Comunicazione asincrona nei sistemi distribuiti: best practices e impatto su prestazioni e affidabilità

16 Febbraio 2024 - 9 min. read

Decoupled Architectures | Microservices

Introduzione

L'asincronia è un concetto ampiamente utilizzato, ma che può portare con sè molte complessità nello sviluppo di software. Possiamo trovare asincronismo in svariati contesti, dalle applicazioni front-end ai sistemi back-end, fino ad arrivare alla comunicazione tra tra servizi in sistemi distribuiti. In base al contesto, il meccanismo con cui si costruisce l'asincronismo, le funzionalità e l'impatto sull'esperienza utente possono variare notevolmente, così come anche i possibili vantaggi e le potenziali criticità.

Ad esempio, in un'applicazione front-end, l'implementazione dell'asincronismo garantisce che gli utenti possano ricevere immediatamente feedback sulle loro azioni mentre ulteriore elaborazione avviene lato backend. Questo migliora sia l'efficienza del sistema, che la UX.

Il back-end solitamente implementa parti asincrone per ottimizzare l'utilizzo delle risorse e facilitare la gestione delle richieste simultanee. Sfruttando i meccanismi asincroni è possibile eseguire parallelamente attività computazionalmente intensive e gestire efficientemente le operazioni di input/output.

Parlando poi di sistemi distribuiti, l'utilizzo di meccanismi asincroni può presentare varie sfide, tra cui la natura inaffidabile della rete, la selezione di protocolli di messaggistica e formati di serializzazione appropriati e l'implementazione di robusti meccanismi di gestione degli errori e di retry. Un sistema distribuito con parti asincrone richiede un'attenta progettazione. Tuttavia, i vantaggi in termini di scalabilità, resilienza e tolleranza agli errori valgono gli sforzi di progettazione aggiuntivi.

In questo articolo esploreremo vari aspetti dell'implementazione della comunicazione asincrona all'interno dei sistemi distribuiti, ma tratteremo anche alcune sfumature dell'asincronia sia nel contesto del front-end che del back-end.

Esaminando in modo completo questi aspetti, puntiamo a far luce sulle complessità, sulle best practices e sulle potenziali insidie che possono sorgere durante gli sviluppi.

Asincronia nello sviluppo front-end

Cominciamo il nostro viaggio dallo sviluppo di applicazioni di front-end.

Per creare applicazioni web con una buona esperienza utente, è fondamentale implementare chiamate API asincrone e, conseguentemente, gestire le risposte asincrone.

I moderni framework di sviluppo front-end, ed anche molte librerie, offrono meccanismi robusti per la comunicazione asincrona con le API. Attraverso tecniche come AJAX e funzionalità JavaScript asincrone come promises e async/await è possibile effettuare richieste e gestirne le risposte senza bloccare il thread principale del browser.

Con questo approccio, è possibile creare interfacce reattive, che creano un'esperienza naturale per l'utente finale riducendo i ricaricamenti delle pagine e il tempo complessivo che l'utente deve attendere per l'output.

L'esperienza utente è generalmente migliore e il consumo complessivo di larghezza di banda viene ridotto, diminuendo i costi e il carico dell'infrastruttura.

Inoltre, la gestione asincrona della comunicazione con il server permette agli sviluppatori frontend di implementare aggiornamenti in tempo reale, arricchendo ulteriormente l'esperienza dell'utente. Ad esempio, sfruttando la comunicazione WebSocket o il long polling, le applicazioni web possono ricevere aggiornamenti istantanei dal backend, aumentando significativamente la gamma di funzionalità che possono essere implementate.

Le principali tecniche di programmazione asincrona per lo sviluppo frontend sono le Callback, le Promise e Async/await.

Le **Callback** sono semplicemente funzioni passate come argomenti ad altre funzioni ed eseguite successivamente, in genere al completamento di un'operazione asincrona come la ricezione della risposta ad una chiamata API.

Sebbene questo meccanismo costituisca la base fondamentale dell'asincromismo in JavaScript, e in precedenza fosse l'unico modo disponibile, può portare facilmente al

fenomeno chiamto “callback hell” e rendere il codice difficile da leggere e mantenere se annidato profondamente.

Le **Promise** rappresentano l'eventuale completamento o fallimento di un'operazione asincrona e consentono il concatenamento di più azioni asincrone. Sono ampiamente utilizzate nel JavaScript più moderno e forniscono una migliore leggibilità e gestione degli errori rispetto alle callback, consentendo di concatenare operazioni utilizzando i metodi .then() e .catch().

Le moderne keyword async/await consentono di scrivere codice asincrono utilizzando una sintassi di tipo sincrono. Usando await si può sospendere l'esecuzione finché una promessa non viene risolta o rifiutata. Async/await è particolarmente utile quando si gestiscono più operazioni asincrone che dipendono l'una dall'altra.

Asincronismo nel backend

L'adozione di componenti asincroni e della programmazione concorrente nello sviluppo backend è sempre più imporante ed è spinta dalla necessità di realizzare applicazioni semrpe più scalabili ed efficienti.

Uno dei principali vantaggi del parallelismo è la riduzione dei tempi di risposta. Ad esempio, suddividere un'attività computazionalmente intensiva in thread paralleli riduce notevolmente il tempo necessario per la risposta ottimizzando contemporaneamente l'utilizzo della CPU. La scrittura di codice concorrente può anche ottimizzare le operazioni di I/O, utilizzando flussi bufferizzati e limitando il tempo in cui il software rimane bloccato per l'I/O e quindi la quantità di cicli CPU sprecati.

La maggior parte degli application server utilizzati come interfacce per il sistema backend possono utilizzare il multi-threading o il multitasking per scalare in modo più efficace e soddisfare il crescente traffico di utenti. Questa scalabilità è essenziale per sfruttare tutta la potenza disponibile nel nodo computazionale.

Ma ci sono anche possibili svantaggi: l'implementazione di componenti asincroni e paralleli introduce ulteriore complessità, richiedendo un'attenta progettazione e un'accurata gestione degli errori. Gestire la concorrenza, la sincronizzazione e la coerenza dei dati diventa più impegnativo e potrebbe portare a bug subdoli.

Un altro aspetto da considerare è che il debug di codice fortemente parallelizzato è solitamente complesso e maggiormente soggetto a errori. Potrebbero verificarsi race condition, situazioni di stallo (deadlock) e problemi relativi alla sincronizzazione, che richiedono strategie di test complete e strumenti di debug.

La tecnica più comune è sfruttare i paradigmi e le librerie di programmazione concorrente per gestire in modo efficiente le operazioni non bloccanti, come callback, promesse o async/await in Node.js o Python con asyncio.

Un altro modo è sfruttare direttamente il threading e il multitasking in linguaggi che lo consentono, ad esempio Java, Rust o Python. Ciò comporta la generazione di thread o processi per gestire attività parallelizzabili, sfruttando così in modo efficace i processori multi-core. Esistono librerie, fornite sia dal linguaggio che da terze parti, per semplificare la gestione delle attività di basso livello, introducendo concetti come pool di thread e implementando meccanismi di sincronizzazione come i lock e i semafori per gestire le risorse condivise in modo sicuro in un ambiente parallelo.

Per attività specifiche, è anche possibile sfruttare prodotti che utilizzano framework di elaborazione distribuita come Apache Kafka, Apache Spark o Apache Flink per elaborare grandi volumi di dati su cluster distribuiti.

Sistemi distribuiti

Nell'architettura dei sistemi distribuiti, i modelli di comunicazione asincroni sono fondamentali per ottenere modularità, scalabilità e resilienza.

A livello infrastrutturale, l'asincronia viene introdotta disaccoppiando i servizi e consentendo una comunicazione senza blocchi tra di loro; l'intera soluzione acquisisce robustezza e solitamente consente un graduale degrado che mette al riparo all'indisponibilità del servizio.

Una delle motivazioni principali alla base della creazione di sistemi complessi utilizzando servizi disaccoppiati risiede in una miglior gestione della complessità. Suddividendo le applicazioni monolitiche in componenti più piccoli, possiamo semplificare lo sviluppo, facilitare l'implementazione indipendente e promuovere l'agilità in risposta ai requisiti in evoluzione. Inoltre, i servizi di disaccoppiamento possono ridurre il rischio di compromissione dell'integrità del sistema complessivo.

Vale anche la pena notare che alcune funzionalità sono naturalmente meglio sviluppate utilizzando servizi di disaccoppiamento o asincroni, quindi un'applicazione complessa può essere composta da alcuni servizi disaccoppiati per il funzionamento asincrono.

Al centro del disaccoppiamento si trova il concetto di comunicazione asincrona, che funge da fulcro per orchestrare le interazioni tra servizi distribuiti.

Esistono molti modelli asincroni diversi, come code di messaggi, sistemi di pubblicazione-sottoscrizione e architetture guidate dagli eventi. Questi forniscono le basi per la costruzione di sistemi resilienti e disaccoppiati.

Le code di messaggi, ad esempio, consentono ai servizi di comunicare in modo asincrono disaccoppiando la produzione e il consumo di messaggi. Ciò consente ai servizi di operare in modo indipendente, elaborando i messaggi secondo il proprio ritmo senza essere strettamente vincolati alla disponibilità o alla reattività di altri componenti.

Allo stesso modo, i bus degli eventi facilitano la comunicazione liberamente accoppiata consentendo ai servizi di pubblicare e iscriversi a eventi di interesse. Questo paradigma basato sugli eventi promuove flessibilità ed estensibilità, permettendo ai servizi di reagire ai cambiamenti dello stato del sistema o agli eventi esterni.

Molti vantaggi sono legati al disaccoppiamento dei servizi e possono essere ottenuti costruendo sistemi distribuiti asincroni. Il più importante è che la comunicazione asincrona **isola e mitiga i guasti a cascata** in modo che un guasto in un servizio non si propaghi necessariamente ad altri, riducendo al minimo l'impatto sull'intero sistema.

Di solito, se un sottoinsieme di microservizi non è integro, solo alcune funzioni dell'intero sistema non funzioneranno correttamente. Se l'applicazione è progettata per un graduale degrado, quindi, c'è la possibilità che l'utente non debba mai fare nulla per far fronte al problema. Ad esempio, quando si utilizza una coda per disaccoppiare un carrello dal servizio ordini, un problema temporaneo in quest'ultimo viene gestito automaticamente e l'utente finale subirà un certo ritardo prima di ricevere la conferma dell'ordine. Nel frattempo può ancora vedere il suo ordine in coda. Viene quindi preso in carico correttamente, anche se necessita ancora di essere confermato.

Un altro vantaggio significativo è che il disaccoppiamento e l'asincronia consentono ai servizi di scalare in modo indipendente, garantendo prestazioni ottimali in condizioni di traffico variabili.

Tutto ciò porta ad una migliore esperienza utente in termini di tempi di risposta più rapidi, aggiornamenti in tempo reale e interazioni più fluide.

Naturalmente, l'interazione dell'utente deve essere considerata e progettata attentamente per garantire un'esperienza fluida e piacevole, soprattutto perché le richieste vengono elaborate in modo asincrono e i loro effetti diventano visibili in seguito. Creare un'esperienza utente non frustrante richiede scelte di progettazione ponderate e una profonda comprensione delle aspettative e dei comportamenti degli utenti.

Una strategia efficace è fornire un feedback progressivo e supportare la trasparenza sullo stato delle operazioni in corso. Invece di lasciare gli utenti all'oscuro mentre le loro richieste vengono elaborate, le applicazioni dovrebbero mostrare un feedback chiaro e tempestivo, indicando che il sistema ha riconosciuto il loro input e sta lavorando attivamente per processarlo.

Ciò può essere ottenuto attraverso indicatori visivi, come spinner di caricamento, barre di avanzamento o messaggi di stato che informano gli utenti sullo stato di avanzamento delle loro richieste e raccolgono gli stati in un report di facile lettura. L'utente potrà procedere tranquillamente nella sua navigazione quando la sua richiesta sarà stata correttamente inoltrata e sarà passata in fase di elaborazione. Naturalmente, non è sempre possibile rendere la risposta indipendente e liberare l'utente, ma quando lo è, è molto meglio riconoscere la richiesta e dare all'utente un modo conveniente per verificare lo stato senza essere costretto ad aspettare.

La gestione efficace degli errori è un altro aspetto critico della progettazione di esperienze user-friendly nei sistemi asincroni. Poiché le operazioni asincrone potrebbero riscontrare errori o guasti durante l'esecuzione, è essenziale anticipare potenziali problemi e fornire agli utenti un feedback chiaro e utilizzabile quando si verificano.

Invece di presentare agli utenti messaggi di errore generici o termini tecnici, le applicazioni dovrebbero sforzarsi di comunicare gli errori in un linguaggio comprensibile all'utente che trasmetta la natura del problema e suggerisca passaggi attuabili per la risoluzione. Inoltre, fornire indicazioni contestuali o collegamenti a risorse di aiuto pertinenti può consentire agli utenti di risolvere i problemi in modo indipendente e ridurre la frustrazione.

Conclusioni

In questo articolo ci siamo tuffati nel campo dell'asincronia e del disaccoppiamento. Abbiamo discusso l'argomento a diversi livelli applicativi, arrivando ad analizzare anche l'impatto che tutto ciò può avere sull'esperienza utente.

Se avete considerazioni in merito, non esisteate a scriverci!

Nel frattempo, se volete approfondire le tecniche menzionate nell'articolo, ecco qualche link che fa per voi:

- Un tuffo nelle architetture cloud disaccoppiate con gli Event Bus
- 3 modi per disaccoppiare i tuoi microservizi: code SQS, load balancing ELB e sistema di notifiche SNS

- Disaccoppiare servizi con SQS e Lambda trigger

Ci vediamo al prossimo articolo!

About Proud2beCloud

Proud2beCloud è il blog di beSharp, APN Premier Consulting Partner italiano esperto nella progettazione, implementazione e gestione di infrastrutture Cloud complesse e servizi AWS avanzati. Prima di essere scrittori, siamo Solutions Architect che, dal 2007, lavorano quotidianamente con i servizi AWS. Siamo innovatori alla costante ricerca della soluzione più all'avanguardia per noi e per i nostri clienti. Su Proud2beCloud condividiamo regolarmente i nostri migliori spunti con chi come noi, per lavoro o per passione, lavora con il Cloud di AWS. Partecipa alla discussione!

Alessio Gandini

Cloud-native Development Line Manager @ beSharp, DevOps Engineer e AWS expert.Computer geek da quando avevo 6 anni, appassionato di informatica ed elettronica a tutto tondo. Ultimamente sto esplorando l'esperienza utente vocale e il mondo dell'IoT.Appassionato di cinema e grande consumatore di serie TV, videogiocatore della domenica.
