Università di Catania

DISTRIBUTED SYSTEMS AND BIG DATA
A.A. 2025/26

Homework 2 DSBD

STUDENTI:

Alessia Fichera 1000084787

Davide Pantò 1000081854

Abstract

L'architettura del sistema si è evoluta passando da un modello a microservizi sincrono a un'architettura resiliente e centralizzata. Rispetto alla versione precedente, sono stati introdotti componenti fondamentali per garantire la scalabilità, il disaccoppiamento tra i servizi e la tolleranza ai guasti verso sistemi esterni.

1. **API Gateway**

Per migliorare la sicurezza e semplificare l'interazione con i client, è stato introdotto un API Gateway (basato su Nginx).

- Funge da unico punto di ingresso per tutte le richieste esterne.
- Maschera la complessità della rete interna dei microservizi. Il client non comunica più direttamente con le singole istanze, UserManager o DataCollector su porte diverse, ma invia tutte le richieste alla porta standard HTTP (80), che vengono poi instradate internamente al servizio di competenza in base al percorso (`/user` o `/data`).

2. Aggiornamento Schema DataDB

In accordo con le nuove funzionalità di monitoraggio, la tabella Interessi all'interno del DataDB è stata estesa per ospitare le regole di allarme personalizzate per ogni utente. Oltre a collegare un utente a un aeroporto, la tabella ora definisce i criteri specifici che attivano le notifiche.

Tabella Interessi: Questa tabella memorizza le preferenze di monitoraggio degli utenti. Rispetto alla versione precedente, sono stati aggiunti i campi *high_value* e *low_value* per supportare la logica condizionale dell'Alert System.

Questi nuovi attributi sono essenziali per il componente *Alert System*. Durante la fase di *Processing*, il worker interroga questa tabella per confrontare i dati in tempo reale (conteggio dei voli scaricati) con i valori soglia (high_value, low_value) configurati dall'utente, determinando così se produrre o meno un evento di notifica su Kafka.

3. Infrastruttura di Messaging Asincrono (Apache Kafka)

Per disaccoppiare i processi di raccolta dati da quelli di monitoraggio e notifica, è stato integrato un **Message Broker** (Kafka).

- Il sistema adotta un approccio "Producer-Consumer". Il Data Collector non invoca direttamente il sistema di alert, ma pubblica eventi su un canale condiviso.
- Topologie di Comunicazione:

    - **Canale di Aggiornamento Dati:** Utilizzato per segnalare che nuovi dati di volo sono disponibili e pronti per essere analizzati.
    - **Canale di Notifica:** Utilizzato per trasmettere gli allarmi generati, contenenti le informazioni necessarie per contattare l'utente .

4. Pipeline di Alerting e Notifica

La logica di monitoraggio è stata separata in due microservizi distinti che operano in background, coordinati da un timestamp generato dal Data Collector che garantisce la sincronizzazione degli allarmi con l'ultimo aggiornamento dei dati.

- **Timestamping e Sincronizzazione:** Il **DataCollector** è l'autorità temporale del sistema. Al termine di ogni ciclo di scaricamento, genera un timestamp e lo inserisce nel payload del messaggio pubblicato sul topic Kafka to-alert-system. Questo approccio certifica l'orario di validità del dato direttamente alla fonte. Il messaggio viene utilizzato dupliciamente:
    - Dall'**Alert System**: Notifica che nuovi dati sono pronti per essere analizzati.
    - Dallo stesso **DataCollector** (in auto-consumo) che ascolta il messaggio da lui prodotto per aggiornare il proprio stato interno (variabile globale timestamp). Questo meccanismo permette di esporre l'API REST /check_time_update (gestita tramite Flask), offrendo all'utente la possibilità di verificare su richiesta l'orario dell'ultimo aggiornamento dati completato con successo.
- **Alert System (Stream Processor):**

  Questo componente agisce come nodo di elaborazione intermedio, svolgendo il duplice ruolo di **Consumer** e **Producer**:

    - Consumer: Intercetta il segnale di aggiornamento dal DataCollector per avviare la verifica delle soglie non appena i nuovi dati sono effettivamente disponibili nel database.
    - Processing: Interroga il database per verificare puntualmente se i nuovi dati violano le condizioni (valori High/Low) definite dagli utenti.
    - Producer: In caso di violazione delle soglie, produce un nuovo evento sul topic di notifica (to-notifier), disaccoppiando così la logica di rilevamento dell'anomalia da quella di invio della comunicazione.
- **Alert Notifier System (Consegna):**
    - Agisce come *Consumer* degli eventi di allarme.
    - Si occupa esclusivamente dell’invio di un email agli utenti interessati per informarli che il valore dei voli di loro interesse risulta essere maggiore o minore delle soglie indicate.

5. Resilienza e Tolleranza ai Guasti (Circuit Breaker)

Il sistema implementa il pattern **Circuit Breaker** per garantire resilienza nelle comunicazioni con l'API OpenSky. Il componente gestisce la stabilità del servizio alternando tre stati distinti:

- **Stato CLOSED :** È la condizione di normale funzionamento. In questo stato, tutte le chiamate verso l'API esterna vengono eseguite regolarmente.
- **Stato OPEN :** Si attiva automaticamente quando viene raggiunta la soglia di **2 fallimenti consecutivi**. In questo stato, il circuito blocca preventivamente qualsiasi richiesta verso OpenSky sollevando un'eccezione, senza tentare la connessione, per evitare tempi di attesa inutili e sovraccarico.
- **Stato HALF-OPEN:** Dopo un periodo di recovery timeout (impostato a 15 secondi), il circuito permette il passaggio di una singola "richiesta sonda". Se questa ha successo, il sistema torna allo stato Closed; se fallisce, ritorna immediatamente allo stato Open.

Comportamento del Sistema in caso di Guasto:

Quando il circuito passa allo stato Open, il Data Collector reagisce modificando la propria strategia operativa:

1. Interrompe immediatamente i tentativi di download massivo dei voli.
2. Entra in una modalità di Fallback, riducendo l'intervallo di polling da 12 ore a 1 minuto.
3. Esegue controlli leggeri e frequenti (richiedendo lo stato di un singolo volo) sfruttando il meccanismo *Half-Open* per verificare la disponibilità del servizio.
4. Non appena il servizio torna disponibile (transizione a *Closed*), ripristina automaticamente il ciclo di aggiornamento completo ogni 12 ore.

Questa scelta implementativa risiede nel bilanciare **tolleranza, reattività e disponibilità**. La soglia di **2 fallimenti** è scelta per tollerare un singolo errore di rete, ma reagire prontamente in caso di disservizio reale. Aspettare il ciclo successivo di **12 ore** dopo un guasto confermato comporterebbe un inaccettabile periodo di inattività. Per questo, il sistema commuta immediatamente su **richieste leggere** ogni **60 secondi**. Questo accelera l'**auto-ripristino**, permettendo di rilevare il ritorno in linea di OpenSky e ripristinare il download completo nel minor tempo possibile.

Diagramma Architetturale:

Il diagramma rappresenta l'architettura dell'applicazione, mostrando le principali interazioni tra i microservizi:

- **Data Collector**: Micro Servizio che raccoglie in modo continuativo le informazioni sui voli dall'API esterna OpenSky Network, aggiornando nelle tabelle del DataDB.
- **User Manager** :Micro servizio che gestisce le richieste di registrazione, login e cancellazione dell'utente nello UserDB.
- **gRPC Service User**: Verifica l'autenticazione prima di processare richieste sui voli.
- **gRPC Service Data**: Rimuove gli interessi dell'utente quando viene cancellato dal sistema.
- **DataDB**: Contiene le tabelle con le informazioni sui voli.
- **UserDB**: Contiene le tabelle con le informazioni sugli utenti.
- **Redis:** funge da cache temporanea per intercettare i tentativi duplicati di registrazione ed evitare interrogazioni inutili al database.

Componenti aggiuntive:

- **API Gateway (Nginx)**: Punto di ingresso unico che instrada le richieste REST verso i microservizi competenti (User Manager o Data Collector).
- **Broker Kafka**: Bus di messaggistica asincrono che disaccoppia la raccolta dati dalla pipeline di alerting, distribuendo i trigger di aggiornamento.
- **Alert System**:Componente Consumer/Producer che verifica le soglie nel DB e genera gli eventi di allarme.
- **Alert Notifier System**: Consumer dedicato esclusivamente all'invio delle notifiche email agli utenti.

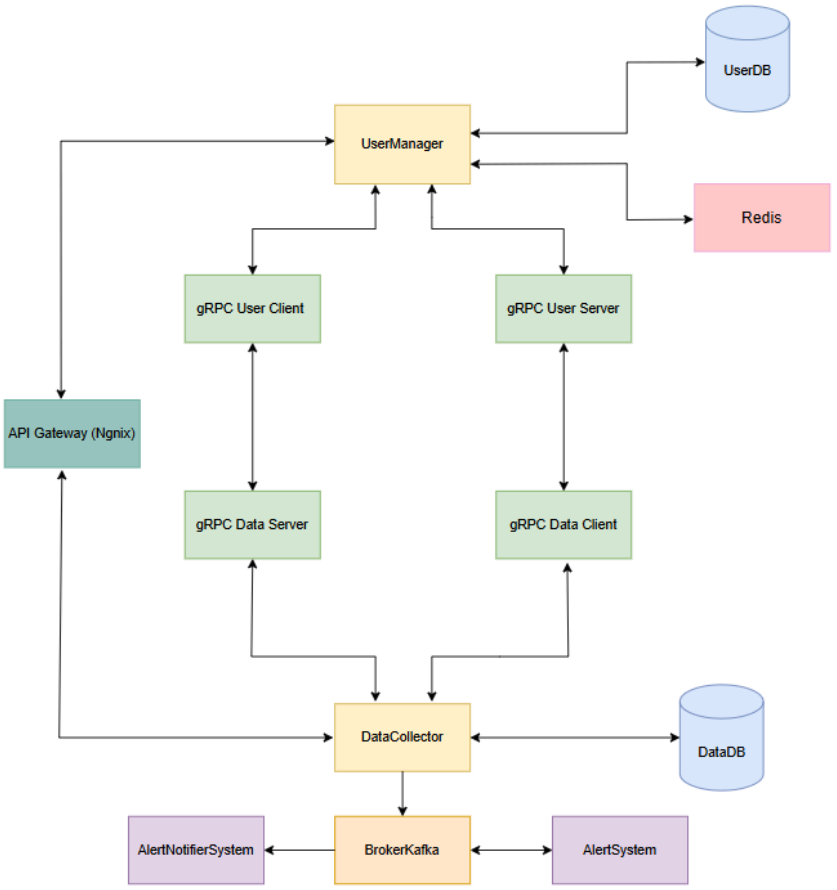


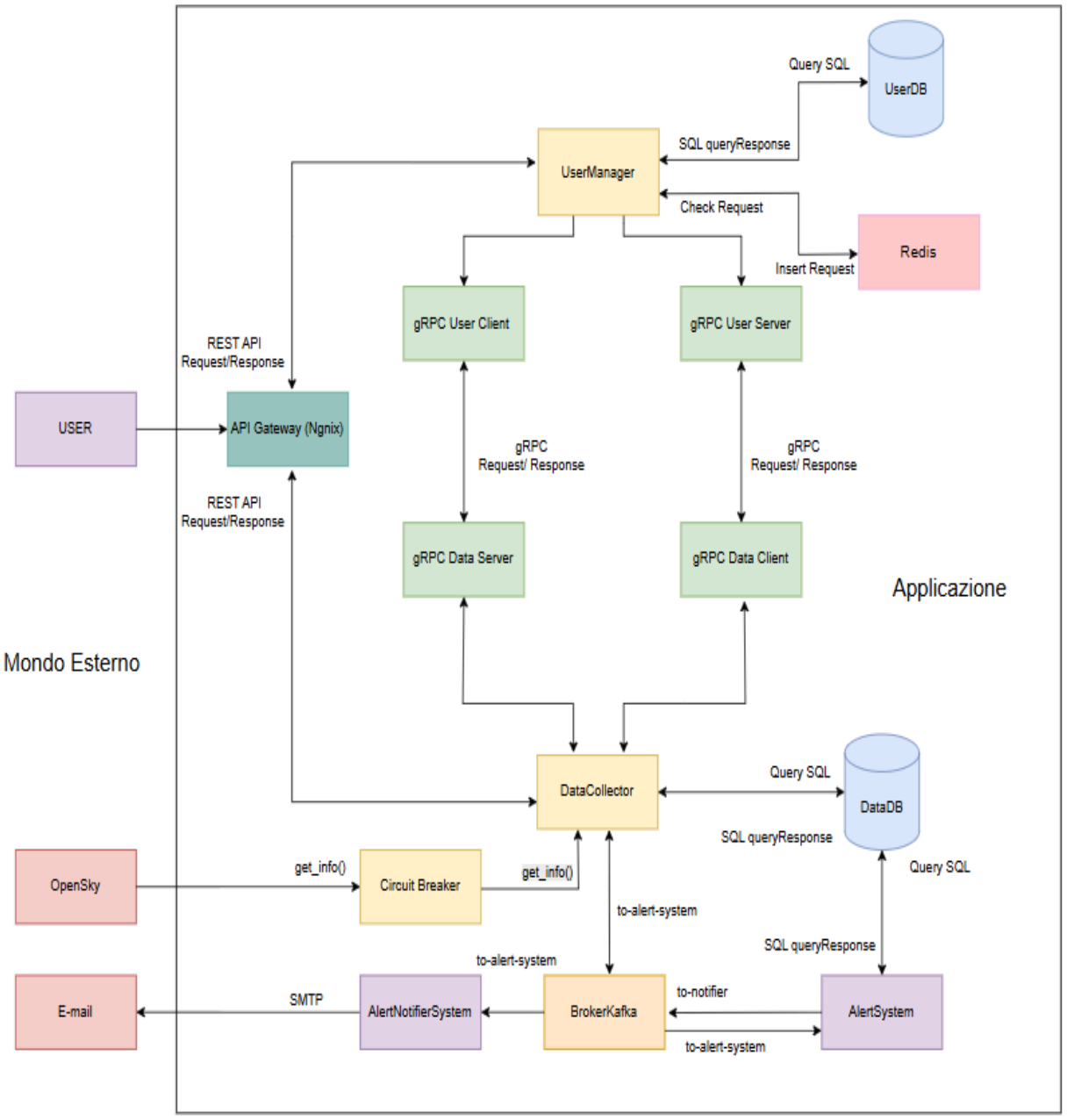
Diagramma delle Interazioni

Il diagramma mostra un'architettura a microservizi composta da **UserManager** e **DataCollector**, che espongono interfacce **REST API** verso l'utente (mondo esterno) e comunicano tra loro tramite protocollo **gRPC**.

- **Comunicazione gRPC (Bidirezionale):**
  - **User Client → Data Server:** Lo User Manager invia comandi al Data Collector (cancellazione dati).
  - **Data Client → User Server:** Il Data Collector interroga lo User Manager (verifica autenticazione).
- **Database Dedicati:** Ogni servizio gestisce il proprio archivio (**UserDB** e **DataDB**) tramite query SQL.
- **OpenSky:** Servizio esterno interrogato dal DataCollector per reperire le informazioni sui voli in tempo reale.
- **Redis:** funge da cache temporanea per intercettare i tentativi duplicati di registrazione ed evitare interrogazioni inutili al database.
- **API Gateway (Nginx)**: Punto di ingresso unico che instrada le richieste REST verso i microservizi competenti (User Manager o Data Collector).
- **Broker Kafka**: Bus di messaggistica asincrono che disaccoppia la raccolta dati dalla pipeline di alerting, distribuendo i trigger di aggiornamento.
- **Alert System**:Componente Consumer/Producer che verifica le soglie nel DB e genera gli eventi di allarme.
- **Alert Notifier System**: Consumer dedicato esclusivamente all'invio delle notifiche email agli utenti.

  Componenti aggiuntive:

- **Circuit Breaker:** Pattern di resilienza interposto tra il Data Collector e OpenSky; monitora i fallimenti delle chiamate esterne per prevenire sovraccarichi o attese inutili in caso di disservizi dell'API remota.
- **Servizio E-mail:** Server di posta esterno contattato via SMTP dall'Alert Notifier System per la consegna effettiva delle email agli utenti.


Query SQL
UserDB
SQL queryResponse
UserManager
Check Request
Redis
Insert Request
gRPC User Client
gRPC User Server
REST API
Request/Response
USER
API Gateway (Ngnix)
gRPC
Request/ Response
gRPC
Request/ Response
REST API
Request/Response
gRPC Data Server
gRPC Data Client
Applicazione
Mondo Esterno
DataCollector
Query SQL
DataDB
SQL queryResponse
Query SQL
OpenSky
get_info()
Circuit Breaker
get_info()
to-alert-system
SQL queryResponse
to-alert-system
E-mail
SMTP
AlertNotifierSystem
BrokerKafka
to-notifier
AlertSystem
to-alert-system

Lista delle API Data Collector:

Nome API	Descrizione	Messaggio di Richiesta	Messaggio di Risposta
sendInterest()	Registra l'interesse di un utente per un aeroporto,una modalità* specificata e include le soglie di allarme per l'Alert System.	**SendInterestRequest** email (string) token (string) airport_code (string) mode (boolean) h_value (int) l_value (int)	**SendInterestResponse** message (string)
delete_interest()	Rimuove l'interesse registrato di un utente.	**DeleteInterestRequest** email (string) token (string) airport_code (string) mode (boolean) h_value (int) l_value (int)	**DeleteInterestResponse** message (string)
get_info()	Recupera le informazioni sui voli per un determinato aeroporto e modalità*.	**GetInfoRequest** email (string) token (string) airport_code (string) mode (boolean)	**GetInfoResponse** count (int) voli (list of objects): partenza (string) ora_arrivo (string) ora_partenza (string) arrivo (string) codice (string)
get_last_one()	Recupera l'ultimo volo in arrivo e in partenza disponibile per l'aeroporto.	**GetLastValueRequest** email (string) token (string) airport_code (string)	**GetLastValueResponse** count (int) voli (list of objects): partenza (string) ora_arrivo (string) ora_partenza (string) arrivo (string) codice (string)
get_avgs()	Calcola la media dei voli (arrivi e partenze) negli ultimi N giorni.	**GetAveragesRequest** email (string) token (string) airport_code (string) n_days (int/string)	**GetAveragesResponse** media arrivi (double) media partenze (double)

check_time_update()	Restituisce il timestamp dell'ultimo aggiornamento dati ricevuto via Kafka.	**CheckTimeRequest** email (string) token (string)	**CheckTimeResponse** message (string)

*modalità: si intende aeroporto di arrivo o di partenza.

Lista delle API User Manager:

Nome API	Descrizione	Messaggio di Richiesta	Messaggio di Risposta
login()	Esegue l'autenticazione di un utente verificando le credenziali (email e password).	**LoginRequest** email (string) password (string)	**LoginResponse** message (string)
registrazione()	Registra un nuovo utente nel sistema salvando email, username e password (con hashing).	**RegistrationRequest** email (string) username (string) password (string)	**RegistrationResponse** message (string)
cancellazione()	Elimina definitivamente un account utente e i relativi dati (sessione e interessi), previa verifica della password.	**DeleteAccountRequest** email (string) password (string)	**DeleteAccountResponse** message (string)