corpo della richiesta*. Ecco uno

schema dettagliato:

HTTP method	CRUD operations	idempotent	Safe	Request body	Cacheable response
POST	C	no	no	yes	(no)
GET	R	yes	yes	no	yes
PUT	U, C	yes	no	yes	no
DELETE	D	yes	no	no	no
HEAD	R	yes	yes	no	yes
OPTIONS	R	yes	yes	no	no
TRACE	-	yes	yes	yes	no
PATCH	U, C	no	no	yes	no

Figura 19: Metodi e proprietà HTTP

Per una comprensione più dettagliata delle operazioni REST sulle risorse, facciamo un esempio: consideriamo courses una risorsa che rappresenta un insieme di corsi, mentre courses/id una risorsa che rappresenta un singolo corso:

Method URI pair	Meaning
POST <code>courses</code>	Create a new course with given representation, and add it to the collection as a new resource under courses. Return its URI
GET <code>courses</code>	Read the whole collection of courses
GET <code>courses/id</code>	Read only course id in the collection
GET <code>courses?year=1</code>	Read only the courses of the first year in the collection
PUT <code>courses</code>	Update the whole collection with the given representation
PUT <code>courses/id</code>	Update only course id in the collection with the given representation
DELETE <code>courses</code>	Delete the whole collection of courses
DELETE <code>courses/id</code>	Delete only course id in the collection

Figura 20: Operazioni sulle risorse: Esempio

2.5 HATEOAS (Hypermedia as the Engine of Application State)

HATEOAS è l'acronimo di *Hypermedia as the Engine of Application State*. È un vincolo dello stile architettonico REST (*Representational State Transfer*) che regola la progettazione e l'interazione dei sistemi distribuiti, in particolare dei servizi web.

In un sistema distribuito che segue il vincolo HATEOAS, i collegamenti ipermediali sono utilizzati per rappresentare in modo dinamico le azioni disponibili o le transizioni di stato che un client può eseguire su una risorsa.

- **Collegamenti ipermediali (Hypermedia):** Ogni risposta dal server include non solo i dati richiesti, ma anche dei link ipermediali che indicano le possibili azioni future che il client può eseguire su quella risorsa. Ad esempio, una risorsa "ordine" potrebbe includere link per "annullare l'ordine" o "verificare lo stato di spedizione".
- **Navigazione dinamica:** I client scoprono le azioni disponibili tramite i link forniti dal server, anziché avere una mappa fissa degli endpoint. Questo rende il sistema più flessibile, in quanto i cambiamenti del server possono essere gestiti dal client senza dover aggiornare il suo codice.
- **Decoupling (Disaccoppiamento):** HATEOAS disaccoppia il client dal server. Il client non deve conoscere gli endpoint o le operazioni specifiche in anticipo; può scoprire dinamicamente come interagire con il

server grazie agli ipermedia.

Tuttavia, può anche aggiungere complessità alla progettazione delle API e all'implementazione dei client, poiché questi ultimi devono comprendere e interpretare i collegamenti ipermediali forniti dal server. Una documentazione adeguata e il rispetto dei principi di progettazione HATEOAS sono importanti per il successo dell'implementazione di questo vincolo nei sistemi distribuiti.

2.6 Richardson Maturity Model

In questa sezione, discuteremo il Richardson Maturity Model, che **fornisce un quadro di riferimento per valutare il livello di RESTfulness dei servizi web o delle API**. Il modello, proposto da Leonard Richardson, consiste in quattro livelli che descrivono l'evoluzione delle architetture RESTful:

- **Livello 0: La palude del POX (Plain Old XML)**

A questo livello, l'API utilizza principalmente HTTP come meccanismo di trasporto per i payload XML. C'è non c'è una chiara separazione delle preoccupazioni tra le risorse e le loro rappresentazioni e i metodi HTTP non vengono utilizzati per le operazioni sulle risorse. Questo livello non è considerato pienamente RESTful, poiché manca di principi chiave come l'identificazione delle risorse tramite URI e l'uso di metodi HTTP per le operazioni.

- **Livello 1: Risorse**

A questo livello, l'API inizia a utilizzare i metodi HTTP per eseguire operazioni CRUD (Create, Read, Update, Delete) sulle risorse. Ogni risorsa è identificata da un URI, ma l'API utilizza ancora principalmente i payload, tipicamente in formato XML, per lo scambio di dati. Sebbene questo livello introduca l'identificazione delle risorse e l'uso di base dei metodi HTTP, mancano ancora alcuni principi chiave di RESTful, come l'uso di ipermedia e di messaggi autodescrittivi.

- **Livello 2: Verbi HTTP**

A questo livello, l'API accetta pienamente l'uso dei metodi HTTP per le operazioni sulle risorse, secondo la loro semantica. I metodi HTTP come GET, POST, PUT e DELETE sono utilizzati per eseguire operazioni sulle risorse e i codici di stato HTTP sono utilizzati per indicare l'esito delle richieste. Le risposte dell'API possono essere in vari formati, come XML, JSON o altri. Questo livello è più RESTful in quanto aderisce ai principi dell'identificazione delle risorse, della comunicazione client-server senza stato e dell'uso di metodi HTTP standard.

- **Livello 3: Controlli ipermediiali**

A questo livello più alto, l'API utilizza controlli ipermediali, tipicamente sotto forma di collegamenti ipertestuali, per guidare le transizioni di stato di un'applicazione client. I client possono scoprire e interagire dinamicamente con le risorse attraverso i controlli ipermediali, rendendo l'API più autodescrittiva e disaccoppiata. Questo livello abbraccia pienamente il principio HATEOAS (Hypermedia As The Engine Of Application State), rendendo l'API altamente RESTful.

Il Richardson Maturity Model viene spesso utilizzato come linea guida per la progettazione di servizi web o API RESTful che aderiscono ai principi del Representational State Transfer (REST), uno stile architettonico per la progettazione di applicazioni in rete.

3. Sintassi e schemi astratti

In questa sezione, discuteremo l'approccio pratico alla progettazione di API REST attraverso l'uso di sintassi e schemi astratti. Una delle **sfide principali** nella progettazione di sistemi RESTful è **garantire la trasparenza dei dati**, in particolare quando si ha a che fare con sistemi eterogenei che possono avere diversi protocolli e formati di dati.

Il marshalling, noto anche come **serializzazione**, si riferisce al **processo di conversione dei dati da una rappresentazione interna a un formato adatto alla trasmissione in rete**. L'unmarshaling, o **deserializzazione**, è il **processo inverso** di conversione dei dati ricevuti in una rappresentazione interna. Questi processi sono fondamentali nelle API RESTful, in quanto consentono la comunicazione e lo scambio di dati tra sistemi diversi con rappresentazioni dei dati potenzialmente diverse.

L'uso di **sintassi astratte e schemi** fornisce un modo pratico per definire la struttura e il formato dei dati nelle API REST, garantendo coerenza e interoperabilità tra i sistemi. Le **sintassi astratte** definiscono la struttura e la semantica dei dati in modo indipendente dal linguaggio, mentre gli **schemi** forniscono un modo per specificare la sintassi concreta e le regole di validazione per i dati in un formato o in una codifica specifici, come JSON o XML. Definendo sintassi e schemi astratti, le API REST possono raggiungere la trasparenza e l'interoperabilità dei dati, consentendo a sistemi con protocolli e formati di dati diversi di comunicare e scambiare efficacemente i dati.

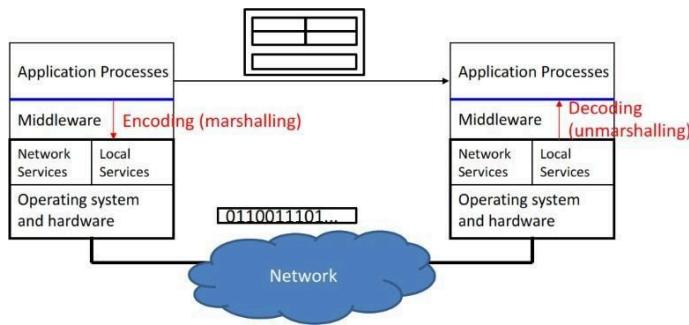


Figura 21: Marshaling e unmarshaling dei dati

Le applicazioni si affidano tipicamente a modalità standard per la rappresentazione dei dati indipendenti dal sistema, che possono includere soluzioni binarie (*come ASN.1, XDR, Protocol Buffers, ecc.*) o orientate ai caratteri (*come XML, JSON, YAML, ecc.*).

Questi standard sono generalmente costituiti da:

1. Un **linguaggio** che definisce tipi di dati astratti, noto anche come **sintassi astratta**.
2. **Rappresentazioni di dati "neutre"** che non sono legate ad alcun sistema specifico e possono essere utilizzate per rappresentare qualsiasi tipo di dato astratto definito dal linguaggio.

Per i ricevitori di dati è necessario un meccanismo di rappresentazione dei dati per separarli, convalidarli e decodificarli correttamente. A seconda dello scenario, il ricevitore può avere bisogno di una conoscenza preliminare del tipo di dati, oppure può dedurre il tipo di dati dai dati stessi.

3.1 Il linguaggio JSON Schema

In questa sezione vedremo come sia possibile specificare schemi JSON e utilizzarli per validare messaggi JSON. Poiché utilizziamo un documento JSON per descrivere un tipo di dati JSON, questa descrizione è leggibile e serializzabile; in questo modo è facile inviare questi dati in rete. I tipi di dati che possono essere descritti con questo linguaggio corrispondono a tipi disponibili solo in altri linguaggi di programmazione (*ad esempio, Javascript*).

Di seguito viene fornito un esempio:

```

1  {
2      "$schema": "http://json-schema.org/draft-07/schema#",
3      "title": "Schema di esempio",
4      "description": "Questo è uno schema JSON di esempio",
5      "type": "oggetto",
6      "property": {
7          "property1": {
8              "type": "stringa",
9              "description": "Questa è la descrizione della proprietà1".
10         },
11         "property2": {
12             "type": "intero",
13             "description": "Questa è la descrizione della proprietà2".
14         }
15     }
16 }
```

Listato 1: Esempio di schema JSON

Lo schema JSON ha la seguente struttura:

- **Sezione Metadati:**
 - \$schema: Specifica la versione dello schema JSON utilizzato.
 - titolo: Fornisce un titolo per lo schema.
 - descrizione: Fornisce una descrizione dello schema.
- **Descrizione del tipo:**
 - tipo: Specifica il tipo di oggetto JSON. In questo esempio, è impostato su "object" per definire un tipo di oggetto.
 - proprietà: Definisce le proprietà dell'oggetto JSON, ciascuna con il proprio tipo e la propria descrizione.

3.1.1 Tipi di dati dello schema JSON

Lo schema JSON supporta i seguenti tipi di dati:

- **stringa:**
 - tipo: Specifica il tipo di dati come "string".
 - formato: Specifica un formato per la stringa (ad esempio, "data", "e-mail", "uri").
 - pattern: Specifica un modello di espressione regolare a cui la stringa deve corrispondere.
 - minLength: Specifica la lunghezza minima della stringa.
 - maxLength: Specifica la lunghezza massima della stringa.
 - enum: Specifica un elenco di valori di stringa consentiti.
- **numero:**
 - tipo: Specifica il tipo di dati come "numero".
 - formato: Specifica un formato per il numero (ad esempio, "float", "double").
 - minimo: Specifica il valore minimo consentito del numero.
 - massimo: Specifica il valore massimo consentito del numero.
 - exclusiveMinimum: Specifica se il valore minimo è esclusivo (cioè non consentito).

- exclusiveMaximum: Specifica se il valore massimo è esclusivo (cioè non consentito).
- multipleOf: Specifica un multiplo per cui il numero deve essere divisibile.

– **intero:**

- tipo: Specifica il tipo di dati come "intero".
- formato: Specifica un formato per l'intero (ad esempio, "int32", "int64").
- minimo: Specifica il valore minimo consentito dell'intero.
- massimo: Specifica il valore massimo consentito del numero intero.
- exclusiveMinimum: Specifica se il valore minimo è esclusivo (cioè non consentito).
- exclusiveMaximum: Specifica se il valore massimo è esclusivo (cioè non consentito).
- multipleOf: Specifica un multiplo per cui l'intero deve essere divisibile.

– **booleano:**

- tipo: Specifica il tipo di dati come "booleano".

– **oggetto:**

- tipo: Specifica il tipo di dati come "oggetto".
- proprietà: Specifica le proprietà dell'oggetto, ciascuna con il proprio schema.
- additionalProperties: Specifica se sono consentite proprietà aggiuntive e il loro schema.
- richiesto: Specifica un elenco di nomi di proprietà richieste.
- minProperties: Specifica il numero minimo di proprietà consentite.
- maxProperties: Specifica il numero massimo di proprietà consentite.
- dipendenze: Specifica le dipendenze tra le proprietà².
- enum: Specifica i valori consentiti tramite enumerazione (utilizzando la sintassi JSON).

– **array:**

- tipo: Specifica il tipo di dati come "array".
- elementi: Specifica lo schema degli elementi della matrice.
- additionalItems: Specifica se sono ammessi elementi aggiuntivi e il loro schema.
- minItems: Specifica il numero minimo di elementi consentiti nella matrice.
- maxItems: Specifica il numero massimo di elementi consentiti nella matrice.
- uniqueItems: Specifica se tutti gli elementi dell'array devono essere unici.

– **nullo:**

- tipo: Specifica il tipo di dati come "null".

Nota: La proprietà "type" è comune a tutti i tipi di dati e specifica il tipo di dati del valore. Altre proprietà e vincoli possono variare a seconda del tipo di dati.

²Le dipendenze nello schema JSON possono essere definite in due modi:

* **Dipendenze di proprietà:** Questo tipo di dipendenza specifica che una proprietà dipende dalla presenza o dall'assenza di un'altra proprietà nello stesso oggetto. Può essere definita usando la parola chiave "dependencies" con il nome della proprietà come chiave e un array di nomi di proprietà dipendenti come valore.

* **Dipendenze dallo schema:** Questo tipo di dipendenza specifica che lo schema di una proprietà dipende dalla presenza o assenza di un'altra proprietà nello stesso oggetto. Può essere definita usando la parola chiave "dependencies" con il nome della proprietà come chiave e uno schema JSON come valore.

3.1.2 Combinazione di schemi

Uno schema JSON può essere il risultato di una combinazione di schemi; è possibile combinarli utilizzando uno di questi attributi:

- allOf: richiede la validità di tutti i sotto-schemi.
- anyOf: richiede la validità rispetto ad almeno un sub schema.
- oneOf: richiede la validità rispetto a un solo sub schema.
- not: non richiede validità.

In alternativa, può esserci anche una combinazione di schemi basata su if, then, else o facendo riferimento a un altro schema usando \$ref, questo riferimento può essere di tre tipi:

- Riferimento interno: riferimento a una parte dello schema. In questo caso, è possibile farlo "definendo" un pezzo di schema riutilizzabile con la proprietà "definitions". Ecco un esempio:

```
1   {
2     "definitions": {
3       "address": {
4         "type": "oggetto",
5         "property": {
6           "road": {"tipo": "string" },
7           "city": {"tipo": "string" },
8           "country": {"type": "string" }
9         }
10      }
11    },
12    "tipo": "oggetto",
13    "property": {
14      "orderDate": {"type": "string", "format": "data"},  
15      "quantity": {"tipo": "integer" },
16      "shippingAddress": { "$ref": "#/definitions/indirizzo" },
17      "billingAddress": { "$ref": "#/definitions/indirizzo" },
18    }
19  }
20 }
```

Listato 2: Esempio di \$ref che utilizza uno schema interno

- Riferimento esterno: riferimento a uno schema esterno nel file system locale.
- Riferimento esterno: riferimento a uno schema esterno disponibile in rete.

4. Progettazione dell'interfaccia del servizio (Service Interface Design)

Nel ciclo di vita del software, la progettazione delle interfacce è una parte fondamentale della fase di progettazione del software. Si tratta di definire l'architettura, compresi i moduli e le loro interfacce. Durante la fase di progettazione, è necessario affrontare vari problemi, come determinare come suddividere il sistema in moduli, decidere la granularità dei moduli, progettare le interfacce per la comunicazione tra i moduli e pianificare la distribuzione dei moduli in sistemi software distribuiti.

La progettazione del software è un'attività creativa che richiede un'attenta considerazione di vari fattori. Tuttavia, sono stati identificati alcuni principi fondamentali che guidano il processo di progettazione. Uno di questi principi è l'**Information Hiding**, che enfatizza una chiara separazione tra interfacce e implementazioni. Ciò aiuta a creare sistemi software modulari e manutenibili, in cui le interfacce forniscono un'astrazione ben definita per la comunicazione tra i moduli, mentre i dettagli dell'implementazione rimangono nascosti.

4.1 Principi di suddivisione

Nella progettazione del software, i principi di partizione vengono utilizzati per dividere un sistema o un'applicazione software in componenti o moduli più piccoli e gestibili. Questi principi mirano a migliorare la progettazione complessiva, la manutenibilità e le prestazioni del sistema. Ecco una breve panoramica di ciascuno dei principi di partizionamento:

- **Massimizzare la coesione interna:** I moduli all'interno di un sistema dovrebbero essere progettati per avere un'elevata coesione interna, il che significa che gli elementi all'interno di ciascun modulo sono strettamente correlati e lavorano insieme per uno scopo comune. In questo modo si creano componenti modulari e autosufficienti, più facili da comprendere, mantenere e aggiornare.
- **Ridurre al minimo l'accoppiamento:** I moduli devono essere progettati per avere un basso accoppiamento, cioè devono avere dipendenze minime da altri moduli. In questo modo, si creano componenti non accoppiati che sono più indipendenti e possono essere modificati o sostituiti senza influenzare l'intero sistema.
- **Prevedere e favorire i cambiamenti futuri e il riutilizzo:** I moduli devono essere progettati in modo da anticipare e accogliere i cambiamenti futuri e promuovere la riusabilità. Si tratta di considerare i potenziali cambiamenti o le modifiche che potrebbero essere necessari in futuro e di progettare i moduli in modo flessibile e modulare per consentire una facile modifica, estensione o riutilizzo dei componenti. Ciò contribuisce alla creazione di sistemi adattabili e scalabili, in grado di evolvere nel tempo in base ai cambiamenti dei requisiti.
- **Bilanciare il carico dei sottosistemi distribuiti:** Nei sistemi distribuiti o negli ambienti in cui i componenti sono distribuiti su più host o nodi, il carico di lavoro o i compiti devono essere distribuiti uniformemente tra i vari sottosistemi distribuiti. Ciò contribuisce a ottimizzare le prestazioni e la scalabilità del sistema distribuito, assicurando che nessun singolo sottosistema sia sovraccaricato da un carico di lavoro eccessivo, mentre altri sottosistemi rimangono sottoutilizzati.

Seguendo questi principi di partizione, i progettisti di software possono creare sistemi ben organizzati, modulari e manutenibili, che si adattano ai cambiamenti futuri e promuovono la riusabilità. Questi principi contribuiscono alla qualità e all'efficienza complessiva della progettazione del software, portando a sistemi software robusti e scalabili.

4.2 Scelta del livello di granularità

Nella progettazione del software, determinare il livello di granularità appropriato per la suddivisione di un sistema in moduli è fondamentale per ottenere un sistema ben progettato ed efficiente. Ecco alcune considerazioni chiave da tenere a mente:

- **Trovare il giusto compromesso:** la suddivisione di un sistema in moduli richiede di trovare il giusto equilibrio tra granularità e coesione. Se il livello di granularità è troppo fine, cioè il sistema è diviso in moduli molto piccoli, può avere un impatto negativo sulle prestazioni e sull'accoppiamento del sistema. Il partizionamento a grana fine può comportare un overhead in termini di comunicazione e coordinamento tra i moduli, con conseguente riduzione delle prestazioni. Inoltre, la suddivisione a grana fine

i moduli possono essere strettamente accoppiati, il che significa che le modifiche a un modulo possono avere un effetto a catena su altri moduli, aumentando l'accoppiamento tra i moduli e rendendo il sistema più difficile da mantenere e aggiornare.

- **Evitare una suddivisione a grana troppo grossa:** D'altra parte, se il livello di granularità è troppo grossolano, cioè il sistema è diviso in moduli monolitici di grandi dimensioni, può influire negativamente sulla coesione interna, sulla flessibilità e sul riutilizzo del sistema. Il partizionamento a grana grossa può dare origine a moduli troppo grandi e complessi, rendendo più difficile la comprensione e la manutenzione della logica interna e della funzionalità di ciascun modulo. Può anche limitare la flessibilità e l'adattabilità del sistema ai cambiamenti, in quanto le modifiche o gli aggiornamenti possono richiedere la modifica di ampie porzioni del sistema. Inoltre, i moduli a grana grossa potrebbero non essere facilmente riutilizzabili in altri contesti o progetti, riducendo il potenziale di riutilizzo del codice e l'efficienza.

In sintesi, la scelta del giusto livello di granularità per il partizionamento di un sistema in moduli è una decisione critica nella progettazione del software. È necessario trovare il giusto compromesso tra il partizionamento a grana fine e quello a grana grossa, considerando fattori quali le prestazioni, l'accoppiamento, la coesione interna, la flessibilità e il riutilizzo. Considerando attentamente questi fattori, i progettisti di software possono creare un sistema ben organizzato, modulare e manutenibile che raggiunga il giusto equilibrio tra granularità e coesione, portando a un sistema software efficiente e scalabile.

4.3 Progettazione dell'interfaccia: Best practices

Come già presentato, la progettazione dell'interfaccia è un aspetto critico della progettazione del software che comporta la creazione di una descrizione formale delle interfacce per i moduli o i componenti di un sistema. È opportuno prendere in considerazione le seguenti best practice:

- **Coerenza con i principi di suddivisione:** Le interfacce devono essere progettate in modo da essere in linea con i principi di suddivisione del sistema. Ciò include l'**esposizione solo delle informazioni necessarie al client del servizio, secondo il principio dell'information hiding**. Le interfacce devono inoltre **avere un'elevata coesione interna**, il che significa che le funzionalità offerte dall'interfaccia devono essere strettamente correlate e limitare il numero di interazioni necessarie per accedere ai suoi servizi, riducendo così al minimo l'accoppiamento tra i componenti. Se l'interfaccia è **orientata agli oggetti**, si dovrebbe adottare l'**ereditarietà per promuovere la riusabilità**.
- **Prevedere i casi particolari:** Le interfacce devono prevedere tutti i possibili casi particolari, comprese le eccezioni. Questa pratica aiuta il debug e contribuisce alla costruzione di un sistema robusto e sicuro. Anticipando le potenziali eccezioni e gestendole con grazia, il sistema può essere meglio preparato a gestire situazioni inaspettate.
- **Preferire i metodi idempotenti:** I metodi idempotenti sono quelli che producono lo stesso risultato indipendentemente dal numero di volte in cui vengono eseguiti. Progettare interfacce con metodi idempotenti può aiutare a limitare l'incertezza sui risultati delle chiamate distribuite, migliorando l'affidabilità e la coerenza del sistema.
- **Considerazione delle prestazioni:** La progettazione dell'interfaccia deve tenere conto delle prestazioni. Limitare il numero di interazioni tra i componenti può migliorare le prestazioni, ma è necessario considerare anche la dimensione dei messaggi trasmessi attraverso l'interfaccia. La frammentazione, il buffering e il tempo di elaborazione possono influire negativamente sulle prestazioni, pertanto le interfacce devono essere progettate tenendo conto di questi fattori.
- **Trasmissione selettiva dei dati:** Le interfacce dovrebbero consentire ai client di selezionare quali dati inviare/ricevere e in che quantità, per evitare di trasmettere dati non necessari. Ciò può contribuire a ridurre l'overhead della rete, a migliorare l'efficienza e a minimizzare l'impatto della trasmissione di dati non necessari sulle prestazioni del sistema.

In poche parole, seguendo le migliori pratiche di progettazione delle interfacce presentate sopra, si possono ottenere interfacce ben progettate che promuovono la riusabilità, la robustezza e l'efficienza del sistema software.

4.4 Progettazione dell'interfaccia: Approcci

In questa sede verranno presentati i migliori approcci per la progettazione delle interfacce. Per comprendere più a fondo ogni approccio, esamineremo lo stesso esempio utilizzando l'approccio considerato. Immaginiamo di avere un servizio che facilita l'esecuzione di operazioni su un **conto bancario**. Questo conto bancario è identificato da un **ID conto** e può essere sottoposto a tre operazioni: *lettura* di informazioni sul conto, *deposito* di un importo specifico sul conto e *prelievo* di un importo specifico dal conto. Inoltre, il servizio consente di specificare se l'importo del prelievo richiesto può essere ridotto se l'intero importo non è disponibile sul conto. Sia l'operazione di deposito che quella di prelievo richiedono una descrizione, che è una stringa che può fornire informazioni aggiuntive o un contesto.

4.4.1 Centrato sul metodo

Nell'approccio centrato sul metodo, la progettazione delle interfacce si concentra sulle operazioni e sui loro argomenti di input/output. Segue un approccio simile a quello dei linguaggi di programmazione classici, in cui viene utilizzato un punto finale fisso. Questo approccio è comunemente utilizzato nei paradigmi tradizionali di programmazione procedurale o orientata agli oggetti. L'esempio descritto in precedenza può essere facilmente formalizzato nel modo seguente, secondo l'approccio centrato sul metodo:

```
1 public interface AccountReader {
2     public Account getAccountInfo(String id)
3         throws UnknownAccountIdException;
4 }
5
6 public interface ImprovedAccountUpdater {
7     public void addDeposit(
8         String id,
9         float amount,
10        String description,
11        String transId)
12        throws ReplicatedTransIdException,
13        UnknownAccountIdException;
14
15    public float addWithdrawal(
16        String id,
17        float requestedAmount,
18        boolean reducibile,
19        String description,
20        String transId)
21        throws ReplicatedTransIdException,
22        UnknownAccountIdException,
23        NoAvailabilityException;
24 }
```

4.4.2 Centralità del messaggio

Nell'approccio incentrato sui messaggi, la progettazione delle interfacce è incentrata sui messaggi e sugli endpoint. Si utilizza un'interfaccia fissa, a funzionamento singolo, come "send(Message)", in cui la progettazione è incentrata sullo scambio di messaggi tra componenti o servizi. Questo approccio è comunemente usato nelle architetture di messaggistica o guidate dagli eventi, in cui i componenti comunicano in modo asincrono attraverso i messaggi.

4.4.3 Constrained

L'approccio constrained è una via di mezzo tra l'approccio centrato sui metodi e quello centrato sui messaggi. Comporta un'interfaccia fissa a operazioni multiple ed è comunemente usato in stili architettonici come *Representational State Transfer (REST)*. In questo approccio, la progettazione si

concentra sui punti finali, sull'allocazione delle operazioni e sui loro argomenti di ingresso/uscita, pur mantenendo un certo livello di vincoli sulle operazioni che possono essere eseguite. Questo approccio offre un equilibrio tra flessibilità e struttura, consentendo interazioni vincolate tra componenti o servizi.

4.5 Problemi e buone pratiche specifiche di REST

La progettazione di una buona API RESTful va oltre la semplice considerazione dei principi generali e il rispetto dei vincoli REST. Ci sono altri aspetti importanti da considerare per garantire una progettazione API RESTful efficace.

- **Progettazione/organizzazione delle risorse:** Occorre prestare attenzione al modo in cui le risorse sono progettate e organizzate nell'API RESTful. Le risorse devono essere identificate e rappresentate in modo logico e intuitivo, seguendo una convenzione di denominazione e una struttura di URL coerente. Gli endpoint delle risorse devono essere progettati in modo semplice e facilmente comprensibile e devono rappresentare lo stato della risorsa.
- **Mappatura delle operazioni concettuali con i metodi:** La mappatura delle operazioni concettuali con i metodi HTTP (ad esempio, GET, POST, PUT, DELETE) deve essere fatta con attenzione. Le operazioni che possono essere eseguite su una risorsa devono essere allineate con i metodi HTTP standard e l'uso di questi metodi deve essere conforme alla loro semantica. Ciò garantisce che l'API sia intuitiva, facile da usare e che segua i principi RESTful.
- **Documentazione e autodescrizione dell'API:** Una documentazione adeguata e le capacità di autodescrizione sono fondamentali per un'API RESTful. La documentazione deve essere completa, chiara e aggiornata e deve fornire informazioni su come utilizzare l'API, le risorse disponibili, gli endpoint e le operazioni. Le funzionalità di autodescrizione delle API, come i controlli Hypermedia, possono essere utilizzate per fornire informazioni sulle funzionalità dell'API, sulle azioni disponibili e sui collegamenti alle risorse correlate.
- **Seguire i modelli di progettazione:** Esistono modelli di progettazione e best practice consolidati per la progettazione di API RESTful, come il modello HATEOAS (Hypermedia as the Engine of Application State), che consente ai clienti di navigare nell'API e di scoprire le risorse e le azioni disponibili in modo dinamico. Seguire questi modelli di progettazione può migliorare la manutenibilità, la scalabilità e l'evolvibilità dell'API.
- **Verificare la conformità a REST:** Non tutte le API etichettate come "API REST" possono aderire pienamente ai principi RESTful. È importante verificare e assicurare che la progettazione dell'API segua rigorosamente i principi e i vincoli dell'architettura REST per ottenere i vantaggi desiderati di un'API RESTful.

4.6 Mappare le risorse in URI

Quando si mappano le risorse agli URI nella progettazione delle API, si possono usare delle convenzioni per rendere l'API facile da capire e autodescrittiva. Ad esempio, l'uso di **nomi plurali per le collezioni** e di **nomi singolari per le non collezioni** può aiutare a mantenere la coerenza. Inoltre, l'uso di **URL che riflettono le relazioni gerarchiche** tra le risorse può migliorare l'intuitività dell'API.

Tuttavia, secondo il principio HATEOAS (Hypermedia as the Engine of Application State), gli URL dovrebbero essere "**opachi**", il che significa che i client non dovrebbero basarsi sulla struttura degli URL per la navigazione, ma dovrebbero invece seguire i link forniti nelle risposte per scoprire e interagire con le risorse. Questo approccio consente una maggiore flessibilità nell'evoluzione dell'API senza interrompere le implementazioni dei client che si basano su modelli specifici di URL.

Nella comunità dei progettisti di API sono in corso dibattiti e pareri discordanti sull'opportunità che gli URL siano opachi o seguano delle convenzioni. Alcuni sostengono che l'uso di convenzioni negli URL può aiutare la scopribilità e la facilità d'uso delle API, mentre altri sostengono la stretta aderenza al principio HATEOAS e l'uso di URL opachi per promuovere la flessibilità e il disaccoppiamento tra client e server. In definitiva, la scelta tra l'uso di convenzioni o di URL opachi nella progettazione delle API dipende dai requisiti e dagli obiettivi specifici dell'API e dalle preferenze del team di progettazione.

Una volta mappate le risorse, è possibile mappare anche le operazioni disponibili su di esse. Quando si definiscono le operazioni in un servizio, in genere si seguono le seguenti fasi:

- Identificare le risorse e i metodi HTTP da utilizzare per ogni operazione.
- Controlla le corrispondenze tra risorse e metodi. Se non si trovano corrispondenze, ciò può indicare che mancano alcune risorse o che devono essere aggiunte.

Inoltre, per ogni coppia risorsa/metodo consentita, vengono definiti i seguenti dettagli:

- Come il metodo deve essere invocato, compresi i parametri di query accettati, le intestazioni della richiesta e il contenuto del corpo.
- I possibili risultati, compresi i codici di stato che possono essere restituiti, e le intestazioni e i contenuti del corpo della risposta corrispondenti per ciascun codice di stato.

Questa mappatura delle operazioni alle risorse, ai metodi e ai dettagli ad essi associati aiuta a progettare un'API ben strutturata e autodescrittiva, rendendo più facile per i client capire come interagire con il servizio. Di seguito viene fornito un esempio che considera uno schema di bibliografia e questo tipo di richiesta:

"Cerca articoli per parola chiave, tipo (articolo/libro) e anno di pubblicazione".

Resource	Verb	Query params	Status		Resp. Body
biblio/items	GET	keyword: string type: "article" "book" beforeInclusive:date afterInclusive:date	200	OK	filtered items (items)
biblio/items/{id}	GET		200	OK	item
			404	Not found	

Figura 22: Esempio di operazioni di mappatura

È anche possibile avere e mappare altri tipi di operazioni, non solo quelle *READ*:

Resource	Verb	Req. body	Status		Resp. body
biblio/items	POST	item	201	Created	item
			400	Bad Request	string
biblio/items/{id}	PUT	item	204	No Content	
			400	Bad Request	string
			404	Not Found	
biblio/items/{id}	DELETE		204	No Content	
			404	Not Found	

Figura 23: Esempio di ulteriori operazioni di mappatura

4.7 Considerazioni sull'efficienza

Oltre alla semplice mappatura delle richieste, è importante anche fare delle considerazioni su cosa si può fare e quali sono le migliori pratiche per aumentare le prestazioni durante le richieste e le risposte. Più precisamente, occorre considerare la dimensione delle risposte che possono essere restituite da un servizio. Ecco alcune buone pratiche comuni per evitare messaggi di grandi dimensioni:

- **Parametri di interrogazione:** Invece di restituire tutti i dati in un'unica risposta, consentite agli utenti di specificare i parametri della query per recuperare solo i dati di cui hanno bisogno. Ciò può includere il filtraggio, l'ordinamento o la limitazione del numero di risultati.
- **Paginazione:** Quando si ha a che fare con grandi raccolte di dati, si può considerare l'uso della paginazione per recuperare i dati in pezzi più piccoli, anziché restituirli tutti in una volta. Ciò può contribuire a ridurre le dimensioni della risposta e a migliorare le prestazioni.
- **Utilizzo di riferimenti:** Invece di includere le rappresentazioni complete delle risorse correlate nella risposta, utilizzare riferimenti o collegamenti per indicare la posizione delle risorse correlate. I client possono quindi recuperare la rappresentazione completa delle risorse correlate solo quando necessario, riducendo le dimensioni della risposta iniziale.

Inoltre, dal lato delle richieste, per ottimizzare la larghezza di banda della rete e ridurre l'elaborazione non necessaria sul lato server, è possibile sfruttare le richieste condizionali HTTP. Esse consentono ai client di specificare che un'operazione deve essere eseguita solo in determinate condizioni. Ecco alcune intestazioni comuni utilizzate per le richieste condizionali:

- **If-Modified-Since:** Questa intestazione specifica la data e l'ora dell'ultima modifica della risorsa. Il server eseguirà l'operazione richiesta solo se la risorsa è stata modificata dopo il valore specificato in questa intestazione.
- **If-Unmodified-Since:** Questa intestazione specifica la data e l'ora dell'ultima modifica della risorsa. Il server eseguirà l'operazione richiesta solo se la risorsa non è stata modificata dal valore specificato in questa intestazione.
- **If-Match:** Questa intestazione specifica il tag di entità (ETag) della **risorsa**. Il server eseguirà l'operazione richiesta solo se l'**ETag** della risorsa corrisponde a uno dei valori elencati in questa intestazione.
- **If-None-Match:** Questa intestazione specifica il tag di entità (ETag) della **risorsa**. Il server eseguirà l'operazione richiesta solo se l'**ETag** della risorsa non corrisponde a nessuno dei valori elencati in questa intestazione.

Per quanto riguarda le richieste condizionali, di seguito vengono presentati tre importanti casi d'uso:

- **Gestione della cache del client:** è possibile inviare nuovamente la risorsa (una sua rappresentazione) solo se modificata dall'ultima volta che è stata messa in cache.

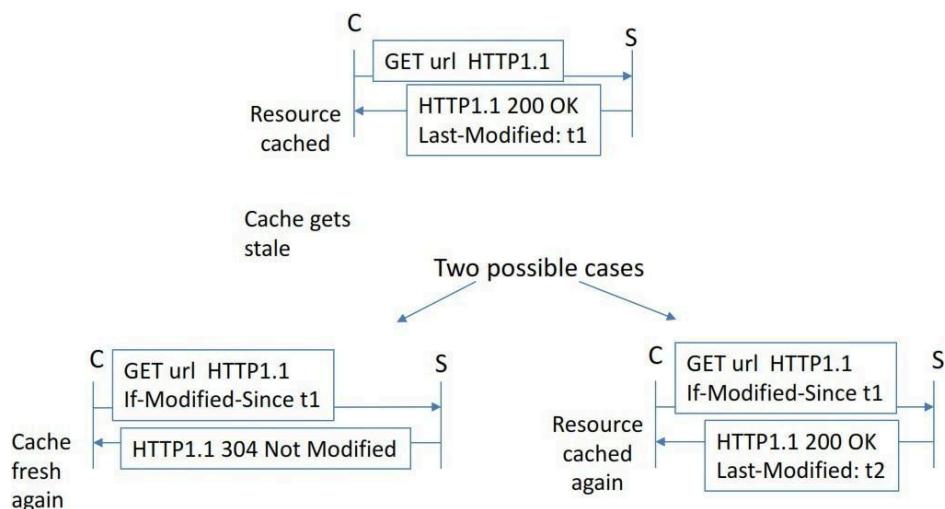


Figura 24: Richieste condizionate: Caso d'uso della cache

- **Lost Update Race Condition:** quando più client lavorano sulla stessa risorsa (aggiornandola), può accadere che uno dei due aggiornamenti venga perso, poiché l'altro client non ha riletto la risorsa.

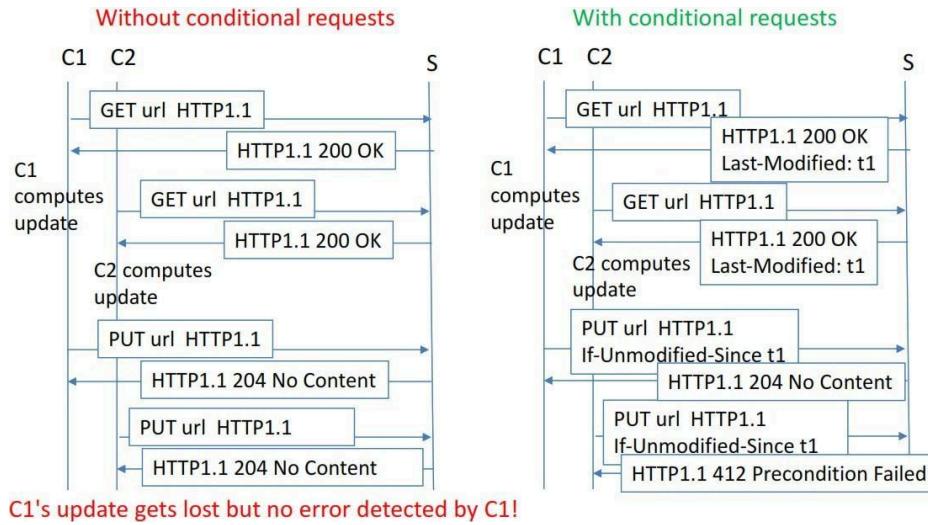


Figura 25: Richieste condizionate: Condizione di gara per l'aggiornamento della lettura

- **Race Condition di Put (Creazione):** quando il metodo put è usato da più client per creare o aggiornare una risorsa simultaneamente, potrebbe accadere che un primo client crei la risorsa e subito dopo il secondo la sovrascriva, senza alcuna condizione preliminare.

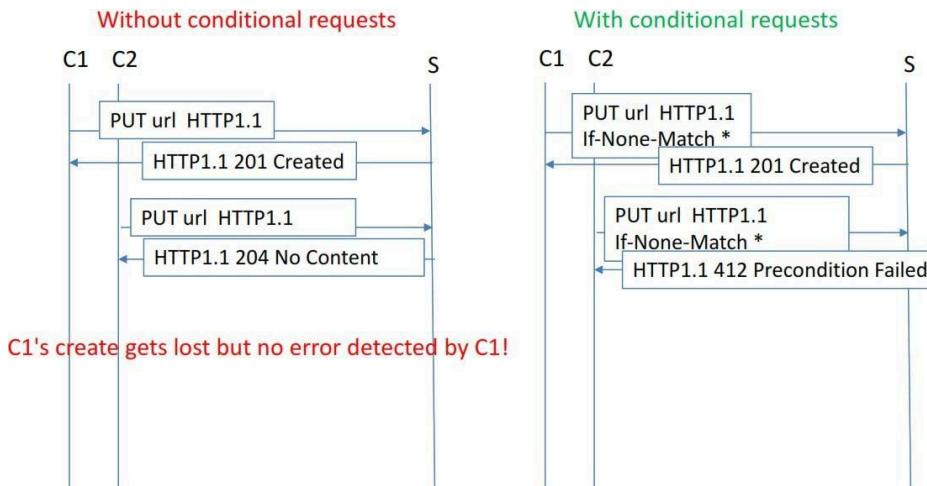


Figura 26: Richieste condizionali: Condizione di gara Put(Create)

5. Chiamata di procedura remota (RPC)

La **chiamata di procedura remota** (RPC) è un **protocollo di comunicazione** che consente a un **programma** di eseguire una **procedura** o una **funzione** in uno spazio di indirizzi diverso, in genere su un computer o un server remoto, come se si trattasse di una chiamata di procedura locale. Consente a un sistema distribuito di invocare funzioni o metodi tra diversi processi o macchine, fornendo ai programmi un modo per interagire e comunicare tra loro attraverso una rete.

Il concetto di RPC si basa sull'idea di astrarre il processo di invocazione di una procedura remota per renderlo trasparente al programma chiamante. Dal punto di vista del programma client, l'invocazione di una procedura remota tramite RPC appare simile alla chiamata di una procedura o funzione locale. Il programma client non deve essere a conoscenza dei dettagli della comunicazione di rete sottostante o della posizione della procedura remota. In pratica, si tratta di un'architettura client-server, in cui il client richiede un'operazione mentre il server la esegue.

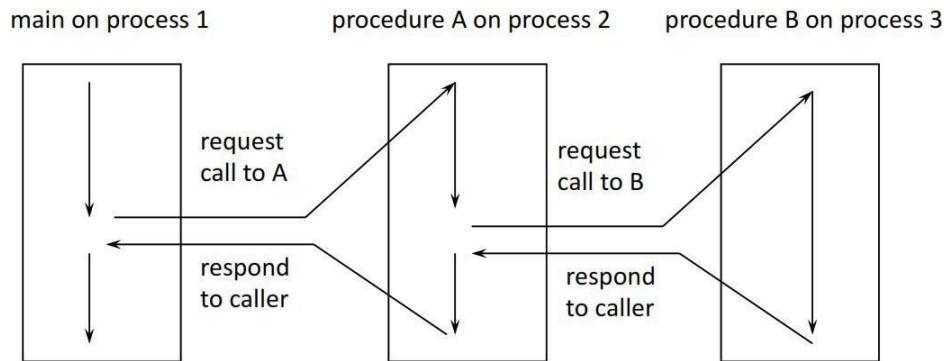


Figura 27: RPC: Chiamata di procedura remota

Ecco una panoramica di alto livello sul funzionamento di RPC:

1. **Invocazione del client**: Il programma client effettua una chiamata di procedura come se stesse chiamando una procedura locale. I parametri della chiamata di procedura vengono marshallizzati (serializzati) in un formato adatto alla trasmissione in rete.
2. **Stub client**: Il sistema di runtime RPC sul lato client intercetta la chiamata di procedura e smista i parametri in un messaggio. Quindi invia il messaggio in rete al server remoto.
3. **Comunicazione di rete**: Il sistema runtime RPC lato client utilizza un protocollo di rete, come TCP/IP, per trasmettere il messaggio contenente la chiamata di procedura al server remoto.
4. **Stub del server (chiamato anche skeleton)**: Il sistema runtime RPC lato server riceve il messaggio dalla rete, estrae le informazioni sulla chiamata di procedura e invoca la procedura o la funzione del server corrispondente.
5. **Esecuzione del server**: La procedura o la funzione del server esegue l'operazione richiesta utilizzando i parametri forniti.
6. **Risposta del server**: Dopo aver eseguito la procedura, il server restituisce i risultati (se presenti) al client.
7. **Comunicazione di rete**: Il sistema di runtime RPC sul lato server raccoglie i risultati in un messaggio e lo invia al client attraverso la rete.
8. **Stub client**: Il sistema runtime RPC lato client riceve il messaggio dal server, estrae i risultati e li restituisce al chiamante originale.

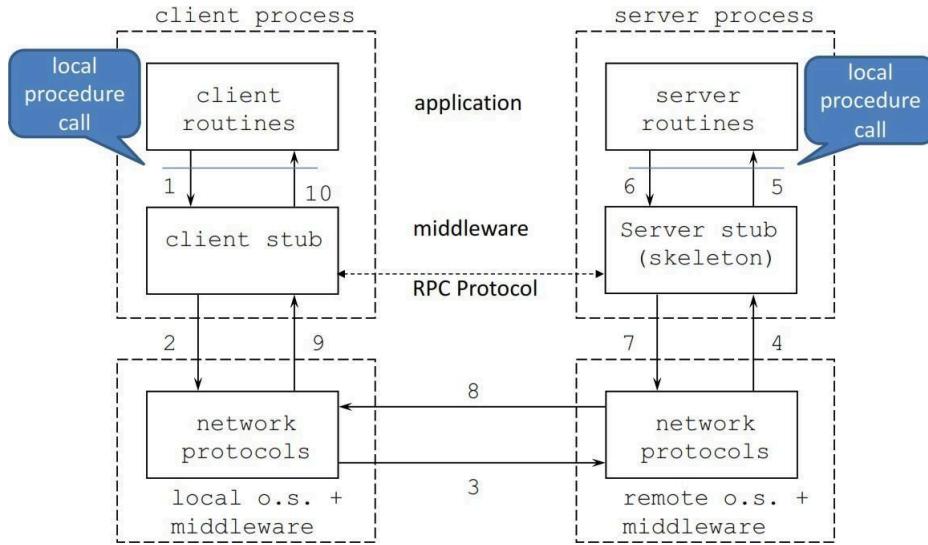


Figura 28: RPC: Modello di chiamata di procedura remota

L'aspetto chiave di RPC è la **trasparenza** che offre al programma chiamante. I programmi client e server possono essere **scritti in linguaggi di programmazione diversi** e possono essere eseguiti su **sistemi operativi o architetture hardware differenti**. L'RPC astrae le complessità della comunicazione di rete, della serializzazione/deserializzazione e della gestione dei protocolli, consentendo agli sviluppatori di concentrarsi sugli aspetti funzionali dell'applicazione distribuita. Naturalmente, bisogna tenere conto del fatto che **non è possibile ottenere la piena trasparenza della distribuzione** (*si veda la sottosezione 1.2.1*); ciò è facilmente comprensibile se si considera che il **tempo di esecuzione non è lo stesso di un sistema con una procedura locale** e che, inoltre, può accadere che il **client debba compiere un'azione senza aver ricevuto una risposta** (fallimento); in questo senso, una procedura locale non può fallire.

L'RPC è ampiamente utilizzato nei sistemi distribuiti, nelle architetture client-server e nei servizi Web per consentire la comunicazione e l'interazione senza soluzione di continuità tra diversi componenti o servizi. Semplifica lo sviluppo di applicazioni distribuite fornendo un modello di programmazione familiare e nascondendo i dettagli di rete di basso livello. I framework e le tecnologie RPC più diffusi includono gRPC, Apache Thrift, CORBA e Java RMI (Remote Method Invocation).

5.1 Il middleware RPC

Esistono due modi per implementare RPC:

1. **Supporto basato sul linguaggio:** In questo approccio, l'RPC è offerto come caratteristica di un linguaggio di programmazione, come **Java RMI** (Remote Method Invocation). Sfrutta le tecniche di serializzazione specifiche del linguaggio per facilitare la comunicazione tra client e server. Questo approccio offre un'elevata trasparenza, nel senso che l'invocazione di una procedura remota appare simile a una chiamata di procedura locale. Tuttavia, è spesso **limitato alle applicazioni scritte in un unico linguaggio**.
2. **Generazione di stub:** In questo approccio, il middleware RPC genera stub basati su un **linguaggio di definizione delle interfacce** (IDL, Interface Definition Language). **La definizione dell'interfaccia descrive i metodi e le loro firme che possono essere invocati in remoto**. A partire dall'IDL, il middleware RPC genera stub client e stub server. Questi stub fungono da proxy rispettivamente per il client e il server e gestiscono il marshalling e il unmarshaling degli argomenti delle procedure e dei valori di ritorno. Questo approccio consente di utilizzare RPC in diversi linguaggi di programmazione.

Quando si implementa un sistema RPC, il middleware deve affrontare diverse sfide:

- **Eterogeneità:** il chiamante e il chiamato possono essere sviluppati con linguaggi di programmazione diversi, funzionare su piattaforme hardware o software diverse e avere rappresentazioni di dati diverse. Il middleware RPC deve gestire la conversione (marshaling/unmarshaling) degli argomenti delle procedure e dei valori di ritorno per garantire la compatibilità.

- **Comunicazione client-server:** Il client e il server vengono in genere eseguiti come processi separati su host diversi. La **posizione del server deve essere determinata** prima di emettere la chiamata alla procedura remota, spesso attraverso il **linking dinamico**. Inoltre, il middleware RPC deve gestire la sincronizzazione di più chiamate in arrivo sul lato server.
- **Passaggio di argomenti per riferimento:** Il middleware RPC deve definire il modo in cui gli argomenti possono essere passati per riferimento, considerando che gli spazi degli indirizzi del client e del server sono solitamente disgiunti. Questo include la gestione dell'allocazione e della deallocazione della memoria per i dati referenziati.
- **Fallimento parziale:** In un ambiente distribuito, è possibile che si verifichino guasti durante il processo RPC. Il middleware RPC deve fornire meccanismi per rilevare e gestire i guasti parziali, assicurando che il sistema rimanga affidabile e resiliente.

5.2 Gestione dei fallimenti parziali (semantica delle chiamate remote)

Quando si effettuano chiamate di procedura remota, la semantica della chiamata differisce da quella delle chiamate locali a causa della natura distribuita del sistema. Le chiamate remote possono incontrare diversi problemi, tra cui **crash del server**, **interruzioni di rete** e **duplicazioni di messaggi**. Per gestire questi problemi, i protocolli middleware RPC forniscono meccanismi per affrontare i guasti parziali. Tuttavia, è importante che i programmatori siano consapevoli di questi problemi e li gestiscano di conseguenza.

Nel contesto delle chiamate remote, sono possibili due tipi di ritorno:

1. **Chiamata riuscita:** La chiamata remota viene completata con successo e il risultato o la risposta previsti vengono restituiti al client. Ciò indica che la procedura remota è stata eseguita senza errori o eccezioni.
2. **Chiamata non riuscita:** La chiamata remota incontra un errore o un fallimento durante l'esecuzione. Ciò può essere dovuto a vari motivi, quali crash del server, problemi di rete o eccezioni lanciate durante l'esecuzione della procedura. In questi casi, il client può ricevere una risposta di errore o un'eccezione che indica il fallimento della chiamata alla procedura remota.

È importante che i **programmatori gestiscano in modo appropriato le chiamate remote**, sia quelle riuscite che quelle non riuscite, nel loro codice applicativo. Devono gestire le eccezioni, convalidare i risultati restituiti e implementare strategie appropriate di gestione degli errori per garantire la correttezza e la robustezza del sistema distribuito. Quando si effettuano chiamate a procedure remote, la semantica della chiamata differisce da quella delle chiamate locali a causa della natura distribuita del sistema. Le chiamate remote possono incontrare diversi problemi, tra cui crash del server, interruzioni di rete e duplicazioni di messaggi. I protocolli middleware RPC possono gestire questi problemi solo parzialmente e i programmatori devono esserne consapevoli.

A seconda del protocollo RPC utilizzato, si possono ottenere diverse **semantiche di chiamata**:

- **At Least Once:** con la semantica almeno una volta, il protocollo RPC garantisce che la **chiamata di procedura remota** venga eseguita **almeno una volta**. In caso di fallimento, il **client riprova la chiamata finché non riceve una risposta positiva**. Questa semantica garantisce che la richiesta venga elaborata, ma può portare a una **duplicazione dell'esecuzione** della procedura se la risposta viene persa o ritardata. In questo caso, le **procedure devono essere idempotenti** (cioè non sensibili al numero di chiamate), in modo che, se eseguite più volte, possano fornire un risultato idempotente. In questo caso il protocollo del livello di trasporto è **UDP** e il protocollo RPC incorpora meccanismi per associare le risposte alle richieste e gestire le ritrasmissioni.
- **At Most Once:** la semantica At Most Once garantisce che la **chiamata di procedura remota venga eseguita al massimo una volta**. Il protocollo RPC adotta misure per garantire l'eliminazione delle richieste duplicate. Questo si ottiene tipicamente assegnando un **identificatore unico a ogni richiesta** e facendo in modo che il server mantenga un **registro delle richieste elaborate** per individuare i duplicati. Se una richiesta viene persa o fallisce, il client può riemettere la richiesta senza preoccuparsi dei duplicati. In questo caso, è possibile utilizzare il protocollo di trasporto **TCP** e il protocollo RPC associa la **risposta alla richiesta attraverso la connessione**. Altrimenti, nel trasporto **UDP**, il protocollo RPC include l'**associazione risposta-richiesta**, la **gestione della ritrasmissione** e il **controllo dei duplicati sul server**.

- **Exactly Once:** la semantica Exactly Once mira a garantire che la **chiamata di procedura remota venga eseguita esattamente una volta**. Questa è la **garanzia più forte** tra le tre semantiche. Raggiungere la semantica "exactly once" può essere impegnativo, in quanto richiede un'attenta coordinazione tra il client e il server. Per ottenere la semantica "exactly once" si possono usare vari meccanismi, come la **sequenza dei messaggi**, il **rilevamento dei duplicati** e le **operazioni idempotenti**.

La scelta della semantica delle chiamate dipende dai requisiti del sistema distribuito e dai compromessi tra affidabilità, prestazioni e semplicità.

5.3 Altri problemi RPC

Quando si parla di RPC, ci sono principalmente altri due campi di problemi da affrontare:

- **Sicurezza:** I principali requisiti di sicurezza in RPC includono il controllo degli accessi, l'autenticazione, la riservatezza e l'integrità.
- **Prestazioni:** Un problema significativo delle prestazioni in RPC è che una chiamata remota è molto più lenta rispetto a una chiamata locale, spesso di ordini di grandezza.

5.4 Diversi tipi di meccanismi di richiesta/risposta RPC

Esistono diversi tipi di meccanismi RPC in termini di interazione richiesta/risposta.

5.4.1 RPC classico (sincrono)

Interazione di base richiesta/risposta in cui un client invia una richiesta a un server e si mette in attesa di una risposta. La richiesta è composta da:

- **Identificazione della procedura**
- **Parametri di ingresso**

Mentre la risposta inviata dal server:

- **Indicazione di successo/fallimento**
- **Valore di ritorno** e parametri di uscita

5.4.2 RPC a una via (asincrono)

Questo è il caso in cui il client inizia l'interazione inviando una richiesta, ma non aspetta una risposta. In questo caso siamo sicuri che non verranno lanciate eccezioni e che l'operazione eseguita dal server andrà a buon fine (o almeno ci contiamo). L'interazione termina con l'invio della richiesta.

5.4.3 RPC a due vie (asincrono)

È simile all'RPC sincrono, ma il client non si blocca in attesa della risposta, in modo che il client principale sia ancora in grado di eseguire altre azioni. La gestione della risposta è delegata a un meccanismo separato, ad esempio un thread dedicato o una callback.

5.4.4 RPC with Streaming

In RPC, il modello di streaming può essere applicato sia al lato della richiesta che al lato della risposta. Questo significa che un messaggio non è più un'entità singola, ma una sequenza di messaggi trasmessi tra client e server. Esistono quattro modalità principali:

1. **Server-Side Streaming**
 - Il client invia una singola richiesta al server.
 - Il server restituisce una sequenza (stream) di messaggi in risposta.
 - Esempio: invio di notifiche o aggiornamenti in tempo reale.
2. **Client-Side Streaming**
 - Il client invia una sequenza di messaggi come richiesta.

- Il server elabora lo stream di messaggi e restituisce una singola risposta quando ha terminato.
 - Esempio: caricamento di dati batch.
3. **Bidirectional Streaming**
- Sia il client che il server inviano sequenze di messaggi in modo simultaneo e indipendente.
 - I messaggi possono essere scambiati in entrambe le direzioni finché lo stream è attivo.
 - Esempio: applicazioni di chat o video streaming interattivi.

5.5 In cima alle API Socket o RPC?

Dobbiamo riflettere su cosa succede se ci affidiamo solo a TCP come protocollo di trasporto invece di affidarci a un middleware RPC per un'applicazione distribuita. Fondamentalmente, i socket forniscono solo servizi di comunicazione, quindi in questo caso dobbiamo risolvere molti altri problemi nella nostra applicazione, come la codifica dei dati scambiati, le procedure di interazione, la gestione dei fallimenti parziali e così via, con conseguente aumento della complessità. In genere l'idea è quella di sviluppare middleware in cima alle API Socket o a protocolli personalizzati a livello di applicazione. D'altra parte, sviluppare l'applicazione su RPC permette di svilupparla come normale codice centralizzato, ma tenendo presente che le interazioni saranno distribuite. I moduli dell'applicazione sono distribuiti su più host e l'applicazione viene testata nell'ambiente distribuito; in questo caso il programmatore può concentrarsi più sulla logica dell'applicazione che sulla comunicazione. Un altro punto chiave importante da sottolineare è che le applicazioni possono essere viste come una combinazione di servizi. I servizi sono progettati e sviluppati come componenti riutilizzabili che interagiscono tramite RPC, il che comporta molti vantaggi: i servizi sono semplici componenti riutilizzabili, lo scaling out può essere ottenuto creando istanze multiple di servizi che si basano sul volume delle richieste.

6. HTTP/2

Prima di immergersi in gRPC, è importante capire i fondamenti del protocollo HTTP/2. Introdotto inizialmente da Google e successivamente standardizzato dall'IETF, HTTP/2 mantiene la stessa semantica di HTTP/1.1 (RESTful), ma migliora l'efficienza e le prestazioni grazie a una diversa codifica dei dati e all'utilizzo di TCP. Una caratteristica notevole è il push non richiesto delle risorse, che consente al server di inviare risorse al client senza richieste esplicite. Questo meccanismo migliora notevolmente l'efficienza complessiva e la reattività della comunicazione web.

6.1 HTTP/2 vs. HTTP/1.1

Le differenze tra HTTP/2 e HTTP/1.1 sono numerose. In particolare, le differenze principali riguardano: la codifica dei dati, il multiplexing, la compressione e la possibilità di push.

	HTTP/2	HTTP/1.1
Codifica dei dati	Binario	Orientamento al personaggio
Multiplexing	Su una singola connessione TCP	Su più connessioni TCP
Compression	Intestazione e corpo	Solo corpo
Push Possibility	Sì	No

Quando si parla di scambio di messaggi binari in HTTP/2, entrano in gioco due concetti importanti: **frame e stream**. Un **frame rappresenta un messaggio binario scambiato tra un client e un server**. Un **flusso**, invece, **si riferisce a una sequenza indipendente e bidirezionale di frame**. Ogni flusso corrisponde a una sequenza ordinata di operazioni logiche, come Request/Response o Push, che sono essenzialmente messaggi HTTP.

In HTTP/2, i **flussi sono multiplexati su una singola connessione TCP**, il che significa che **più flussi possono essere trasmessi simultaneamente**. L'ordine dei frame all'interno di ciascun flusso viene preservato, garantendo l'integrità del flusso di messaggi. Inoltre, vengono introdotti meccanismi di controllo del flusso basati su finestre per regolare la velocità di trasmissione dei dati per ogni flusso.

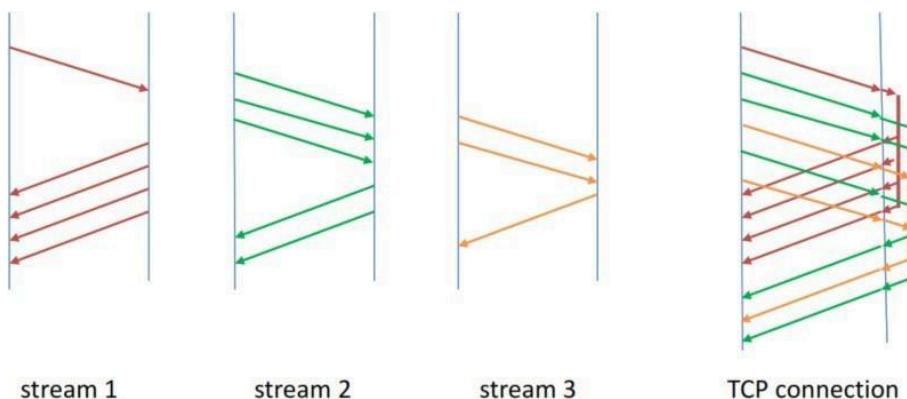


Figura 29: HTTP/2: Multiplexing del flusso

Una **caratteristica notevole di HTTP/2 è il push del server**. Ciò consente al server di inviare proattivamente rappresentazioni aggiuntive al client, anticipando le sue richieste future. Questo approccio aiuta a ridurre la latenza scambiando l'uso della larghezza di banda. Ad esempio, quando una pagina web contiene collegamenti che fanno riferimento ad altre risorse, invece di aspettare che il client richieda ciascuna risorsa individualmente, il server può inviare queste risorse al client in modo preventivo.

Ecco perché HTTP/2 è preferito:

1. **Miglioramento della prestazione:**
 - o **Multiplexing:** HTTP/2 permette il multiplexing, cioè la possibilità di inviare più richieste e risposte

contemporaneamente su una singola connessione TCP, quindi lo si apre una singola volta e si risparmia tempo a causa della continua apertura e chiusura della connessione. Questo elimina il problema di "head-of-line blocking" che si verifica in HTTP/1.1, dove ogni richiesta deve essere completata prima che la successiva possa iniziare.

- **Compressione delle intestazioni:** HTTP/2 usa una compressione delle intestazioni delle richieste e delle risposte, riducendo la quantità di dati trasmessi.
 - **Prioritizzazione delle richieste:** HTTP/2 consente di dare priorità ad alcune richieste, migliorando la gestione delle risorse di rete, specialmente quando sono coinvolti più flussi di dati.
2. **Efficienza nella comunicazione binaria:**
- **gRPC** (un popolare framework RPC) si basa su HTTP/2 per la trasmissione dei dati. Utilizza il formato binario di **Protocol Buffers** (Protobuf) per la serializzazione dei dati, che è molto più efficiente rispetto al formato JSON usato in REST. Questo rende gRPC estremamente performante rispetto a soluzioni basate su HTTP/1.1.
3. **Connessioni persistenti:**
- Con HTTP/2, le connessioni TCP sono riutilizzate per più richieste, riducendo il sovraccarico dovuto all'apertura di nuove connessioni per ogni richiesta, come accadeva in HTTP/1.1 (dove era necessario stabilire una nuova connessione per ogni richiesta o ogni "keep-alive").
4. **Streaming bidirezionale:**
- HTTP/2 consente lo streaming bidirezionale di dati, un aspetto importante per applicazioni che richiedono una comunicazione continua tra client e server, come nel caso di gRPC, che supporta **streaming** sia lato client che lato server.
5. **Server Push:**
- In HTTP/2, una delle funzionalità avanzate che consente il push del server è il **Server Push**. La caratteristica di "server push" permette al server di inviare risorse aggiuntive al client in anticipo, cioè prima che il client le richieda esplicitamente. Ciò può migliorare notevolmente la latenza e le prestazioni, specialmente in contesti di applicazioni web, riducendo il numero di round-trip necessari per caricare tutte le risorse di una pagina.

Differenze con HTTP/1.1:

- **HTTP/1.1** è meno performante in scenari ad alta latenza e a basso throughput, poiché non supporta il multiplexing delle richieste su una singola connessione, e la gestione delle intestazioni è meno efficiente. Quindi, mentre HTTP/1.1 può ancora essere utilizzato per alcune applicazioni, non è ideale per architetture moderne basate su microservizi e RPC, dove le prestazioni e la scalabilità sono cruciali.

Early Hints

Early Hints è una funzionalità introdotta con **HTTP/2** e **HTTP/3** per migliorare la velocità di caricamento delle pagine web, in particolare nelle architetture moderne basate su **server** e **client** che richiedono molte risorse esterne (come CSS, JavaScript e immagini). L'idea alla base di **Early Hints** è che il server possa inviare al client (ad esempio, un browser web) delle indicazioni sulle risorse da caricare **prima** che la risposta finale completa della richiesta HTTP sia stata inviata. In pratica, **Early Hints** è un codice di stato HTTP (es. **103 Early Hints**) che permette al server di inviare al client delle "**informazioni preliminari**" sulle risorse che probabilmente saranno necessarie per completare una richiesta. Questo avviene **prima** che la risposta finale (con il corpo del contenuto) venga inviata dal server. Questa funzionalità è particolarmente utile per accelerare il **rendering delle pagine** in scenari dove il client deve caricare molte risorse esterne (come immagini, file CSS, JavaScript), poiché il client può iniziare a scaricarle immediatamente, **senza dover aspettare che il server completi la risposta**.

6.2 Avvio, negoziazione e aggiornamento della connessione HTTP/2

Quando il client sa che il server supporta HTTP/2, il processo di comunicazione inizia stabilendo una connessione TCP. Il client avvia la connessione inviando la stringa di prefazione di 24 byte ("PRI * HTTP/2.0\r\nSM\r\n") seguito da un frame SETTINGS. Il server risponde con la sua stringa di prefazione e un frame SETTINGS. I flussi possono quindi essere aperti per ulteriori comunicazioni.

Nei casi in cui il client non ha conoscenze pregresse, è necessaria la negoziazione HTTP/2. Per la negoziazione basata su TLS, comunemente utilizzata nelle connessioni HTTPS, il client può utilizzare il protocollo TLS Application Layer Protocol Negotiation (ALPN). Il client richiede al server uno dei diversi protocolli, tra cui HTTP/2. Il server seleziona un protocollo e lo comunica al server. Il server sceglie un protocollo e lo comunica al client nella risposta. Se viene selezionato HTTP/2, si verifica lo stesso processo di

handshake menzionato in precedenza, con lo scambio di stringhe di prefazione e frame SETTINGS. In alternativa, è possibile utilizzare la negoziazione basata su HTTP. In questo approccio, il client può negoziare HTTP/2 tramite il meccanismo HTTP/1.1 Upgrade. Il client invia una richiesta HTTP/1.1 che include un'intestazione **Upgrade** che indica il desiderio di passare a HTTP/2. Se il server supporta HTTP/2, risponde con una **risposta 101 Switching**. Se il server supporta HTTP/2, risponde con un codice di stato 101 Switching Protocols e include un'intestazione Upgrade che conferma il passaggio a HTTP/2. Anche in questo caso, il successivo processo di handshake avviene come descritto in precedenza. In entrambi i casi, il processo di negoziazione consente al client e al server di stabilire la comunicazione utilizzando il protocollo HTTP/2, garantendo uno scambio di dati efficiente e ottimizzato.

6.3 Gestione degli errori

Oltre ai meccanismi di segnalazione degli errori forniti da HTTP/1 (*segnalazione degli errori a livello di risposta*), HTTP/2 offre ulteriori meccanismi di segnalazione degli errori sia a livello di flusso che di connessione.

- **Errori a livello di flusso:** Questi errori si verificano nella gestione dei singoli flussi. HTTP/2 consente una segnalazione più precisa degli errori relativi a flussi specifici, fornendo informazioni dettagliate sui problemi riscontrati durante l'elaborazione di un particolare flusso.
- **Errori a livello di connessione:** Questi errori si verificano nella gestione della connessione TCP. HTTP/2 include meccanismi per segnalare errori che riguardano l'intera connessione, come ad esempio problemi legati alla creazione o al mantenimento della connessione TCP stessa.

Offrendo la segnalazione degli errori a livello di flusso e a livello di connessione, HTTP/2 migliora la precisione e la granularità della segnalazione degli errori rispetto a HTTP/1. Ciò consente di ottenere informazioni più dettagliate e accurate sulle richieste non completate e sugli errori riscontrati durante la comunicazione tra client e server. Oltre ai meccanismi di gestione degli errori menzionati in precedenza, in HTTP/2 esistono altri modi per conoscere lo stato delle richieste non completate.

- Riquadro GOAWAY: Il frame GOAWAY viene utilizzato per indicare il numero di flusso più alto che il server ha elaborato. Questa informazione è preziosa per i client per determinare la sicurezza di ritentare le richieste. Tutte le richieste inviate su flussi con un numero superiore a quello indicato nel frame GOAWAY possono essere ritentate in modo sicuro, poiché è garantito che non sono state elaborate dal server.
- Codice di errore STREAM RIFIUTATO: Il codice di errore REFUSED STREAM può essere incluso in un frame RST STREAM. Quando un server invia un frame RST STREAM con il codice di errore REFUSED STREAM, indica che lo stream viene chiuso prima che sia avvenuta qualsiasi elaborazione. In questo caso, qualsiasi richiesta inviata sul flusso resettato può essere riproposta senza problemi.

Utilizzando il frame GOAWAY e il codice di errore REFUSED STREAM, HTTP/2 fornisce ai client meccanismi per raccogliere informazioni sullo stato delle richieste non completate. Ciò consente ai client di prendere decisioni informate sull'opportunità di riprovare le richieste o di intraprendere azioni appropriate in base alla risposta del server.

7. gRPC

gRPC è un **meccanismo RPC** (Remote Procedure Call) che opera **sulla base di HTTP/2**. Sfrutta le caratteristiche di HTTP/2 orientate alle prestazioni, come i messaggi binari e la compressione delle intestazioni. Analogamente a SOAP, **gRPC utilizza HTTP/2 come livello di trasporto**. La tipica **rappresentazione dei dati utilizzata in gRPC è costituita dai protocol buffers**, che forniscono un modo **indipendente dal linguaggio per definire la struttura dei messaggi**. Alcune caratteristiche chiave di gRPC sono:

- **Supporto multilingue**: gRPC supporta più linguaggi di programmazione, consentendo a client e server di essere implementati in lingue diverse e di comunicare senza problemi.
- **Semantica "at most once"**: gRPC garantisce che una procedura remota venga eseguita al massimo una volta, garantendo affidabilità e coerenza nella comunicazione tra client e server.
- **Gestione della sicurezza**: gRPC supporta funzioni di sicurezza come Transport Layer Security (TLS), garantendo una comunicazione sicura sulla rete.
- **Molteplici modalità di funzionamento**: gRPC supporta diverse modalità di funzionamento, tra cui la **comunicazione sincrona, asincrona e orientata ai flussi (stream-oriented)**. Questa flessibilità consente agli sviluppatori di scegliere l'approccio più adatto in base ai requisiti dell'applicazione.

Utilizzando i vantaggi di HTTP/2 e dei buffer di protocollo, gRPC offre una comunicazione efficiente e affidabile tra sistemi distribuiti, rendendola una scelta popolare per la costruzione di moderne architetture di microservizi.

7.1 Protocol Buffer: un meccanismo di serializzazione per gRPC

Protocol Buffers è una rappresentazione dei dati indipendente dal sistema e un linguaggio di definizione dell'interfaccia (IDL) che fornisce un linguaggio astratto per definire la struttura dei messaggi. Offre un formato di rappresentazione binario che è più compatto ed efficiente rispetto ad altri formati di interscambio dati come JSON o XML, soprattutto perché HTTP/2 utilizza già la rappresentazione binaria dei messaggi.

Alcune caratteristiche chiave dei buffer di protocollo includono:

- **Rappresentazione indipendente dal sistema**: I buffer di protocollo offrono un modo indipendente dal linguaggio per definire la struttura dei messaggi. Ciò significa che è possibile definire i formati dei messaggi una sola volta e generare binding di codice per tutti i principali linguaggi di programmazione, consentendo una comunicazione senza soluzione di continuità tra sistemi diversi.
- **Rappresentazione binaria**: I buffer di protocollo **utilizzano un formato binario per la serializzazione dei dati**, che consente una codifica e una decodifica più efficiente dei messaggi. Il formato binario compatto contribuisce a ridurre le dimensioni del carico utile e a migliorare le prestazioni della trasmissione dei dati.
- **Binding per i principali linguaggi di programmazione**: I buffer di protocollo offrono binding specifici per tutti i principali linguaggi di programmazione, tra cui Java, C++, Python, Go e altri. Questi binding consentono agli sviluppatori di lavorare facilmente con Protocol Buffers nel loro linguaggio di programmazione preferito e di sfruttare il codice generato per una serializzazione e deserializzazione efficiente dei messaggi.

Offrendo una rappresentazione dei dati indipendente dal sistema e fornendo binding specifici per il linguaggio, i buffer Protocol semplificano lo sviluppo di applicazioni multiplataforma e interoperabili. Il suo formato binario compatto e l'efficiente generazione di codice ne fanno una scelta popolare per la strutturazione e la serializzazione dei dati in vari ambiti, dalle architetture a microservizi ai sistemi distribuiti. A questo punto, diamo un'occhiata ai tipi di messaggio: in gRPC, i **tipi di messaggio sono strutture di dati** definite dall'utente che possono essere utilizzate per RPC. Questi tipi di messaggio consistono in una collezione di campi tipizzati, **denominati e numerati**. Ogni campo può essere opzionale e singolare (permettendo 0 o 1 ripetizioni) o opzionale e ripetuto (permettendo 0 o più ripetizioni).

Ad esempio, si consideri il tipo di messaggio "Libro" che ha i campi "isbn", "titolo" e "autori". Il campo "isbn" ha il numero 1, il campo "titolo" ha il numero 2 e il campo "autori" ha il numero 3. Se un campo è

assente al momento dell'invio del messaggio, il suo valore predefinito verrà restituito al momento dell'invio del messaggio. Se un campo è assente al momento dell'invio del messaggio, il suo valore predefinito sarà restituito quando il messaggio viene letto dal destinatario. In queste note non si parlerà di tipi di dati, enum, tipi di mappe e altri dettagli di gRPC.

Questo approccio strutturato alla definizione dei tipi di messaggio consente uno scambio di dati efficiente e standardizzato in gRPC attraverso diversi linguaggi di programmazione.

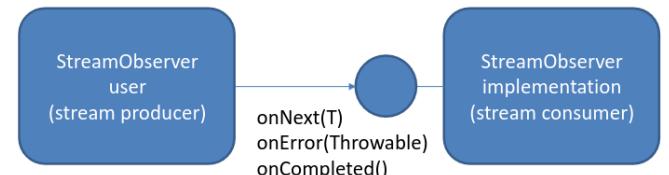
7.2 gRPC: Tipi di RPC supportati

gRPC supporta vari tipi di chiamate di procedura remota (RPC) che rispondono a diversi modelli e requisiti di comunicazione:

- **RPC sincrono classico:** è l'approccio RPC tradizionale, in cui il client fa una richiesta al server e attende una risposta. La comunicazione è sincrona, cioè il client è bloccato finché non riceve una risposta dal server.
- **RPC asincrono a due vie:** in questo tipo di RPC, il client invia una richiesta al server, ma invece di attendere una risposta, continua la sua esecuzione. Il server elabora la richiesta e invia la risposta al client separatamente. Ciò consente l'esecuzione simultanea sia sul lato client che su quello server.
- **Modalità Streaming:** gRPC supporta la **modalità streaming**, sia per il lato client che per il lato server, permette di gestire un flusso **continuo** di dati. In questa modalità, il client e il server possono inviare dati **in tempo reale** senza dover aspettare che l'intero messaggio sia pronto. Ciò significa che i dati possono essere trasmessi in **segmenti** o **parti** man mano che vengono prodotti o consumati, permettendo una comunicazione continua e in tempo reale.

StreamObserver viene utilizzato in gRPC per gestire lo **streaming di dati in modalità asincrona**,

permettendo al server e al client di inviare e ricevere dati in tempo reale (streaming) in modo efficiente. In gRPC, i servizi possono essere definiti in diversi tipi di comunicazione, tra cui streaming (sia dal client che dal server), e **StreamObserver** è lo strumento che facilita la gestione di queste comunicazioni bidirezionali o unidirezionali. **StreamObserver** è presente sia sul **lato client** che sul **lato server**. Sul **lato server**, viene utilizzato per gestire il **flusso di messaggi** che arrivano dal **client** (per lo streaming delle richieste) o per **invio di messaggi di risposta** in streaming **al client**. Sul **lato client**, viene utilizzato per **ricevere risposte** in streaming **dal server** e per **inviare richieste** in streaming.



1. **Response Streaming** (Server-Side)
 - a. **Produttore (Server):** Il server invia un flusso continuo di risposte.
 - b. **Consumatore (Libreria):** La libreria gRPC gestisce la ricezione dei dati da inviare al client.
 - c. **Nota:** Anche nei metodi non-streaming, la risposta può essere gestita con uno streaming interno.
2. **Request Streaming** (Server-Side)
 - a. **Consumatore (Server):** Il server riceve un flusso continuo di richieste dal client.
 - b. **Produttore (Libreria):** La libreria gRPC gestisce l'invio della risposta al client.
3. **Ruoli Opposti nel Client-Side**
 - a. **Response Streaming (Client):** Il client è il consumatore dei dati inviati dal server.
 - b. **Request Streaming (Client):** Il client è il produttore dei dati inviati al server.

8. Sincronizzazione

In un sistema distribuito, i processi operano in **modo asincrono**, il che significa che possono avanzare a **velocità diverse** e subire **ritardi variabili**. Tuttavia, ci sono scenari in cui la sincronizzazione diventa necessaria per imporre l'ordine degli eventi o per soddisfare specifici requisiti temporali. Due approcci comuni per ottenere la sincronizzazione nei sistemi decentralizzati sono:

- **Sincronizzazione degli orologi fisici:** Questo approccio prevede la sincronizzazione degli orologi fisici dei diversi processi del sistema. Assicurando che tutti i processi abbiano una nozione comune di tempo, diventa più facile coordinare le azioni e imporre un ordine temporale. Per sincronizzare gli orologi tra i nodi distribuiti si possono usare tecniche come il Network Time Protocol (NTP).
- **Orologi logici:** Nei casi in cui la sincronizzazione temporale precisa non è fattibile o non è richiesta, gli orologi logici forniscono una forma di sincronizzazione più rilassata. Gli orologi logici assegnano agli eventi timestamp logici basati su relazioni causali. Consentono di ordinare gli eventi senza fare affidamento sul tempo fisico e sono particolarmente utili per mantenere la coerenza nei sistemi distribuiti.

Questi meccanismi di sincronizzazione svolgono un ruolo cruciale nei sistemi distribuiti, garantendo il coordinamento, la coerenza e la comunicazione affidabile tra i processi.

8.1 Sincronizzazione dell'orologio fisico

La sincronizzazione dell'orologio fisico nei sistemi distribuiti può essere ottenuta con vari metodi. Alcune possibilità includono:

- **Ricevitori UTC:** La sincronizzazione tramite ricevitori UTC (Coordinated Universal Time) fornisce risultati molto accurati con una precisione fino a 50ns. Tuttavia, questo approccio può essere costoso da implementare.
- **NTP:** il Network Time Protocol (NTP) è una soluzione più economica per la sincronizzazione dell'orologio. Si basa su server temporali e può raggiungere un'accuratezza ragionevole, sebbene non sia così preciso come i ricevitori UTC.

La precisione della sincronizzazione NTP può variare a seconda delle condizioni della rete:

Tabella 1: Precisione di sincronizzazione NTP

Tipo di rete	Precisione
LAN	< 1ms
Internet pubblico	10-50 ms
Internet pubblico in condizioni di congestione	> 100 ms

La sincronizzazione fisica dell'orologio è un aspetto importante dei sistemi distribuiti, in quanto garantisce un comportamento coerente e coordinato tra i diversi nodi.

8.2 Orologi logici

Nei sistemi distribuiti, il raggiungimento di un accordo sull'ordine degli eventi è spesso sufficiente ai fini della sincronizzazione. Gli orologi logici sono algoritmi distribuiti che possono essere utilizzati per ottenere questo tipo di accordo.

Gli orologi logici forniscono un modo per stabilire un ordine parziale degli eventi in un sistema distribuito senza fare affidamento sulla sincronizzazione dell'orologio fisico. Invece, ogni processo del sistema mantiene il proprio orologio logico, che viene incrementato al verificarsi degli eventi. L'ordine degli eventi viene determinato in base ai valori di questi orologi logici.

Esistono diversi algoritmi per l'implementazione degli orologi logici, come gli **orologi Lamport** e gli **orologi vettoriali**. Questi algoritmi consentono ai processi di assegnare timestamp o valori vettoriali agli eventi, permettendo di determinare le relazioni di causalità tra gli eventi.

Utilizzando gli orologi logici, i sistemi distribuiti possono ottenere la sincronizzazione in termini di ordinamento degli eventi, anche in assenza di sincronizzazione fisica degli orologi. Ciò consente ai processi di

ragionare sulle relazioni causali e sulle dipendenze tra gli eventi in un ambiente distribuito.

8.2.1 Orologi Lamport

Gli **orologi di Lamport** sono una tecnica usata nei **sistemi distribuiti** (dove ci sono più computer o processi che lavorano insieme) per **tenere traccia dell'ordine in cui accadono gli eventi in questi sistemi**. Questo è importante perché, nei sistemi distribuiti, i processi non sono sincronizzati da un orologio centrale, quindi ogni processo ha il suo "orologio" che avanza indipendentemente dagli altri.

L'obiettivo principale degli orologi di Lamport è quello di stabilire una **relazione causale** tra gli eventi che avvengono nei vari processi. In altre parole, vogliamo sapere se un evento in un processo ha causato un altro evento in un altro processo. Tuttavia, non possiamo conoscere il **tempo esatto** di ciascun evento (perché non ci sono orologi globali), ma possiamo stabilire un **ordine parziale**.

La **chiave dell'algoritmo** degli orologi di Lamport è la relazione che chiamano "**happens-before**" (che significa "accade prima"). Questa relazione ci dice in quale ordine si verificano gli eventi in un sistema distribuito.

La **relazione happens-before** negli orologi di Lamport può essere osservata nei seguenti casi:

- Quando l'**evento a** si verifica prima dell'**evento b** all'interno dello **stesso processo**.
 - Ad esempio, se un processo ha un evento A che accade prima di un evento B, allora possiamo dire che A **happens-before** B. In altre parole, A deve essere accaduto prima di B, perché fanno parte dello stesso processo e non possono sovrapporsi temporalmente.
- Quando l'**evento a** è l'**invio di un messaggio** e l'**evento b** è la **ricezione dello stesso messaggio** da parte di **processi diversi**.
 - Supponiamo che il processo P1 invii un messaggio a P2. L'evento di invio nel processo P1 **happens-before** l'evento di ricezione nel processo P2. Questo è un **esempio di relazione causale** tra eventi di processi diversi.

Utilizzando gli orologi di Lamport, i processi di un sistema distribuito possono stabilire un ordinamento parziale degli eventi che cattura le relazioni causali tra gli eventi. Tuttavia, è importante notare che gli orologi Lamport non forniscono una nozione globale di tempo fisico e non garantiscono l'accuratezza in termini di ordinamento in tempo reale.

Gli orologi di Lamport sono un elemento fondamentale nei sistemi distribuiti per ragionare sull'ordine degli eventi e garantire la coerenza nei protocolli di passaggio dei messaggi.

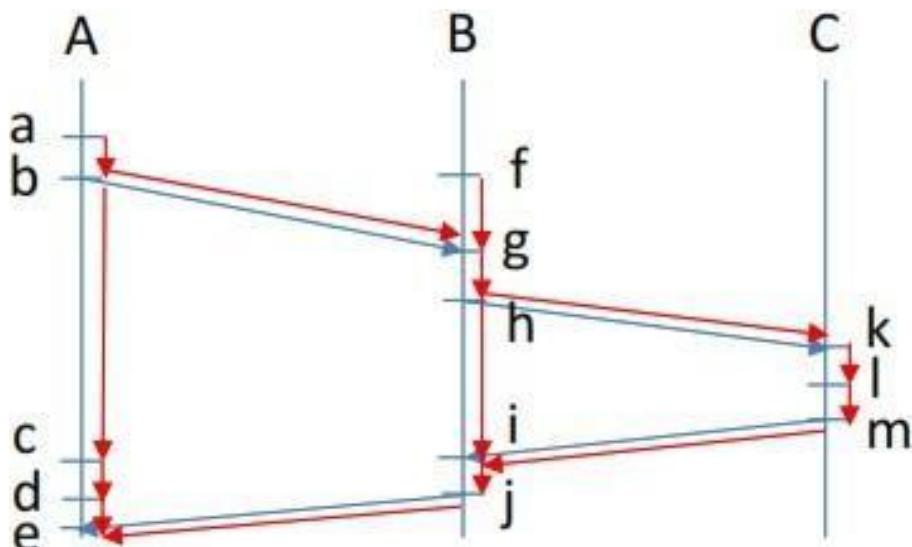


Figura 30: Orologio di Lamport

L'algoritmo dell'orologio di Lamport funziona come segue:

- Ogni processo P_i mantiene un contatore temporale locale C_i
- Il processo P_i marca il tempo di ogni evento locale a con il valore corrente di C_i , indicato come

$C_i(a)$, dopo aver incrementato C_i

- Il processo P_i marca il timestamp di ogni messaggio che invia con il timestamp $C_i(s)$ assegnato all'evento di invio s .
- Quando il processo P_i riceve un messaggio con timestamp C_r
 - Se $C_r > C_i$, il processo P_i regola il proprio orologio locale impostando $C_i = C_r$
 - Dopo aver regolato l'orologio, il processo P_i registra l'evento di ricezione come di consueto.

La fase di regolazione dell'orologio assicura che l'orologio locale C_i del processo P_i sia sempre uguale o superiore a qualsiasi timestamp ricevuto da altri processi.

La fase di regolazione dell'orologio può essere espressa in modo equivalente come $C_i = \max(C_i, C_r)$, assicurando che l'orologio locale sia regolato al valore massimo tra il suo valore attuale e il timestamp ricevuto.

Orologi Lamport: Discussione

Quando si utilizzano gli orologi Lamport, è importante notare che ad alcuni eventi in processi diversi può essere assegnato lo stesso timestamp. Per evitare questa situazione, è possibile includere l'**ID del processo nel timestamp**. In questo caso, il timestamp dell'evento a che si verifica nel processo P_i sarebbe rappresentato come $\langle C_i(a), i \rangle$. Questa ipotesi si basa sull'unicità degli ID dei processi all'interno del sistema distribuito. È importante sottolineare che gli **orologi di Lamport forniscono un ordinamento parziale degli eventi** in base ai loro timestamp. Se l'evento a si verifica prima dell'evento b , allora è vero che $C(a) < C(b)$. **Tuttavia, non è sempre vero il contrario** ($(C(a) < C(b)) \nrightarrow (a \rightarrow b)$). È possibile che due eventi abbiano timestamp diversi ($C(a) < C(b)$) anche se non sono causalmente correlati ($a \nrightarrow b$).

$\langle C_i(x), i \rangle < \langle C_j(y), j \rangle$ iff $C_i(x) < C_j(y)$ or $C_i(x) = C_j(y)$ and $i < j$ questo nel caso si includesse l'ID

Negli orologi di Lamport, avere lo stesso timestamp non implica necessariamente una relazione causale tra eventi. Se due eventi (A) e (B) hanno lo stesso timestamp o timestamp differenti, non puoi sapere se sono concorrenti o meno. La relazione causale (A → B) è garantita solo se un timestamp è maggiore di un altro. Se due eventi hanno lo stesso timestamp, il meccanismo degli orologi di Lamport non offre informazioni aggiuntive per stabilire se esiste una relazione causale oppure no.

L'algoritmo dell'orologio di Lamport può essere implementato utilizzando una **funzione di timestamping** che incrementa e restituisce il valore del contatore. Inoltre, i processi comunicano tra loro inviando e ricevendo **messaggi**. Ogni volta che un processo invia un messaggio, deve allegare al messaggio un **timestamp** che rappresenta l'ora logica in cui il messaggio è stato inviato. Nel momento in cui un altro processo riceve questo messaggio, deve **confrontare** il timestamp del messaggio con il proprio contatore locale. Se il timestamp del messaggio è maggiore del proprio contatore, il processo **aggiorna** il suo contatore locale al valore del timestamp ricevuto (per assicurarsi che il proprio orologio sia sempre "allineato" a quello del sistema). Per implementare questo, possiamo usare dei **ganci (hooks)** che vengono eseguiti ogni volta che si invia o si riceve un messaggio. Questi ganci sono semplicemente delle funzioni che si occupano di **aggiungere il timestamp ai messaggi** quando vengono inviati e di **aggiornare il contatore locale** quando i messaggi vengono ricevuti.

I processi che non **causally related** si dicono **concurrent**.

Nell'esempio i contatori partono da 0 ma questo non è importante, possono partire da un qualsiasi numero, alla ricezione del messaggio, i contatori vengono confrontati e viene preso il max tra di essi.

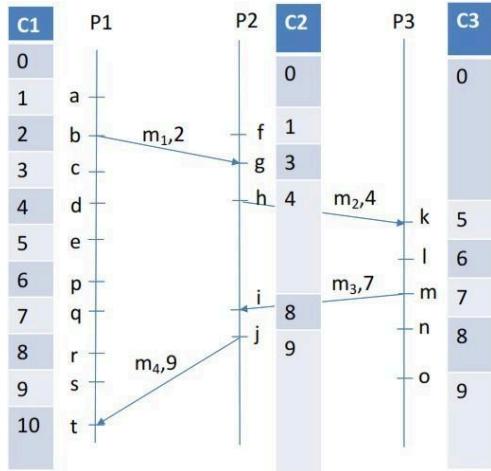


Figura 31: Esempio di algoritmo di clock Lamport

Limiti degli orologi di Lamport

- **Ordine parziale, non totale:** Anche se gli **orologi di Lamport** possono dirci che un evento A è causato da un evento B (ad esempio, l'invio di un messaggio causa la sua ricezione), **non possono dirci sempre con certezza in che ordine sono avvenuti altri eventi che non sono direttamente correlati**. Due eventi in processi diversi possono avere timestamp diversi senza che uno sia veramente "prima" dell'altro.
 - Esempio: Se P1 ha un timestamp di 5 e P2 ha un timestamp di 6, non possiamo dire con certezza se P1 sia accaduto prima di P2 o viceversa. Solo se ci sono messaggi tra i processi possiamo avere una relazione causale chiara.

Vantaggi:

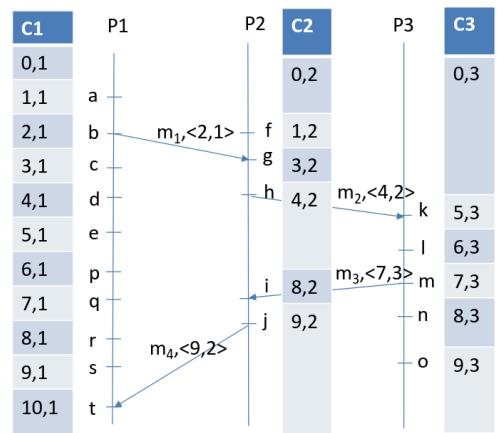
- **Coerenza e sincronizzazione:** Gli **orologi di Lamport** permettono ai sistemi **distribuiti** di stabilire un **ordine tra gli eventi** e garantire che i processi coordinino correttamente le loro operazioni, anche se non sono sincronizzati da un orologio centrale.

Esempio di orologio di Lamport con Total Ordering

L'algoritmo degli orologi di Lamport fornisce un **ordine parziale** degli eventi, cioè possiamo stabilire quale evento è avvenuto prima di un altro solo se c'è una **relazione causale** diretta (come l'invio e la ricezione di un messaggio). Tuttavia, se due eventi **non sono direttamente correlati** (ad esempio, non c'è un messaggio che li collega), **non possiamo stabilire con certezza quale sia avvenuto prima**.

Per ottenere un **ordine totale**, dobbiamo introdurre un criterio aggiuntivo quando i timestamp sono uguali. Questo può essere fatto, per esempio, **utilizzando l'ID del processo insieme al timestamp**, in modo da garantire che ogni evento abbia un timestamp unico. In tal modo, possiamo confrontare sempre due eventi e determinare quale è avvenuto prima, anche se non sono causali l'uno rispetto all'altro, ottenendo così un **ordinamento totale**.

Example with Total Ordering



Il **total ordering** è un meccanismo che si applica agli orologi di Lamport per garantire un ordine totale tra tutti gli eventi, anche se questi sono concorrenti. Tuttavia, **non risolve il problema della distinzione tra eventi concorrenti**; semplicemente impone un **ordine arbitrario** per tutti gli eventi nel sistema.

Applicazione: Multicast totale ordinato

In alcuni scenari, come nel caso dei **server replicati**, è necessario un **multicast totalmente ordinato**, in cui **tutti i messaggi multicast devono essere consegnati nello stesso ordine a ciascun ricevitore**. Questa applicazione è particolarmente importante quando ci sono più **server replicati** identici, con l'obiettivo di garantire la **tolleranza ai guasti e la riduzione della latenza**.

In questa configurazione, le richieste di servizio possono essere inviate a qualsiasi server del gruppo. Quando riceve una richiesta, il server la invia in multicast a tutti gli altri server del gruppo.

Il requisito chiave in questo scenario è che le operazioni devono essere eseguite in ogni replica nello stesso ordine. Il **multicast con ordine totale garantisce che tutti i server elaborino i messaggi multicast in modo coerente e sincronizzato**. Garantendo lo stesso ordine di esecuzione, il sistema raggiunge la coerenza desiderata tra le repliche.

Il multicast con ordine totale è fondamentale per mantenere l'integrità e la correttezza dei server replicati. Garantisce che tutte le repliche elaborino le richieste in un ordine coerente, **evitando potenziali incoerenze o conflitti** che potrebbero derivare dall'esecuzione di operazioni in ordini diversi.

Impiegando un **multicast totalmente ordinato**, i sistemi possono ottenere un **comportamento affidabile e coerente** tra i **server replicati**, consentendo la tolleranza ai guasti e garantendo una riduzione della latenza grazie all'elaborazione parallela, pur mantenendo l'ordine di esecuzione richiesto.

L'algoritmo Multicast with total ordering utilizza orologi Lamport, mantenuti in ogni processo del sistema. L'algoritmo assicura che ogni messaggio multicast sia marcato dal mittente con il timestamp dell'evento di invio. I messaggi ricevuti vengono quindi accodati e ordinati in base ai loro timestamp.

Ecco le fasi dell'algoritmo di multicast a ordinamento totale:

1. Gli **orologi di Lamport** sono mantenuti in **ogni processo** del sistema.
2. Quando un **processo invia un messaggio multicast**, **marca il messaggio** con il **timestamp** dell'evento di **invio**.
3. Alla **ricezione** di un **messaggio multicast**, **ogni ricevitore aggiunge il messaggio** alla sua **coda di messaggi**, **ordinando** i messaggi in base ai loro **timestamp**.
4. **Ogni ricevitore conferma** la **ricezione** del messaggio agli **altri ricevitori**.
5. Una volta **ricevuti gli acknowledgment** per il messaggio in testa alla coda da parte di tutti gli altri ricevitori, il messaggio viene messo in coda e consegnato all'applicazione per un'ulteriore elaborazione.

Utilizzando gli **orologi di Lamport** e l'**approccio basato sulle code**, l'**algoritmo di multicast a ordine totale** garantisce che **i messaggi vengano consegnati ed elaborati nello stesso ordine da tutti i ricevitori**. Il meccanismo di riconoscimento fornisce un modo per sincronizzare i ricevitori e garantire che un messaggio venga rimosso dalla coda solo quando tutti i ricevitori ne hanno confermato la ricezione.

Questo algoritmo è essenziale in scenari in cui il mantenimento di un ordine coerente di consegna dei messaggi è cruciale, come ad esempio in ambienti con server replicati. Applicando il multicast con ordine totale, l'algoritmo facilita l'elaborazione affidabile e sincronizzata dei messaggi tra tutti i ricevitori, garantendo la coerenza e la correttezza desiderate nel sistema.

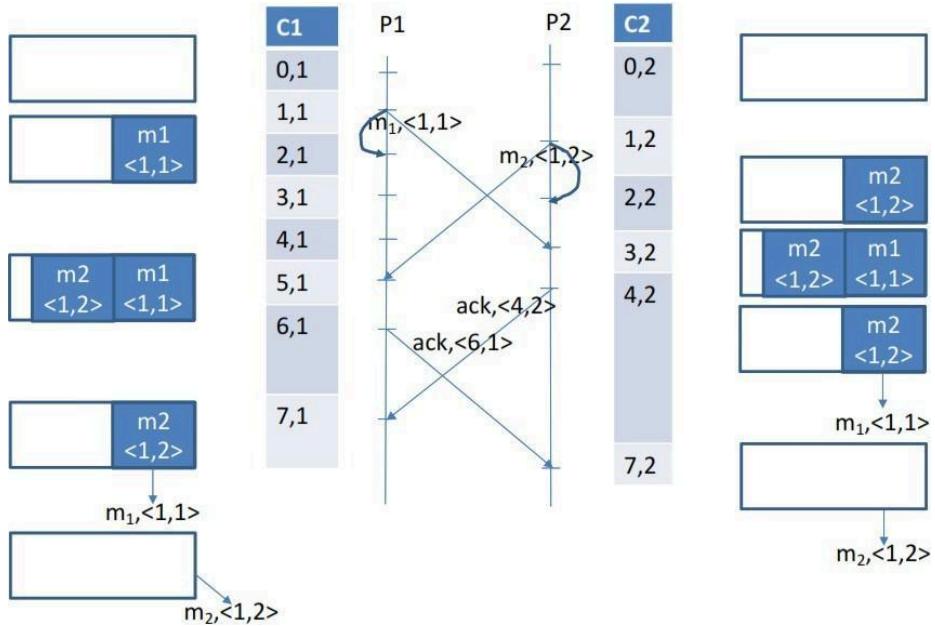


Figura 32: Esempio di algoritmo multicast a ordine totale

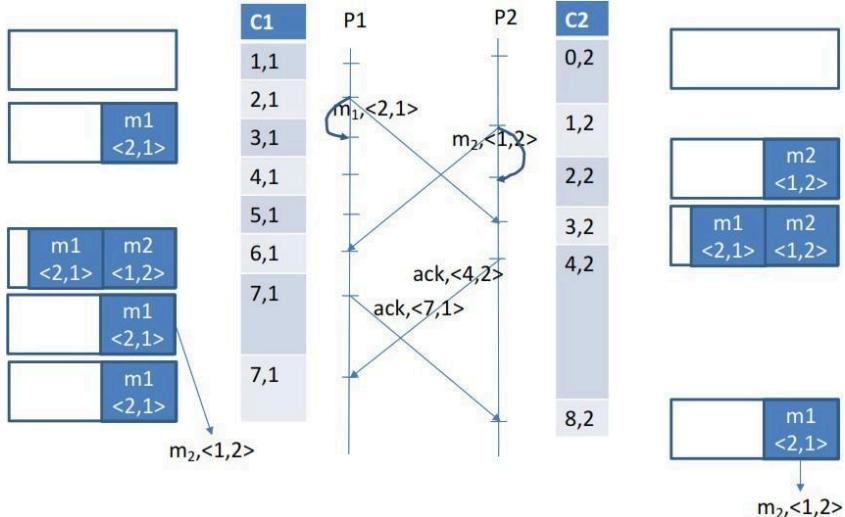


Figura 33: Algoritmo multicast a ordine totale: Altro scenario

Multicast a ordinamento totale: Discussione

È stato dimostrato che l'**algoritmo Multicast with total ordering consegna i messaggi** a tutti i processi di destinazione nello stesso ordine, date le seguenti ipotesi:

1. Nessun messaggio viene perso durante la trasmissione.
2. I messaggi provenienti dallo stesso mittente vengono ricevuti dai destinatari nello stesso ordine in cui sono stati inviati.

Quando queste ipotesi sono vere, l'algoritmo garantisce che l'**ordine di consegna dei messaggi sia coerente tra tutti i ricevitori**. Questo è particolarmente importante quando i messaggi in multicast rappresentano operazioni da eseguire sui dati replicati nello stesso ordine.

Implementando l'algoritmo di multicast a ordine totale, il **sistema realizza la replica della macchina a stati**. La **replica della macchina a stati** assicura che ogni replica **elabori le operazioni nello stesso ordine**, garantendo la coerenza dei dati replicati. L'algoritmo consente un'esecuzione affidabile e sincronizzata delle operazioni sui dati replicati, indipendentemente dall'ordine di ricezione delle operazioni da parte delle repliche.

Vale la pena di notare che **l'algoritmo si basa sulle ipotesi di cui sopra**. Se i messaggi vengono persi o ricevuti fuori ordine, la garanzia dell'algoritmo di consegna dei messaggi totalmente ordinati potrebbe non essere valida e potrebbe essere necessario implementare meccanismi aggiuntivi, come la ritrasmissione o la gestione degli errori, per gestire tali scenari.

Nel complesso, l'algoritmo di multicast con ordine totale fornisce una soluzione robusta per ottenere una consegna coerente dei messaggi e un'esecuzione sincronizzata delle operazioni in ambienti replicati. Garantendo lo stesso ordine di elaborazione dei messaggi in tutti i ricevitori, consente una replica affidabile e tollerante delle macchine a stati, contribuendo alla correttezza e all'integrità complessive del sistema.

8.2.2 Vector Clocks

Sebbene gli **orologi di Lamport** forniscano un **ordinamento totale degli eventi**, è importante notare che questo **ordinamento non implica necessariamente la causalità**. In altre parole, se $C(a) < C(b)$, non garantisce che l'evento a preceda causalmente l'evento b.

Per ovviare a questa limitazione, sono stati introdotti gli orologi vettoriali come meccanismo per fornire un ordinamento parziale degli eventi che catturi la causalità. Negli **orologi vettoriali**, **ogni processo mantiene** un **vettore di timestamp**, dove ogni elemento rappresenta il valore logico dell'orologio di un particolare processo.

La proprietà dell'orologio vettoriale $C(a) < C(b)$ è definita come segue:

- Se **$C(a) < C(b)$** , significa che l'evento **a** precede **causalmente** l'evento **b**. In altre parole, l'evento **a** è stato la causa dell'evento **b**.
- Se **$C(a) > C(b)$** , significa che l'evento **b** precede causalmente l'evento **a**.
- Se invece **$C(a) = C(b)$** non sono comparabili (cioè nessuno dei due vettori è maggiore dell'altro), allora gli eventi **a** e **b** sono **concomitanti** (cioè, sono avvenuti indipendentemente l'uno dall'altro, senza una relazione causale).

Utilizzando gli orologi vettoriali, è possibile determinare la relazione causale tra gli eventi, anche in un sistema distribuito. Ogni processo incrementa il proprio valore di orologio logico ogni volta che esegue un evento locale e include il proprio orologio vettoriale nei messaggi che invia agli altri processi. Quando riceve un messaggio, un processo aggiorna il proprio orologio vettoriale prendendo il valore massimo per ogni elemento tra il proprio orologio vettoriale e l'orologio vettoriale ricevuto.

Il confronto degli orologi vettoriali consente di identificare gli eventi causalmente correlati. Se $C(a) < C(b)$, implica che l'evento a precede causalmente l'evento b. Questa proprietà consente di rilevare la causalità all'interno di un sistema distribuito, aiutando in varie applicazioni come l'ordinamento degli eventi, le istantanee distribuite e i protocolli di coerenza distribuiti.

È importante notare che gli orologi vettoriali forniscono un ordinamento parziale e non possono catturare le esatte dipendenze causali tra tutti gli eventi. Tuttavia, offrono un approccio pratico ed efficiente per catturare la causalità nei sistemi distribuiti, consentendo di ragionare sulla causalità e di preservare le relazioni causali tra gli eventi.

L'algoritmo **Vector Clocks** consente di catturare la causalità in un sistema distribuito. L'algoritmo prevede le seguenti fasi:

1. **Ogni processo ha un vettore di orologi**: Ogni processo nel sistema distribuito mantiene un **vettore di orologi** che ha tante voci quanti sono i processi nel sistema. Ad esempio, se ci sono tre processi **P1, P2 e P3**, ogni processo avrà un vettore di tre valori, uno per ogni altro processo. Il valore in posizione **i** del vettore di un processo **Pi** rappresenta il contatore di eventi logici che **Pi** ha osservato nel processo **Pi**.
2. **Inizializzazione dell'orologio vettoriale**: All'inizio, ogni processo imposta il proprio vettore come $[0, 0, 0]$ (o comunque con i valori iniziali a zero) perché all'inizio nessun evento è stato registrato.
3. **Aggiornamento del vettore al verificarsi di un evento locale**: Quando un processo **Pi** esegue un evento locale (come una computazione o una scrittura), il processo **incrementa il proprio contatore locale** (cioè il valore del proprio orologio) e aggiorna il proprio vettore, incrementando il valore in corrispondenza del proprio indice nel vettore.
4. **Invio di un messaggio**: Quando un processo invia un messaggio, **invia anche il proprio vettore di orologio** al destinatario del messaggio. In questo modo, il destinatario conoscerà la versione dell'orologio di tutti i processi al momento dell'invio del messaggio.
5. **Ricezione di un messaggio**: Quando un processo **Pi** riceve un messaggio con un vettore di orologio **VCr**

(dove r è l'indice del mittente), **aggiorna il proprio orologio vettoriale**. L'aggiornamento avviene **prendendo il valore massimo** tra ciascun valore del proprio vettore e quello ricevuto nel messaggio. Questo assicura che il processo abbia il massimo della conoscenza temporale tra i suoi eventi e quelli degli altri processi.

6. **Marcare l'evento di ricezione:** Dopo aver aggiornato il proprio vettore, il processo **Pi** marca l'evento di ricezione con il nuovo valore del vettore, il che aiuta a mantenere traccia dell'ordine temporale di tutti gli eventi, inclusi quelli causati da comunicazioni inter-processo.

Mantenendo e aggiornando gli orologi vettoriali in ogni processo, l'algoritmo assicura che le dipendenze causalı tra gli eventi possano essere catturate. La regolazione dell'orologio vettoriale durante la ricezione dei messaggi consente ai processi di sincronizzare la loro conoscenza del tempo e di incorporare le informazioni di timestamp provenienti da altri processi.

Il confronto degli orologi vettoriali tra i diversi processi consente di determinare le relazioni causalı tra gli eventi, aiutando in varie applicazioni come l'ordinamento degli eventi, le istantanee distribuite e i protocolli di coerenza distribuiti.

È importante notare che l'algoritmo Vector Clocks richiede ai processi di scambiare e aggiornare le informazioni sull'orologio vettoriale durante il passaggio dei messaggi, consentendo loro di mantenere una visione coerente del tempo in tutto il sistema distribuito e di catturare con precisione la causalità tra gli eventi. Lo scenario di Lamport Clock presentato in precedenza (*figura: 31*) è ora presentato nella versione Vector Clock:

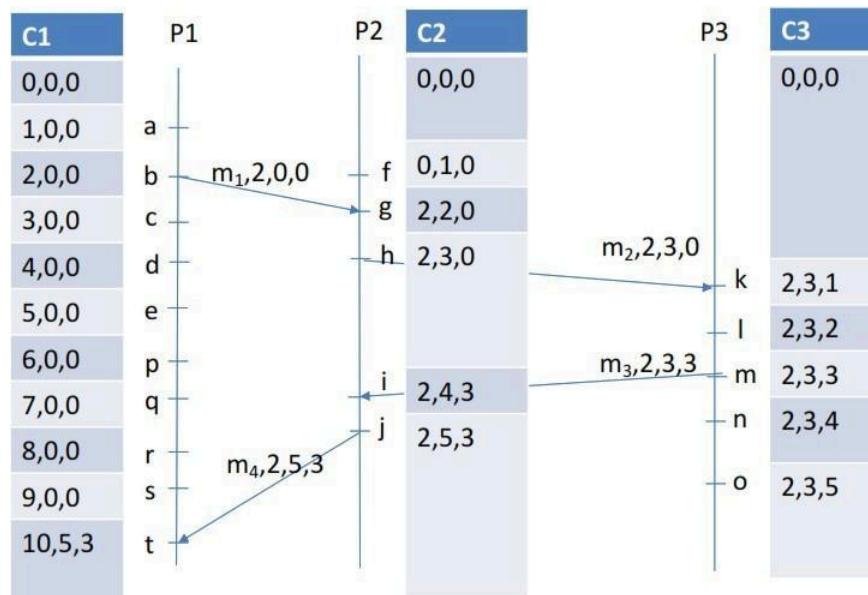


Figura 34: Algoritmo del clock vettoriale

Limiti degli orologi vettoriali

- **Complessità:** Gli **orologi vettoriali richiedono più memoria e banda rispetto agli orologi di Lamport**, perché ogni processo deve mantenere un vettore con una dimensione pari al numero di processi nel sistema, e deve inviare l'intero vettore nei messaggi.
- **Gestione dei conflitti:** Sebbene gli **orologi vettoriali** possano rilevare la causalità, **non forniscono una soluzione per i conflitti tra eventi concorrenti** che avvengono in sistemi distribuiti. È necessario usare altri protocolli (come Paxos o Raft) per risolvere i conflitti che potrebbero emergere.

Esempio di applicazione dell'orologio vettoriale: Multicast ordinato causale

Nel contesto della comunicazione multicast in un sistema distribuito, è importante garantire l'ordine causale. L'ordine causale significa che **ogni messaggio multicast deve essere consegnato solo dopo che tutti i messaggi causalmente correlati inviati in precedenza sono stati consegnati**. Tuttavia, i messaggi non causalmente correlati inviati in precedenza possono essere consegnati in qualsiasi ordine.

Per ottenere un multicast ordinato causale, si possono utilizzare orologi vettoriali. Ogni processo mantiene un

vettore locale di contatori temporali, denotato come $VC[i]$, dove i rappresenta l'identificatore del processo. Quando invia un messaggio multicast, il mittente lo marca con il valore corrente del suo orologio vettoriale. Alla ricezione di un messaggio multicast, un processo aggiorna il suo orologio vettoriale prendendo il massimo valore del suo orologio vettoriale locale e il timestamp ricevuto.

Utilizzando gli orologi vettoriali, un sistema distribuito può garantire che i messaggi multicast siano consegnati in modo causalmente ordinato, rispettando le dipendenze tra i messaggi e consentendo al tempo di consegnare in qualsiasi ordine i messaggi non causalmente correlati.

Differenza tra Lamport Clock e Vector Clock

La differenza tra **Lamport Clock** e **Vector Clock** risiede nella capacità di rappresentare correttamente la **causalità** e identificare **eventi concorrenti** in un sistema distribuito.

Il **Lamport Clock** si basa sulla relazione "**happens before**" ($a \rightarrow b$), garantendo che se un evento a causa l'evento b , allora il **timestamp assegnato a a sarà minore di quello assegnato a b** . Tuttavia, il **contrario non è vero**: se il timestamp di a è minore di quello di b , non si può affermare con certezza che a abbia causato b , poiché il **Lamport Clock non cattura le relazioni causali in modo completo**. Inoltre, **non consente di distinguere eventi concorrenti** (ovvero eventi che non sono correlati da una relazione di causalità), poiché il confronto dei timestamp può comunque indicare un ordine arbitrario anche quando gli eventi sono indipendenti.

Il **Vector Clock**, invece, rappresenta **un'evoluzione più precisa**. Ogni evento è associato a un vettore di timestamp che tiene traccia del conteggio degli eventi noti per ciascun processo. **Questo approccio consente di catturare completamente la causalità**: se un evento a causa b , allora il vettore di a sarà strettamente minore del vettore di b e viceversa. Inoltre, i vector clock **permettono di identificare eventi concorrenti**. Se i vettori di due eventi non sono confrontabili (ad esempio, alcuni valori di un vettore sono maggiori e altri minori rispetto all'altro), allora i due eventi sono indipendenti e non correlati causalmente.

Confronto tra Lamport e Vector Clock

Proprietà	Lamport Clock	Vector Clock
Happens Before	Se $a \rightarrow b$, allora $LC(a) < LC(b)$	Se $a \rightarrow b$, allora $VC(a) < VC(b)$
Viceversa valida?	No ($LC(a) < LC(b)$ non implica $a \rightarrow b$)	Si ($VC(a) < VC(b)$ implica $a \rightarrow b$)
Concorrenza rilevata?	No	Si (se i vettori non sono confrontabili)
Precisione causale	Limitata	Completa

Definizione di relazione causale (causal relationship)

Le relazioni causali in un sistema distribuito si basano sull'idea che alcuni eventi (messaggi) possono influenzare altri, stabilendo un ordine logico tra di essi. Per comprendere queste relazioni nel contesto del multicast ordinato causale, consideriamo tre tipi principali di relazioni:

1. Relazione di causalità diretta

Un messaggio m_1 causa direttamente un altro messaggio m_2 se:

- a. m_1 viene inviato da un processo P , e successivamente P invia m_2 .

Ciò significa che m_2 dipende logicamente da m_1 , quindi m_1 deve essere consegnato prima di m_2 .

2. Relazione di causalità transitiva

Un messaggio m_1 causa transitivamente un messaggio m_3 se esiste un messaggio intermedio m_2 tale che:

- a. m_1 causa m_2 , e
- b. m_2 causa m_3 .

3. Assenza di relazione causale (messaggi concorrenti)

Due messaggi m_1 e m_2 sono concorrenti se:

- a. Nessuno dei due è causato dall'altro, né direttamente né transitivamente.
- b. Non esiste alcun evento o messaggio che li collega logicamente.

In altre parole, i messaggi concorrenti non hanno alcun ordine obbligatorio di consegna.

9. Coordination Algorithm

Nei sistemi distribuiti, il coordinamento tra più processi è fondamentale per garantire un corretto utilizzo delle risorse e un'esecuzione ordinata. Gli **algoritmi di coordinamento** svolgono un ruolo fondamentale per ottenere la **sincronizzazione**, la **mutua esclusione**, l'**elezione del leader** e il **consenso tra i processi**. Questi algoritmi **consentono ai processi di collaborare in modo efficiente, mantenere la coerenza e prendere decisioni collettive** in un ambiente distribuito.

Questa sezione esplora vari algoritmi di coordinazione che affrontano le sfide più comuni dei sistemi distribuiti. Si inizia esaminando il concetto di mutua esclusione, che garantisce l'accesso esclusivo alle risorse condivise. Vengono discussi diversi tipi di algoritmi di mutua esclusione, come gli approcci basati sui token e sui permessi.

Un altro problema di coordinamento critico è l'elezione di un leader o di un coordinatore tra un gruppo di processi. Esaminiamo il problema dell'elezione e i suoi requisiti, presupposti e soluzioni. L'**algoritmo Bully**, che **elegge un coordinatore** attraverso lo **scambio di messaggi**, e l'**algoritmo Ring**, che utilizza una **struttura logica ad anello** per il **processo di elezione**, sono spiegati in dettaglio.

Infine, si accenna al concetto di consenso, che prevede il raggiungimento di un accordo tra i processi su un valore di output comune. Il consenso è un problema di coordinamento più generale e può essere affrontato con tecniche di elezione del leader o di mutua esclusione.

In questa sezione, si approfondiscono gli algoritmi di coordinamento fondamentali che facilitano una collaborazione efficiente e affidabile tra i processi distribuiti. La comprensione di questi algoritmi ci permette di progettare sistemi distribuiti robusti, in grado di gestire l'accesso concorrente alle risorse condivise, di eleggere i leader e di prendere decisioni collettive.

9.1 Mutual Exclusion

Per garantire l'accesso reciprocamete esclusivo alle risorse condivise da più processi in un sistema distribuito, esistono diversi tipi di algoritmi:

- **Basato su gettoni (token)**: Un token circola tra i processi, consentendo l'accesso alla risorsa solo al possessore del token.
- **Basato sui permessi**: I processi richiedono e ottengono permessi da un gestore centrale o dai propri pari per accedere alla risorsa.
 - **Centralizzato**: Un gestore centrale controlla l'accesso alle risorse condivise.
 - **Distribuito (Decentralizzato)**: Gli algoritmi decentralizzati, come quelli basati su orologi di Lamport, gestiscono l'accesso usando un protocollo di accordo tra i processi.

9.1.1 Token Ring Mutual Exclusion

In questo algoritmo, i processi sono organizzati in un overlay ad **anello**. Nel sistema c'è un **token che circola** continuamente **sull'anello**. Quando un processo P vuole accedere a una risorsa, **attende il token**. Una volta **ricevuto il token**, il processo P **inizia ad accedere alla risorsa e mantiene il token** fino al termine dell'accesso. Quindi, **passa il token** al processo successivo nell'anello. **Ogni processo** deve ricevere il **token** prima di poter accedere alla risorsa. Ogni volta che un processo ottiene il token, invia un **messaggio** per passarne il controllo al prossimo processo. Il **numero di messaggi per accesso** dipende dalla posizione del processo che vuole accedere alla risorsa rispetto al token. In media, se i processi sono distribuiti in modo uniforme, il numero di messaggi per ciascun accesso è circa **n/2** (dove n è il numero di processi).

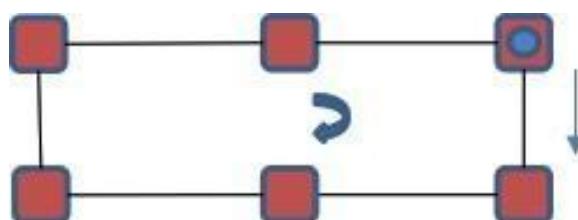


Figura 35: Algoritmo di mutua esclusione Token Ring

Vantaggi:

- Efficienza: Minimo overhead di comunicazione (in media, 1 messaggio per accesso).
- Non ci sono conflitti poiché esiste un solo token.
- Facile da implementare in reti affidabili.

Svantaggi:

- Il **token può andare perso** (es. a causa di un crash), e servono meccanismi per rigenerarlo.
- Non è **ideale in reti instabili**, poiché i ritardi possono bloccare l'accesso alla risorsa.

9.1.2 Centralized Mutual Exclusion (permission based)

Questo algoritmo prevede un **gestore centrale (C)** responsabile della gestione delle **risorse condivise**. Quando un **processo P** vuole accedere a una risorsa, **invia una richiesta al gestore centrale e attende una risposta ai permessi**. Il gestore centrale ritarda la concessione dei permessi quando la risorsa è già impegnata.

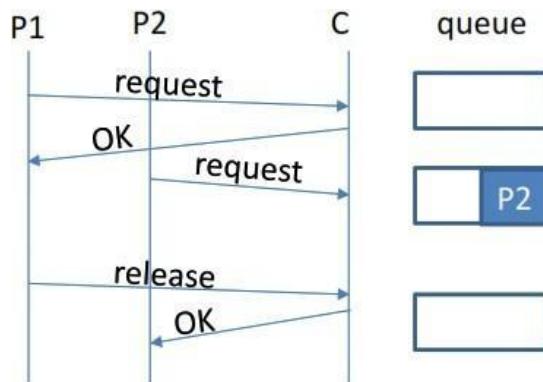


Figura 36: Algoritmo di mutua esclusione centralizzato

9.1.3 Decentralized Mutual Exclusion (permission based)

L'algoritmo distribuito di mutua esclusione si basa sugli **orologi di Lamport** per il **multicast totalmente ordinato**. Quando un processo richiedente vuole accedere a una risorsa, invia una **richiesta a tutti i processi** (incluso se stesso) e **attende il permesso da ogni processo**. In caso di conflitto, se anche un altro processo vuole accedere alla risorsa, la richiesta con il timestamp più basso ha la precedenza.

Vantaggi:

- Non c'è bisogno di un token, quindi non ci sono problemi legati alla sua perdita.
- Adatto in ambienti dove tutti i processi devono essere coinvolti nel controllo.

Svantaggi:

- Alto overhead di comunicazione: Ogni accesso richiede **2(n-1) messaggi** (richieste + risposte).
- Maggiore latenza rispetto al metodo token-based.
- **Rischio di deadlock o starvation** in caso di errori di rete.

9.1.4 Confronto delle prestazioni

Dopo aver descritto il precedente algoritmo di esclusione, è molto significativo confrontare le loro prestazioni:

Algorithm		#Messages/access	delay before entry (# messages)
Permission based	Centralized	3	2
	Distributed	2 (N-1)	2 (N-1)
Token based	Token ring	1...	0 ... N-1

Figura 37: Confronto delle prestazioni

9.2 Election

Il problema dell'elezione consiste nell'eleggere un processo da un gruppo di processi per svolgere un determinato lavoro o assumere un ruolo specifico tra gli altri (ad esempio, un coordinatore). In questo contesto sono necessarie alcune ipotesi:

- Ogni processo ha un **identificatore univoco** $\text{id}(P)$.
- Ogni processo **conosce tutti gli altri processi del gruppo**.
- I **processi del gruppo** possono essere **attivi** o **inattivi**, ma i **canali sono affidabili**.

L'algoritmo deve eleggere il processo up con l'id più alto e, alla fine dell'algoritmo, tutti i processi devono concordare su chi sia il processo eletto.

9.2.1 Election: Bully algorithm

L'**algoritmo Bully** viene utilizzato per eleggere un **coordinatore** da un **gruppo di processi**. Ogni processo ha un identificatore univoco. Quando un processo rileva che il **coordinatore è assente**, inizia un'elezione inviando un **messaggio di ELEZIONE** ai **processi di numero superiore**. Se non riceve risposta, il processo vince l'elezione, altrimenti si arrende. Il **vincitore informa gli altri processi della sua elezione**.

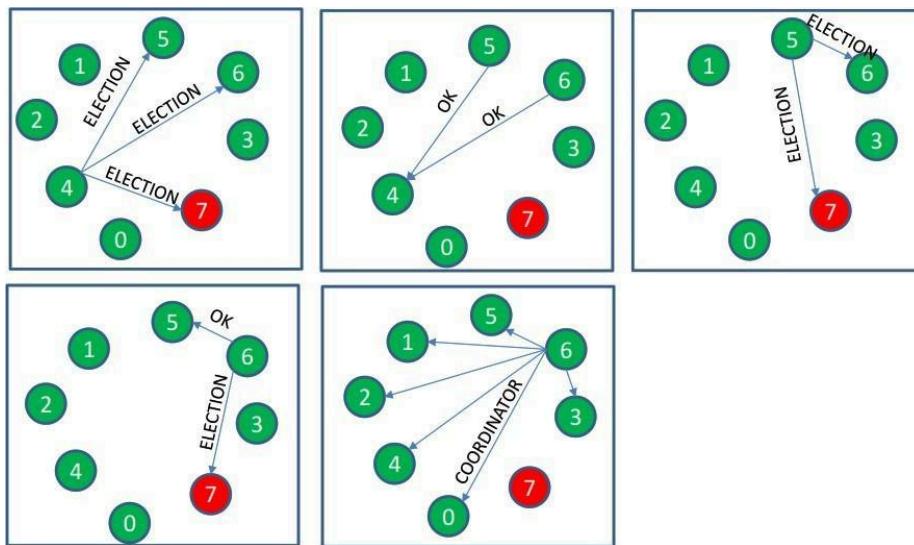


Figura 38: Algoritmo del bullo elettorale

9.2.2 Election: Ring algorithm

L'**algoritmo ad anello** elegge un **coordinatore** utilizzando una **struttura logica ad anello**. Quando un processo rileva che il coordinatore è assente, inizia un'elezione inviando un **messaggio di ELEZIONE** al suo **successore nell'anello**. Se non riceve risposta, inoltra il messaggio al successore successivo finché non riceve una risposta. Il messaggio di ELEZIONE **contiene l'elenco dei mittenti**. Quando il messaggio torna al processo di partenza, interrompe la circolazione e calcola il vincitore, che è il processo con l'**identificatore**

più alto tra quelli che hanno partecipato all'elezione. Il processo che ha determinato il vincitore dell'elezione invia quindi un **messaggio COORDINATORE** a tutti gli altri processi nell'anello per comunicare l'esito dell'elezione.

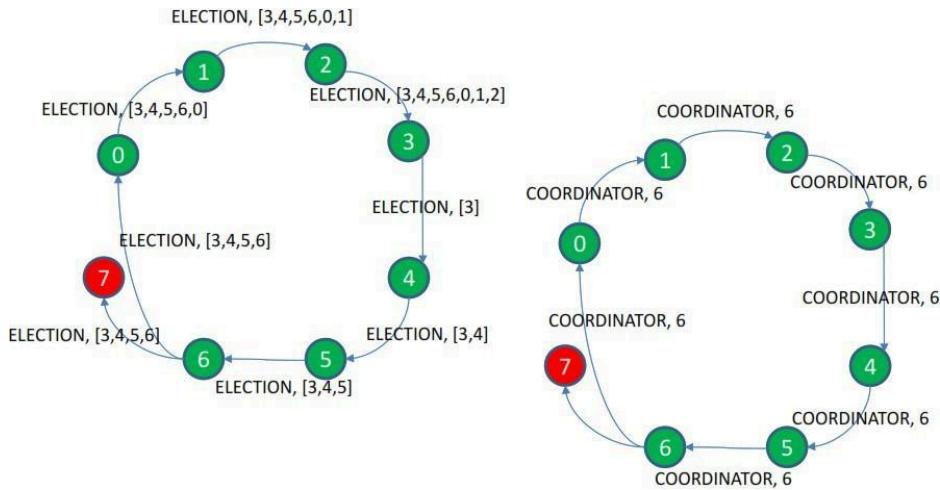


Figura 39: Algoritmo dell'anello elettorale

9.3 Consensus

Il **consenso** è un problema di coordinamento fondamentale nei sistemi distribuiti, in cui un insieme di **n** processi (o nodi) deve **decidere insieme su un valore comune**, pur avendo ciascun processo il proprio valore di ingresso (proposto). L'obiettivo del consenso è che **tutti i processi**, a prescindere dai valori che propongono inizialmente, concordino su **un unico valore di uscita**, che verrà poi utilizzato per decisioni future nel sistema.

Proprietà e relazioni con altri problemi

- **Mutua esclusione:** Il problema della **mutua esclusione** (garantire che solo un processo alla volta acceda a una risorsa critica) può essere visto come un **caso speciale del problema di consenso**. In una situazione di mutua esclusione, i processi devono "concordare" su quale processo accede alla risorsa.
- **Elezione del leader:** Anche l'**elezione di un leader** (come nel caso dell'algoritmo Bully o Ring) è un **caso particolare di consenso**. In un'elezione del leader, i processi devono concordare su quale processo deve diventare il coordinatore (leader), che è il "valore di uscita" scelto dal sistema.

Criteri da tenere a mente quando si valuta le performance di un algoritmo di coordinazione

1. **Complessità dei messaggi**
 - Descrizione: Si riferisce al numero di messaggi scambiati tra i processi durante l'esecuzione dell'algoritmo.
2. **Complessità temporale** (o complessità computazionale)
 - Descrizione: Misura il tempo richiesto per eseguire l'algoritmo in relazione alla dimensione del sistema (numero di processi, nodi o risorse).
3. **Complessità spaziale**
 - Descrizione: Rappresenta la quantità di memoria necessaria per eseguire l'algoritmo, inclusi i dati di stato e le strutture ausiliarie.
4. **Resilienza ai guasti**
 - Descrizione: Indica la capacità dell'algoritmo di continuare a funzionare correttamente nonostante guasti di processi o della rete (es. crash di nodi, perdita di messaggi).
5. **Scalabilità**
 - Descrizione: Rappresenta la capacità dell'algoritmo di mantenere buone prestazioni man mano che il numero di processi o risorse cresce.
6. **Latency (latenza)**
 - Descrizione: Misura il tempo che intercorre tra l'invio di una richiesta e la sua risposta.
7. **Fairness (Equità)**

- a. Descrizione: Riguarda la distribuzione equa delle risorse o dei diritti di accesso tra i processi.
- 8. **Overhead**
 - a. Descrizione: Indica il carico extra imposto dall'algoritmo sui processi, sia in termini di calcolo che di comunicazione.
- 9. Robustezza
 - a. Descrizione: La capacità dell'algoritmo di gestire situazioni impreviste o eccezionali, come messaggi duplicati, timeout, o cambiamenti dinamici nella topologia della rete.
- 10. **Convergenza**
 - a. Descrizione: Per algoritmi come quelli di consenso, la convergenza indica il tempo necessario per raggiungere uno stato stabile in cui tutti i processi concordano su una decisione.

10. Data Replication and Consistency

La **replica dei dati** è una tecnica utilizzata per **migliorare l'affidabilità e le prestazioni di un sistema**. In questa sezione ci concentreremo sulla replica dei dati legata alle prestazioni, che comporta tecniche di scaling out per la scalabilità dimensionale e geografica. Idealmente, la replica dei dati dovrebbe essere trasparente per l'utente, ma introduce una sfida legata alla coerenza. Quando i dati replicati cambiano, possono sorgere stati incoerenti, rendendo temporaneamente le repliche non identiche e interrompendo la trasparenza. Tuttavia, possiamo cercare di nascondere queste incoerenze al costo di una latenza aggiuntiva e di un sovraccarico di prestazioni.

10.1 Cost of Consistency

Per **risolvere le incoerenze**, le **modifiche ai dati** vengono **copiate** in **tutte le repliche**. Questo comporta un **sovraccarico di banda** e di **calcolo** per l'esecuzione delle copie e aggiunge **latenza** per nascondere gli stati incoerenti. Sebbene la replica possa migliorare le prestazioni e la scalabilità, la coerenza delle repliche comporta un **costo in termini di prestazioni**. L'equilibrio tra coerenza e prestazioni **dipende dai requisiti specifici di coerenza** (consistency requirements).

10.2 Consistency Model

Un **modello di consistenza** funge da **contratto tra i processi e l'archivio dati**. **Definisce le proprietà di consistenza** garantite dal **data store** nelle **operazioni di lettura/scrittura**. In altre parole, il modello di consistenza stabilisce le **regole** che determinano in che modo i dati vengono visualizzati e aggiornati dai processi che accedono al sistema.

La **coerenza rigorosa** (strict consistency) è il modello di consistenza più forte e più semplice da comprendere. Garantisce che:

- Ogni **operazione di scrittura** su un dato è immediatamente visibile a tutte le repliche.
- Non è mai possibile leggere un dato obsoleto: se un processo scrive un valore, ogni altro processo che legge lo stesso dato vedrà subito quel valore

Quindi, la **coerenza rigorosa** (strict consistency) garantisce che ogni modifica dei dati venga propagata a tutte le repliche prima di eseguire l'operazione successiva sull'archivio dati. Questo modello è facile da capire e da usare, ma **comporta un costo elevato in termini di prestazioni**, poiché ogni scrittura deve essere sincronizzata immediatamente su tutte le repliche, aumentando i tempi di latenza. In pratica, per migliorare le prestazioni, i requisiti di coerenza vengono allentati, rendendo il modello meno semplice per i programmati. In generale, questo è il modello di sistema di riferimento:

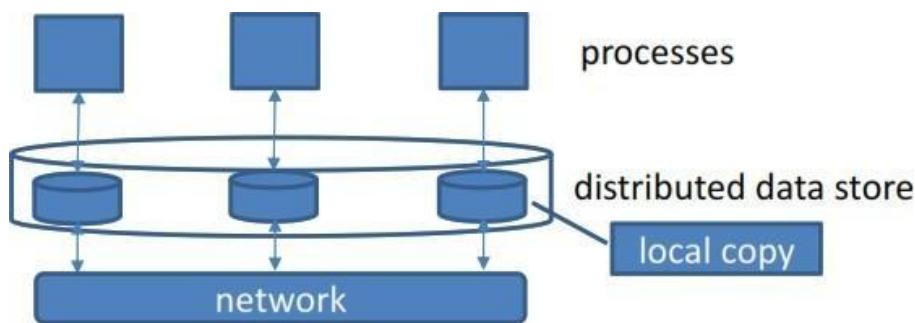


Figura 40: Modello di coerenza

10.2.1 Data-Centric Consistency Models (Modelli di coerenza incentrati sui dati)

Le proprietà di coerenza possono essere espresse in **termini di comportamento delle operazioni di lettura/scrittura sull'archivio dati globale**. Ad esempio, la coerenza rigorosa garantisce che ogni lettura restituisca gli ultimi dati scritti. Allentando la coerenza rigorosa, si allentano i vincoli sulle operazioni di lettura e scrittura.

10.2.2 Continuous Consistency

La **Continuous Consistency** è un modello di consistenza che cerca di bilanciare prestazioni e coerenza, consentendo un certo grado di **deviazione** dai principi della **strict consistency**. In altre parole, non è richiesta una coerenza assoluta in tempo reale, ma ogni lettura deve restituire un valore che sia "**vicino**" a quello più recente scritto. La **tolleranza alla deviazione** dipende da tre misure principali:

1. **Numeric Deviation:** La differenza tra i dati numerici tra le repliche (come i valori numerici di un contatore).
2. **Deviazione di stallo (staleness deviation):** La differenza nel **tempo dell'ultimo aggiornamento** tra le repliche. Se una replica non è stata aggiornata recentemente, la lettura potrebbe restituire dati obsoleti, ma solo per un certo periodo di tempo.
3. **Deviation in the order of writes:** A volte le operazioni di scrittura vengono applicate a una copia locale prima di essere propagate alle altre repliche, causando una **deviazione temporanea** nell'ordine delle operazioni.

La **Continuous Consistency** offre quindi un **continuum di opzioni**, dove le deviazioni ammesse possono essere regolate per migliorare le **prestazioni**, riducendo il carico di sincronizzazione tra le repliche, ma sempre con la garanzia che le letture siano abbastanza vicine al valore più recente scritto.

10.2.3 Unità di consistenza (Conits)

Quando si **misura la coerenza**, è comune riferirsi a **unità di dati** chiamate **conits** piuttosto che all'intero archivio di dati. La granularità dei conits introduce dei compromessi.

- **Conits di grandi dimensioni** possono portare a **incoerenze significative**.
 - Se ogni conit rappresenta una grande porzione di dati, è più facile mantenere la coerenza tra le repliche, ma può portare a incoerenze significative se qualche replica non è aggiornata in tempo. La causa di queste incoerenze è che una modifica su una grande unità di dati potrebbe non propagarsi rapidamente a tutte le repliche, rendendo più difficile mantenere la coerenza in tempo reale.
- **Conits di piccole dimensioni** richiedono la **gestione di un numero maggiore di unità**.
 - Con unità più piccole, ogni singolo conit può essere aggiornato e sincronizzato più rapidamente, migliorando la **coerenza**. Tuttavia, la gestione di un numero maggiore di conits implica **maggiori costi di coordinamento** e **maggior complessità** nel garantire che tutte le repliche siano aggiornate correttamente, aumentando il carico di lavoro del sistema.

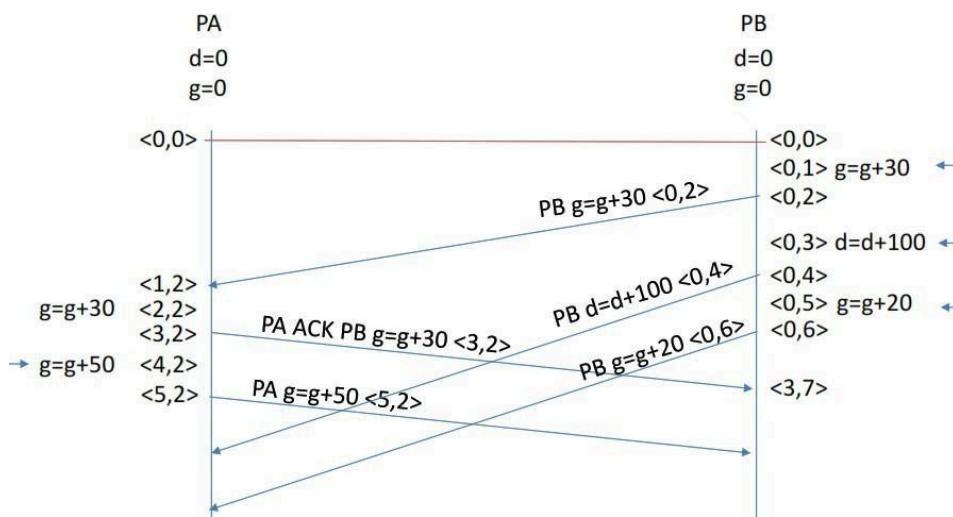


Figura 41: Esempio: (vedi Van Steen, Tanenbaum)

10.2.4 Sequential Consistency (Lamport)

Il **modello di coerenza sequenziale** di Lamport definisce una regola per determinare come devono essere visibili le operazioni di lettura e scrittura in un sistema distribuito. In particolare, stabilisce due condizioni

chiave:

- Sequenzialità globale:** Le operazioni di lettura e scrittura eseguite da tutti i processi devono sembrare come se fossero state eseguite in un **ordine sequenziale unico**, ossia in una sequenza di operazioni che avvengono una dopo l'altra, come in un singolo flusso di esecuzione. Questo significa che il sistema deve garantire che tutte le operazioni siano viste in un certo ordine, e tutte le repliche devono essere in accordo su questo ordine.
- Ordine per processo:** Ogni processo deve osservare le sue **operazioni locali** (le letture e scritture che esegue) nell'ordine in cui le ha effettivamente eseguite, come specificato dal programma del processo. Ad esempio, se un processo esegue prima una scrittura e poi una lettura, l'ordine di queste operazioni deve essere rispettato anche nel contesto globale.

Proprietà del modello

- Deterministico a livello globale:** Anche se le operazioni possono essere intercalate in modi diversi (nondeterminismo), tutti i processi devono vedere lo stesso ordine di esecuzione, ovvero lo stesso **interleaving** delle operazioni. Ogni operazione di lettura restituisce lo stesso valore, come se tutte le operazioni del sistema fossero state eseguite in un unico ordine sequenziale possibile.
- Semplicità:** La coerenza sequenziale è meno forte della coerenza rigorosa (strict consistency) ma offre un buon compromesso tra prestazioni e consistenza, ed è meno costosa da implementare rispetto alla coerenza rigorosa.

P1	W(x)a
P2	W(x)b
P3	R(x)b
P4	R(x)a

P1	W(x)a
P2	W(x)b
P3	R(x)b
P4	R(x)a

Figura 42: Esempio: Consistenza sequenziale Clock Lamport

10.2.5 Casual Consistency

La **coerenza causale** garantisce che le scritture potenzialmente correlate dal punto di vista causale devono essere viste da tutti i processi nello stesso ordine. Tuttavia, le scritture concomitanti possono essere viste in un ordine diverso su macchine diverse. La coerenza causale è più debole della coerenza sequenziale.

P1	W(x)a	W(x)c
P2	R(x)a	W(x)b
P3	R(x)a	R(x)c
P4	R(x)a	R(x)b

P1	W(x)a
P2	R(x)a
P3	W(x)b
P4	R(x)a

Figura 43: Esempio: Coerenza causale

La **causal relation** avviene tra la $W(x)a$ e $W(x)b$ tra il **processo P1** e il **processo P2** perché P2 prima di scrivere in x , legge prima il suo valore e legge a , quindi sappiamo a priori che questa lettura è avvenuta dopo la scrittura di P1, di conseguenza l'operazione di scrittura di P2 è **causal related** all'operazione di scrittura di P1.

10.2.6 Entry Consistency

La **consistenza di ingresso** è un modello di consistenza che si riferisce a gruppi di operazioni eseguite atomicamente, spesso indicate come sezioni critiche. Le proprietà di consistenza in entry consistency sono espresse in termini di operazioni di lettura, scrittura, blocco e sblocco. L'acquisizione di un blocco può avere successo solo se tutte le scritture nella sezione critica sono state eseguite in modo atomico. L'accesso esclusivo a un blocco può avvenire solo se nessun altro processo ha accesso esclusivo o non esclusivo a quel blocco, mentre l'accesso non esclusivo a un blocco è consentito solo se ogni precedente accesso esclusivo è

stato completato.

P1	L(x) W(x)a L(y) W(y)b U(x) U(y)	
P2	L(x) R(x)a R(y)null	
P3	L(y) R(x)null R(y)b	

P1	L(x) W(x)a L(y) W(y)b U(x) U(y)	
P2	L(x) R(x)null R(y)null	
P3	L(y) R(x)null R(y)b	

Figura 44: Esempio: Entry Consistency

10.2.7 Eventual Consistency

In alcune applicazioni, sono accettabili garanzie di consistenza più deboli. La **Eventual Consistency** è un modello di consistenza utilizzato nei sistemi distribuiti in cui **non viene garantito immediatamente** che tutte le copie dei dati siano sincronizzate, ma si garantisce che alla fine **tutte le copie convergano verso uno stato consistente**, purché non ci siano ulteriori aggiornamenti. Questo modello è particolarmente adatto per scenari in cui:

- Ci sono molte più **lettura** rispetto alle **scritture**.
- È importante avere **alta disponibilità e velocità** di accesso ai dati, anche a costo di una temporanea incoerenza.

La **eventual consistency** funziona bene quando ci sono **pochi aggiornamenti** e quando i processi accedono sempre alla stessa replica, ma la mobilità dei processi può causare problemi.

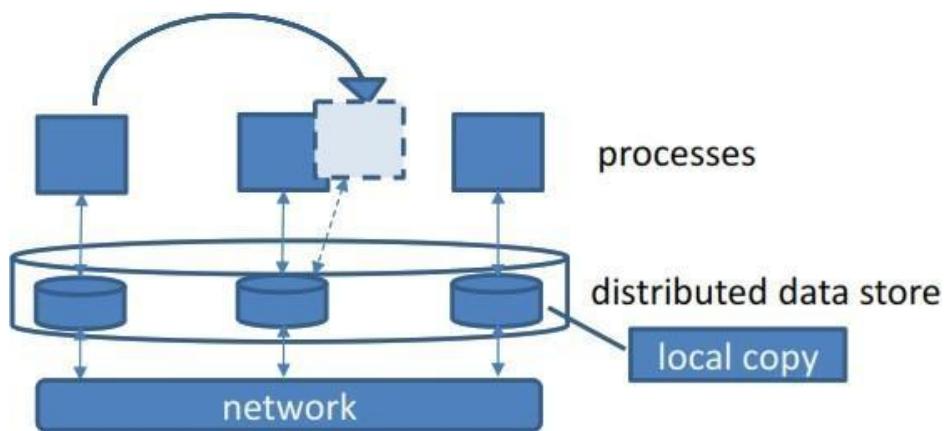


Figura 45: Esempio: Eventual Consistency

10.2.8 Client-centered consistency models

Mentre i **modelli di consistenza incentrati sui dati** forniscono proprietà di consistenza dei dati a livello di sistema, i **modelli di consistenza incentrati sul client** forniscono proprietà di consistenza per ogni **singolo processo o client**. Questi modelli possono essere aggiunti per limitare i problemi legati alla **process mobility**

quando si usa la **consistenza eventuale**.

xj	version j of variable x
Wk(xj)	Pk writes xj
Wk(x1;x2)	Pk writes x2 which follows from x1
Wk(x1 x2)	Pk writes x2 concurrently with x1 (potential write-write conflict)

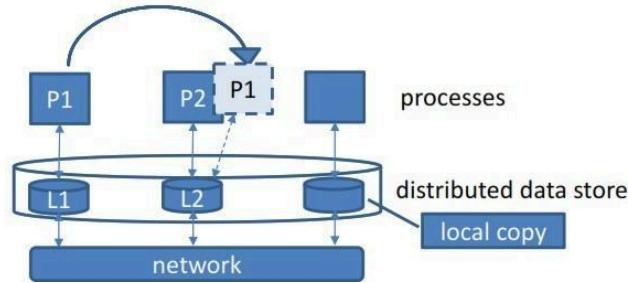


Figura 46: Esempio: Modelli di coerenza centrata sul cliente con coerenza eventuale

Operazioni descritte

1. **Wk(xj)**:
 - Significa che il processo **Pk** ha scritto una versione **xj** della variabile **x**.
2. **Wk(x1;x2)**:
 - Il processo **Pk** scrive una nuova versione **x2**, che è una versione successiva e **deriva causalmente** da **x1**.
 - Questo implica una relazione di **dipendenza causale**: **x2** esiste solo dopo che **x1** è stato scritto.
3. **Wk(x1|x2)**:
 - Il processo **Pk** scrive **x1** e **x2** **contemporaneamente**, senza dipendenze tra loro.
 - Qui potrebbe verificarsi un **conflitto di scrittura (write-write conflict)** perché non c'è un ordine chiaro tra **x1** e **x2**.
4. **Rk(xj)**:
 - Il processo **Pk** legge la versione **xj** della variabile **x**.

Letture monotone (Monotonic Reads) Nel modello di coerenza delle **lettute monotone**, se un processo legge un dato, le letture successive dello stesso dato restituiranno sempre quel valore o uno più aggiornato.

L1	W1(x1)	R1(x1)	✓	
L2	W2(x1;x2)	R1(x2)		✗

Figura 47: Esempio: Letture monotone

Scritture monotone (Monotonic Writes) Nel modello di coerenza delle **scritture monotone**, se un processo scrive su un dato, quella scrittura deve essere completata prima che lo stesso processo possa fare un'altra scrittura sullo stesso dato.

L1	W1(x1)	
L2	W2(x1;x2) W1(x2;x3)	

L1	W1(x1)	
L2	W2(x1 x2) W1(x2;x3)	

Figura 48: Esempio: Scritture monotone

Read Your Writes Nel modello di coerenza “**read your writes**”, se un processo scrive un dato, qualsiasi lettura successiva dello stesso dato da parte dello stesso processo rifletterà sempre quella scrittura.

L1	W1(x1)	
L2	W2(x1;x2) R1(x2)	

L1	W1(x1)	
L2	W2(x1 x2) R1(x2)	

Figura 49: Esempio: Leggere le scritture

Writes Follow Reads Nel modello di coerenza “**writes follow reads**”, se un processo scrive su un dato dopo averlo letto, la scrittura sarà basata sullo stesso valore letto o su uno più aggiornato.

L1	W1(x1) R2(x1)	
L2	W3(x1;x2) W2(x2;x3)	

L1	W1(x1) R2(x1)	
L2	W3(x1 x2) W2(x1 x3)	

Figura 50: Esempio: Le scritture seguono le letture

10.3 Gestione delle repliche

La gestione delle repliche consiste nel decidere quali server o contenuti replicare e dove collocare le repliche. È un aspetto importante della replica dei dati per garantire prestazioni e affidabilità. Principalmente, esistono due modi diversi di eseguirla:

- **Replicazione dei server:** si concentra sulla determinazione dei server da replicare e sul problema del posizionamento, che comporta la decisione di dove collocare i server replicati. Questo aspetto è particolarmente importante per ottenere il **bilanciamento del carico** e la **tolleranza ai guasti**.
- **Replicazione dei contenuti:** si tratta invece di decidere quali contenuti devono essere replicati e il problema del posizionamento delle repliche. L'obiettivo è **ottimizzare la disponibilità dei contenuti** e **ridurre la latenza di accesso ai dati**.

10.3.1 Aggiornamento delle strategie di propagazione

Le strategie di propagazione degli aggiornamenti definiscono il modo in cui gli aggiornamenti vengono propagati in un sistema replicato. Si possono usare diverse strategie a seconda del **rappporto lettura-aggiornamento**, dei **requisiti di coerenza** e di altri fattori. Fondamentalmente, la propagazione degli aggiornamenti può assumere diverse forme:

- **Propagazione della notifica degli aggiornamenti:** nota anche come **protocolli di invalidazione**, prevede la notifica degli aggiornamenti alle repliche. Questa strategia funziona meglio in scenari con un basso rapporto lettura-aggiornamento.
- **Propagazione dei dati:** comporta la propagazione dei dati effettivi alle repliche. Questa strategia è più adatta a scenari con un elevato rapporto lettura-aggiornamento.
- **Propagazione delle operazioni di aggiornamento:** nota anche come replica attiva, prevede la propagazione delle operazioni di aggiornamento stesse. Tuttavia, questo approccio non è sempre fattibile o conveniente in tutti gli scenari.

10.3.2 Protocolli Push (basati su server) e Pull (basati su client)

I protocolli pull e push sono approcci diversi alla sincronizzazione delle repliche in un sistema replicato. I **protocolli push** sono generalmente utilizzati per le **repliche permanenti e avviate dal server**. Sono adatti quando è richiesta una **forte coerenza** e funzionano bene in scenari con un **elevato rapporto letturaaggiornamento**. I **protocolli pull**, invece, sono generalmente utilizzati per le **repliche avviate dal client**. Sono più adatti quando è accettabile una **consistenza debole** e funzionano bene in scenari con un **basso rapporto letturaaggiornamento**. Tuttavia, il tempo di risposta può aumentare con le mancanze della cache.

	Push	Pull
State at server	list of client replicas+caches	none
Messages sent	update (+ fetch if invalidation)	poll (+ fetch if changed)
Response time at client	0 (or fetch time if invalidation)	poll (+fetch) time

Figura 51: Protocolli Push vs Pull

Esempio di Protocollo Pull (Basato su Client)

Immagina un sistema di **content delivery network (CDN)** utilizzato per servire **contenuti multimediali** (come video o immagini) a utenti in tutto il mondo. In questo caso, i client (utenti) richiedono i contenuti ai server di replica quando necessario, ma non c'è una sincronizzazione automatica dei dati da parte del server.

Scenario:

1. **Cliente A** (situato in Europa) richiede di visualizzare un video.
2. Il server principale (**S1**), che si trova negli Stati Uniti, non invia automaticamente il contenuto al server europeo. Invece, **Cliente A** fa una richiesta esplicita per ottenere il video dal server più vicino.
3. Il server europeo (**S2**), che non ha ancora una copia del video, esegue una **richiesta pull** al server principale (**S1**) per ottenere il contenuto.
4. Una volta che il server europeo (**S2**) ottiene il video, lo memorizza nella cache per fornire velocemente il contenuto a eventuali altri utenti che ne faranno richiesta.
5. Se un altro **Cliente B** richiede lo stesso video, il server europeo (**S2**) può rispondere rapidamente, in quanto ha già la copia del contenuto.

Vantaggi:

- Scalabilità: il carico sui server è ridotto, poiché i client richiedono i dati solo quando ne hanno bisogno.
- Minor traffico di rete: i server non inviano dati continuamente, ma solo quando un client richiede una risorsa.

Svantaggi:

- Coerenza debole: non c'è garanzia che tutti i client abbiano sempre la versione più aggiornata del contenuto, poiché il server replica i dati solo quando un client richiede l'aggiornamento.
- Maggiore latenza nelle prime richieste: se una replica non ha il contenuto nella cache, il server deve fare una richiesta al server principale, causando un aumento della latenza.

10.4 Protocolli per la Continous Consistency

I protocolli per la **Continous Consistency** prevedono che i processi eseguano scritture sulle copie locali in modo provvisorio. Ogni processo propaga le sue scritture locali agli altri processi, che quindi rilevano i conflitti e applicano gli aggiornamenti alle loro copie locali. Le metriche di coerenza vengono monitorate e, in caso di deviazioni, vengono intraprese azioni correttive, come l'interruzione dell'esecuzione di altre scritture sulla copia locale o la richiesta di propagazione degli aggiornamenti.

10.5 Protocolli per la Sequential Consistency

I protocolli per la **Sequential Consistency** si concentrano sulla garanzia che l'esecuzione di operazioni di lettura e scrittura su un sistema replicato si comporti come se fossero eseguite in un certo ordine sequenziale. Due protocolli comunemente utilizzati per ottenere la coerenza sequenziale sono i **Primary Based Protocols** e i **Replicated Write Protocols**.

10.5.1 Primary Based Protocols

I protocolli basati sul primario assegnano un **processo primario** responsabile del **coordinamento** delle **operazioni di scrittura** su un particolare elemento di dati. Esistono due varianti principali di **Primary Based Protocols**:

- **Protocolli di scrittura remota (backup primario):** Nei **protocolli di scrittura remota** con **backup primario**, il processo che detiene la **copia primaria** del dato è fisso, ossia non cambia. Quando un processo desidera eseguire una scrittura su un dato, invia la richiesta di scrittura al **processo primario** che detiene la copia originale del dato. Questo primario a sua volta **aggiorna** tutte le **repliche dei dati** e invia un ACK al processo client bloccato in attesa del completamento della scrittura. Poiché le scritture sono consentite solo attraverso un **primario**. L'ordine in cui sono state fatte le scritture assicura **sequential consistency**. Questo processo primario è responsabile di coordinare tutte le operazioni di scrittura e di garantire la consistenza dei dati.
- **Protocolli di scrittura locale:** Nei **protocolli di scrittura locale**, la **copia primaria** di un dato viene trasferita al processo che deve scrivere. In pratica, un client che desidera scrivere su un dato replicato può farlo migrando la copia primaria del dato su un server di replica locale al client. Il client può quindi fare molte operazioni di scrittura localmente mentre gli aggiornamenti ad altre repliche sono effettuati in modo asincrono. Vi è solo un primario in qualsiasi momento.

10.5.2 Replicated Write Protocols

I **Replicated Write Protocols** consentono di eseguire operazioni di scrittura su repliche diverse, ma garantiscono che l'ordine delle operazioni rimanga coerente tra tutte le repliche. Questo è cruciale per assicurare che tutte le copie dei dati siano allineate correttamente, evitando incoerenze tra le repliche.

I due approcci principali utilizzati per ottenere protocolli di scrittura replicata sono:

- **Protocolli di replica attiva:** I **protocolli di replica attiva** sono progettati per garantire che le operazioni di scrittura vengano eseguite in modo coerente su tutte le repliche di un dato, mantenendo un **ordine globale** delle operazioni. Ciò significa che le operazioni di scrittura sono applicate nello stesso ordine in tutte le repliche, garantendo così una **coerenza sequenziale**. Questo tipo di replica è utile in scenari in cui è fondamentale che tutte le repliche siano sempre sincronizzate in modo preciso.
- **Protocolli basati sul quorum:** I **protocolli basati sul quorum** sono un meccanismo per garantire la coerenza nei sistemi distribuiti attraverso un **consenso maggioritario (majority consensus)** tra i processi per eseguire operazioni di lettura e scrittura. In questi protocolli, le operazioni vengono eseguite solo se ottengono l'approvazione dalla maggioranza dei nodi, assicurando che tutte le repliche dei dati siano sincronizzate in modo coerente.

10.6 Cache Coherence Protocols

I protocolli di coerenza della cache sono specifici per i sistemi che coinvolgono la cache, tipicamente nei sistemi client-server come le applicazioni web. Questi protocolli mirano a mantenere la coerenza e l'uniformità tra le cache.

10.6.1 Strategia di coerenza

Le strategie di coerenza determinano il modo in cui le incongruenze vengono rilevate e gestite all'interno del sistema. Esistono tre strategie di coerenza comuni:

- **Il client convalida la consistenza prima di procedere:** In questa strategia, il client convalida la consistenza dei dati prima di procedere con una transazione. Se vengono rilevate incongruenze, la transazione non viene eseguita.
- **Il client convalida la coerenza mentre procede (approccio ottimistico):** Questa strategia prevede che il client proceda con la transazione e convalidi la coerenza durante il processo. Se la convalida fallisce, indicando incongruenze, la transazione viene interrotta.
- **Il client convalida la consistenza alla fine della transazione:** In questa strategia, il client completa la transazione e ne convalida la coerenza alla fine. Se vengono trovate incongruenze, la transazione viene interrotta.

10.6.2 Strategia di applicazione (Enforcement Strategy)

La **strategia di applicazione** riguarda il modo in cui la **coerenza** dei dati viene garantita all'interno di un sistema distribuito, specialmente quando i dati vengono memorizzati in **cache** per migliorare le prestazioni. I due approcci principali per far rispettare la coerenza nelle cache di lettura e scrittura sono:

- **Cache di sola lettura con meccanismi push o pull:** In questo approccio, le **cache** sono utilizzate solo per la **lettura** dei dati, mentre gli aggiornamenti (scritture) avvengono altrove. Le cache vengono aggiornate tramite due meccanismi principali:
 - **Meccanismo Push:** In questo caso, quando i dati originali vengono aggiornati, il sistema "spinge" questi cambiamenti verso le cache. Questo significa che quando i dati vengono modificati nella fonte primaria (ad esempio un database o un server), le modifiche vengono immediatamente propagate alle cache, mantenendo così la coerenza tra la fonte dei dati e la cache.
 - **Meccanismo Pull:** In alternativa, quando una cache deve essere aggiornata, il sistema "tira" i dati aggiornati dalla fonte primaria. Questo accade quando una richiesta di lettura arriva alla cache, che a sua volta consulta la fonte principale per ottenere i dati più recenti. In questo caso, la cache potrebbe essere aggiornata solo quando c'è una richiesta di lettura o quando è necessario.
- **Cache in lettura e scrittura con protocolli di scrittura locale basati sul primario:**
 - In questo approccio, le **cache** non sono solo di lettura, ma anche di **scrittura**, il che significa che i client possono eseguire scritture direttamente sulla loro cache. Tuttavia, per evitare incoerenze tra le repliche dei dati, vengono utilizzati protocolli di scrittura di tipo **Primary Based Protocols**. In questo modello, un client che desidera scrivere su un dato deve ottenere un "blocco" su di esso, ovvero il client prende in esclusiva la **copia primaria** del dato. Questo processo è simile a una **lock-based approach**, dove il client si assicura che nessun altro processo possa scrivere su quel dato finché non ha completato la sua operazione. La cache del client che ha preso il blocco diventa un "**primario temporaneo**", il che significa che è autorizzata a fare modifiche ai dati senza interferenze da parte di altri client.
 - Poiché altre cache possono anche avere i dati in memoria, è possibile che altri client vogliano scrivere sugli stessi dati. Quando ciò accade, si attiva una **strategia di risoluzione dei conflitti**.

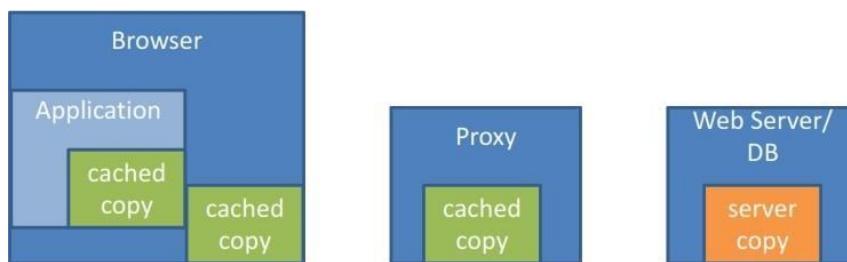


Figura 52: Gestione della cache nelle applicazioni web

11. WebSocket

I **WebSocket** sono diventati una scelta popolare per l'implementazione di applicazioni interattive e in tempo reale. Forniscono canali di comunicazione affidabili, a bassa latenza e bidirezionali, rendendoli adatti a vari casi d'uso come il lavoro cooperativo, le interfacce utente altamente interattive, i giochi, le videoconferenze e le notifiche asincrone dai server ai client.

Sebbene le interazioni richiesta-risposta siano state ampiamente utilizzate per molte applicazioni, non sono sempre adatte a scenari che richiedono notifiche push in tempo reale. Le soluzioni tradizionali, come il **polling** e il **long polling**, si sono dimostrate inefficienti, soprattutto per le **notifiche frequenti**. **COMET**, che include tecniche come lo **HTTP streaming** e la **subscription**, è un altro approccio, ma richiede che il client svolga un **ruolo di server** e non è pienamente soddisfacente per la comunicazione bidirezionale a bassa latenza.

I **WebSocket** sono un **protocollo a livello di applicazione** progettato per fornire canali di comunicazione affidabili, a bassa latenza e bidirezionali. Offrono un servizio di comunicazione **simile a TCP (TCP like)** con alcune differenze e sfruttano l'infrastruttura web esistente. Ciò include il riutilizzo delle stesse porte HTTP (80, 443), la compatibilità con i proxy e altri intermediari web e la capacità di lavorare fianco a fianco con le normali comunicazioni basate su HTTP. I **WebSocket** seguono anche il **modello di security** basato sulla **politica di origine (SOP)**. La comunicazione nei **WebSocket** si basa su una **comunicazione basata su messaggi** attraverso una struttura di **framing binario stratificata su TCP**. Ciò consente una trasmissione dei dati efficiente e flessibile.

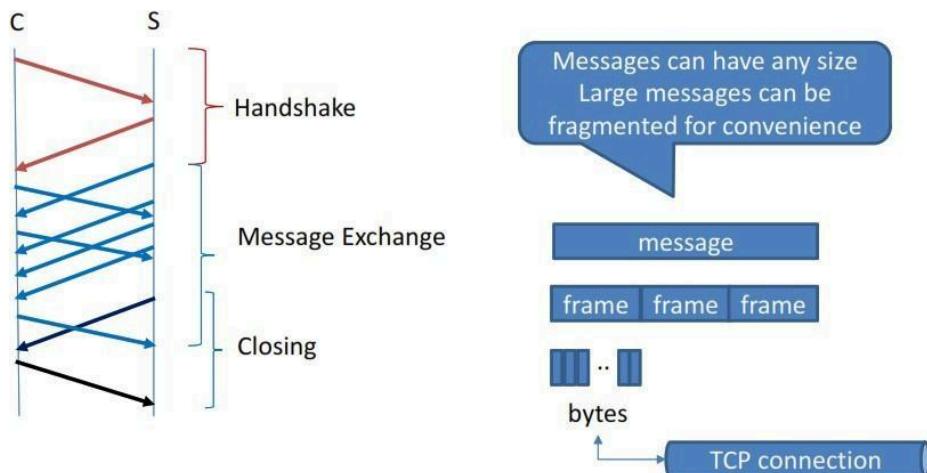


Figura 53: Protocollo Web Socket (RFC 6455)

Handshake Process (HTTP-based Upgrade)

1. **Initial Connection:** A **WebSocket connection starts as an HTTP request**. The client (e.g., a browser) sends an **HTTP Upgrade request to the server**, asking to switch from the HTTP protocol to **WebSocket**.
2. **The request includes headers like Upgrade**: websocket and **Connection: Upgrade**, which indicate the client wants to establish a WebSocket connection.
3. **Server Response:** If the server supports **WebSockets**, it responds with an **HTTP 101 status code**, indicating that it is **switching protocols**, and sends headers confirming the upgrade to **WebSocket**.

11.1 WebSockets vs TCP

I **WebSocket** forniscono connessioni TCP con alcune caratteristiche aggiuntive, come:

- Modello di sicurezza per browser basato sull'origine del Web.
- Meccanismo di indirizzamento e di denominazione dei protocolli che supporta:
 - Più endpoint su una porta
 - Più nomi di host su un indirizzo IP.

- Meccanismo di frame e messaggistica senza limiti di lunghezza dei messaggi.
- Meccanismo di chiusura aggiuntivo in banda che funziona con proxy e intermediari.

11.2 URI WebSocket

Gli endpoint **Websocket** sono identificati in modo univoco da URI, simili agli URI HTTP, ma con lo schema "ws" o "wss". Lo schema "ws" è usato per le connessioni **WebSocket** su TCP, mentre lo schema "wss" è usato per le connessioni WebSocket su TLS su TCP.



Figura 54: Stretta di mano

11.3 Tipi di frame

I **WebSocket** definiscono diversi tipi di frame per il controllo e i dati:

- **Control Frames**: sono utilizzati per scopi specifici, come la chiusura della connessione e le operazioni di ping/pong.
- **Cornici di dati**: trasportano il carico effettivo dei messaggi scambiati tra il client e il server. Possono essere ulteriormente classificati come frame binari, frame di testo (codificati UTF-8) o frame di continuazione.

11.4 Messaggi e chiusura

I **WebSocket** gestiscono i messaggi come sequenze di frame di dati, tutti dello stesso tipo. Esistono due tipi di messaggi:

- **Messaggi non frammentati**: vengono trasmessi come un singolo frame di dati, quindi il bit FIN è impostato su 1 nel singolo frame.
- **Messaggi frammentati**: vengono suddivisi in più frame per la trasmissione, in modo che solo il FIN dell'ultimo fotogramma è impostato a 1.

Entrambi gli endpoint possono avviare un **handshake di chiusura** in **WebSocket**. Una volta inviato un frame di chiusura, non vengono più scambiati frame di dati. L'altro endpoint risponde con un frame di chiusura, se non ne ha già inviato uno. Dopo l'handshake di chiusura, la connessione TCP sottostante viene chiusa. Questa è una differenza sostanziale rispetto a una connessione **Socket TCP/IP**, dove, sebbene la comunicazione sia bidirezionale, se uno dei due partecipanti chiude la connessione, l'altra direzione rimane ancora attiva. Al contrario, nel **WebSocket**, il messaggio di chiusura causa la terminazione completa della comunicazione in entrambe le direzioni.

11.5 Utilizzo di WebSocket

I **WebSocket** forniscono l'infrastruttura di base per la trasmissione dei dati, ma le applicazioni che utilizzano i **WebSocket** spesso implementano protocolli di livello superiore su di essi. Questi protocolli di livello superiore definiscono tipi di messaggi, metadati e procedure per requisiti applicativi più specifici. I sottoprotocolli, che sono protocolli denominati **costruiti su WebSocket**, possono anche essere utilizzati per fornire ulteriori funzionalità e struttura alla comunicazione **WebSocket**. Di seguito viene descritta la programmazione di **WebSocket** in Javascript³:

³In queste note non verrà discussa la programmazione di Web Sockets, per la quale si rimanda alle diapositive.

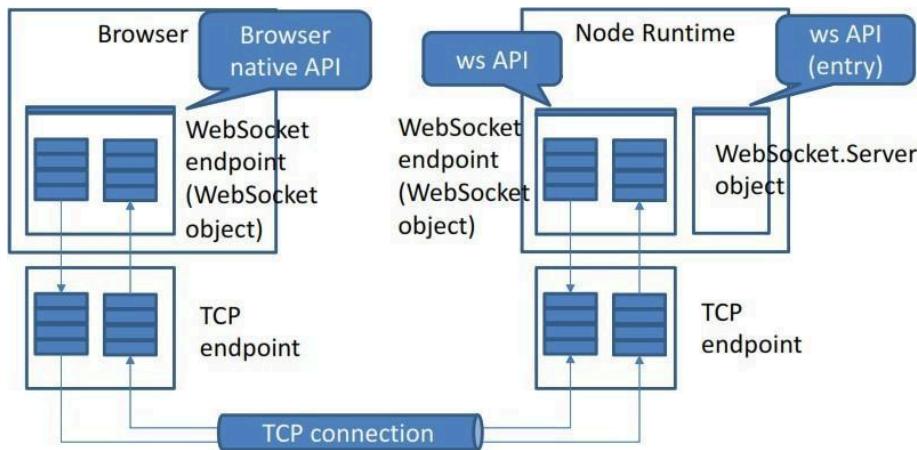


Figura 55: Programmazione Web Socket con Javascript

12. Sockets TCP/IP

I socket TCP/IP sono un metodo di comunicazione tra due dispositivi o programmi su una rete. Un socket è essenzialmente una combinazione di un indirizzo IP e di un numero di porta e rappresenta un punto finale di una comunicazione di rete.

I socket TCP/IP sono molto utilizzati nelle applicazioni client-server, in cui un **programma client** comunica con un **programma server** attraverso una **rete**. Il **client** e il **server** creano **ciascuno un socket** e lo usano per stabilire una **connessione tra loro**. Una volta stabilita la connessione, i due programmi possono scambiarsi dati. L'API Socket estende il modello di I/O di Unix, unificando il trattamento di file e connessioni di rete attraverso i descrittori (*file descriptors*). In TCP/IP esistono principalmente **due tipi di socket**:

- **Stream socket**: forniscono un flusso bidirezionale di dati affidabile, ordinato e controllato da errori tra i due endpoint.
- **Datagram Sockets**: forniscono una consegna di dati inaffidabile, non ordinata e non controllata.
- **Raw Sockets**: forniscono un accesso diretto ai servizi forniti dai protocolli di livello 2-3.

Per stabilire una connessione socket TCP/IP, il client e il server devono seguire una serie di passaggi, noti come "*handshake*". Questo comporta lo scambio di pacchetti tra i due endpoint per stabilire e confermare la connessione. Una volta stabilita la connessione, i dati possono essere inviati e ricevuti utilizzando il socket.

I socket TCP/IP sono utilizzati in un'ampia gamma di applicazioni, dalla navigazione web alla posta elettronica al trasferimento di file. Sono una componente essenziale della comunicazione di rete e consentono ai programmi di comunicare e scambiare dati su Internet.

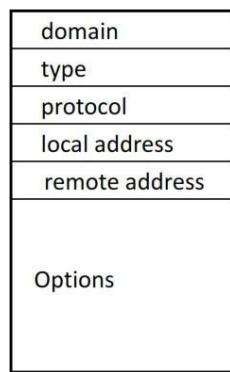


Figura 56: Struttura dei dati associata a una presa generica

Quando si utilizzano i socket TCP, gli endpoint coinvolti nella comunicazione possono assumere stati o forme diverse a seconda del loro ruolo nel processo di comunicazione. I socket TCP sono ampiamente utilizzati per comunicazioni affidabili, bidirezionali e orientate al flusso.

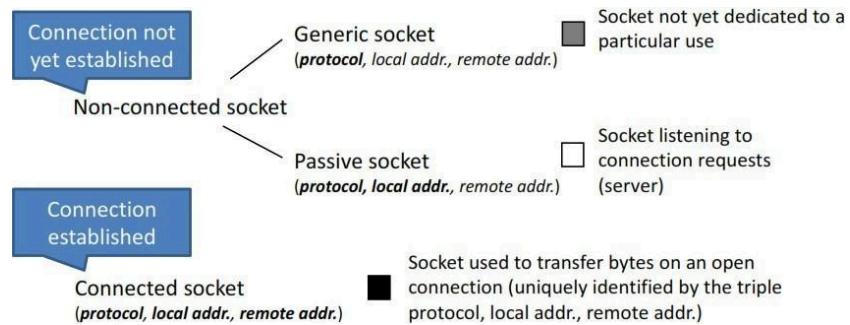


Figura 57: Prese di corrente con ruoli diversi

Quando parliamo di ruoli e di interazione con i socket, dobbiamo anche occuparci delle diverse fasi della connessione. È possibile suddividere l'interazione in tre fasi principali:

- **Handshake:** in questa fase client e server eseguono diverse azioni, quali:
 - Allocare le risorse locali per la comunicazione.
 - Specificare gli endpoint.
 - Aprire una connessione (lato client).
 - Attendere la creazione di una connessione (lato server).
- **Scambio di messaggi:** è la fase successiva e più lunga in cui:
 - I dati vengono inviati e ricevuti su una connessione (compresi i dati urgenti).
 - Il client e il server ricevono una notifica quando arrivano i dati.
- **Chiusura:** ultima fase utilizzata per:
 - Termina con grazia una connessione o interrompe una connessione.
 - Rispondere alle richieste di terminazione aggraziata e alle condizioni di interruzione.
 - Rilasciare le risorse quando una connessione termina

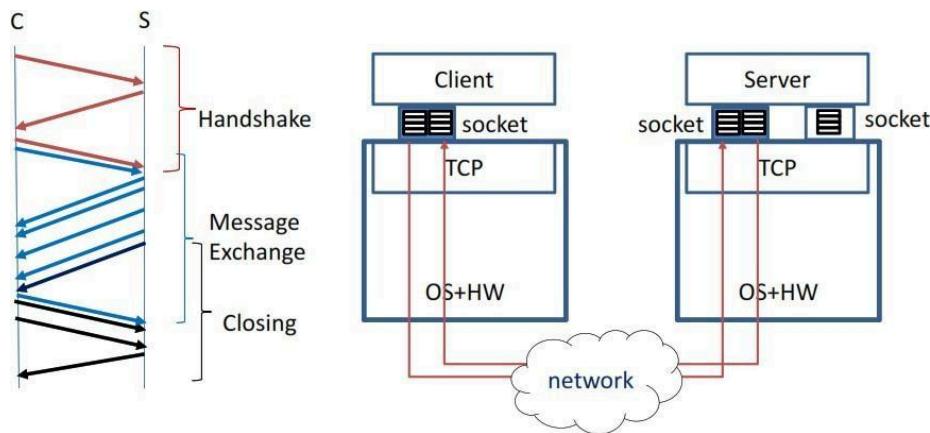


Figura 58: Prese collegate e passive

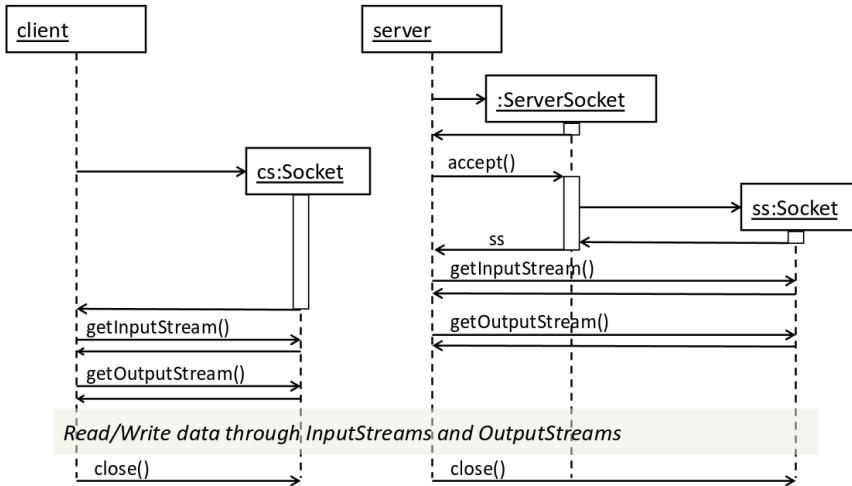
Un socket è definito da diversi parametri, tra cui il **dominio** (es. Internet IPv4 o IPv6), il **tipo** (STREAM, DGRAM) e il **protocollo** (TCP, UDP, ecc.). Ogni socket può essere configurato con opzioni specifiche per ottimizzare il comportamento, come la dimensione dei buffer per l'invio/ricezione di dati o l'attivazione di messaggi di controllo (keepalive) per verificare la vitalità della connessione.

Comunicazione tramite TCP

Per il protocollo TCP, che garantisce una connessione affidabile:

1. Il **server** crea un **ServerSocket** e resta in attesa di connessioni con il metodo **accept()**. Una volta stabilita la connessione, un oggetto **Socket** gestisce lo scambio di dati.
2. Il **client**, invece, avvia una connessione usando un oggetto **Socket**, specificando l'indirizzo del server e la porta di destinazione.

Le operazioni principali comprendono l'invio e la ricezione di dati, la gestione della terminazione della connessione e il rilascio delle risorse.



Gestione delle connessioni in Java

L'API Socket in Java, contenuta nel pacchetto **`java.net`**, replica l'API originale, focalizzandosi sui domini Internet. Include classi come **Socket** per connessioni **STREAM**, **ServerSocket** per **server passivi**, e **DatagramSocket** per la gestione di **datagrammi**. La comunicazione utilizza flussi di input e output, come **InputStream** e **OutputStream**, che semplificano la lettura e scrittura dei dati.

Blocking Operations and Timeouts

Alcune operazioni in rete sono di tipo bloccante, il che significa che il programma si ferma e aspetta che l'operazione venga completata prima di proseguire. Ad esempio, l'operazione **accept** (che accetta una connessione in ingresso), la creazione di un nuovo **Socket** (che attende di stabilire una connessione), le letture su un **InputStream** (che bloccano il programma fino a quando non arriva almeno un byte di dati) e le scritture su un **OutputStream** (che attendono che ci sia spazio sufficiente per inviare i dati) sono tutte operazioni che possono causare il blocco del programma.

Quando questo comportamento non è desiderato, una soluzione comune è eseguire queste operazioni in un thread separato, in modo che non blocchino il flusso principale dell'applicazione. Inoltre, se l'operazione di lettura su un **InputStream** rischia di durare troppo a lungo, è possibile impostare un timeout tramite l'opzione **SO_TIMEOUT** del socket, che limita il tempo massimo di attesa per i dati prima che venga sollevata un'eccezione di timeout. Questo permette di gestire meglio le situazioni in cui l'attesa potrebbe essere troppo lunga, migliorando la reattività dell'applicazione.

Server sequenziali e concorrenti

Un **server sequenziale** elabora le richieste una alla volta, processandole in ordine di arrivo (**FIFO**). Questo approccio, sebbene semplice, presenta **limiti significativi** in **termini di efficienza**: ogni richiesta deve attendere la completa elaborazione delle precedenti, anche quando la CPI è inattiva, causando **ritardi** o timeout in caso di carico elevato. Se il **tasso di richieste è elevato** (con tempi di arrivo e di servizio simili), è probabile che nuove richieste arrivino mentre il server è occupato, causando il **possibile scarto delle richieste** o il **timeout per i client**. Una soluzione è servire le richieste in modo concorrente.

Un **server concorrente** risolve questo problema servendo **più richieste contemporaneamente**. In questo modo, la probabilità che una richiesta venga scartata o che il client vada in timeout diminuisce. In Java, la concorrenza può essere implementata con thread creati su richiesta o con un pool di thread preconfigurato, gestito tramite la libreria **`java.util.concurrent`**. Ci sono diversi approcci per implementare un **server concorrente**:

- Un approccio consiste nel simulare la concorrenza all'interno di un singolo thread, creando un **nuovo thread ogni volta** che arriva una richiesta. Un esempio di server TCP concorrente con creazione di thread su richiesta prevede che un **Master Thread** accetti continuamente le connessioni e, quando una nuova connessione arriva, venga creato un **Slave Thread** per gestirla.
- Un altro approccio prevede invece l'uso di un **pool di thread pre-creati** all'avvio, pronti per

essere assegnati alle richieste non appena arrivano. In questo modo, i thread già pronti vengono assegnati alle richieste in arrivo, riducendo i tempi di risposta senza sacrificare i benefici della concorrenza.

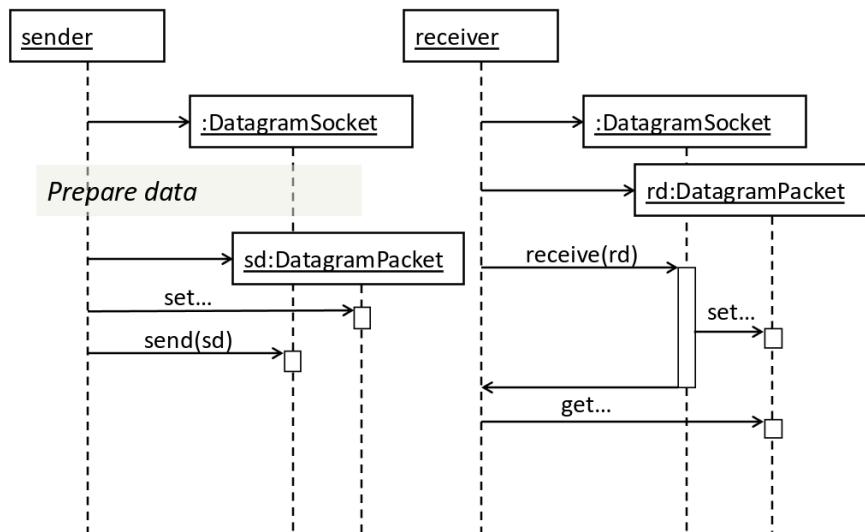
Java supporta la gestione dei pool di thread tramite gli **Executors** (nella libreria `java.util.concurrent`). Esistono diversi tipi di executor, che permettono di configurare il ciclo di vita dei thread (ad esempio, il numero massimo e minimo di thread) e di definire politiche per il riavvio dei thread che falliscono.

Comunicazione tramite UDP

A differenza di TCP, UDP è un protocollo senza connessione, adatto a trasmissioni leggere e rapide dove l'affidabilità non è prioritaria. In Java, **DatagramSocket** e **DatagramPacket** consentono l'invio e la ricezione di datagrammi. Ogni pacchetto viene trattato come un'entità indipendente, rendendo UDP ideale per applicazioni come il broadcast.

In Java, per utilizzare i socket UDP si impiegano due classi principali:

- **DatagramSocket:**
 - Viene creato con il costruttore `DatagramSocket(int localPort, InetAddress localAddr)`, che permette di specificare una porta locale e un indirizzo IP.
 - È possibile "connettere" il socket a un indirizzo remoto con il metodo `connect(InetAddress remoteAddr, int remotePort)`.
 - Il metodo `close()` distrugge il socket e rilascia le risorse.
- **DatagramPacket:**
 - Viene utilizzato per **inviare o ricevere dati**, e viene creato con `DatagramPacket(byte[] buf, int length, InetAddress destAddr, int destPort)`, dove si specificano il buffer di dati, la lunghezza, l'indirizzo di destinazione e la porta di destinazione.



Codifica dei dati

I protocolli **TCP** e **UDP** trasportano **byte**, ma la codifica e decodifica dei dati (marshalling/unmarshalling) è responsabilità del programmatore. Si possono utilizzare librerie per la codifica standard (es. JSON, XML) o classi Java come **DataInputStream**, **DataOutputStream** e **String**. È importante prestare attenzione a dettagli come l'endianness degli interi e la gestione del charset.

Perchè TCP non è abbastanza adatto per rilevare i peer process crashes?

TCP non è completamente adatto a rilevare il crash di un processo peer perché opera a livello di trasporto, garantendo una trasmissione affidabile dei dati, ma non monitora lo stato dell'applicazione del peer. Ecco i

principali motivi:

1. **TCP si concentra sull'affidabilità della connessione, non sulla vitalità del peer:** TCP assicura che i dati vengano trasmessi correttamente e in ordine, ma non monitora lo stato del processo applicativo. Se il processo del peer si arresta senza chiudere correttamente la connessione, la connessione TCP potrebbe rimanere attiva senza che il sistema rilevi il problema.
2. **Mancanza di un feedback immediato in caso di crash:** Le connessioni TCP rimangono attive finché il sistema operativo e la rete del peer sono funzionanti. Se il processo si arresta mentre il sistema operativo è ancora attivo, il socket può rimanere aperto, poiché TCP non ha visibilità sullo stato del processo applicativo. Inoltre, non esistono meccanismi integrati per monitorare continuamente lo stato del processo applicativo.
3. **Dipendenza dai timeout di TCP:** TCP utilizza meccanismi di timeout per rilevare problemi, ma non sono ottimali:
 - **Keepalive TCP:** i messaggi keepalive possono verificare se la connessione è ancora attiva, ma sono disabilitati per default e hanno intervalli di tempo lunghi.
 - **Timeout di ritrasmissione:** questi timeout si attivano solo quando sono in corso trasmissioni attive, quindi non sono efficaci durante i periodi di inattività.Questi timeout possono impiegare minuti per rilevare un crash del peer, il che li rende poco adatti in scenari che richiedono una risposta rapida.
4. **Difficoltà nel distinguere il crash del peer da altri problemi:** TCP non è in grado di distinguere tra:
 - Un crash del processo del peer.
 - Un'interruzione temporanea della rete o una partizione di rete.In entrambi i casi, la connessione TCP può diventare non responsiva, ma TCP non può determinare la causa esatta senza l'ausilio di meccanismi aggiuntivi.

Soluzione: heartbeat a livello applicativo (richieste di ping/pong)

Per superare queste limitazioni, si implementano solitamente controlli di vitalità a livello applicativo, come ad esempio:

1. L'invio periodico di messaggi di heartbeat per verificare che il peer risponda entro un determinato intervallo di tempo.
2. L'utilizzo di timeout applicativi per rilevare un possibile crash del peer nel caso in cui non arrivi una risposta in tempo.

13. Tolleranza ai guasti (Fault Tolerance)

La tolleranza ai guasti nei sistemi distribuiti si riferisce alla capacità di un sistema di continuare a fornire servizi anche in presenza di guasti parziali. L'obiettivo è garantire che il sistema sia in grado di riprendersi automaticamente dai guasti, di rilevarli e di continuare a funzionare con un impatto minimo sulle prestazioni complessive.

13.1 Guasti, errori e fallimenti

Nell'ambito della tolleranza ai guasti, vengono comunemente utilizzati i termini *guasto*, *errore* e *anomalia*. Un guasto si riferisce a un **difetto** o a una **deviazione dal comportamento** previsto di un componente del sistema e può essere **transitorio**, **intermittente** o **permanente**. Quando un guasto si manifesta e influisce sul comportamento del sistema, viene chiamato **errore**. Se un errore porta all'incapacità del sistema di svolgere la funzione richiesta, viene considerato una **failure**. Le **failure** possono essere classificate in vari tipi, tra cui **crash**, **omissioni (omission)**, **errori di temporizzazione (timing)**, **risposte errate (incorrect response)** e **comportamenti arbitrari (arbitrary)**.

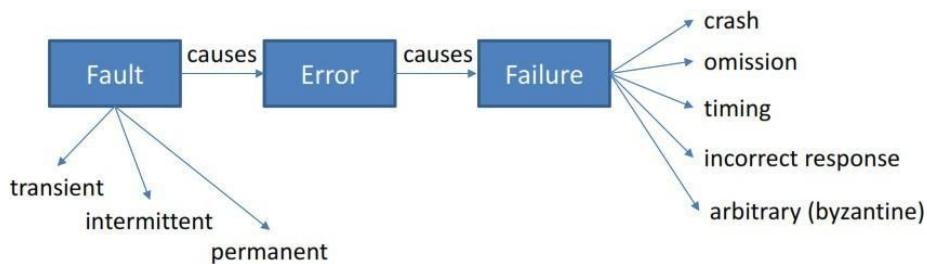


Figura 59: Concetti principali di tolleranza ai guasti

La **dipendenza** è un concetto chiave della tolleranza ai guasti e comprende diversi attributi, tra cui la **disponibilità**, l'**affidabilità**, la **sicurezza** e la **manutenibilità**. Questi attributi sono fondamentali per la costruzione di sistemi in grado di resistere ai guasti e garantire la fornitura continua di servizi. L'affidabilità può essere raggiunta attraverso una combinazione di tecniche di prevenzione dei guasti, tolleranza ai guasti, rimozione dei guasti e previsione dei guasti.

Più specificamente, la tolleranza ai guasti nei sistemi distribuiti si riferisce alla capacità del sistema di continuare a fornire servizi anche in presenza di guasti parziali, ma mira anche a garantire che le prestazioni complessive non siano seriamente compromesse e che il sistema possa recuperare automaticamente da questi guasti.

Per ottenere la tolleranza ai guasti, è necessario adottare le seguenti misure:

- **Rilevamento dei guasti:** È necessario rilevare guasti e malfunzionamenti nel sistema. Per identificare i guasti si possono utilizzare varie tecniche, come il **monitoraggio**, i **meccanismi di heartbeat** o i **meccanismi di timeout**.
- **Procedure di ripristino:** Una volta rilevato un guasto, è necessario implementare le procedure di ripristino appropriate. Queste procedure possono comprendere il **riavvio dei componenti guasti**, la **riconfigurazione del sistema** o l'**attivazione di risorse di backup**.
- **Funzionamento continuo:** Il sistema deve continuare a funzionare anche in presenza di guasti parziali, in attesa della riparazione o del ripristino. In questo modo si garantisce che il sistema possa continuare a fornire servizi, anche se eventualmente con un certo degrado delle prestazioni.

Garantendo la tolleranza ai guasti, i sistemi distribuiti possono mantenere la loro disponibilità e affidabilità, consentendo loro di resistere ai guasti e di continuare a fornire servizi agli utenti.

Failure Type	Detection Techniques	Recovery/Repair Techniques
crash	timeout	replacement, restart, redundancy
omission	timeout	redundancy (including retransmission)
timing	timeout	redundancy
incorrect response	input validation /comparison	redundancy
arbitrary	input validation /comparison	redundancy

Figura 60: Tecniche di rilevamento, recupero e riparazione

13.2 Tecniche di ridondanza

La ridondanza è un approccio fondamentale nella tolleranza ai guasti. Comporta l'uso di componenti o processi ridondanti per garantire la resilienza del sistema. La **ridondanza temporale** si ottiene ritrasmettendo i messaggi o ripetendo le operazioni fallite. La **ridondanza fisica**, invece, prevede l'aggiunta di altre apparecchiature, processi o copie di dati. Una soluzione comune per la ridondanza fisica è la replica dei processi, in cui vengono mantenute più copie di un processo.

Tipologie di Crash

I crash e le disconnessioni della rete sono situazioni comuni nei sistemi distribuiti, e la gestione dei guasti in questi scenari è cruciale per mantenere la disponibilità e l'affidabilità del sistema.

1. Crash del Processo

Il crash di un processo è uno degli eventi più comuni e può accadere per vari motivi, come errori di applicazione, errori di memoria o guasti hardware. Un crash di processo di solito porta alla terminazione immediata di una comunicazione con il peer, ma come si gestisce il crash nel contesto di una connessione TCP?

- **Rilevamento:** Quando un processo crasha, il sistema operativo (OS) rileva l'errore, e il livello TCP inizia a inviare segnali di reset (RST) ai peer connessi. Questo comportamento dipende dalla configurazione del sistema e dalla rete sottostante.
- **Risposta:** Quando il processo crasha, TCP inizia a inviare un pacchetto RST, segnalando che la connessione è stata chiusa bruscamente. I peer ricevono il pacchetto RST e terminano la connessione in modo simile. Tuttavia, questo potrebbe non essere immediatamente evidente, poiché la rete potrebbe avere un certo ritardo nel rilevare il crash.
- **Comportamento dopo il crash:** I peer connessi tentano di inviare pacchetti al processo che ha subito il crash, ma questi pacchetti non vengono mai ricevuti. In assenza di un timeout o di un altro meccanismo di gestione dei guasti, la connessione potrebbe rimanere in uno stato incoerente.

2. Crash dell'Host (con successivo riavvio)

Nel caso di un crash dell'host (ossia, il crash dell'intero sistema o server), la gestione del guasto è un po' più complessa, in quanto l'intero sistema potrebbe essere riavviato.

- **Comportamento del TCP:** Quando l'host si riavvia, TCP risponde ai pacchetti di connessione provenienti dai peer pre-crash con pacchetti RST (Reset). Questo accade perché il riavvio del sistema fa sì che il TCP stia effettivamente tentando di rispondere a pacchetti che appartenevano a una connessione pre-crash. La nuova connessione, infatti, non è più associata alla connessione precedente, e TCP risponde con un pacchetto RST per segnalare la chiusura della connessione.
- **Problemi e soluzioni:** Questo comportamento è utile nel senso che garantisce che le connessioni non stiano cercando di inviare dati a un sistema non più operativo. Tuttavia, se i peer non sono preparati a gestire i pacchetti RST, la connessione potrebbe essere persa senza che i peer siano consapevoli del guasto.

3. Crash dell'Host (senza riavvio successivo) o Disconnessione Permanente della Rete

Nel caso in cui un host crashi senza successivo riavvio o la rete venga disconnessa permanentemente (es. guasti fisici della rete, disconnessione dell'interfaccia di rete), la situazione diventa più complicata. Questo può accadere per vari motivi, come la perdita di connettività, problemi fisici sul server, o guasti hardware di rete.

- **Timeout dei peer:** I peer che cercano di comunicare con l'host crashato o disconnesso non ricevono risposte e alla fine scadono dopo un periodo di attesa (timeout). La durata di questi timeout dipende dalla configurazione di rete, dal protocollo TCP e dalle impostazioni della finestra di ricezione.
- **Timeout meccanismi personalizzati:** Sebbene il timeout TCP predefinito possa essere utile, in alcuni scenari potrebbe essere necessario implementare timeout personalizzati. Gli sviluppatori possono decidere di gestire i timeout utilizzando applicazioni di livello superiore per fornire una gestione più affidabile dei guasti. Ciò può includere l'implementazione di meccanismi come:
 - **Retry logic:** Tentativi di riconnessione automatica in caso di timeout.
 - **Messaggi di errore:** Avvisi agli utenti o al sistema che la connessione non è disponibile.
 - **Determinazione della causa:** Meccanismi per capire se il problema è legato al server, alla rete o a un guasto hardware.
- **Durata del timeout:** I timeout possono durare da pochi secondi a minuti, a seconda delle impostazioni. I timeout di rete lunghi, tuttavia, possono avere un impatto sulle prestazioni del sistema e sull'esperienza utente, specialmente nei sistemi con alta disponibilità.

Gestione dei Guasti TCP con il Socket API

Nel contesto delle applicazioni distribuite, la Socket API è uno degli strumenti principali per la gestione delle comunicazioni di rete. Con la Socket API, gli sviluppatori possono implementare diversi meccanismi di gestione dei guasti.

- **Timeout della socket:** Quando un processo non riceve una risposta da un peer, il timeout della socket entra in gioco. Se la connessione non viene ripristinata entro il periodo di timeout configurato, la connessione viene interrotta.
- **Rilevamento delle connessioni chiuse:** Quando un peer si disconnette, TCP invia un pacchetto FIN (finish), segnalando la chiusura della connessione. Se un processo cerca di inviare dati dopo che la connessione è stata chiusa, il sistema restituisce un errore.

13.3 Rilevamento dell'arresto anomalo del processo

Rilevare gli arresti anomali dei processi nei sistemi distribuiti può essere difficile a causa della natura asincrona del sistema e dell'assenza di limiti superiori ai ritardi dei messaggi. Un processo può rilevare il fallimento dell'arresto anomalo di un altro processo, spegnendosi quando non arrivano dati da quel processo. Tuttavia, i rilevamenti di crash possono essere errati o ritardati. I diversi tipi di crash, come i **fail-stop**, i **fail-silent**, i **fail-safe** e i **fail-arbitrari**, hanno diversi livelli di osservabilità e gravità (severity).

13.3.1 Esempio: Rilevamento dell'arresto anomalo di un processo connesso via TCP

In un sistema in cui i processi comunicano tramite **TCP** (Transmission Control Protocol), il protocollo stesso è in grado di rilevare eventi come la chiusura o il ripristino di una connessione da parte del peer. Questi eventi vengono notificati al processo attraverso l'**API Socket** durante operazioni di lettura o scrittura.

Tuttavia, in situazioni più critiche, come il crash dell'host remoto o disconnessioni permanenti della rete, il rilevamento dell'arresto anomalo può diventare complesso. In questi casi, il protocollo TCP potrebbe non segnalare immediatamente l'interruzione. Per affrontare tali scenari, è necessario adottare **meccanismi di timeout aggiuntivi** per individuare tempestivamente l'assenza di attività e gestire correttamente l'interruzione della connessione.

Questi meccanismi includono:

- **Timeout di lettura/scrittura:** Limita il tempo massimo per cui un'operazione di I/O può rimanere bloccata.
- **Keep-alive:** Una funzione TCP che invia periodicamente segnali per verificare la presenza del peer.
- **Timeout personalizzati:** Implementati a livello applicativo per monitorare l'inattività e prendere decisioni adeguate, come chiudere la connessione o riavviare il processo.

Questo approccio garantisce una gestione più robusta delle connessioni in ambienti TCP, riducendo il rischio di malfunzionamenti dovuti a condizioni di rete o di sistema impreviste.

	Read	Write
connection closed	returns -1	successful
connection reset	throws SocketException	throws SocketException

Figura 61: Esempio: Rilevamento dell'arresto anomalo di un processo collegato via TCP

13.4 Affidabilità attraverso la ridondanza: Gruppi di processo

Per ottenere **tolleranza ai guasti** e garantire **affidabilità (reliability)** in un sistema, si può utilizzare il concetto di **gruppi di processo**. Un gruppo di processi è un insieme di processi che lavorano insieme come se fossero un'unica entità, anche se fisicamente possono essere distribuiti su più nodi. L'obiettivo di un gruppo di processi è di rendere il sistema resistente ai guasti, in modo che se uno o più processi falliscono, gli altri possano continuare a funzionare.

Caratteristiche principali dei gruppi di processo:

1. **Gestione dell'appartenenza al gruppo:** I processi devono sapere chi fa parte del gruppo. Devono esserci meccanismi che permettono ai processi di entrare o uscire dal gruppo in modo coordinato.
2. **Leadership:** In molti casi, è necessario che un processo agisca come **coordinatore** del gruppo. Questo processo leader è responsabile di prendere decisioni cruciali e coordinare le operazioni all'interno del gruppo.
3. **Replica dei dati:** Per garantire che i dati siano disponibili anche in caso di guasto, i dati devono essere **replicati** tra i processi del gruppo. Questo assicura che ci siano copie ridondanti dei dati, pronte per essere utilizzate se un processo fallisce.

Tipi di organizzazione dei gruppi di processo:

1. **Hierarchical Organization:**
 - In una struttura gerarchica, i processi sono organizzati in livelli, con un **processo primario** che agisce come **coordinatore** principale.
 - Il processo primario ha il compito di gestire le operazioni e prendere le decisioni, mentre gli altri processi subordinati eseguono le operazioni di backup o supporto.
 - In questo caso, vengono utilizzati **Primary Based Protocols**, in cui il processo primario è responsabile di coordinare le operazioni e garantire la coerenza all'interno del gruppo.
2. **Flat Organization:**
 - In una struttura piatta, non c'è un coordinatore principale, ma tutti i processi sono trattati in modo più simmetrico.
 - In questo caso, per garantire la tolleranza ai guasti e la coerenza, vengono utilizzati protocolli più complessi, come:
 - **Replicated Write Protocols**, che assicurano che le scritture siano replicate correttamente tra i vari processi.
 - 1. **Active Replica**, dove tutte le repliche dei dati sono attive e operano contemporaneamente.
 - 2. **Quorum Based Protocols**, che richiedono che una certa maggioranza dei processi (un quorum) confermi una scrittura o una lettura prima che venga considerata valida.

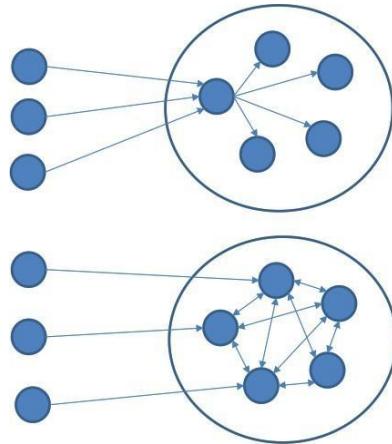


Figura 62: Organizzazioni dei gruppi di processo

13.5 Ottener la tolleranza ai guasti desiderata

Il livello di **tolleranza ai guasti** di un gruppo di processo, indicato come **k-fault tolerance**, rappresenta il numero massimo di guasti che il gruppo può sopportare senza compromettere il funzionamento. Un gruppo è k-fault tolerant può continuare a operare anche se fino a k processi falliscono.

Per ottenere una tolleranza ai guasti k , è necessario avere almeno $2k + 1$ processi nel gruppo. La scelta dei **protocolli di consenso** (come quorum o leader) dipende dal tipo di guasti previsti nel sistema, come guasti di crash, omissione o arbitrari. Protocolli più complessi sono richiesti per guasti arbitrari (Byzantine).

Failure type	#Processes needed for k-fault tolerance	Consensus protocol Examples
crash, omission, timing, network	$k+1$	flooding
+incorrect response	$2k+1$	voting, Paxos
+arbitrary (byzantine)	$3k+1$	PBFT, Blockchain

Figura 63: Ottener la tolleranza ai guasti desiderata

13.6 Il teorema CAP o Brewer's Theorem

Il teorema **CAP** (*Consistency, Availability, Partition Tolerance*) afferma che in un sistema distribuito è impossibile ottenere contemporaneamente tutte e tre le proprietà: *consistenza*, *disponibilità* e *tolleranza alle partizioni*. Tuttavia, è importante notare che il teorema CAP è un risultato teorico e non una regola assoluta che si applica a tutti i sistemi distribuiti in ogni scenario. Il motivo per cui il teorema CAP è stato oggetto di dibattito e di critiche è che presenta un modello semplificato di sistemi distribuiti, facendo determinate ipotesi e compromessi. Ecco alcuni motivi per cui alcuni sostengono che il teorema CAP non sia del tutto accurato:

- **Eccessiva semplificazione:** Il teorema CAP semplifica eccessivamente la complessità dei sistemi distribuiti considerando solo tre proprietà. I sistemi distribuiti del mondo reale hanno spesso dimensioni aggiuntive, come la **latenza**, la **scalabilità**, la **durabilità** e le **prestazioni**, che non sono affrontate esplicitamente dal teorema CAP.
- **Variazioni del mondo reale:** Il teorema CAP presuppone che le partizioni della rete siano possibili, ma non considera le variazioni e la gravità delle partizioni che possono verificarsi negli scenari reali. In pratica, sistemi diversi possono gestire le partizioni in modo diverso in base ai loro requisiti specifici e ai loro compromessi.
- **Il CAP come filosofia progettuale:** I critici sostengono che il teorema CAP sia più una filosofia di

progettazione che un teorema rigoroso. Incoraggia i progettisti di sistemi a dare priorità a determinate proprietà in base ai requisiti dell'applicazione, piuttosto che considerarle come mutuamente esclusive. In realtà, i progettisti spesso effettuano compromessi e impiegano tecniche per raggiungere un equilibrio tra le proprietà del CAP.

- **CAP nelle basi di dati distribuite:** Il teorema CAP è comunemente associato ai database distribuiti e la sua applicazione in questo contesto ha ricevuto un'attenzione significativa. Diversi sistemi di database distribuiti, come i database NewSQL e NoSQL, hanno messo in discussione ed esteso i presupposti del teorema CAP, fornendo diversi modelli di consistenza e approcci per gestire le partizioni.

È importante considerare che il teorema **CAP fornisce una base teorica per la comprensione dei compromessi** nei sistemi distribuiti, ma **non deve essere considerato come una regola assoluta** che governa tutti i sistemi distribuiti. I progettisti e gli architetti di sistemi devono valutare attentamente i requisiti, i vincoli e le proprietà desiderate quando progettano e implementano sistemi distribuiti.

14. MQTT

MQTT (MQ Telemetry Transport) è un **protocollo di trasporto di messaggistica client-server publish-subscribe**, tipicamente basato su **TCP**. È stato progettato per essere aperto, leggero e semplice, il che lo rende adatto ad ambienti con vincoli come la comunicazione IoT (Internet of Things) e M2M (Machine-to-Machine). MQTT privilegia l'**ingombro ridotto del codice**, garantisce la **semplicità** e offre una **comunicazione efficiente** su una **larghezza di banda di rete limitata** e su **dispositivi a basso consumo**. Fornisce inoltre funzionalità di **affidabilità** e **tolleranza ai guasti** per gestire canali di comunicazione inaffidabili caratterizzati da connessioni wireless, rumore e frequenti disconnessioni.

14.1 Architettura MQTT

MQTT segue un'architettura publish-subscribe, che lo distingue dai tradizionali sistemi di code di messaggi. In questa architettura, il broker consegna le notifiche in modo sincrono agli abbonati collegati. Tuttavia, MQTT consente anche di utilizzare meccanismi di conservazione/persistenza dei messaggi.

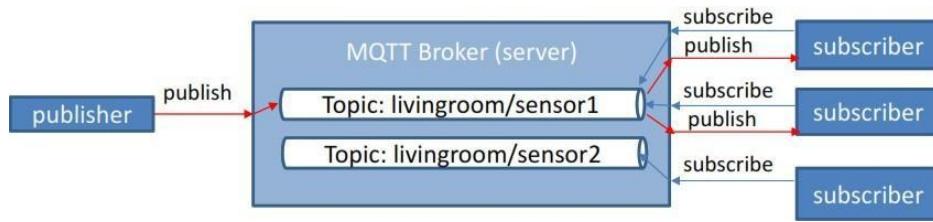


Figura 64: Architettura MQTT

14.2 Topics MQTT

Nel sistema publish-subscribe di MQTT, i client non hanno bisogno di conoscersi direttamente. Le unità indirizzabili in MQTT sono i canali chiamati **topics**. I client possono comunicare tra loro a patto che conoscano i topics condivisi. I **topics MQTT** sono identificati in modo univoco all'interno di un broker utilizzando nomi gerarchici simili a percorsi. Ad esempio, un topic può essere chiamato "appartamento1/salotto/temperatura". I topics non vengono creati o distrutti esplicitamente, esistono intrinsecamente e i client devono semplicemente farvi riferimento.

14.2.1 Topics speciali

I topics che iniziano con il simbolo \$ sono riservati e non possono essere utilizzati dalle applicazioni per i propri scopi. I topics che iniziano con \$SYS forniscono informazioni relative al sistema. Ad esempio:

- \$SYS/broker/version
- \$SYS/broker/uptime
- \$SYS/broker/clients/connected
- \$SYS/broker/clients/disconnected
- \$SYS/broker/clients/total

14.2.2 Filtri per topic

I filtri per topics sono utilizzati per fare riferimento a collezioni di topics utilizzando i caratteri jolly. MQTT supporta due tipi di caratteri jolly:

- **Jolly a livello singolo**: rappresentato dal simbolo "+". Ad esempio, "appartamento1/+/temperatura" corrisponde a topics come "appartamento1/camera_da_letto/temperatura" e "appartamento1/salotto/temperatura".

- **Carattere jolly multilivello:** rappresentato dal simbolo "#", che deve trovarsi alla fine del topic. Ad esempio, "appartamento1/#" corrisponde ad topics come "appartamento1/camera da letto/temperatura", "appartamento1/soggiorno/temperatura" e "appartamento1/cucina/umidità", catturando tutti i sottoargomenti diversi da "appartamento1".

14.3 Il protocollo MQTT

I client, sia **publisher** che **subscriber**, stabiliscono una **connessione con il broker MQTT** utilizzando una **connessione di livello 4**, tipicamente **TCP o TCP+TLS**, che fornisce un flusso di byte bidirezionale, ordinato e senza perdite. MQTT può essere implementato su TCP o TCP sicuro (TLS) per una maggiore sicurezza. Poiché le connessioni TCP possono subire guasti, un client può subire una temporanea disconnessione dal server.

Non appena viene stabilita la connessione, ha luogo un handshake MQTT di richiesta/risposta, che comprende i seguenti passaggi:

- **Identificazione e autenticazione del client:** Il client fornisce informazioni di identificazione e si autentica al broker, garantendo un accesso autorizzato alla rete MQTT. Il client decide il suo ClientID generandolo in modo univoco attraverso delle tecniche di generazione id, anche attraverso random number molto grandi.
- **Associazione con lo stato di sessione del client, se presente:** Se il client ha precedentemente stabilito una sessione con il broker, può associarsi allo stato di sessione corrispondente. In questo modo, i client possono mantenere la continuità attraverso le disconnessioni e riprendere la comunicazione senza problemi. Il flag **Clean Start** viene utilizzato durante la fase di connessione di un client al broker. Quando un client si connette al broker, può decidere se avviare una **nuova sessione** o **riprendere** una sessione precedente.
 - **Clean Start = true (1):** Quando il flag è attivato, il client richiede al broker di **inizializzare una nuova sessione** senza recuperare alcuna informazione da una sessione precedente. Se il client si disconnette e si riconnette successivamente con lo stesso **clientId**, il broker non terrà traccia della sessione precedente, e quindi non ci saranno messaggi arretrati in attesa di essere consegnati.
 - **Clean Start = false (0):** Se il flag è disabilitato, il client sta chiedendo di **riprendere la sessione precedente**, se esiste. Il broker conserverà le sottoscrizioni, i messaggi non consegnati e altri dati associati a quel **clientId**. Se il client si disconnette e si riconnette, il broker invierà i messaggi arretrati (se ci sono) e manterrà la sessione attiva. Questo comportamento è utile quando il client desidera mantenere uno stato persistente, ad esempio, per continuare a ricevere messaggi non consegnati durante la sua disconnessione.
- **Possibilità di impostare/negoziare le preferenze del client:** Durante l'handshake, i client hanno la possibilità di impostare e negoziare le preferenze con il broker. Ciò include aspetti quali i **livelli di QoS**, la **persistenza dei messaggi** e altri parametri specifici del protocollo.

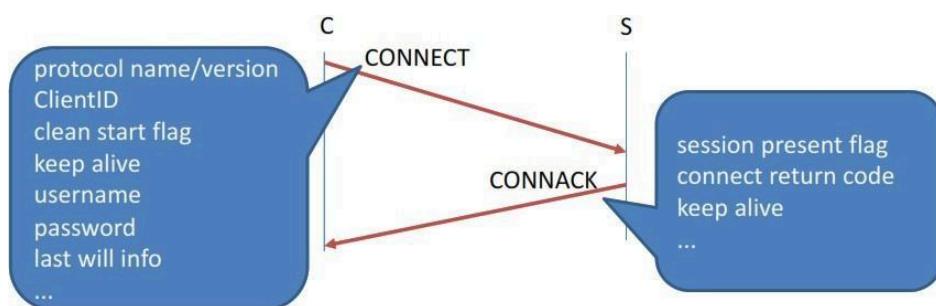


Figura 65: Handshake MQTT

14.4 Operazioni

Dopo l'handshake iniziale, un client può eseguire una sequenza di operazioni scambiando pacchetti di controllo con il broker. Le operazioni supportate da MQTT includono:

- **Publish:** I client possono pubblicare messaggi su un topic specifico, che vengono poi distribuiti agli abbonati interessati.
- **Subscribe:** I client possono iscriversi a specifici topics per ricevere i messaggi pubblicati su tali topics da altri client.
- **Unsubscribe:** I clienti possono annullare l'iscrizione ai topics precedentemente sottoscritti, non ricevendo più i messaggi pubblicati su tali topics.
- **Ping:** i client possono inviare un pacchetto di controllo Ping al broker per mantenere la vitalità della connessione e rilevare eventuali disconnessioni.

Un client può **terminare una sessione con grazia** inviando un pacchetto di controllo **Disconnect** al broker.

14.4.1 Pubblicare la qualità del servizio (QoS)

L'operazione **Publish** di MQTT supporta tre livelli di **Qualità del Servizio (QoS)**, ognuno dei quali offre diverse garanzie sulla consegna dei messaggi. Questi livelli determinano il modo in cui i messaggi vengono inviati, confermati e gestiti tra il client e il broker, con diverse implicazioni per la latenza, l'affidabilità e l'overhead.

1. Il **QoS 0** rappresenta il livello più basso di qualità del servizio (**best effort**). In questo caso, il messaggio viene inviato una sola volta **senza alcuna garanzia di consegna (fire-and-forget)**. Il mittente invia il messaggio e non attende alcuna conferma dal destinatario. Se il messaggio non arriva a destinazione, non viene ritrasmesso, rendendo questa modalità adatta per applicazioni dove la perdita di messaggi occasionali è accettabile. È il livello più veloce e con il minor overhead, ma la mancanza di conferme di ricezione implica che la consegna non è garantita.



Figura 66: QoS 0

2. Il **QoS 1** garantisce che il **messaggio venga consegnato almeno una volta**. Il mittente invia il messaggio e attende una conferma dal destinatario. Se la conferma non viene ricevuta entro un determinato **periodo di timeout**, il **mittente ritrasmette il messaggio**, il che può comportare la **ricezione di più copie dello stesso messaggio**. Questo livello di QoS è utile quando è importante assicurarsi che i messaggi vengano ricevuti, ma non è fondamentale che arrivino una sola volta. È necessario un po' più di overhead rispetto al QoS 0 a causa delle conferme di ricezione, ma offre una maggiore affidabilità.

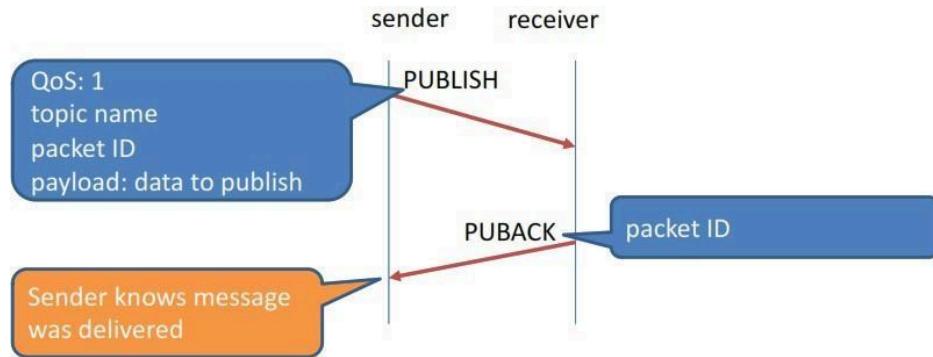


Figura 67: QoS 1

- Il **QoS 2** è il livello più alto di qualità del servizio e garantisce che ogni messaggio venga consegnato **esattamente una volta**. Questo livello include un meccanismo complesso che impedisce la duplicazione dei messaggi, attraverso una serie di scambi tra il mittente e il broker per garantire che il messaggio venga ricevuto una sola volta. QoS 2 è adatto in scenari dove è cruciale evitare duplicati, come nelle transazioni finanziarie o nelle applicazioni in cui la coerenza dei dati è fondamentale. Tuttavia, comporta un maggiore overhead e può introdurre ritardi significativi, a causa dei passaggi aggiuntivi necessari per garantire che il messaggio arrivi senza duplicazioni.

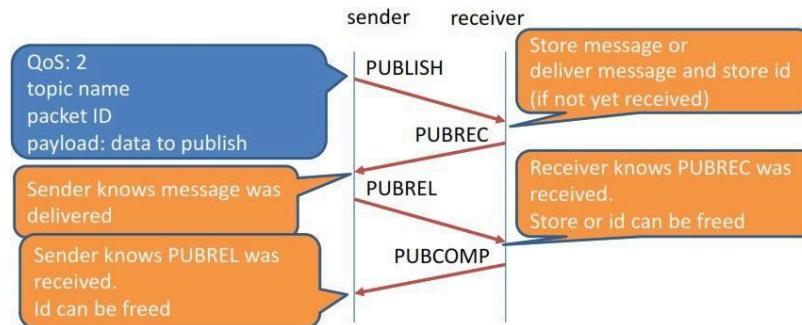


Figura 68: QoS 2

14.4.2 Subscription e annullamento della subscription

La sottoscrizione e la disiscrizione in MQTT sono operazioni fondamentali che consentono ai client di ricevere o cessare di ricevere messaggi su specifici topic. Il processo coinvolge l'invio di richieste al broker, che gestisce le sottoscrizioni in base alle politiche interne e alle capacità del sistema. Ecco una descrizione del processo:

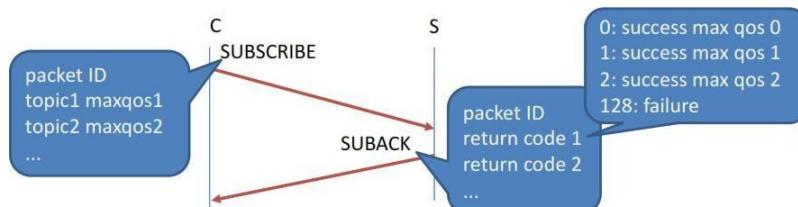


Figura 69: Subscription

- **Subscription:** Quando un client desidera ricevere messaggi pubblicati su topics specifici, invia una richiesta di sottoscrizione al broker MQTT. La richiesta di sottoscrizione include uno o più filtri per topic che definiscono i topic di interesse. Ogni filtro può anche specificare il **livello massimo di qualità del servizio** (QoS) desiderato per quel topic. I filtri per topic sono particolarmente potenti perché possono includere caratteri jolly, consentendo al client di iscriversi a una serie di topics in modo flessibile. Per esempio, un filtro come "casa/+temperatura" corrisponde a tutti i topics della forma "casa/<sala>/temperatura", dove il simbolo "+" rappresenta un singolo livello nella gerarchia dei topics.
- **Filtri per topic:** I filtri per topic sono usati per definire l'insieme dei topic a cui il client vuole iscriversi. I filtri dei topics possono includere caratteri jolly per una corrispondenza flessibile. Ad esempio, un filtro per topic come "casa/+temperatura" corrisponderebbe ad topics come "casa/salotto/temperatura" e "casa/camera da letto/temperatura", dove il carattere jolly "+" rappresenta un singolo livello nella gerarchia dei topics.
- **Concessione delle subscriptions:** Una volta ricevuta la richiesta di sottoscrizione, il broker MQTT valuta i filtri dei topics richiesti dal client. Il **broker ha la possibilità di accettare o rifiutare le sottoscrizioni in base alle proprie politiche interne e alle risorse disponibili**.
- **Riduzione della QoS:** Inoltre, il **broker può anche decidere di modificare il livello massimo di QoS** associato a ciascun filtro per topic. Ad esempio, se il broker rileva delle limitazioni dovute a risorse insufficienti o priorità QoS, potrebbe ridurre il livello di QoS richiesto dal client per garantire che i messaggi vengano comunque ricevuti, ma con un QoS inferiore a quello inizialmente richiesto.

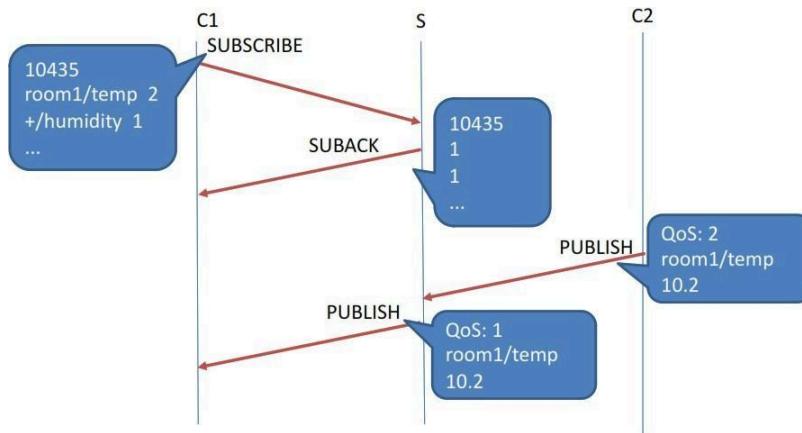


Figura 70: Esempio: Declassamento QoS

Nell'esempio, C1 decide di sottoscriversi per alcuni topic indicando il **packetID** e il **QoS per ogni topic**. Il **broker MQTT** in questo caso, secondo **proprie regole interne**, decide di fare un **downgrade del QoS** a un livello precedente. Quando C2 decide di pubblicare un messaggio sul topic room1/temp con QoS pari a 2, tra C2 e il broker MQTT verrà seguito il protocollo QoS 2 con i relativi ACK ma tra il broker MQTT e tutti i subscriber di quel topic verrà effettuato il protocollo QoS 1 come da lui deciso.

Quando un **subscriber** si iscrive a un **topic**, può specificare un **livello di QoS preferito**. Il **broker utilizza il livello di QoS più basso** tra:

1. Quello con cui il **messaggio** è stato pubblicato.
2. Quello **richiesto dal subscriber**.

Questo garantisce che i messaggi consegnati ai subscribers rispettino il livello di affidabilità desiderato.

Quando un client non desidera più ricevere messaggi su determinati topics, può inviare una **richiesta di disiscrizione al broker MQTT**. La richiesta di disiscrizione **specificata i topics** dai quali il client desidera cessare di ricevere aggiornamenti. In questo caso, il broker interrompe la consegna dei messaggi per i topics indicati nella richiesta di disiscrizione.

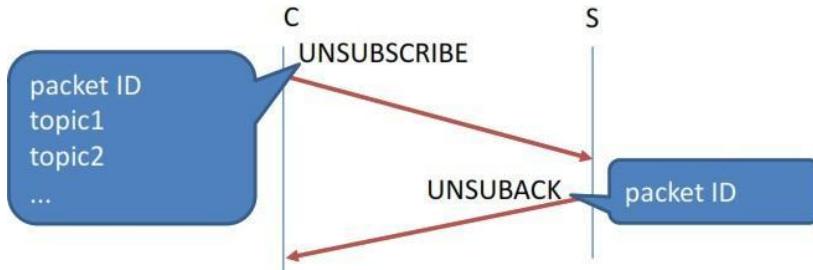


Figura 71: Disiscrizione

Il sistema di sottoscrizione e disiscrizione in MQTT offre una notevole flessibilità, poiché consente ai broker di gestire selettivamente le sottoscrizioni, modificando anche i livelli di QoS in base alle risorse e alle priorità. Questo approccio permette una gestione efficiente della comunicazione tra i client, adattandosi alle capacità e ai vincoli del broker e del sistema complessivo.

Archiviazione persistente

Il broker MQTT mantiene alcune informazioni di stato per i client iscritti disconnessi, garantendo la continuità delle loro sessioni. Queste includono:

- **Sessioni persistenti:** Il broker tiene traccia delle informazioni di sessione per i client disconnessi, consentendo loro di riprendere le sottoscrizioni e le operazioni di pubblicazione QoS 1 e QoS 2 quando si riconnettono.
- **flag RETAIN:** I subscribers possono contrassegnare i messaggi come conservati, indicando che l'ultimo valore pubblicato per un particolare topic deve essere conservato dal broker. Quando un client si iscrive a un topic con un valore conservato, il broker pubblica immediatamente il messaggio conservato al client. Il broker memorizza solo l'ultimo messaggio con flag retain arrivato, considerandolo come “**last known state**”, inoltre anche per semplicità e consistenza nella gestione dati.
- **Informazioni sullo stato del client:** Anche i client devono mantenere le proprie informazioni di stato per le operazioni di pubblicazione QoS 1 e QoS 2, per garantire una consegna affidabile dei messaggi.

14.5 Last Will (Testament):

MQTT fornisce un meccanismo chiamato "Last Will" (Ultima Volontà) o "Testament" per gestire le disconnessioni inaspettate dei client. Se un client si disconnette inaspettatamente, il broker può rilevare la disconnessione ed eseguire le ultime volontà del client. La funzione Last Will include i seguenti campi quando ci si connette al broker:

- **Topic del testamento:** Specifica il topic in cui pubblicare l'ultimo messaggio testamentario.
- **Messaggio testamentario:** Definisce il contenuto dell'ultimo messaggio testamentario.
- **QoS del testamento:** Specifica il livello QoS a cui pubblicare l'ultimo messaggio di volontà (il livello di affidabilità con cui il messaggio deve essere recapitato).
- **Flag Retain del testamento:** Indica se l'ultimo messaggio di testamento deve essere conservato dal broker. Se impostato su "true", il messaggio LWT sarà memorizzato dal broker e verrà inviato a tutti i nuovi client che si iscrivono a quel topic.

14.6 MQTT over Websocket

MQTT over WebSocket consente ai browser di agire come client MQTT, utilizzando WebSocket per trasmettere le comunicazioni. Poiché le API dei browser non permettono l'accesso diretto alle connessioni TCP, MQTT over WebSocket offre una soluzione alternativa, sfruttando la tecnologia WebSocket per consentire la comunicazione in tempo reale tra il browser e il broker MQTT.

Questa funzionalità permette di utilizzare MQTT anche in ambienti web, dove il browser agisce come client

MQTT, stabilendo una connessione attraverso WebSocket invece che tramite TCP tradizionale. I WebSocket, infatti, sono compatibili con i browser e permettono la comunicazione bidirezionale e persistente tra client e server, rendendo possibile l'adozione di MQTT in applicazioni web senza la necessità di plugin o librerie esterne.

In questo modo, i browser possono interagire in modo efficiente con il broker MQTT, consentendo l'invio e la ricezione di messaggi in tempo reale, con la stessa efficienza e affidabilità di una connessione MQTT su TCP, ma compatibile con l'ambiente web.

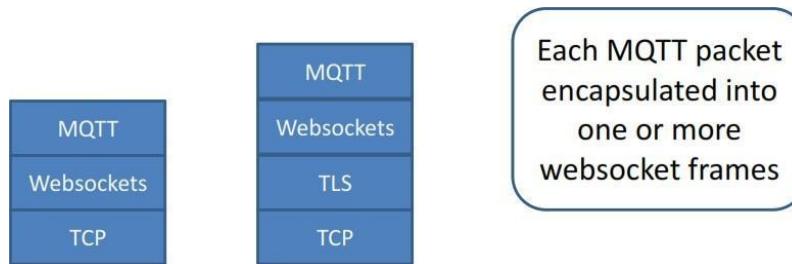


Figura 72: MQTT su Websocket

14.7 Linee guida per la progettazione dell'interfaccia Pub/Sub

Il modello Pub/Sub (Publish/Subscribe) è un'architettura di messaggistica che consente una comunicazione disaccoppiata tra i componenti di un sistema. I subscriber pubblicano messaggi (eventi o dati) a un broker, e gli abbonati ricevono questi messaggi in base alle loro sottoscrizioni. Quando si progetta un sistema Pub/Sub basato su MQTT, è fondamentale prendere in considerazione diversi aspetti per ottimizzare la sua efficienza, manutenibilità e scalabilità. Le principali linee guida sono le seguenti:

1. Livello di granularità dei messaggi e dei topics

La granularità dei topics è un elemento cruciale nella progettazione di un sistema Pub/Sub. Topics troppo generali possono portare a un sovraccarico di messaggi non necessari per gli abbonati, mentre una granularità fine consente di ridurre il carico sul broker MQTT e migliorare l'efficienza. Tuttavia, è necessario bilanciare la granularità con il livello di interattività desiderato, poiché una maggiore granularità può aumentare la complessità e i costi operativi (in termini di gestione dei topic e della connessione).

- **Granularità fine:** Consente un controllo più preciso sui messaggi, ma può aumentare il numero di topic.
- **Granularità grossolana:** Riduce il numero di topic, ma può far sì che gli abbonati ricevano più messaggi di quelli di interesse.

2. Gestione degli errori

Quando si progettano sistemi basati su MQTT, è essenziale avere una gestione strutturata degli errori. Invece di restituire messaggi generici di errore, si consiglia di definire topic e messaggi specifici per la gestione degli errori. Questo approccio facilita la gestione delle condizioni di errore, migliorando la trasparenza e la reattività del sistema.

- **Errori strutturati:** Utilizzare topic come error/system o error/device/{deviceId} per specificare il tipo di errore e la sua origine.
- **Messaggi di errore:** Includere informazioni dettagliate sul tipo di errore, il suo impatto e le possibili azioni correttive.

3. Operazioni idempotenti

Le operazioni idempotenti sono fondamentali in un sistema Pub/Sub, poiché i messaggi possono essere inviati più volte a causa di riavvii del client, errori di rete o disconnessioni. La progettazione di messaggi idempotenti garantisce che l'esecuzione ripetuta di un'operazione non causi effetti collaterali indesiderati.

- **Progettare operazioni idempotenti:** Ogni messaggio deve essere progettato in modo tale che, se ricevuto

più volte, non modifichi lo stato del sistema in modo imprevisto.

- **Stato esplicito:** Se un'operazione modifica uno stato, assicurarsi che lo stato sia facilmente identificabile e che l'operazione possa essere ripetuta senza effetti collaterali.

4. Documentazione chiara e completa

Una documentazione ben strutturata è essenziale per un sistema MQTT, in quanto aiuta gli sviluppatori a comprendere facilmente come interagire con il sistema. La documentazione deve includere dettagli sui topic, sui tipi di messaggi e sulla struttura dei dati. Inoltre, i nomi dei topics dovrebbero essere autoesplicativi per facilitare l'uso del sistema.

- **Definire i topic:** Fornire una lista completa dei topic utilizzati, con descrizioni delle loro finalità.
- **Descrivere la struttura dei messaggi:** Dettagliare la struttura dei messaggi, specificando i campi, i formati e le possibili variabili.

14.8 Linee guida specifiche per MQTT

Oltre alle linee guida generali per il design di sistemi Pub/Sub, ci sono alcune pratiche specifiche relative a MQTT che dovrebbero essere seguite per garantire compatibilità, interoperabilità e facilità di utilizzo:

1. Utilizzare caratteri ASCII nei nomi dei topic

I nomi dei topics (topic) MQTT devono essere composti da caratteri ASCII semplici per garantire la compatibilità e l'interoperabilità tra diverse piattaforme e sistemi. Utilizzare caratteri speciali o non standard può causare problemi di codifica o limitazioni in alcuni ambienti.

- **Evita caratteri non ASCII:** Limita l'uso di caratteri speciali o non standard nei nomi dei topics.
- **Usa nomi chiari e descrittivi:** I topic dovrebbero riflettere chiaramente il contenuto o l'ambito del messaggio.

2. Utilizzare i nomi dei topics come identificatori

I nomi dei topics in MQTT possono essere utilizzati come identificatori per trasmettere metadati o informazioni aggiuntive sul messaggio. Progettando una struttura di nomi dei topics ben definita, è possibile includere informazioni contestuali senza la necessità di campi o intestazioni aggiuntive nel messaggio stesso.

- **Metadati nei topic:** Ad esempio, sensor/temperature/room1 potrebbe indicare un sensore di temperatura situato in una specifica stanza.
- **Gerarchia dei topic:** Usare una struttura gerarchica nei nomi dei topics (separata da "/") per migliorare l'organizzazione e l'interpretazione.

3. Lasciare la struttura dei topics aperta all'estensione

La progettazione dei topic deve essere flessibile e facilmente estensibile, per consentire l'integrazione di nuove funzionalità e miglioramenti senza dover rivedere radicalmente l'architettura del sistema. Un design estensibile riduce il rischio di conflitti e rende il sistema più adattabile a lungo termine.

- **Mantenere l'apertura per nuove funzionalità:** La struttura dei topic deve poter evolvere, aggiungendo nuovi livelli o categorie quando necessario (ad esempio, sensor/{type}/{location}/status).
- **Versionamento:** Considerare l'uso di versioni nei topic (ad esempio, sensor/v1/temperature) per facilitare le modifiche e le migrazioni future.