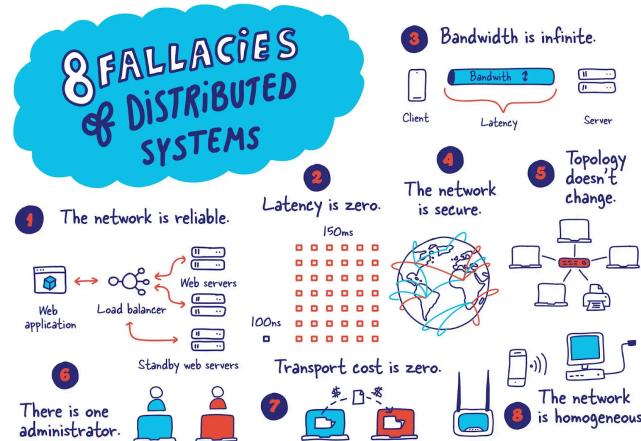


Comunicazione

Fallacie del Calcolo Distribuito

Il calcolo distribuito introduce specifiche assunzioni che possono portare a malintesi quando si progettano e implementano sistemi. Di seguito sono elencate alcune delle principali fallacie del calcolo distribuito:



Network is Reliable

Questa fallacia **assume che le connessioni di rete siano sempre stabili, senza interruzioni**.

Tuttavia, le reti possono fallire a causa di problemi hardware, errori di configurazione o guasti transitori. È fondamentale che le applicazioni gestiscano potenziali guasti di rete in modo elegante.

```
@RestController
public class NetworkReliabilityController {

    @PostMapping("/reliable")
    public ResponseEntity<String> reliableService(@RequestBody String data) {
        NetworkService remoteService = new NetworkService();
        String response = remoteService.process(data);
        return ResponseEntity.ok(response);
    }
}
```

Questa scrittura in realtà è poco flessibile: problema della gestione degli errori, generazione dei log, problema del sovraccarico della rete per riuscire a risolvere la richiesta. – RICORDARE CHE LA CHIAMATA è REMOTA – In alternativa, l’uso dell’iniezione di dipendenze consente una migliore testabilità e una chiara separazione delle preoccupazioni:

```
@RestController
public class NetworkReliabilityController {
    private final NetworkService remoteService;

    public NetworkReliabilityController(NetworkService remoteService) {
        this.remoteService = remoteService;
    }

    @PostMapping("/reliable")
    public ResponseEntity<String> reliableService(@RequestBody String data) {
        String response = remoteService.process(data);
        return ResponseEntity.ok(response);
    }
}
```

Gestire l'HttpTimeoutException può essere complesso: - Il servizio remoto sta ancora elaborando (timeout legittimo)? - I dati sono effettivamente arrivati al servizio remoto? - Dobbiamo mostrare all'utente un errore, registrarlo o riprovare?

La Latency è Zero

Questa fallacia assume che la comunicazione tra i servizi avvenga istantaneamente, trascurando che **le richieste e le risposte richiedono tempo**.

Diverse operazioni comportano latenze variabili, e questa variabilità può influire significativamente sulle prestazioni.

| Processo | Durata | Normalizzata |
|------------------------------------|--------|--------------|
| 1 ciclo CPU | 0.3ns | 1s |
| Accesso cache L1 | 1ns | 3s |
| Accesso cache L2 | 3ns | 9s |
| Accesso cache L3 | 13ns | 43s |
| Accesso DRAM (dalla CPU) | 120ns | 6min |
| SSD I/O | 0.1ms | 4days |
| HDD I/O | 1-10ms | 1-12 mesi |
| Internet: San Francisco a New York | 40ms | 4 anni |
| Internet: San Francisco a Londra | 80ms | 8 anni |
| Internet: San Francisco a Sydney | 130ms | 13 anni |
| Ritrasmissione TCP | 1s | 100 anni |
| Riavvio del container | 4s | 400 anni |

La Larghezza di Banda è Infinita

Questa fallacia **assume che la rete possa gestire qualsiasi quantità di dati inviati o ricevuti senza degrado delle prestazioni**.

In realtà, la larghezza di banda è limitata e superare questo limite può portare a problemi di prestazioni.

- 1 Gbps = 128MBps
- Dopo aver dedotto l'overhead TCP/IP: 64MBps
- Dopo aver dedotto l'overhead di serializzazione: 32MBps

Con l'avvicinarsi di questo limite (ad esempio, aggiungendo più macchine sulla stessa rete), i ritrasmetti TCP diventano più frequenti, aumentando la latenza e influenzando l'affidabilità complessiva.

La Rete è Sicura

Questa fallacia **assume che tutte le comunicazioni sulla rete siano intrinsecamente sicure**.

La sicurezza deve essere progettata nelle comunicazioni di rete per proteggere contro intercettazioni, manomissioni e accessi non autorizzati. Gli attacchi di ingegneria sociale possono sfruttare la complessità nelle organizzazioni.

Ad esempio, un attaccante potrebbe trovare admin DBA su LinkedIn e manipolarli per trasferire dati sensibili senza adeguate misure di sicurezza.

La Topologia Non Cambia

Questa fallacia **assume che la struttura della rete rimanga costante**.

In realtà, i sistemi distribuiti possono cambiare a causa di scalabilità, guasti o riconfigurazione, influenzando la scoperta e la disponibilità dei servizi. Ad esempio, la dipendenza dai metodi di callback può introdurre problemi di affidabilità se le comunicazioni di rete vengono interrotte, portando a potenziali fallimenti del sistema a causa di tempi di attesa prolungati.

C'è Un Solo Amministratore

Questa fallacia **assume che una sola autorità gestisca tutti gli aspetti del sistema distribuito**.

Anche se ciò può essere vero in piccoli ambienti, con l'espansione dei team e delle responsabilità, il coordinamento diventa essenziale per evitare modifiche conflittuali, soprattutto durante aggiornamenti o cambiamenti di configurazione.

- **Tutto andrà liscio se più amministratori distribuiscono aggiornamenti contemporaneamente?**
- **Chi comprende gli impatti di specifiche configurazioni quando molti sviluppatori apportano modifiche?**

Investire tempo in configurazioni chiare e ben documentate è fondamentale per mantenere l'integrità del sistema.

Il Costo di Trasporto è Zero

This fallacy assumes that transferring data across the network incurs no cost, neglecting the reality that data transfer can lead to operational costs, and resource usage, especially when dealing with large volumes of data.

- Network hardware has upfront and ongoing costs
- Serialization before crossing the network (and deserialization on the other side) takes time.
- In the cloud, It can be a big cost factor (e.g., 70K\$ cloud bill at the end of the month)

The Network is Homogeneous

Tale errore **presuppone che tutti i componenti della rete siano simili, trascurando il fatto che i diversi servizi possono essere implementati in linguaggi di programmazione diversi, eseguiti su piattaforme diverse o funzionare con configurazioni diverse.**

In passato, l'interoperabilità tra .NET e Java era un processo relativamente semplice (2005).

Il panorama tecnologico attuale include Go, Rust, Python, MongoDB, Cassandra e sistemi liberamente integrati su HTTP (ad esempio, REST), con conseguenti sfide quali:

- serializzatori/deserializzatori JSON incompatibili;
- passaggio da modelli di dati relazionali a NoSQL.

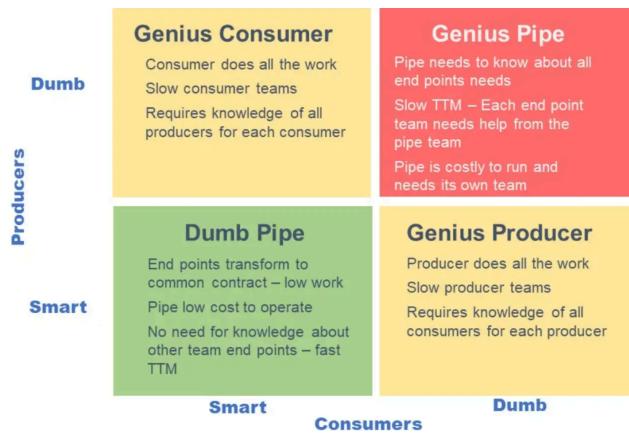
Si prevede un'ulteriore peggioramento prima di una possibile ripresa.

Stili di comunicazione

La sfida più grande nel passare da un'applicazione monolitica a un'applicazione basata su microservizi risiede nel cambiare il meccanismo di comunicazione. Una conversione diretta da chiamate di metodo in processo a chiamate RPC a servizi causerà una comunicazione eccessiva e non efficiente che non funzionerà bene in ambienti distribuiti.

Le sfide di progettazione dei sistemi distribuiti sono così famose che esiste persino un canone noto come le [Fallacie del Calcolo Distribuito](#), che **elena assunzioni sbagliate che gli sviluppatori fanno spesso quando si spostano da design monolitici a design distribuiti.**

Smart Endpoints Dumb Pipes



La comunità dei microservizi promuove la filosofia degli smart endpoints and dumb pipes secondo cui:

- La logica di business è gestita a livello di servizio (endpoint intelligenti), mentre la comunicazione tra i servizi è mantenuta semplice (tubi stupidi).
- Questo approccio **evita middleware complessi (come gli ESB nella SOA tradizionale)** rendendo i tubi **esclusivamente responsabili per il trasporto dei messaggi**, non per la trasformazione o l'orchestrazione.

Il risultato è un accoppiamento debole, una migliore scalabilità e resilienza, poiché i servizi possono evolversi indipendentemente senza dipendere da uno strato di comunicazione complesso.

Tassonomia

Le interazioni client-servizio formano la spina dorsale dei sistemi distribuiti e possono essere categorizzate lungo due dimensioni chiave:

- **la relazione tra il client e il servizio** → *uno-a-uno vs. uno-a-molti*
- **la natura del tempo di risposta** → *sincrono vs. asincrono*

Relazione tra il client e il servizio

Uno-a-Uno: In questo stile di interazione, ogni richiesta del client è indirizzata a un servizio specifico e la risposta proviene dallo stesso servizio. Questo approccio è semplice e semplifica la relazione tra client e servizi, ma può portare a un **accoppiamento stretto**.

- **Esempio:** Un utente richiede le informazioni del proprio conto a un servizio bancario; la richiesta è elaborata dal servizio conti, che recupera e invia i dati pertinenti.

Uno-a-Molti: In questo modello, **una singola richiesta del client può invocare più servizi**. Questo è utile in scenari in cui più servizi devono collaborare per soddisfare una richiesta. Questo approccio consente una maggiore flessibilità e scalabilità, poiché differenti servizi possono essere aggiornati o sostituiti in modo indipendente.

- **Esempio:** Una richiesta per un itinerario di viaggio potrebbe essere inviata a un servizio che interagisce con più servizi: un servizio di prenotazione voli, un servizio di prenotazione hotel e un servizio di noleggio auto.

Tempo di risposta

Sincrono: Nelle interazioni sincrone, **il client invia una richiesta e attende una risposta, bloccando spesso ulteriori azioni fino a quando non riceve la risposta**. Questo modello è intuitivo ma può portare a ritardi se il servizio impiega tempo a elaborare la richiesta.

- **Esempio:** Un client invia un modulo su un sito web e attende che il server confermi l'invio prima di continuare.

Questo accoppiamento stretto può portare a colli di bottiglia nelle prestazioni se il servizio è lento a rispondere.

Asincrono: Le interazioni asincrone consentono al client di continuare a elaborare altre attività senza attendere una risposta. Il client può ricevere la risposta in un secondo momento o essere informato quando la risposta è pronta.

Questo approccio non bloccante migliora la reattività e l'esperienza utente, ma può complicare la gestione degli errori e la gestione dello stato.

- **Esempio:** Un utente carica un grande file su un servizio di archiviazione cloud. Invece di aspettare il completamento del caricamento, l'utente può continuare a lavorare, e il servizio lo informa quando il caricamento è terminato.

Tipi di interazione

Richiesta/Risposta (Uno-a-Uno, Sincrono): In questo stile classico di interazione, un client invia una richiesta a un servizio e attende una risposta diretta. L'aspettativa è di avere una risposta tempestiva, il che può portare a un accoppiamento stretto tra il client e il servizio. Questo modello è comune nelle applicazioni web tradizionali e nelle API dove è richiesto un feedback immediato.

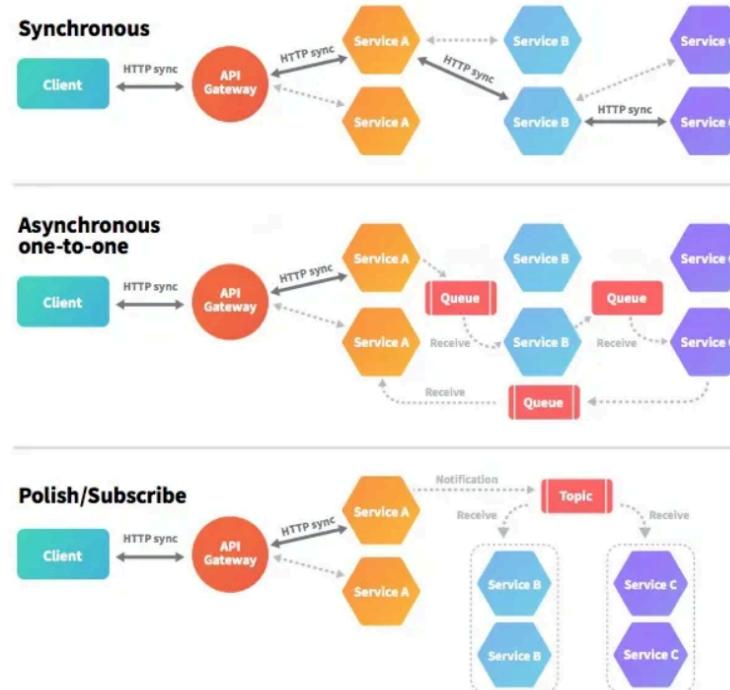
Richiesta/Risposta Asincrona (Uno-a-Uno, Asincrono): In questa interazione, un client invia una richiesta ma non attende una risposta immediata. Invece, può continuare a elaborare altre attività. Il servizio risponderà eventualmente, ma il client non si blocca mentre attende. Questo modello aiuta a migliorare l'esperienza utente prevenendo freeze o ritardi dell'interfaccia utente.

Publish/Subscribe (Uno-a-Molti, Asincrono): Questo modello di interazione consente a un client di pubblicare messaggi su un argomento o canale che possono essere consumati da più servizi. I servizi interessati si iscrivono a questi messaggi, che possono essere elaborati in modo indipendente dal pubblicatore.

Questo disaccoppia i client dai servizi, consentendo un'architettura più flessibile.

Publish/Subscribe Asincrono (Uno-a-Molti, Asincrono): In questa variazione, un client pubblica un messaggio di richiesta a più servizi e attende risposte per un periodo di tempo specificato.

Questo consente al client di ricevere input da vari servizi senza bloccare la propria operazione, aumentando l'efficienza.



Comunicazioni Sincrone

Definizioni e Problemi Conosciuti

La serializzazione è il processo di conversione di strutture dati o oggetti in un formato che può essere facilmente trasmesso o archiviato. Questo formato è spesso uno stream di byte o testo, che può essere successivamente deserializzato (convertito nuovamente) nella struttura dati o nell'oggetto originale. Questo concetto può portare ad una maggiore latenza nella risoluzione della richiesta.

```
@RestController
@RequestMapping("/api")
public class UserController {

    @PostMapping("/user")
    public Response createUser(@RequestBody User user) {
        return new Response("Utente " + user.getFirstName() + " registrato con successo!", true);
    }
}
```

⚠ Il concetto di accoppiamento spaziale si riferisce al grado di dipendenza tra diversi componenti o servizi in un sistema in un dato momento.

Un alto grado di accoppiamento spaziale significa che i componenti sono strettamente connessi, richiedendo una conoscenza diretta dell'esistenza, delle interfacce o delle posizioni reciproche. Ciò può portare a una minore flessibilità e a una maggiore complessità di manutenzione. ⇒ La modifica di un componente comporta la modifica di tutti i componenti da esso dipendenti

```
public class Payment {
    private double amount;
    private String paymentMethod;
    private boolean isApproved;
}

@RestController
public class OrderController {

    public Order createOrder(Order order) {
        // Chiamata diretta alla posizione del servizio di pagamento, porta, endpoint (accoppiamento stretto)
        String url = "http://payment-service:8080/api/payments";
        RestClient restClient = RestClient.builder().build();
        Payment payment = restClient.get()
            .uri(url)
            .retrieve()
            .body(new ParameterizedTypeReference<>() {});
        // ...
    }
}
```

⚠ Il concetto di accoppiamento temporale si verifica quando i componenti o i servizi devono essere disponibili e reattivi nello stesso momento per funzionare correttamente. ⇒ Una delle due piaghe irrisolvibili nella comunicazione sincrona.

Questo accade spesso nei modelli di comunicazione sincrona, dove un componente deve attendere che un altro elabori una richiesta prima di procedere. Nei casi in cui una catena di più servizi deve comunicare, **la latenza accumulata può degradare significativamente le prestazioni. [non risolvibile con approcci sincroni!]**

Accoppiamento API si riferisce al grado di dipendenza tra un client e un'API.

Un'API altamente accoppiata significa che le modifiche nell'API possono facilmente rompere il client, mentre un'API a basso accoppiamento fornisce maggiore flessibilità e resilienza ai cambiamenti.

Cosa succede se dobbiamo modificare la classe *Payment*?

```
public class Payment {
    private double amount;
    private String currency;
    private String paymentMethod;
    private boolean isApproved;
}
```

Over-fetching: si verifica quando **un'API restituisce più dati di quanto il client necessiti effettivamente, portando a un utilizzo inefficiente della banda e a un tempo di elaborazione aumentato**. Questo avviene tipicamente nelle API REST con strutture di risposta fisse, dove un client non può specificare esattamente quali campi richiede.

Scenario: Un client desidera solo il titolo e l'autore di un libro, ma l'API restituisce l'intero oggetto libro, inclusi campi non necessari come ISBN, descrizione, editore, ecc.

```
public class Book {
    private String title;
    private String author;
    private String isbn;
    private String description;
    private String publisher;
    private int pages;

    // ...
}

@RestController
@RequestMapping("/books")
public class BookController {

    @GetMapping
    public Iterable<Book> findAll() {
        return List.of(
            new Book("Spring Boot", "John Doe", "123456789", "Guida completa", "TechPub", 500),
            new Book("Microservizi", "Jane Smith", "987654321", "Spiegazione dettagliata", "CloudPub", 300)
        );
    }
}
```

Under-fetching (noto anche come chattiness): si verifica quando **un client richiede dati da un'API ma non riceve tutte le informazioni necessarie in una singola risposta**. Di conseguenza, il client deve effettuare ulteriori richieste per recuperare i dati mancanti, portando a inefficienze e a una latenza aumentata.

Scenario: L'endpoint `/books` restituisce un modello semplificato per i libri. Se il client ha bisogno anche dell'editore, deve effettuare richieste aggiuntive per recuperare i dettagli dell'editore come: `GET /books/{title}`

```
public class BookBasicDTO {
    private String title;
    private String author;
}

@RestController
@RequestMapping("/books")
public class BookController {
```

```

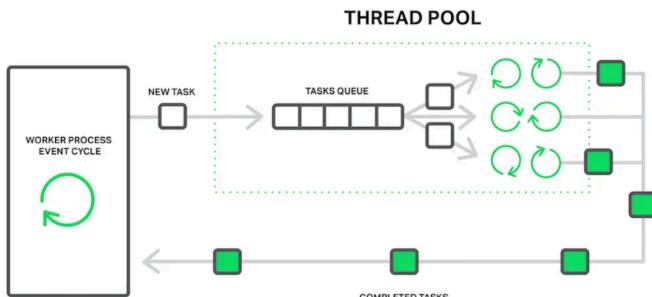
    @GetMapping
    public Iterable<BookBasicDTO> findAll() {
        List<Book> books = List.of(
            new Book("Spring Boot", "John Doe", "123456789", "Guida completa", "TechPub", 500),
            new Book("Microservizi", "Jane Smith", "987654321", "Spiegazione dettagliata", "CloudPub", 300)
        );

        // Restituisce solo titolo e autore
        return books.stream()
            .map(book -> new BookBasicDTO(book.getTitle(), book.getAuthor()))
            .toList();
    }
}

```

⚠️ **Esaурimento del pool di thread (sul client!)**: i client che attendono una risposta dal server consumano risorse di sistema (thread, memoria), il che può essere problematico in ambienti ad alta concorrenza. **[non risolvibile con approcci sincroni!]**

Scenario: Un client interroga l'endpoint `/books` 100 volte al secondo. Ogni richiesta richiede in media 1 secondo per essere soddisfatta. In qualsiasi momento, il **client** ha circa 100 thread nello stato di attesa (in attesa della risposta del server). Dato che ogni thread richiede 1 MB di RAM, cosa succede se il servizio libri smette di rispondere per 10 secondi? Il client avrà bisogno di 1 GB di RAM solo per gestirli!



Il modello ONE THREAD/REQUEST non funziona!

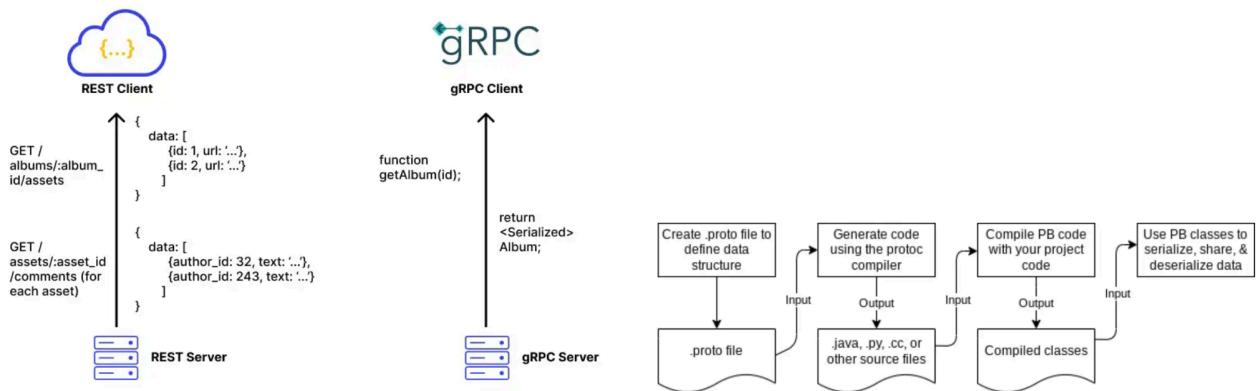
REST (Trasferimento di Stato Rappresentazionale)

Limitazioni di REST:

| Caratteristica | REST |
|---------------------------------------|------|
| Serializzazione | S |
| Accoppiamento Temporale | S |
| Accoppiamento API | S |
| Over-fetching | S |
| Under-fetching (chattiness) | S |
| Esaürimento del Pool di Thread | S |

gRPC (Chiamata di Procedura Remota di Google)

REST → USATO PER LA SUA SEMPLICITA' MA SOFFRE DI TUTTI I PROBLEMI...



Come gRPC Risolve le Limitazioni di REST:

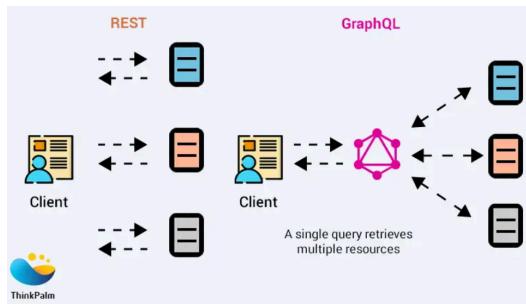
- Serializzazione:** gRPC utilizza Protocol Buffers (Protobuf), un formato binario compatto, per la serializzazione dei dati invece di JSON o XML (comunemente usato nelle API REST). I formati binari sono molto più efficienti in termini di dimensioni e velocità perché occupano meno spazio e sono più rapidi da serializzare e deserializzare.
- Accoppiamento API:** gRPC definisce contratti API rigorosi, assicurando un tipo forte e riducendo il rischio di modifiche che rompono la compatibilità. Con il supporto integrato per la compatibilità retrospettiva e prospettica, l'evoluzione dell'API è più fluida rispetto a REST, dove le modifiche nei payload JSON o nelle specifiche possono più facilmente introdurre incompatibilità.
- Over-fetching:** gRPC riduce l'over-fetching attraverso una serializzazione binaria efficiente.
- Under-fetching (chattiness):** gRPC minimizza i round trip multipli sfruttando il multiplexing [HTTP/2](#), abilitando comunicazioni efficienti.

| Caratteristica | REST | gRPC |
|---------------------------------------|------|---------|
| Serializzazione | S | Ridotto |
| Accoppiamento Temporale | S | S |
| Accoppiamento API | S | Ridotto |
| Over-fetching | S | Ridotto |
| Under-fetching (chattiness) | S | Ridotto |
| Esaурimento del Pool di Thread | S | S |

Limitazioni di gRPC:

- Complessità:** Protobuf introduce più complessità in termini di definizioni di schema e generazione del codice.
- Supporto per il Browser:** Sebbene gRPC sia eccellente per le comunicazioni backend e tra servizi, non è ben supportato negli ambienti di browser, limitando il suo uso per le applicazioni frontend.
- Tipizzazione Rigida:** Sebbene la tipizzazione rigida garantisca robustezza, introduce anche rigidità, poiché le modifiche all'API richiedono una gestione attenta dei contratti Protobuf.

GraphQL (Meta)



Esempio di Query GraphQL

```
query {
  user(id: "123") {
    id
    name
    email
    posts {
      title
      content
    }
  }
}
```

- Questa query richiede dati per un utente specifico con `id: "123"`.
- Recupera `id`, `name` e `email` dell'utente.
- Recupera anche un elenco di post scritti dall'utente, inclusi il `title` e il `content` di ogni post.

Come GraphQL Risolve le Limitazioni di REST:

- **Accoppiamento API:** A differenza di REST, dove diverse risorse sono esposte tramite più endpoint, GraphQL opera attraverso un singolo endpoint per tutte le query, semplificando la struttura API. A differenza di REST, che spesso richiede versioning per gestire le modifiche, lo schema flessibile di GraphQL consente modifiche non-breaking, come l'aggiunta di nuovi campi senza influenzare i client esistenti.
- **Over-fetching e under-fetching (chattiness):** GraphQL consente ai client di richiedere solo i campi di cui hanno bisogno, affrontando il problema di REST di restituire dati non necessari (over-fetching) o di effettuare più richieste per recuperare i dati necessari (under-fetching).
- **Serializzazione:** Il numero ridotto di richieste e risposte (poiché viene utilizzato un singolo endpoint flessibile) riduce anche il carico complessivo di serializzazione.

| Caratteristica | REST | GraphQL |
|---------------------------------------|------|---------|
| Serializzazione | S | Ridotto |
| Accoppiamento Temporale | S | S |
| Accoppiamento API | S | Ridotto |
| Over-fetching | S | Ridotto |
| Under-fetching (chattiness) | S | Ridotto |
| Esaурimento del Pool di Thread | S | S |

Limitazioni di GraphQL:

- **Complessità nell'ottimizzazione delle query:** Sebbene GraphQL offra flessibilità ai client, comporta anche una maggiore responsabilità per il server nell'ottimizzare le query. Se non gestite correttamente, query complesse o profondamente annidate possono portare a colli di bottiglia nelle prestazioni sul lato server.
- **Sfide di caching:** Il caching in GraphQL è più complesso rispetto a REST, poiché le query possono essere dinamiche e granulari. Le API REST possono sfruttare più facilmente il caching basato su HTTP (basato sugli endpoint), mentre GraphQL richiede strategie di caching più sofisticate.
- **Sovraccarico degli schemi:** Il sovraccarico della gestione degli schemi e dei resolver potrebbe non giustificare i benefici in tutti i casi d'uso.

Riepilogo

Sebbene **REST**, **gRPC** e **GraphQL** offrano vantaggi distinti in contesti specifici, presentano tutti limitazioni a causa della loro natura sincrona. gRPC e GraphQL affrontano molte delle inefficienze di REST, come i colli di bottiglia nelle prestazioni, l'over-fetching/under-fetching e le problematiche di evoluzione dell'API. Tuttavia, le sfide intrinseche della comunicazione sincrona spesso richiedono un passaggio verso **comunicazioni asincrone**.

Stili di comunicazione (RESTClient)

Fornitori RESTful

Ci si riferisce all'esempio *rest-social-network* ([labs/rest-social-network](#)):

Il **post-service** gestisce i post su un social network, senza memorizzare dettagli sugli utenti. Di seguito è riportata la classe del modello:

```
@AllArgsConstructor
@NoArgsConstructor
@RequiredArgsConstructor
@Data
@Entity
public class Post {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @NotNull @EqualsAndHashCode.Include private String userUUID;
    @NotNull @EqualsAndHashCode.Include private LocalDateTime timestamp;
    @NotNull private String content;
}
```

Il controller REST gestisce le richieste HTTP in ingresso e risponde con i dati appropriati. Supporta due endpoint chiave:

- GET `/posts` → Restituisce tutti i post.
- GET `/posts/{userUUID}` → Restituisce tutti i post creati da un utente specifico.

```
@RestController
@RequestMapping("/posts")
public class PostController {
    PostRepository postRepository;

    public PostController(PostRepository postRepository) {
        this.postRepository = postRepository;
    }

    @GetMapping("/{userUUID}")
```

```

public Iterable<PostDTO> findByUuid(@PathVariable String userUUID) {
    Iterable<Post> foundPosts = postRepository.findByUserUUID(userUUID);
    return mapToDTO(foundPosts);
}

@GetMapping
public Iterable<PostDTO> findAll() {
    Iterable<Post> foundPosts = postRepository.findAll();
    return mapToDTO(foundPosts);
}

private Iterable<PostDTO> mapToDTO(Iterable<Post> posts) {
    return StreamSupport.stream(posts.spliterator(), false)
        .map(p -> new PostDTO(p.getUserUUID(), p.getTimestamp(), p.getContent()))
        .collect(Collectors.toList());
}
}

```

Consumatori RESTful

Il **user-service** gestisce i dati sugli utenti ed espone i seguenti endpoint:

- **GET /users** → Restituisce tutti gli utenti (solo dettagli locali).
- **GET /users/{userUUID}** → Restituisce dettagli locali e tutti i post di un utente specifico (**necessità di comunicazione in questo punto!**).

```

@RestController
@RequestMapping("/users")
public class UserController {
    UserRepository userRepository;
    PostIntegration postIntegration;

    public UserController(UserRepository userRepository, PostIntegration postIntegration) {
        this.userRepository = userRepository;
        this.postIntegration = postIntegration;
    }

    @GetMapping
    public Iterable<UserDTO> findAll() {
        Iterable<UserModel> foundUsers = userRepository.findAll();
        return mapToDTO(foundUsers);
    }

    @GetMapping("/{userUUID}")
    public UserDTO findByUuid(@PathVariable String userUUID) {
        Optional<UserModel> optionalUserModel = userRepository.findByUserUUID(userUUID);
        optionalUserModel.orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND));
        UserDTO userDTO = mapToDTO(optionalUserModel.get());

        // comunicazione qui!
        Iterable<PostDTO> posts = postIntegration.findbyUserUUID(userUUID);

        for (PostDTO postDTO : posts) {
            userDTO.getPosts().add(postDTO);
        }

        return userDTO;
    }
}

```

```

private UserDTO mapToDTO(UserModel user) {
    return new UserDTO(
        user.getUserUUID(),
        user.getNickname(),
        user.getBirthDate()
    );
}

private Iterable<UserDTO> mapToDTO(Iterable<UserModel> users) {
    return StreamSupport.stream(users.spliterator(), false)
        .map(u -> new UserDTO(u.getUserUUID(), u.getNickname(), u.getBirthDate()))
        .collect(Collectors.toList());
}
}

```

La parte di comunicazione è interamente gestita da un bean dedicato chiamato *PostIntegration*. I dati di connessione che descrivono la posizione di rete del servizio da consumare (in questo caso, il **post-service**) sono esternalizzati nel file **application.yml** e recuperati tramite l'annotazione **@Value**.

```

@Component
public class PostIntegration {
    String postServiceHost;
    int postServicePort;

    public PostIntegration(
        @Value("${app.post-service.host}") String postServiceHost,
        @Value("${app.post-service.port}") int postServicePort) {
        this.postServiceHost = postServiceHost;
        this.postServicePort = postServicePort;
    }

    public Iterable<PostDTO> findByUserUUID(String userUUID) {
        String url = "http://" + postServiceHost + ":" + postServicePort + "/posts" + "/" + userUUID;
        RestClient restClient = RestClient.builder().build();
        return restClient.get()
            .uri(url)
            .retrieve()
            .body(new ParameterizedTypeReference<>() {});
    }
}

```

Prova del sistema di messaggistica

```

services:
  post-service:
    build: post-service
    environment:
      - SPRING_PROFILES_ACTIVE=docker
  ports:
    - "8080:8080"

  user-service:
    build: user-service
    environment:
      - SPRING_PROFILES_ACTIVE=docker
  ports:
    - "8081:8080"

```

```
mvn clean package -Dmaven.test.skip=true
docker compose up --build --detach
```

Il comando seguente mostra i dati memorizzati localmente sui post (senza dettagli sugli utenti).

```
curl -X GET http://localhost:8080/posts | jq
```

```
[
  {
    "userUUID": "171f5df0-b213-4a40-8ae6-fe82239ab660",
    "timestamp": "2025-03-01T10:30:00",
    "content": "hello!"
  },
  {
    "userUUID": "171f5df0-b213-4a40-8ae6-fe82239ab660",
    "timestamp": "2025-03-01T10:32:00",
    "content": "i'm json!"
  },
  {
    "userUUID": "b1f4748a-f3cd-4fc3-be58-38316afe1574",
    "timestamp": "2025-03-01T10:32:00",
    "content": "looking for an apartment"
  }
]
```

Il comando seguente mostra i dati sugli utenti, arricchiti con i dati sui post.

```
curl -X GET http://localhost:8081/users/171f5df0-b213-4a40-8ae6-fe82239ab660 | jq
```

```
{
  "userUUID": "171f5df0-b213-4a40-8ae6-fe82239ab660",
  "nickname": "hannibal",
  "birthDate": "2000-03-01",
  "posts": [
    {
      "userUUID": "171f5df0-b213-4a40-8ae6-fe82239ab660",
      "timestamp": "2025-03-01T10:32:00",
      "content": "i'm json!"
    },
    {
      "userUUID": "171f5df0-b213-4a40-8ae6-fe82239ab660",
      "timestamp": "2025-03-01T10:30:00",
      "content": "hello!"
    }
  ]
}
```

DTOs

Un **DTO (Data Transfer Object)** è un design pattern utilizzato nell'ingegneria del software per trasferire dati tra diverse parti di un'applicazione, spesso oltre i confini di rete o tra i vari strati all'interno della stessa applicazione.

Lo scopo principale di un DTO è quello di encapsulare dati e ridurre la quantità di informazioni trasmesse attraverso la rete, contenendo solo i campi necessari, ed è comunemente usato in sistemi distribuiti e applicazioni che seguono architetture a strati (ad esempio, MVC o architetture orientate ai servizi).

Caratteristiche Chiave dei DTO

- **Incapsulamento:** I DTO avvolgono i dati in una struttura che nasconde entità complesse o potenzialmente informazioni sensibili, esponendo solo i campi rilevanti per l'operazione di trasferimento dati specifica.
- **Nessuna Logica di Business:** I DTO tipicamente non contengono logica di business, poiché servono solo come contenitori per i dati. Il loro scopo è puramente il trasferimento di dati, quindi solitamente includono solo metodi come getter e setter.
- **Serializzazione:** Poiché i DTO vengono spesso trasferiti tramite una rete o tra i confini applicativi, sono di solito progettati per essere serializzabili (ad esempio, in JSON o XML).

Quando Utilizzare i DTO

- **API e Microservizi:** I DTO sono comunemente utilizzati in API REST e microservizi. Forniscono un modo per definire il formato e il contenuto dei dati scambiati senza esporre modelli interni o entità di database.
- **Riduzione del Carico Dati:** Selezionando solo i campi rilevanti, i DTO possono ridurre la quantità di dati trasmessi, risultando particolarmente utili in applicazioni mobili o in reti a bassa larghezza di banda.
- **Disaccoppiamento degli Strati:** In un'architettura a strati (ad esempio, separando accesso ai dati, logica di business e strati di presentazione), i DTO aiutano a mantenere la separazione fungendo da intermediari tra la logica di business e gli strati di presentazione.

Il mapping automatico tra entità e DTO è una necessità comune, poiché semplifica il processo di conversione dei dati tra i diversi strati di un'applicazione. In Java e Python, sono disponibili librerie per facilitare questo mapping, riducendo il codice boilerplate e migliorando la leggibilità del codice.

Mapping da Entità a DTO

Diversi framework offrono capacità di mapping automatico. Ecco i più popolari:

MapStruct: (Java) MapStruct è una potente libreria di generazione di codice a tempo di compilazione che crea mapper tipo-sicuri tra oggetti Java (ad esempio, entità e DTO). Genera codice a tempo di compilazione, quindi non c'è sovraccarico a tempo di esecuzione, rendendola veloce ed efficiente.

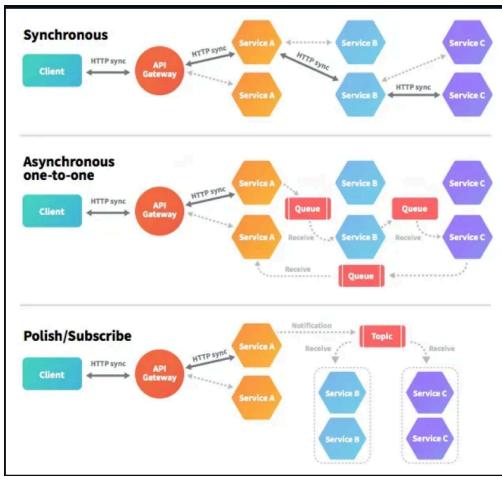
ModelMapper: (Java) ModelMapper è una libreria più flessibile, basata su runtime, che fornisce un approccio basato su convenzione per il mapping degli oggetti. Può mappare automaticamente proprietà con nomi simili ed è altamente personalizzabile.

Marshmallow: (Python) Marshmallow è una libreria popolare per la serializzazione/deserializzazione di oggetti in Python. È tipicamente utilizzata per convertire oggetti in JSON e viceversa, ma può anche essere impiegata per i mapping dei DTO.

Pydantic: (Python) Pydantic è principalmente utilizzato per la validazione dei dati e la gestione delle impostazioni, ma funge anche da eccellente libreria DTO. Fornisce validazione dei dati e conversione tra oggetti Python e formati compatibili con JSON.

Comunicazioni Asincrone

La comunicazione asincrona è un modello architettonale chiave nei sistemi distribuiti. A differenza della comunicazione sincrona, dove un servizio invia una richiesta e attende una risposta, **la comunicazione asincrona consente ai servizi di continuare a elaborare altre attività mentre sono in attesa di una risposta INSERENDO UN COMPONENTE BROKER.**



Vantaggi della Comunicazione Asincrona

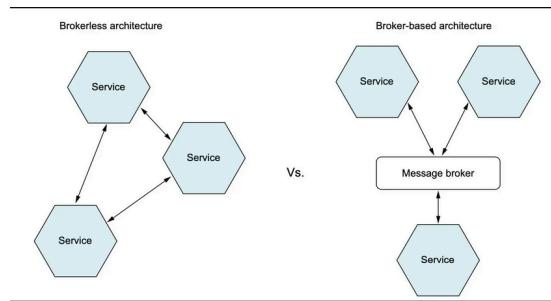
- Servizi Decoupled:** I servizi nelle architetture asincrone sono più disaccoppiate. Il mittente non deve sapere quando o come il destinatario elabora il messaggio (**risolve il disaccoppiamento temporale!**).
- Migliore Utilizzazione delle Risorse:** Nella comunicazione asincrona, le risorse di sistema (come thread e memoria) non sono bloccate in attesa di risposte (**risolve l'esaurimento del thread pool!**). Ciò migliora l'utilizzazione delle risorse, specialmente in ambienti ad alta intensità di throughput.
- Scalabilità Migliorata:** I sistemi asincroni sono più scalabili poiché i servizi non stanno aspettando risposte immediate. Invece, gestiscono le attività in modo indipendente. Le code di messaggi possono anche scalare orizzontalmente, distribuendo il carico tra i servizi.
- Maggiore Resilienza:** Poiché i servizi non dipendono dalla disponibilità immediata degli altri, il sistema è più resiliente ai guasti. Un broker di messaggi può memorizzare i messaggi nel caso in cui un servizio sia offline e consegnarli quando diventa nuovamente disponibile, rendendo il sistema a prova di guasto.

Sfide della Comunicazione Asincrona

- Complessità:** Gestire la comunicazione asincrona introduce complessità aggiuntiva nel design del sistema. Garantire che i messaggi siano consegnati, elaborati e confermati correttamente richiede componenti e configurazioni infrastrutturali aggiuntivi.
- Consistenza:** La consistenza eventuale è una situazione comune nei sistemi asincroni, dove gli aggiornamenti dei dati si propagano in modo asincrono tra i servizi. Sebbene ciò migliori la disponibilità, può portare a problemi di consistenza dei dati poiché gli aggiornamenti possono richiedere tempo per propagarsi.
- Ordine dei Messaggi:** Garantire che i messaggi siano elaborati nell'ordine corretto può essere una sfida, in particolare nei sistemi distribuiti in cui i messaggi potrebbero arrivare in ordine errato. I broker di messaggi avanzati offrono supporto per il sequenziamento dei messaggi, ma ciò aggiunge complessità all'implementazione.
- Gestione degli Errori:** Gli errori nella comunicazione asincrona sono spesso più difficili da rilevare e gestire rispetto ai sistemi sincroni. Quando un servizio non riesce ad elaborare un messaggio, potrebbe non essere immediatamente ovvio. Meccanismi di retry, code per messaggi non elaborati e monitoraggio sono essenziali per gestire gli errori in modo efficace.

Sistemi di Messaging Basati su Broker e Senza Broker

Ci sono due principali approcci per il passaggio di messaggi asincroni: sistemi **basati su broker** e sistemi **senza broker** (noto anche come peer-to-peer o messaggistica diretta).



Sistemi di Messaging Basati su Broker

I sistemi di messaggistica con broker si basano su un **broker di messaggi centrale** per gestire la comunicazione tra diversi servizi. Il broker funge da intermediario, ricevendo messaggi dai produttori e consegnandoli ai consumatori. I comuni sistemi basati su broker includono:

- RabbitMQ (<https://www.rabbitmq.com>)
- Apache Kafka (<http://kafka.apache.org>)
- ActiveMQ (<http://activemq.apache.org>)
- AWS SQS (<https://aws.amazon.com/sqs/>)
- Microsoft Azure Service Bus (<https://azure.microsoft.com/en-us/products/service-bus/>)
- Google Cloud Pub/Sub (<https://cloud.google.com/pubsub>)

Vantaggi

- **Decoupling:** I produttori e i consumatori non hanno bisogno di sapere dell'esistenza reciproca (**risolve il coupling spaziale!**). Interagiscono solo con il broker, rendendo il sistema loosely coupled e più facile da mantenere e scalare.
- **Affidabilità:** I broker di messaggi memorizzano i messaggi e forniscono **garanzie di consegna** (ad es., almeno-una-volta, esattamente-una-volta). Anche se un consumatore è offline, il broker garantisce che il messaggio sarà consegnato quando il consumatore sarà disponibile.
- **Scalabilità:** Con una configurazione appropriata (ad es. le partizioni di Kafka, i cluster RabbitMQ), i broker possono gestire enormi quantità di dati e scalare orizzontalmente per adattarsi a casi d'uso ad alta intensità di throughput.

Svantaggi

- **Single Point of Failure:** Se non configurato correttamente con replicazione o failover, il **broker può diventare un punto unico di fallimento** nel sistema. In caso di inattività del broker, il flusso dei messaggi può essere interrotto. Come componente centrale, il **broker può anche diventare un collo di bottiglia sotto carichi pesanti** o misconfigurazioni, causando ritardi o fallimenti nella consegna dei messaggi.
- **Overhead di Comunicazione/Latenza:** Ogni messaggio deve passare attraverso il broker prima di essere consegnato al destinatario previsto. Questo introduce anche ulteriori latenze. Il numero di comunicazioni aumenta rispetto all'utilizzo della comunicazione diretta senza broker.

Esempio

La seguente logica di business implica molte chiamate tra servizi e il broker. La latenza viene aggiunta a ciascun passaggio. Se il broker fallisce, l'intero processo fallisce.

```
function AppA (x) {
    y = do_business_logic_A (x);
    return AppB (y);
}

function AppB (x) {
    y = do_business_logic_B (x);
    return AppC (y);
}
```

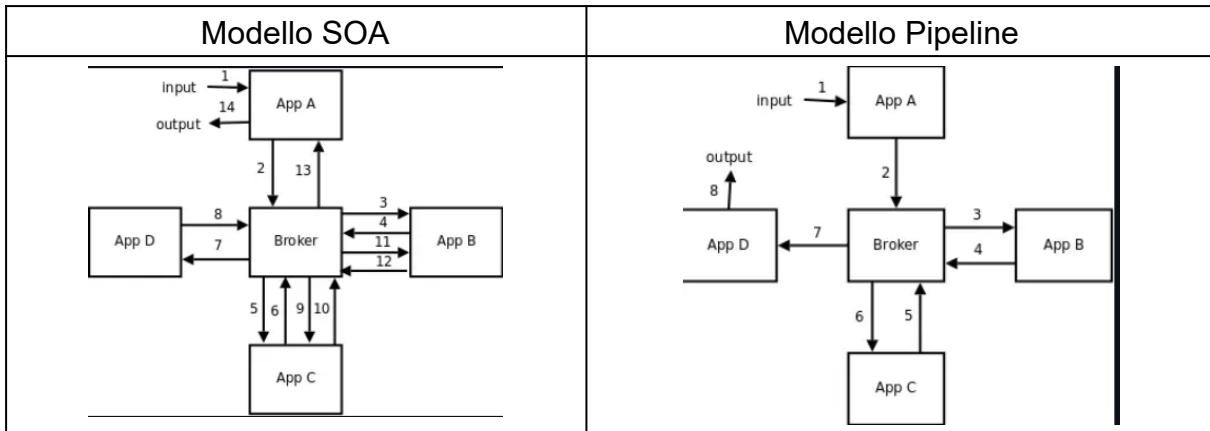
```

function AppC (x) {
    y = do_business_logic_C (x);
    return AppD (y);
}

function AppD (x) {
    return do_business_logic_D (x);
}

```

Nel modello SOA avvengono 14 interazioni per eseguire il tutto, da serializzare e deserializzare i messaggi. Si può usare un modello a Pipeline per diminuire il numero di messaggi scambiati.



Sistemi di Messaging Senza Broker

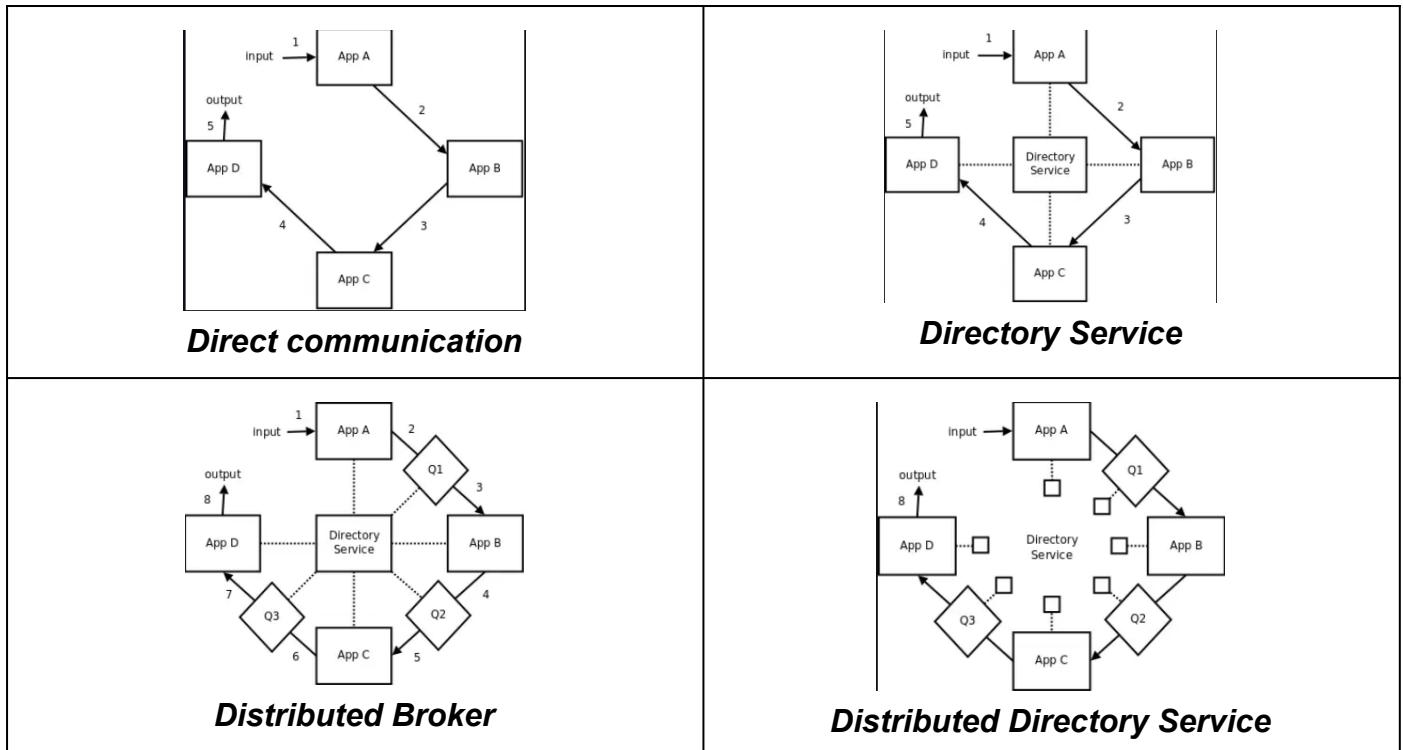
La messaggistica senza broker elimina la necessità di un broker centrale. Invece, i servizi comunicano direttamente tra loro usando delle librerie incorporate sui servizi. I sistemi senza broker utilizzano protocolli come gRPC, ZeroMQ e messaggistica basata su HTTP.

Per una comunicazione efficiente, i peer generano **più thread o utilizzano I/O asincrono per gestire più connessioni contemporaneamente**. Questo garantisce che il peer possa inviare/ricevere messaggi senza bloccare altre attività.

- ZeroMQ (<https://zeromq.org/>)
- NanoMsg (<https://nanomsg.org/>)

Vantaggi

- **Nessun Punto Unico di Fallimento:** I sistemi senza broker evitano che il broker diventi un punto unico di fallimento, rendendo l'architettura più resiliente a determinati tipi di guasti.
- **Bassa Latenza:** I messaggi viaggiano direttamente tra i servizi, riducendo l'overhead aggiuntivo introdotto da un broker. Ciò porta a una comunicazione più rapida e a una minore latenza.
- **Maggiore Controllo:** La comunicazione diretta fornisce ai servizi il pieno controllo su come i messaggi vengono gestiti, migliorando la flessibilità nella gestione di scenari specifici come retry o gestione degli errori.



Svantaggi

- **Coupling Ristretto:** In un sistema senza broker, i servizi devono sapere come comunicare direttamente tra loro. Questo aumenta il coupling tra i servizi e rende più difficile gestire le modifiche (ad es., cambiare la posizione o l'API di un servizio può richiedere di aggiornare tutti i servizi che comunicano con esso).
- **Nessuna Affidabilità Integrata:** Nei sistemi senza broker, le funzionalità di affidabilità come persistenza dei messaggi, retry e garanzie di consegna devono essere implementate dagli sviluppatori. Ciò aumenta la complessità, poiché il sistema manca dei meccanismi di affidabilità forniti dai broker.
- **Nessuna Scalabilità Integrata:** Scalare sistemi senza broker può essere più complesso, specialmente in ambienti ad alta intensità di throughput. Potrebbe essere necessario implementare meccanismi di bilanciamento del carico e failover personalizzati per garantire che il sistema possa gestire grandi quantità di connessioni o messaggi.
- **Nessuna Concorrenza Integrata:** Gestire più connessioni concorrenti o garantire l'ordinamento e la consegna dei messaggi può diventare problematico, richiedendo un'attenta progettazione e gestione della logica di comunicazione.

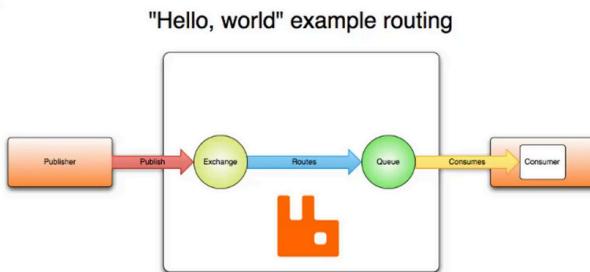
Comunicazioni Asincrone (RabbitMQ)

RabbitMQ è un broker di messaggi open-source ampiamente utilizzato che facilita la comunicazione tra diverse parti di un sistema distribuito. Permette alle applicazioni di inviare e ricevere messaggi attraverso un sistema di messaggistica robusto e flessibile basato sul **Protocollo Avanzato di Accodamento Messaggi (AMQP)**. RabbitMQ supporta vari modelli di messaggistica, rendendolo adatto a diverse casistiche.

Componenti Chiave di RabbitMQ

- **Produttore:** L'applicazione che invia messaggi al broker.
- **Consumatore:** L'applicazione che riceve messaggi da una coda.
- **Coda:** Un buffer che memorizza i messaggi fino a quando non vengono consumati.
- **Scambio:** Un meccanismo di instradamento che determina come i messaggi vengono distribuiti alle code.
- **Collegamento:** Un legame tra uno scambio e una coda che definisce le regole di instradamento per i messaggi.

- Chiave di Routing:** L'attributo di messaggio che viene preso in considerazione dallo scambio quando si decide come instradare un messaggio.



Gli Exchange e le code vanno configurate in modo da rispettare i requisiti del sistema.

Tipi di Scambi

RabbitMQ fornisce diversi tipi di exchange che determinano come i messaggi vengono instradati alle code. I principali tipi di scambi sono:

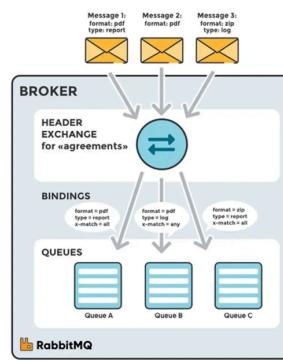
| Tipo di scambio | Descrizione e caso d'uso | Schema |
|-----------------|---|--------|
| Fanout | <p>Descrizione: Uno scambio fanout instrada i messaggi a tutte le code collegate a esso, indipendentemente dalla chiave di routing. Broadcasta i messaggi a più consumatori. → il messaggio in questo esempio viene copiato 3 volte 1 volta per ogni coda</p> <p>Caso d'uso: Utile per scenari in cui i messaggi devono essere consegnati a più iscritti.</p> | |
| Diretto | <p>Descrizione: Uno scambio diretto instrada i messaggi con una chiave di routing specifica alle code collegate allo scambio con la stessa chiave di routing.</p> <p>Caso d'uso: Utile per la comunicazione punto a punto in cui i messaggi devono essere instradati a una coda specifica.</p> <p>Esempio: Un sistema di logging dove i log di diversi livelli di severità (ad esempio, INFO, ERROR) vengono inviati a diverse code in base alla loro severità.</p> | |
| Topic | <p>Descrizione: Uno scambio topic instrada i messaggi a una o più code in base a modelli di wildcard nella chiave di routing. Ciò consente una logica di instradamento più complessa.</p> <p>Caso d'uso: Utile per scenari in cui i messaggi devono essere filtrati in base a più criteri.</p> <p>Esempio: Un servizio di notizie dove gli articoli possono essere contrassegnati con più categorie (ad esempio, sport, politica), consentendo agli iscritti di ricevere solo gli articoli di interesse.</p> | |

Headers

Descrizione: Uno scambio a header instrada i messaggi in base agli attributi dell'intestazione del messaggio piuttosto che alla chiave di routing. Abbina le intestazioni a criteri specificati.

Caso d'uso: Utile per scenari che richiedono instradamenti basati su più attributi anziché su una sola chiave di routing.

Esempio: Un sistema di elaborazione ordini in cui gli ordini vengono instradati in base a più criteri come la posizione del cliente, il tipo di prodotto e la priorità.



Queue

In RabbitMQ, le code sono un componente fondamentale che memorizza i messaggi. Agiscono come intermediari tra i produttori di messaggi e i consumatori. → I messaggi inviati dalle applicazioni vengono instradati attraverso gli exchange e quindi finiscono in coda, in attesa di essere elaborati dalle applicazioni consumer.

Gli attributi chiave delle code in RabbitMQ includono:

- **Archiviazione:** le code archiviano i messaggi fino a quando non vengono elaborati o utilizzati dalle applicazioni.
- **Durabilità:** le code possono essere durevoli, il che significa che sopravvivono al riavvio del broker. La durabilità garantisce che i messaggi non vadano persi anche in caso di riavvio di RabbitMQ.
- **Ordine dei messaggi:** per impostazione predefinita, RabbitMQ mantiene l'ordine dei messaggi all'interno di una coda (FIFO - First-In, First-Out).
- **Proprietà configurabili:** le code hanno proprietà configurabili come la lunghezza massima, i livelli di priorità massimi, il TTL (Time-To-Live) del messaggio, ecc., che consentono l'ottimizzazione per soddisfare requisiti specifici.

Consumatori

In RabbitMQ, i consumatori sono applicazioni o componenti che recuperano ed elaborano i messaggi dalle code. Svolgono un ruolo fondamentale nel flusso di elaborazione dei messaggi all'interno dell'ecosistema RabbitMQ.

Gli aspetti chiave dei consumatori in RabbitMQ includono:

- **Elaborazione dei messaggi:** una volta che un consumatore è connesso e iscritto a una coda, ascolta attivamente i messaggi in arrivo. Dopo aver ricevuto un messaggio dalla coda, il consumer lo elabora in base a una logica predefinita o a requisiti aziendali.
- **Concorrenza e ridimensionamento:** i consumer possono essere ridimensionati orizzontalmente per gestire un aumento dei carichi di messaggi. È possibile creare più istanze di un'applicazione consumer per elaborare contemporaneamente i messaggi provenienti dalla stessa coda, migliorando le prestazioni e la scalabilità.
- **Message Acknowledgement Modes:** RabbitMQ supporta diverse modalità di riconoscimento, come il riconoscimento automatico (i messaggi vengono contrassegnati come riconosciuti non appena vengono consegnati ai consumatori) e il riconoscimento manuale (i consumatori riconoscono esplicitamente i messaggi dopo l'elaborazione).
- **Gestione degli errori:** i consumatori devono gestire gli errori con grazia. Se l'elaborazione di un messaggio non riesce, i consumer possono scegliere di rifiutare, riaccodare o gestire il messaggio non riuscito in base a strategie di gestione degli errori predefinite.

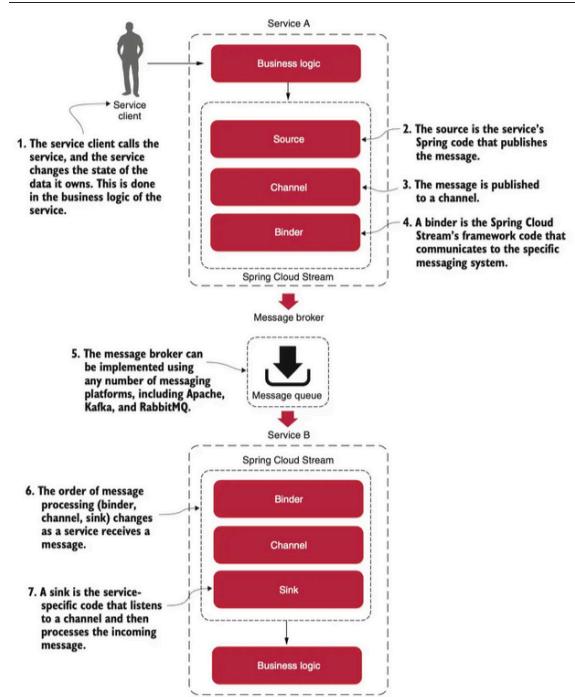
TRE ESERCIZI LABS.

Comunicazioni Asincrone (*Spring Cloud Stream*)

Introduzione

[Spring Cloud Stream](#) permette di astrarre i dettagli di implementazione della piattaforma di messaggistica (ad esempio, Apache Kafka, RabbitMQ). In questo modo, i **dettagli specifici dell'implementazione vengono mantenuti separati dal codice dell'applicazione**. La pubblicazione e il consumo dei messaggi nelle applicazioni vengono gestiti attraverso interfacce di Spring neutre rispetto alla piattaforma.

Iniziamo la nostra discussione esaminando l'architettura di Spring Cloud Stream nel contesto di due servizi che comunicano tramite messaggistica. *Un servizio è il publisher del messaggio, mentre l'altro è il consumer del messaggio.*



Source: prende un [Plain Old Java Object \(POJO\)](#), che rappresenta il messaggio da pubblicare, lo serializza (la serializzazione predefinita è JSON) e pubblica il messaggio su un canale.

Channel: è un'astrazione sulla coda che contiene il messaggio. È sempre associato a un nome di coda di destinazione, ma quel nome di coda non viene mai esposto direttamente al codice, il che significa che possiamo cambiare le code che il canale legge o scrive senza modificare il codice dell'applicazione (solo la configurazione).

Binder: comunica con una piattaforma di messaggistica specifica. La parte binder del framework Spring Cloud Stream ci permette di lavorare con i messaggi senza dover essere esposti alle librerie e API specifiche della piattaforma per pubblicare e consumare messaggi.

Sink: ascolta un canale per i messaggi in arrivo e deserializza il messaggio di nuovo in un oggetto POJO. Da lì, il messaggio può essere elaborato dalla logica di business del servizio Spring.

Definizione degli eventi

Si comunica tra i servizi usando delle *classi evento*.

Un evento può essere definito dai seguenti elementi:

- Il tipo di evento, ad esempio, un evento di creazione o eliminazione
- Una chiave che identifica i dati (ad esempio, un ID messaggio)
- Un elemento di dati, ovvero i dati effettivi nell'evento

- Un timestamp, che descrive quando è avvenuto l'evento

```

@NoArgsConstructor
@AllArgsConstructor
@RequiredArgsConstructor
@Data
@Builder
public class Event<K, T> {
    public enum Type {CREATE, DELETE, UPDATE}
    @NotNull private Type eventType; → TIPO DI EVENTO
    @NotNull private K key; → TIPO DELLA CHIAVE: TANTI EVENTI CON LA STESSA CHIAVE SE CORRELATI.
    @NotNull private T data;
    private ZonedDateTime eventCreatedAt = ZonedDateTime.now();
}

```

Dipendenze del progetto

Per includere Spring Cloud Stream nel nostro progetto, dobbiamo aggiungere `spring-cloud-stream` e almeno un binder (ad esempio, `spring-cloud-starter-stream-rabbit` o `spring-cloud-starter-stream-kafka`) come mostrato di seguito.

La sezione `<dependencyManagement>` viene tipicamente utilizzata nei progetti basati su Spring per gestire le dipendenze di Spring Cloud in modo coerente.

Utilizzando il BOM (Bill of Materials) di `spring-cloud-dependencies`, possiamo assicurarci che vengano utilizzate le dipendenze corrette ed evitare conflitti di versione. Vanno inserite sia nel producer che nel consumer.

```

<properties>
    <spring-cloud.version>2024.0.0</spring-cloud.version> → serve per specificare la versione
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-stream</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
    </dependency>
</dependencies>

```

QUESTE RIGHE GESTISCONO IL BOM.

```

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

```

Pubblicazione degli eventi

Per pubblicare un evento, dobbiamo:

- Creare un oggetto Event
- Usare la classe StreamBridge per pubblicare gli eventi sul topic desiderato
- Aggiungere la configurazione necessaria per la pubblicazione degli eventi

```

@Log4j2
@Component
public class ScheduledTask {
    private final StreamBridge streamBridge;
    // costruttore qui
    @Scheduled(fixedRate = 100)
    public void randomMessage() {
        Event<UUID, Integer> event = new Event<>(
            randomType,
            randomUUID,
            randomData
        );
        sendMessage("message-out-0", event);
        log.info(event.toString());
    }

    private void sendMessage(String bindingName,
                           Event<UUID, Integer> event) {
        Message<Event<UUID, Integer>> message =
        MessageBuilder.withPayload(event)
                     .setHeader("routingKey",
                    event.getType().name())
                     .setHeader("partitionKey", event.getKey())
                     .build();
        streamBridge.send(bindingName, message);
    }
}

```

Per quanto riguarda la configurazione, dobbiamo fornire RabbitMQ come sistema di messaggistica predefinito (inclusi i dati di connettività), JSON come tipo di contenuto predefinito e i topic che devono essere utilizzati.

```

spring.cloud.stream:
  bindings:
    message-out-0:
      contentType: application/json
      destination: queue.messages
      binder: local_rabbit
  binders:
    local_rabbit:
      type: rabbit
      environment:
        spring:
          rabbitmq:
            host: 127.0.0.1
            port: 5672
            username: guest
            password: guest

```

spring.cloud.stream.bindings

- **message-out-0:** Questo è un binding di output (un canale) per inviare messaggi.
 - **contentType:** Imposta il formato del messaggio su application/json, il che significa che i messaggi inviati su questo canale saranno serializzati come JSON.
 - **destination:** Imposta la destinazione (la coda) per i messaggi su queue.messages.
 - **binder:** Specifica local_rabbit come binder da utilizzare, che collega questo binding alla configurazione definita di RabbitMQ.

spring.cloud.stream.binders

- **local_rabbit:** Definisce un binder RabbitMQ personalizzato.
 - **type:** Specifica il tipo come rabbit, indicando che RabbitMQ è il broker di messaggi.
 - **environment:** Configura le impostazioni specifiche dell'ambiente di RabbitMQ.
 - **spring.rabbitmq:** Specifica le proprietà di connessione a RabbitMQ:
 - **host:** Il nome host del server RabbitMQ (127.0.0.1, che è localhost).
 - **port:** La porta per connettersi a RabbitMQ, impostata sulla porta predefinita 5672.
 - **username e password:** Le credenziali per accedere a RabbitMQ, qui impostate su guest.

Ricezione degli eventi

Per poter consumare eventi, dobbiamo fare quanto segue:

- Dichiare i message processors, che sono metodi che consumano gli eventi.
- Aggiungere la configurazione necessaria per consumare gli eventi dal broker.

Un esempio di message processor è dichiarato di seguito. Possiamo vedere che la classe è annotata con **@Configuration**, indicando a Spring di cercare i bean Spring all'interno della classe. La classe fornisce effettivamente un bean che implementa l'interfaccia **Consumer<Event<String, Integer>>**.

```
@Log4j2
@Component
public class EventReceiver {
    @Bean
    public Consumer<Event<String, Integer>> messageProcessor() {
        return event -> {
            log.info(event.toString());

            switch (event.getType()) {
                case CREATE:
                    // fare qualcosa
                    break;
                case DELETE:
                    // fare qualcosa
                    break;
                case UPDATE:
                    // fare qualcosa
                    break;
                default:
                    String errorMessage = "Tipo di evento errato: " + event.getType() + ", previsto un evento
CREATE/DELETE/UPDATE";
                    throw new RuntimeException(errorMessage);
            }
        };
    }
}
```

```
spring.cloud.stream:
function:
  definition: messageProcessor
bindings:
  messageProcessor-in-0:
    binder: local_rabbit
    contentType: application/json
    destination: queue.messages
binders:
  local_rabbit:
    type: rabbit
    environment:
      spring:
        rabbitmq:
          host: 127.0.0.1
          port: 5672
          username: guest
          password: guest
```

- **spring.cloud.stream.function.definition:**

- **messageProcessor:** Definisce un nome di funzione, messageProcessor, che rappresenta la logica che elaborerà i messaggi in arrivo. Questa funzione fungerà da gestore per i messaggi ricevuti sul canale di input configurato.
- **spring.cloud.stream.bindings**
 - **messageProcessor-in-0:** Configura un binding di input (canale) per la funzione messageProcessor.
 - **binder:** Specifica local_rabbit come binder, collegando questo binding alla configurazione personalizzata di RabbitMQ.
 - **contentType:** Imposta il formato del messaggio su application/json, il che significa che i messaggi in arrivo verranno deserializzati da JSON.
 - **destination:** Specifica la coda dei messaggi a cui ascoltare, impostata su queue.messages. Questa coda è dove RabbitMQ inoltrerà i messaggi affinché la funzione messageProcessor li gestisca.

Provare il sistema di messaggistica: LAVINMQ (CLONE DI RABBIT)

[LavinMQ](#) è un broker di messaggi estremamente veloce in grado di gestire grandi quantità di messaggi e connessioni. Implementa il protocollo AMQP (così da poter sostituire trasparentemente RabbitMQ) e può essere eseguito sia su un singolo nodo che su un cluster.

Un publisher, molti consumer (pubblica-sottoscrivi)

```
services:
  publisher:
    image: async-rabbitmq-publisher
    build: async-rabbitmq-publisher
    environment:
      - SPRING_PROFILES_ACTIVE=docker
    depends_on:
      lavinmq:
        condition: service_healthy

  consumer:
    image: async-rabbitmq-consumer
    build: async-rabbitmq-consumer
    environment:
      - SPRING_PROFILES_ACTIVE=docker
    depends_on:
      lavinmq:
        condition: service_healthy
  deploy:
    mode: replicated
    replicas: 3

lavinmq:
  image: cloudamqp/lavinmq:latest
  ports:
    - 5672:5672
    - 15672:15672
  healthcheck:
    test: [ "CMD", "lavinmqctl", "status" ]
    interval: 5s
    timeout: 2s
```

```

    retries: 60
deploy:
  resources:
    limits:
      memory: 512m

```

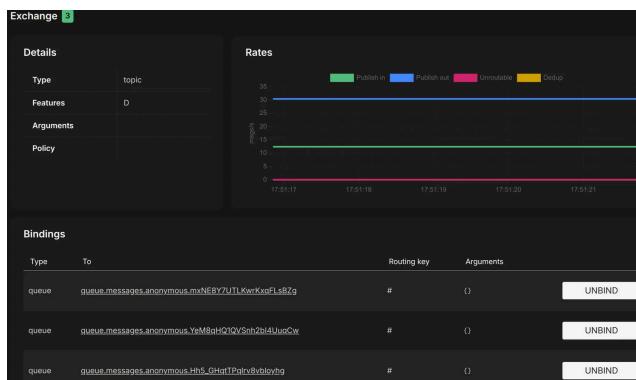
Avvia il sistema con i seguenti comandi:

```

export COMPOSE_FILE=docker-compose-one-to-many.yml
mvn clean package -Dmaven.test.skip=true
docker compose up --build --detach

```

Utilizzando l'interfaccia web di [LavinMQ](#) (login: guest/guest), possiamo vedere che l'exchange queue.messages riceve 10 eventi/s e pubblica gli stessi eventi su n code diverse (una per ogni consumer). Di conseguenza, la velocità di uscita è di 10 eventi/s * n, dove n è il numero di repliche del consumer.



Gruppi di consumer

Il problema è che tutte le istanze di un consumer del messaggio consumano gli stessi messaggi. Questo non è utile per la scalabilità.

Possiamo evitare questo problema utilizzando gruppi di consumer. Ogni binding di consumer può utilizzare la proprietà `spring.cloud.stream.bindings.<bindingName>.group` per specificare un nome di gruppo. I consumer all'interno dello stesso gruppo competono per gli stessi messaggi.

Modifica `docker-compose.yml` per attivare il profilo `groups` per i consumer.

```

consumer:
  image: async-rabbitmq-consumer
  build: async-rabbitmq-consumer
  environment:
    - SPRING_PROFILES_ACTIVE=docker,groups
  depends_on:
    - lavinmq
    condition: service_healthy
  deploy:
    mode: replicated
    replicas: 3

```

Il profilo `groups` aggiunge le seguenti configurazioni (vedi `application.yml`):

```

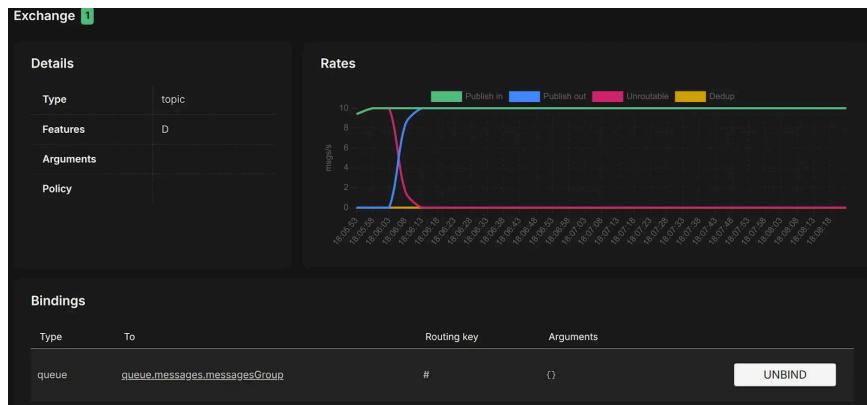
spring.config.activate.on-profile: groups
spring.cloud.stream:
  bindings:
    messageProcessor-in-0:

```

```
group: messagesGroup
```

Avvia il sistema con i seguenti comandi:

```
export COMPOSE_FILE=docker-compose-one-to-many-groups.yml
mvn clean package -Dmaven.test.skip=true
docker compose up --build --detach
```



Utilizzando l'interfaccia web di [LavinMQ](#), possiamo vedere che l'exchange riceve 10 eventi/s e li pubblica su una coda singola (queue.messages.messagesGroup) che ha tre consumer. Di conseguenza, la velocità di uscita è di 10 eventi/s.

Partizioni

Il problema è che ciascun evento viene ricevuto solo da un consumatore. Tuttavia, non abbiamo alcuna garanzia che tutti i messaggi relativi alla stessa entità (ad esempio, lo stesso prodotto, lo stesso ordine) vengano ricevuti dalla stessa istanza del consumatore. Questo potrebbe portare a comportamenti indesiderati. Per risolvere questo problema, possiamo utilizzare le **partizioni**.

```
services:
  publisher:
    image: async-rabbitmq-publisher
    build: async-rabbitmq-publisher
    environment:
      - SPRING_PROFILES_ACTIVE=docker,partitioned
    depends_on:
      lavinmq:
        condition: service_healthy
    deploy:
      resources:
        limits:
          memory: 512m

  consumer-0:
    image: async-rabbitmq-consumer
    build: async-rabbitmq-consumer
    environment:
      - SPRING_PROFILES_ACTIVE=docker,groups,partitioned_instance_0
    depends_on:
      lavinmq:
        condition: service_healthy
    deploy:
      resources:
        limits:
          memory: 512m
```

```

consumer-1:
  image: async-rabbitmq-consumer
  build: async-rabbitmq-consumer
  environment:
    - SPRING_PROFILES_ACTIVE=docker,groups,partitioned_instance_1
  depends_on:
    - lavinmq:
        condition: service_healthy
  deploy:
    resources:
      limits:
        memory: 512m

consumer-2:
  image: async-rabbitmq-consumer
  build: async-rabbitmq-consumer
  environment:
    - SPRING_PROFILES_ACTIVE=docker,groups,partitioned_instance_2
  depends_on:
    - lavinmq:
        condition: service_healthy
  deploy:
    resources:
      limits:
        memory: 512m

```

Il profilo *partitioned* aggiunge le seguenti configurazioni.

Lato Publisher:

```

spring.config.activate.on-profile: partitioned

spring.cloud.stream.bindings.message-out-0.producer:
  partition-key-expression: headers['partitionKey']
  partition-count: 3

```

Lato Consumer:

```

spring.config.activate.on-profile: partitioned_instance_0
spring.cloud.stream.bindings:
  messageProcessor-in-0:
    consumer:
      partitioned: true
      instanceIndex: 0
      instanceCount: 3

---
spring.config.activate.on-profile: partitioned_instance_1
spring.cloud.stream.bindings:
  messageProcessor-in-0:
    consumer:
      partitioned: true
      instanceIndex: 1
      instanceCount: 3

---
spring.config.activate.on-profile: partitioned_instance_2
spring.cloud.stream.bindings:

```

```
messageProcessor-in-0:
  consumer:
    partitioned: true
    instanceIndex: 2
    instanceCount: 3
```

Il valore `instanceCount` rappresenta il numero totale di istanze dell'applicazione tra cui i dati devono essere partizionati. L'`instanceIndex` deve essere un valore univoco tra le diverse istanze, con un valore compreso tra 0 e `instanceCount - 1`. L'indice dell'istanza aiuta ogni istanza dell'applicazione a identificare la partizione unica dalla quale riceve i dati. È richiesto dai binders che utilizzano tecnologie che non supportano nativamente il partizionamento. Ad esempio, con RabbitMQ, c'è una coda per ogni partizione, con il nome della coda contenente l'indice dell'istanza.

Avvia il sistema con i seguenti comandi:

```
export COMPOSE_FILE=docker-compose-one-to-many-partitions.yml
mvn clean package -Dmaven.test.skip=true
docker compose up --build --detach
```

Utilizzando l'[interfaccia web di LavinMQ](#), possiamo vedere che lo scambio di messaggi riceve 5 eventi/s e pubblica gli stessi eventi su tre diverse (nominative) code. Ogni evento viene consumato una sola volta da un solo consumatore. Pertanto, il tasso di output è di 5 eventi/s. Tuttavia, controllando i log, è possibile osservare come ciascun consumatore riceva tutti i cinque messaggi relativi allo stesso ID.

Routing

Considera uno scenario in cui più consumatori devono gestire messaggi di diversi tipi, come CREATE, UPDATE e DELETE. Questo può essere gestito in modo efficiente utilizzando tecniche di routing all'interno di un broker di messaggi.

I produttori possono assegnare ai messaggi una **chiave di routing** (una stringa che rappresenta il tipo di messaggio), consentendo al broker di inoltrare ogni messaggio alla coda appropriata in base alla sua chiave.

```
services:
  publisher:
    image: async-rabbitmq-publisher
    build: async-rabbitmq-publisher
    environment:
      - SPRING_PROFILES_ACTIVE=docker,routed
    depends_on:
      - lavinmq:
          condition: service_healthy
    deploy:
      resources:
        limits:
          memory: 512m

  consumer-0:
    image: async-rabbitmq-consumer
    build: async-rabbitmq-consumer
    environment:
      - SPRING_PROFILES_ACTIVE=docker,routed_instance_CREATE
    depends_on:
      - lavinmq:
          condition: service_healthy
    deploy:
      resources:
        limits:
          memory: 512m
```

```

consumer-1:
  image: async-rabbitmq-consumer
  build: async-rabbitmq-consumer
  environment:
    - SPRING_PROFILES_ACTIVE=docker,routed_instance_UPDATE
  depends_on:
    lavinmq:
      condition: service_healthy
  deploy:
    resources:
      limits:
        memory: 512m

consumer-2:
  image: async-rabbitmq-consumer
  build: async-rabbitmq-consumer
  environment:
    - SPRING_PROFILES_ACTIVE=docker,routed_instance_DELETE
  depends_on:
    lavinmq:
      condition: service_healthy
  deploy:
    resources:
      limits:
        memory: 512m

```

Il profilo *routed* aggiunge le seguenti configurazioni.

Lato Publisher:

```

spring.config.activate.on-profile: routed
spring.cloud.stream:
  rabbit:
    bindings:
      message-out-0:
        producer:
          routingKeyExpression: headers['routingKey']

```

Lato Consumer:

```

spring.config.activate.on-profile: routed_instance_CREATE
spring.cloud.stream:
  rabbit:
    bindings:
      messageProcessor-in-0:
        consumer:
          binding-routing-key: 'CREATE'

---
spring.config.activate.on-profile: routed_instance_UPDATE
spring.cloud.stream:
  rabbit:
    bindings:
      messageProcessor-in-0:
        consumer:
          binding-routing-key: 'UPDATE'

---

```

```

spring.config.activate.on-profile: routed_instance_DELETE
spring.cloud.stream:
  rabbit:
    bindings:
      messageProcessor-in-0:
        consumer:
          binding-routing-key: 'DELETE'

```

Avvia il sistema con i seguenti comandi:

```

export COMPOSE_FILE=docker-compose-one-to-many-routed.yml
mvn clean package -Dmaven.test.skip=true
docker compose up --build --detach

```

Utilizzando l'[interfaccia web di LavinMQ](#), è possibile osservare che lo scambio di messaggi riceve 5 eventi/s e pubblica gli stessi eventi su tre diverse (anonime) code. Ogni evento viene consumato una sola volta da un solo consumatore. Tuttavia, controllando i log, è possibile osservare come ciascun consumatore riceva tutti i messaggi dello stesso tipo.

Infrastructure

Scoperta dei Servizi

In un'architettura distribuita, individuare gli indirizzi IP dei servizi per la comunicazione è essenziale. Questo processo, noto come **scoperta dei servizi**, è stato un concetto fondamentale nel calcolo distribuito sin dalla sua origine.

La scoperta dei servizi è fondamentale per due motivi chiave:

- **Scalabilità Orizzontale:** Le architetture a microservizi richiedono aggiustamenti dinamici, come il ridimensionamento orizzontale mediante l'aggiunta di più istanze di servizio (ad esempio, container aggiuntivi). Questa capacità sposta i team di sviluppo da una scalabilità verticale, consentendo un approccio più resiliente e scalabile.
- **Resilienza:** Per prevenire che i guasti si propaghino ai servizi dipendenti, le architetture a microservizi devono gestire in modo efficace le istanze non sane o non disponibili. I motori di scoperta dei servizi monitorano continuamente la salute del servizio e rimuovono le istanze guaste dal registro, garantendo che il traffico venga reindirizzato verso istanze sane, riducendo al minimo le interruzioni.

Il problema con la scoperta dei servizi basata su DNS

Nel mondo non cloud, la risoluzione della posizione del servizio veniva spesso risolta attraverso una combinazione di DNS e un bilanciatore di carico di rete.

Sebbene questo tipo di modello funzioni bene con applicazioni che hanno un numero relativamente ridotto di servizi in esecuzione su un gruppo di server statici, non funziona bene per architetture a microservizi. Le ragioni di ciò includono: I bilanciatori di carico tradizionali sono **gestiti staticamente**.

Non sono progettati per una registrazione e deregistrazione rapida dei servizi. In uno scenario di bilanciamento di carico tradizionale, la **registrazione di nuove istanze di servizio non avviene quando inizia una nuova istanza di servizio**. Qui di seguito un esempio di configurazione nginx che gestisce un insieme di tre repliche. La sua natura statica è evidente.

```

worker_processes auto;

events {
  worker_connections 1024;
}

http {

```