PROGRAMMAZIONE ORIENTATA AI MICROSERVIZI

Pattern

Tommaso Nanu
tommaso.nanu@alad.cloud

UNIVERSITÀ DI PARMA
DIPARTIMENTO DI SCIENZE MATEMATICHE, FISICHE E INFORMATICHE
Corso di Laurea in Informatica

DDD – Domain-Driven Design

DDD

- Considera i problemi come domini
- Ogni contesto delimitato è servito da un microservizio
- Curva di apprendimento molto ripida
- Organizzazione del codice allineato ai problemi aziendali
- Applicati solo su microservizi complessi (esempio, no su microservizio CRUD)
- Consente di comprendere la complessità del dominio
- Identificare e definire entità, oggetti valore e aggregazioni

DDD e microservizi

Il pattern DDD è considerato una buona scelta per la progettazione di architetture basate su microservizi.

Aiuta, infatti, a definire in modo preciso i confini dei servizi, la loro comunicazione e a gestire modelli complessi.

DDD e microservizi

- **Bounded context.** Ogni microservizio ha il proprio contesto delimitato che, di fatto, il proprio dominio specifico
- **Isolamento dei dati.** Ogni microservizio può (deve) gestire il proprio accesso ai dati e la loro persistenza in modo indipendente. Questo può comprendere l'adozione di diverse tecniche di memorizzazione/motori di database adatti per il contesto specifico (p.e. per un microservizio può essere più adatto un database relazionale, per un altro un Db NoSQL)
- **Decomposizione funzionale.** Il pattern DDD può rendere più agevole la suddivisione della problematica in microservizi in base alle funzionalità di dominio guidata dai requisiti di business

DDD e microservizi

- **Modellazione del dominio.** Ogni microservizio può utilizzare DDD per la modellazione del proprio dominio specifico, definendo entità, oggetti valore e regole di business
- **Linguaggio Ubiquo.** DDD favorisce l'adozione di un linguaggio ubiquo (condiviso) tra i team di sviluppo e analisti funzionali che migliorano la comunicazione e minimizzano i fraintendimenti sui requisiti specifici del dominio

Circuit Breaker

Problema

In un ambiente distribuito può succedere che le chiamate a risorse remote (ad altri microservizi, per esempio) non vadano a buon fine per diverse ragioni:

- Connessione lenta
- Timeout dovuti a sovraccarichi
- Server non disponibile per aggiornamenti o errori

Questi errori, in genere, si risolvono in autonomia dopo poco tempo e un'applicazione cloud affidabile deve essere in grado di gestire ciò, per esempio con una *strategia di retry*, ovvero introducendo un tempo *x*, trascorso il quale riprovare la chiamata remota. Tale tempo *x* può essere incrementale o esponenziale, a seconda del tipo di errore.

Problema

In un contesto in cui l'errore è dovuto a eventi imprevisti e la sua risoluzione potrebbe impiegare molto tempo è inutile continuare a ripetere la chiamata remota.

La soluzione migliore è che l'applicativo chiamante intercetti ed accetti l'errore ed eviti di chiamare continuamente (come conseguenza della strategia di retry) per evitare, per esempio, la creazione di un sovraccarico equivalente ad un attacco DoS (Denial of Service).

In questo contesto, il rischio è un aumento esponenziale del traffico verso il servizio non più funzionante.

È necessario mettere in piedi una sorta di barriera che interrompa le richieste in eccesso, in quanto sicuramente non andranno a buon fine.

Circuit Breaker

- Impedisce temporaneamente l'esecuzione di operazioni la cui probabilità di riuscita è molto bassa
- Evita quindi di bloccare le risorse del sistema (memoria, thread, connessioni al DB, etc..), che potrebbero esaurirsi causando quindi errori in altri microservizi
- Aumenta quindi la resilienza della nostra applicazione

Circuit Breaker

- Impedisce temporaneamente l'esecuzione di operazioni la cui probabilità di riuscita è molto bassa
- Evita quindi di bloccare le risorse del sistema (memoria, thread, connessioni al DB, etc..), che potrebbero esaurirsi causando quindi errori in altri microservizi
- Aumenta quindi la resilienza della nostra applicazione

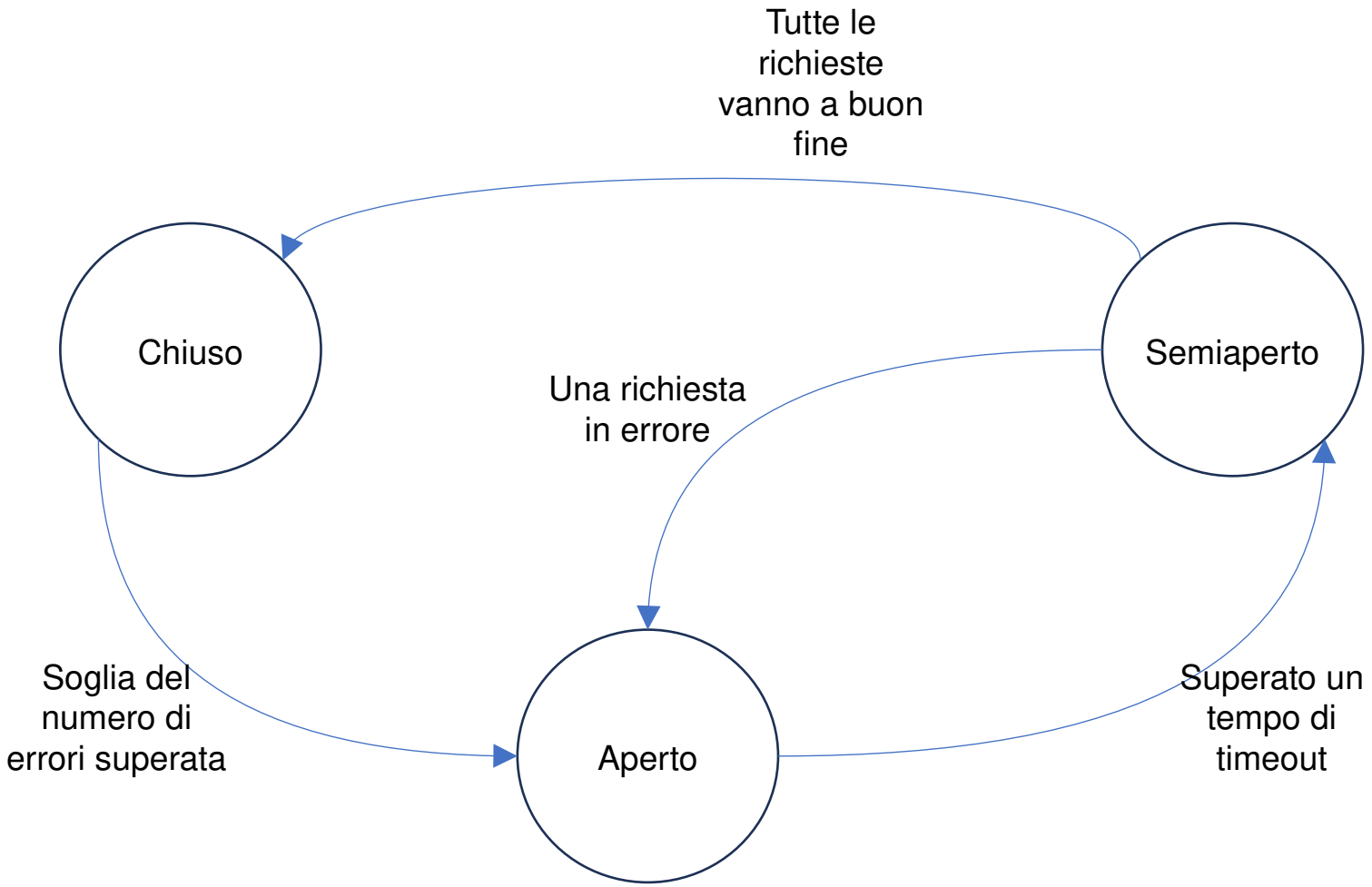
Circuit Breaker

Di fatto il nostro circuit breaker funge da proxy per le richieste, il quale dovrà monitorare il numero degli errori recenti e decidere se consentire il transito oppure bloccarlo.

Il circuito di tale proxy può essere implementato come una macchina a stati:

- ***Chiuso***: il servizio è funzionante e il proxy mantiene il conteggio del numero degli errori. Nel momento in cui tale numero supera una certa soglia il proxy transisce nello stato ***Aperto*** e avvia un timer. Allo scadere transisce nello stato ***Semiaperto***
- ***Aperto:*** la richiesta restituisce subito un esito negativo al chiamante
- ***Semiaperto:*** solo un numero limitato di richieste possono passare. Se queste poche richieste hanno tutte esito positivo, il proxy passa allo stato ***Chiuso***. Se almeno una richiesta va in errore, il proxy passa allo stato ***Aperto*** e avvia un timer; allo scadere transisce nello stato ***Semiaperto***

Circuit Breaker



Conclusioni

- Offre stabilità di sistema
- Riduce al minimo l’impatto sulle prestazioni
- Utilizzo delle informazioni di cambio stato per monitorare l’integrità del sistema

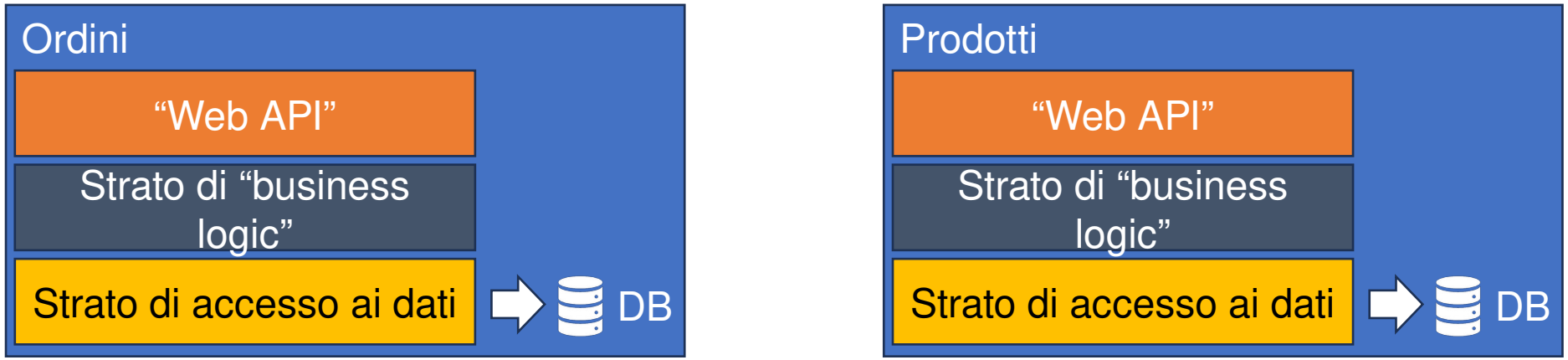
Saga Pattern

Problema

- Transazione: singola unità di lavoro che gode delle proprietà ACID
  - Atomicità: set indivisibile di operazioni, da eseguire tutte o nessuna
  - Coerenza: al termine della transazione i dati sono in uno stato coerente
  - Isolamento: transazioni simultanee non interferiscono tra loro
  - Durabilità: dati persistenti dopo il commit, anche in caso di errore di sistema

Come possiamo garantire le proprietà ACID nel caso in cui la transazione coinvolga più microservizi?

Problema



Nel momento in cui l’utente acquista un prodotto è necessario che il servizio Prodotti aggiorni la quantità disponibile.

Cosa succede se l’aggiornamento della quantità disponibile non va a buon fine? Il prodotto può essere venduto in quantità superiore a quella effettivamente presente in magazzino. Questo è un problema.

Transazioni distribuite

Strumento che ci consente di eseguire tante operazioni atomiche distinte che coinvolgono più microservizi come fosse un'unica transazione.

Si pone due obiettivi:

1. Garanzia delle proprietà ACID
2. Isolamento dei dati: prima del termine della transazione distribuita, l'insieme dei dati coinvolti devono essere accessibili da parte di un altro microservizio non coinvolto?

2PC – Two-phase commit

Il protocollo 2PC (two-phase commit) è un modello ampiamente utilizzato per l'implementazione delle transazioni distribuite.

È presente un coordinatore centrale, responsabile del controllo delle transazioni locali.

1. **Prima fase**: il coordinatore chiede ad ogni partecipante se possono procedere con il commit della loro transazione locale.
2. **Fase di commit**: attivata quando tutti i partecipanti restituiscono Sì, il coordinatore chiede a tutti di eseguire il commit; se almeno uno risponde negativamente, il coordinatore invita tutti ad eseguire una rollback.

Problemi

- Presenza di *Single point of failure*, nel coordinatore
- Il coordinatore dovrà attendere la risposta da tutti i partecipanti: il microservizio più lento detterà le tempistiche di avanzamento globale della transazione distribuita
- Non supportato nei DBMS NoSQL

Saga Pattern

- Modello alternativo per la gestione delle transazioni distribuite
- Ogni partecipante eseguirà una operazione di avanzamento locale (chiamata «pivot transaction»)
- Ogni operazione eseguita all'interno della saga potrà essere ripristinata da una operazione di compensazione (chiamata «compensating transaction»)
- Garantisce che tutte le operazioni di avanzamento vengano completate correttamente, oppure che vengano annullate grazie alle rispettive operazioni di compensamento, mantenendo quindi coerente lo stato globale del sistema
- Le operazioni di compensazione sono idempotenti e riprovabili: ciò ci dà la garanzia di una gestione automatica delle transazioni

Saga Pattern

- Sequenza di transazioni locali, eseguite da ogni partecipante alla saga
- Il partecipante i-esimo aggiorna il proprio database locale e pubblica un messaggio per attivare la transazione locale successiva
- Se la transazione i-esima ha esito negativo, la saga esegue una serie di transazioni di compensazione per annullare il lavoro svolto precedentemente dagli altri partecipanti

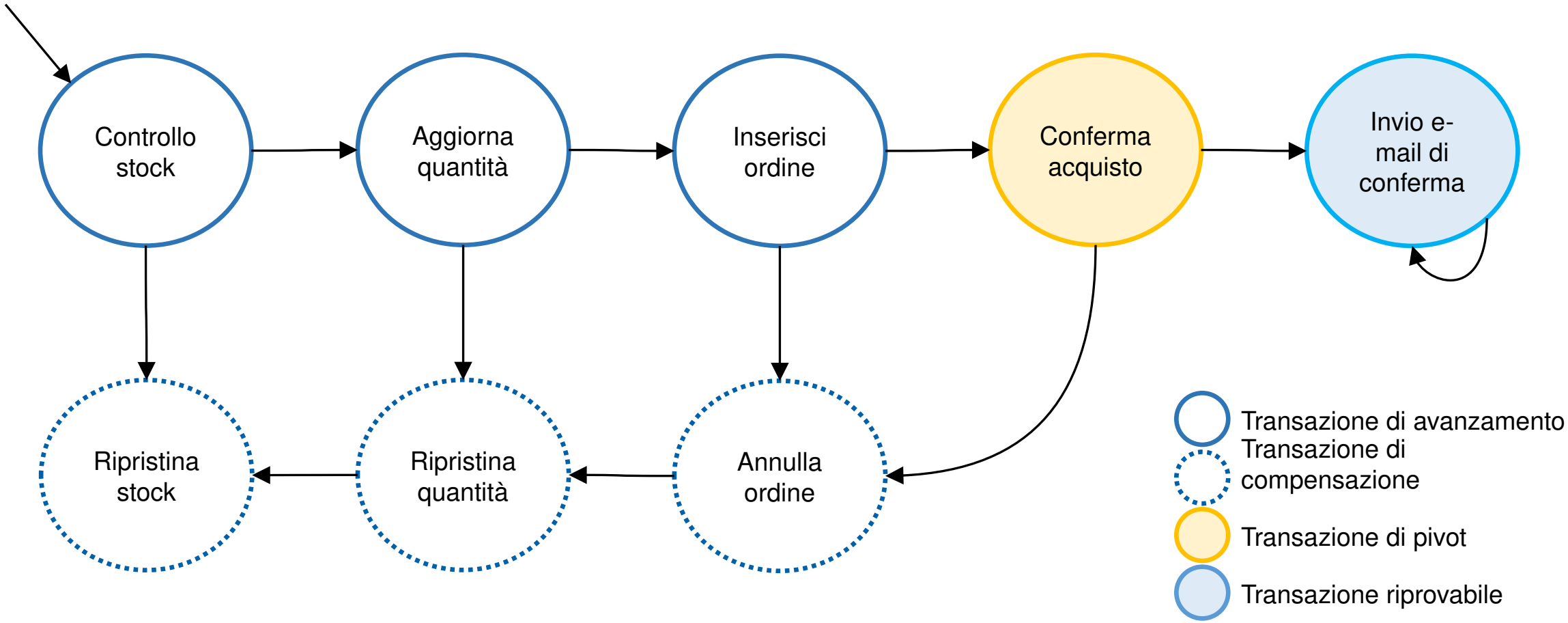
Saga Pattern

- Transazioni di avanzamento: sono quelle transazioni locali che permettono l'avanzamento della saga, prevedono sempre una transazione di compensazione per annullare l'operazione

- Transazioni pivot: al più una, se va a buon fine la saga completerà il suo avanzamento con successo. Può essere una transazione non compensabile né ripetibile oppure l'ultima transazione compensabile oppure la prima transazione ripetibile della saga

- Transazioni riprovabili: segue la transazione pivot, opzionali, e offrono garanzia di esito positivo. Possono essere ripetute più volte

Saga Pattern



Saga Pattern

Saga Pattern

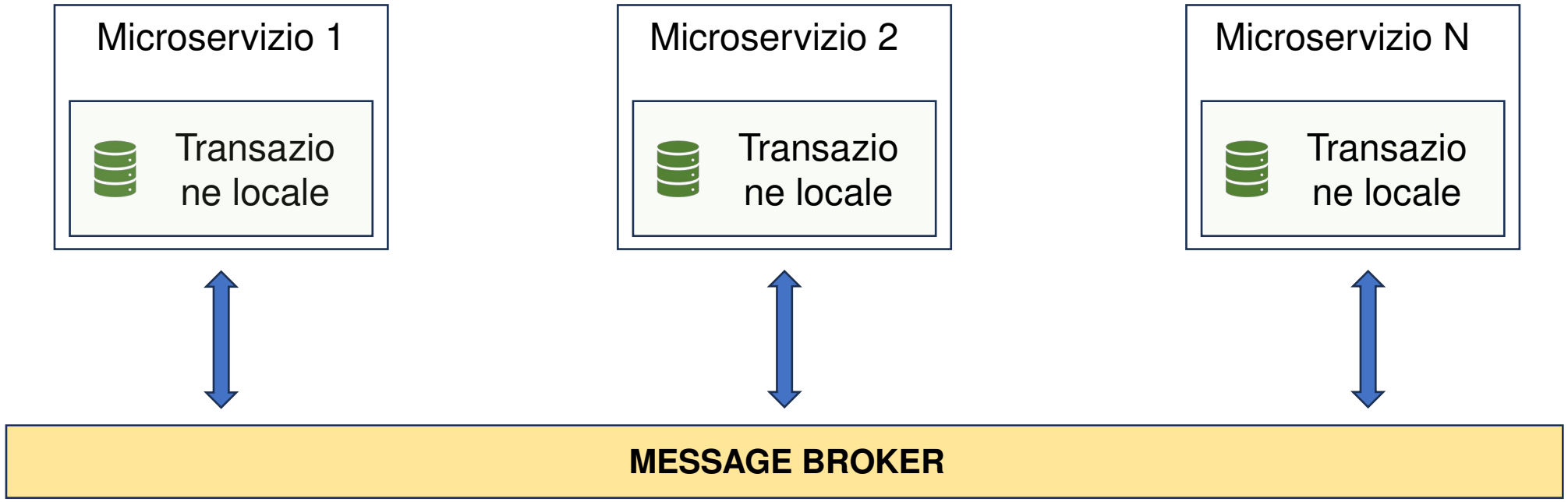
Due possibili approcci implementativi della saga:

- **Coreografia**: nessun coordinatore, i microservizi collaborano per portare a conclusione la saga
- **Orchestrazione**: controllo della saga centralizzato da parte di un orchestratore

Coreografia

- Tutti i microservizi coinvolti collaborano attraverso lo scambio di messaggi (o trigger di eventi), indispensabili per l'avanzamento della saga, in assenza di un punto di controllo centralizzato
- Ogni microservizio sa esattamente quali operazioni fare ed in risposta a quale evento farle
- Tipicamente è presente un Message Broker per lo scambio dei messaggi

Coreografia



Vantaggi

- Non esiste una forte dipendenza con un componente centrale, ovvero non abbiamo un ***Single Point of Failure***, le responsabilità sono distribuite tra i partecipanti alla saga

- Semplice e veloce da sviluppare, in contesti con poche transazioni che coinvolgono un numero ristretto di microservizi

Svantaggi

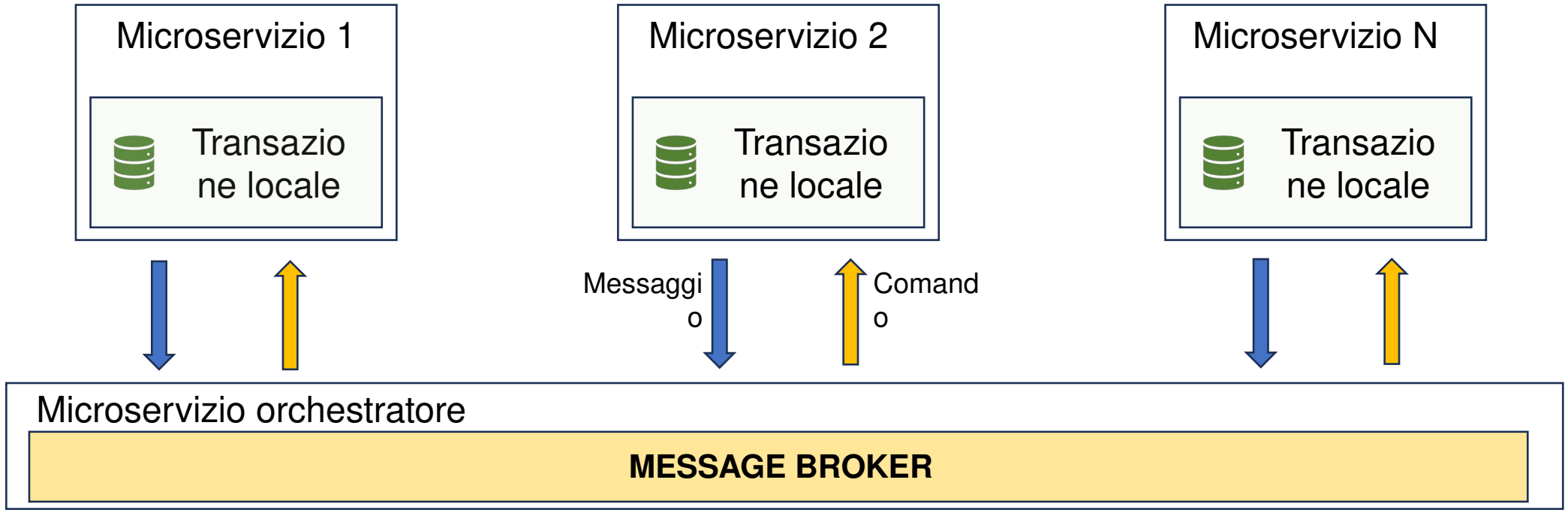
- La complessità cresce velocemente aggiungendo nuovi microservizi
- Flusso di lavoro confuso in quanto è più difficile capire quali servizi gestiscono eventi specifici
- Rischio di presenza di cicli tra le dipendenze di una saga
- Necessaria una fase di organizzazione tra i microservizi coinvolti nella saga, con l'ausilio, per esempio, dell'analista funzionale

Orchestrazione

- Un servizio complementare gestisce e controlla tutta la saga, definito per l'appunto l'orchestratore

- Tipicamente è presente un Message Broker utilizzato dall'orchestratore per comandare i microservizi e dai microservizi per comunicare lo stato delle operazioni

Orchestrazione



Architettura dei microservizi

Vantaggi

- Punto centrale che fornisce una visione di tutta la saga
- Adatto in contesti complessi con molti partecipanti
- Aiuta ad evitare cicli tra i microservizi della saga
- La logica di business è centralizzata in un unico punto
  - Modifica più semplice
  - L'aggiunta di nuovi servizi comporta la sola modifica dell'orchestratore
- Incline ad uno sviluppo *Agile:* ogni gruppo di lavoro può essere autonomo

Svantaggi

- Presenza di un ***Single Point of Failure***, ovvero se va giù l'orchestratore, tutte le saghe saranno ferme

- È necessario implementare un nuovo servizio

- Il numero di interazioni, confrontate con l'approccio a Coreografia, è esattamente doppio: il messaggio per comunicare con l'orchestratore e il messaggio (comando) da inviare al servizio

Coreografia vs Orchestrazione

Una risposta possiamo ottenerla rispondendo a queste domande:

- Quante transazioni distribuite dobbiamo gestire?
- Quanti microservizi saranno coinvolti?
- Quanto velocemente crescerà la nostra applicazione?
- Quanti microservizi nasceranno nel tempo?

La risposta comune a tutte queste domande è:

In un piccolo contesto, con pochi microservizi e poche transazioni distribuite, preferiamo un approccio a coreografia, altrimenti ad orchestratore.

MA non esiste una risposta univoca, dipende dal contesto della nostra applicazione.

Grazie a tutti