Relazione Homework 1 DSBD

Lorenzo Varsallona (1000084765) - Paola Pappalardo (1000035439)

Indice

1 Introduzione e Obiettivi

Il presente documento illustra il funzionamento e le decisioni progettuali adottate per lo sviluppo dell'applicativo. In particolare, nelle sezioni successive si approfondirà quanto è stato utilizzato e il funzionamento dei microservizi implementati, con particolare enfasi sulle scelte architetturali che hanno guidato la progettazione del sistema.

1.1 Architettura Generale

L'applicativo è stato strutturato secondo un'architettura a microservizi, suddiviso in più moduli Python per garantire una suddivisione pulita, elegante e modulare del codice. Questa scelta architetturale consente il riutilizzo di componenti comuni e facilita la manutenzione futura del sistema.

L'applicativo prevede due microservizi principali:

- **User-Manager**: responsabile della gestione degli utenti e della memorizzazione dei dati anagrafici.
- **Data-Collector**: responsabile della raccolta dati di voli da API esterne e della gestione delle preferenze aeroportuali degli utenti.

Questi due microservizi comunicano tra loro tramite un canale gRPC bidirezionale.

1.2 Infrastruttura e Database

Per il database, vista la semplicità attuale del progetto, si è deciso di utilizzare due database PostgreSQL distinti (uno per ogni microservizio) situati all'interno dello stesso container. Questa decisione facilita lo sviluppo e il testing in ambienti locali, consentendo successivamente di scalare i database su container separati o servizi gestiti senza modificare l'applicazione.

1.3 Politica di Idempotenza e Affidabilità

Un requisito fondamentale del progetto è l'implementazione della politica *at-most-once* per evitare la duplicazione dei messaggi. Questo è stato realizzato attraverso un meccanismo di cache basato su *message IDs* (identificatori univoci per ogni richiesta) memorizzati nel database con scadenza temporale (5 minuti).

Quando una richiesta viene elaborata, il message ID e la risposta vengono salvati in cache. Se la stessa richiesta viene ricevuta entro il periodo di scadenza, il sistema restituisce la risposta memorizzata senza rielaborare l'operazione, garantendo così l'idempotenza.

1.4 Fonte Dati e Funzionalità

L'API OpenSky è stata scelta come fonte dati principale per la raccolta delle informazioni di volo. Questa API fornisce accesso a dati reali di aviazione civile, permettendo di implementare tutte le funzionalità richieste dalla specifica, oltre ad alcune funzionalità aggiuntive che verranno approfondite nelle sezioni seguenti.

I dati vengono raccolti secondo una finestra temporale specifica, scalata di 2 ore indietro per garantire la *data availability* dei voli, e questi ultimi vengono sincronizzati periodicamente con una frequenza di 12 ore.

2 Architettura del sistema e decomposizione

In questa sezione si approfondiranno gli aspetti implementativi adottati all'interno dell'applicativo, con particolare attenzione alle scelte tecnologiche e architetturali.

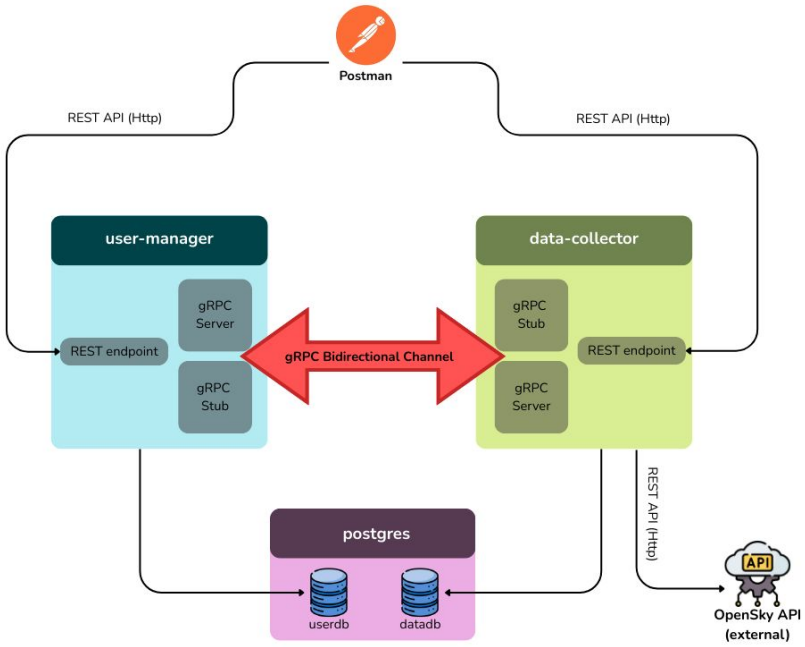

Figura 1: Diagramma architetturale generale del sistema.

2.1 Database

L'applicativo utilizza due database PostgreSQL distinti:

- **userdb**: gestisce i dati anagrafici degli utenti e la cache per l'idempotenza dei messaggi.
- **datadb**: gestisce i dati di volo e le preferenze aeroportuali degli utenti.

Per l'interazione con i database PostgreSQL, è stato scelto **SQLAlchemy** tramite l'integrazione `Flask-SQLAlchemy`. Questa scelta offre numerosi vantaggi:

- **Astrazione dal database**: il codice Python rimane indipendente dal motore SQL specifico.
- **Type safety**: i modelli ORM garantiscono type checking e validazione a livello applicativo.
- **Query builder**: le query vengono costruite direttamente tramite funzioni di SQLAlchemy, riducendo anche il rischio di SQL-Injection.
- **Relationship**: gestione automatica e semplificata delle relazioni tra tabelle.
- **Session management**: gestione automatica delle transazioni e del ciclo di vita degli oggetti in memoria (commit, rollback, ecc).

Inoltre, SQLAlchemy consente di definire il modello dati interamente in Python, sincronizzando automaticamente lo schema del database tramite `db.create_all()`.

2.1.1 Database userdb

Il database userdb contiene due tabelle principali, gestite attraverso i modelli ORM definiti nel file `models.py`:

Tabella `users` Questa tabella memorizza i dati anagrafici degli utenti registrati nel sistema:

Campo	Tipo	Vincolo
`email`	`VARCHAR(255)`	PK
`name`	`VARCHAR(255)`	NOT NULL
`surname`	`VARCHAR(255)`	NOT NULL
`fiscal_code`	`VARCHAR(16)`	Nullable
`bank_info`	`VARCHAR(200)`	Nullable

Tabella `message_ids` Questa tabella implementa il meccanismo di cache per la politica *at-most-once*, memorizzando gli ID dei messaggi già elaborati:

Campo	Tipo	Vincolo
`id`	`VARCHAR(36)`	PK
`response_data`	`JSON`	Nullable
`response_status`	`INTEGER`	Nullable
`expires_at`	`TIMESTAMP`	NOT NULL

L'utilizzo del tipo `JSON` per memorizzare `response_data` consente di salvare risposte di qualsiasi forma senza necessità di schema predefinito. Un processo di pulizia periodico (cleanup job schedulato ogni 5 minuti) rimuove automaticamente i record scaduti, evitando la crescita indefinita della tabella.

2.1.2 Database datadb

Il database datadb contiene due tabelle principali, gestite attraverso i modelli ORM definiti nel file `models.py`:

Tabella `user_airports` Questa tabella implementa la relazione molti-a-molti tra utenti e aeroporti monitorati:

Campo	Tipo	Vincolo
`id`	`INTEGER`	PK
`user_email`	`VARCHAR(255)`	NOT NULL
`airport_code`	`VARCHAR(10)`	NOT NULL

Ogni riga rappresenta l'iscrizione di un utente al monitoraggio di un determinato aeroporto. La coppia `(user_email, airport_code)` deve essere univoca per evitare duplicati.

Tabella `flights` Questa tabella memorizza i dati di volo raccolti periodicamente dall'API OpenSky.

Campo	Tipo	Vincolo
`id`	`INTEGER`	PK
`icao24`	`VARCHAR(20)`	Nullable
`callsign`	`VARCHAR(20)`	Nullable
`est_departure_airport`	`VARCHAR(10)`	Nullable
`est_arrival_airport`	`VARCHAR(10)`	Nullable
`first_seen`	`INTEGER`	Nullable
`last_seen`	`INTEGER`	Nullable
`airport_monitored`	`VARCHAR(10)`	Nullable
`direction`	`VARCHAR(10)`	Nullable

I campi `first_seen` e `last_seen` memorizzano timestamp. Il campo `direction` può assumere i valori 'arrival' o 'departure' per distinguere i voli in arrivo da quelli in partenza. La deduplicazione durante la raccolta si basa sulla combinazione univoca `(icao24, first_seen, airport_monitored, direction)`.
N.B.: se un utente o i suoi interessi vengono eliminati, i voli associati vengono rimossi solo nel caso in cui quell'utente sia l'unico ad aver sottoscritto il relativo codice ICAO. Se invece altri utenti hanno lo stesso interesse, verrà eliminato solo l'interesse dell'utente, mentre i voli resteranno nel database.

2.2 Microservizio user-manager

Il microservizio **user-manager** è responsabile della gestione del ciclo di vita degli utenti: registrazione, eliminazione e gestione delle loro preferenze aeroportuali. Il servizio è costruito su Flask e implementa sia endpoint HTTP REST per l'interazione con i client, sia un server gRPC per la comunicazione bidirezionale con il microservizio data-collector.

2.2.1 Architettura Interna

L'applicazione è strutturata mediante tre componenti principali, definiti in altrettanti moduli Python:

- **user-manager.py**: punto di ingresso dell'applicazione. Inizializza il database tramite `db.create_all()`, avvia lo scheduler di background per la pulizia periodica della cache di idempotenza (ogni 5 minuti), e istanzia il thread gRPC che rimane in ascolto su porta 50051.

- **routes.py**: contiene i tre endpoint REST principali esposti dal servizio, ognuno dei quali gestisce logica di business specifica e comunica con il data-collector tramite gRPC.
- **grpc_service.py**: implementa il server gRPC con i servizi definiti nel file proto (`UserService` e `CollectorService`).

2.2.2 Endpoint REST

Il microservizio espone tre endpoint HTTP principali su porta 5000:

- **POST /addUser**: crea un nuovo utente nel database. Implementa la politica *at-most-once* controllando innanzitutto se il `message_id` esiste già in cache; in caso affermativo, restituisce la risposta precedentemente memorizzata. Successivamente, valida l'unicità dell'email e del codice fiscale. Se tutti i controlli hanno esito positivo, salva l'utente e la coppia `(message_id, response)` nella cache con TTL di 5 minuti. Ritorna HTTP 201 (Created) per successo, 409 (Conflict) per duplicati.

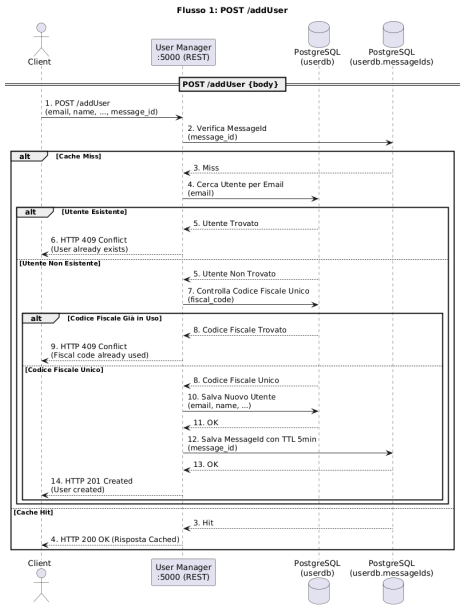

Figura 2: Diagramma di sequenza POST/addUser

- **DELETE /deleteUser**: elimina un utente dal database in base all'email fornita. Dopo aver rimosso l'utente da userdb, comunica immediatamente con il data-collector via gRPC tramite il metodo `CleanupUser()`, notificando al secondo servizio di rimuovere tutti i dati associati a quell'utente (voli, preferenze aeroportuali). Questo garantisce la coerenza tra i due database.

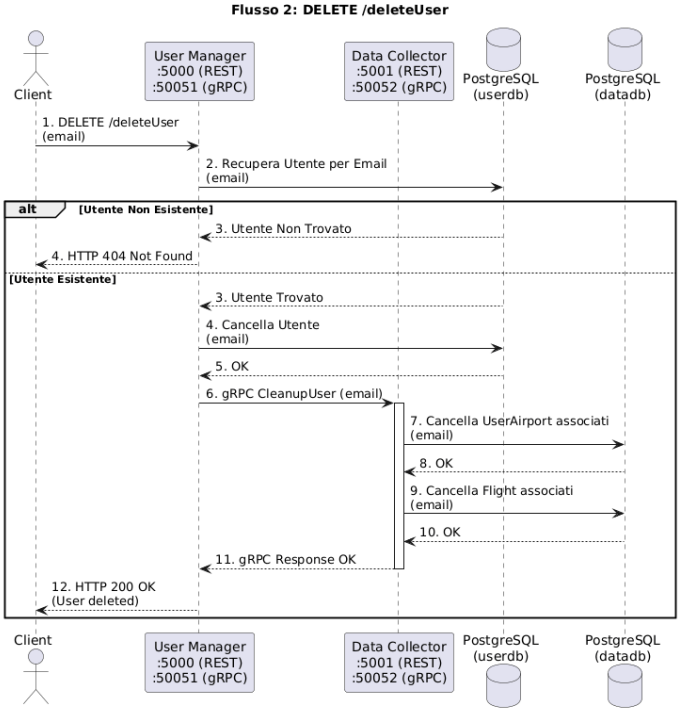

Figura 3: Diagramma di sequenza DELETE/deleteUser

- **DELETE /removeAirportInterest**: rimuove uno specifico aeroporto dalle preferenze monitorate di un utente. Effettua una validazione preliminare sull'esistenza dell'utente, quindi comunica con il data-collector tramite gRPC usando il metodo `RemoveAirportInterest()`, che provvede a rimuovere l'aeroporto dalla tabella `user_airports` in datadb.

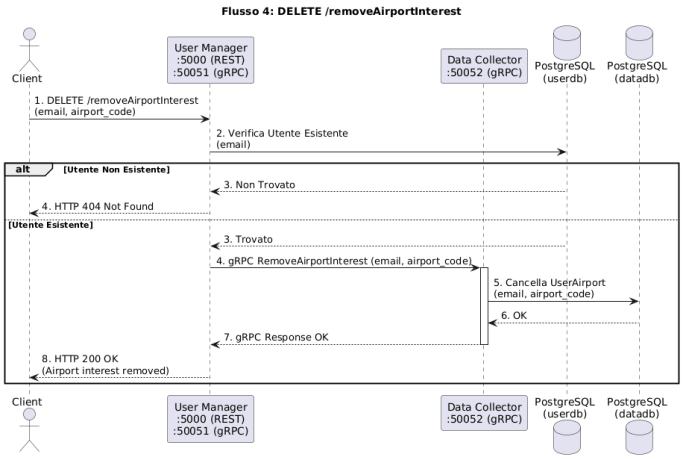

Figura 4: Diagramma di sequenza DELETE/removeAirportInterest

2.2.3 Comunicazione gRPC

Il microservizio implementa due servizi gRPC definiti nel file `user_manager.proto`:

- **UserService**: offre i metodi `CheckUser()` e `GetUser()` per permettere al data-collector di verificare l'esistenza di un utente e recuperarne i dati anagrafici. Questi metodi sono invocati dal data-collector quando un utente effettua operazioni che richiedono validazione.
- **CollectorService**: espone i metodi `CleanupUser()` e `RemoveAirportInterest()` che ricevono richieste dal data-collector per sincronizzare le operazioni.

2.3 Microservizio data-collector

Il microservizio **data-collector** è responsabile della raccolta periodica dei dati di volo dall'API OpenSky e della gestione delle preferenze aeroportuali degli utenti. Analogamente a user-manager, è costruito su Flask ed espone sia endpoint HTTP REST per operazioni sincrone e query, sia un server gRPC per ricevere notifiche di modifica dai dati gestiti da user-manager.

2.3.1 Architettura Interna

L'applicazione è strutturata in diverse componenti principali:

- **data-collector.py**: punto di ingresso dell'applicazione. Inizializza il database tramite `db.create_all()`, avvia uno scheduler di background che esegue la raccolta periodica dei voli ogni 12 ore mediante la funzione `run_scheduled_flights()`, e istanzia il thread gRPC che rimane in ascolto su porta 50052.
- **routes.py**: contiene i molteplici endpoint REST che espongono funzionalità di query sui dati di volo, registrazione delle preferenze aeroportuali, e strumenti di debug. Tutti gli endpoint che modificano dati aeroportuali comunicano con user-manager tramite gRPC per validare l'esistenza degli utenti.
- **grpc_service.py**: implementa il server gRPC con il servizio `CollectorService`, che riceve richieste da user-manager per sincronizzare le operazioni di eliminazione (`CleanupUser` e `RemoveAirportInterest`).
- **services.py**: contiene la logica di raccolta dei dati di volo. La funzione `collect_flights()` interroga l'API OpenSky per arrivals e departures, deduplica i voli, e li salva in datadb.
- **token_manager.py**: gestisce l'autenticazione OAuth2 con l'API OpenSky, fornendo token validi per le richieste autenticate.

2.3.2 Endpoint REST

Il microservizio espone i seguenti endpoint HTTP su porta 5001:

- **POST /register_airports**: registra gli interessi aeroportuali di un utente. Prima valida l'esistenza dell'utente tramite gRPC con user-manager (`CheckUser`), poi salva le coppie `(email, airport_code)` nella tabella `user_airports`. Successivamente, avvia in background un thread che esegue immediatamente la raccolta dati (`collect_flights`) per i soli aeroporti registrati, popolando il database con i voli storici. Ritorna HTTP 200 se la registrazione ha successo.

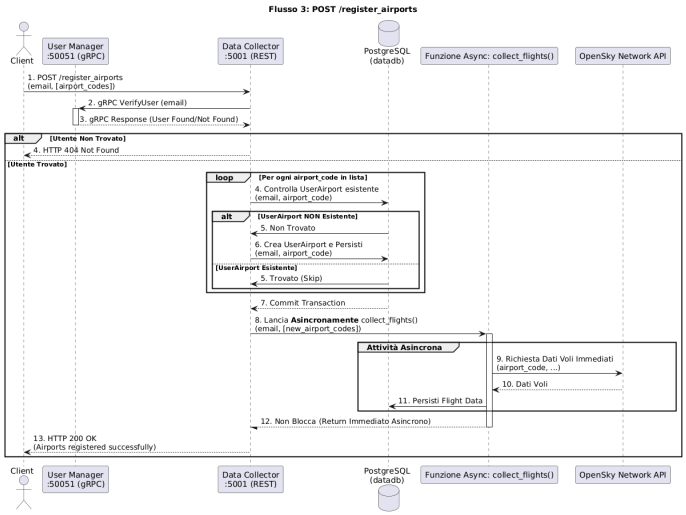

Figura 5: Diagramma di sequenza POST/registerAirports

- **GET /user_info/<email>**: recupera le informazioni complete su un utente, combinando i dati anagrafici provenienti da user-manager (via gRPC `GetUser`) con l'elenco degli aeroporti di interesse salvati localmente. Ritorna 404 se l'utente non esiste in user-manager.

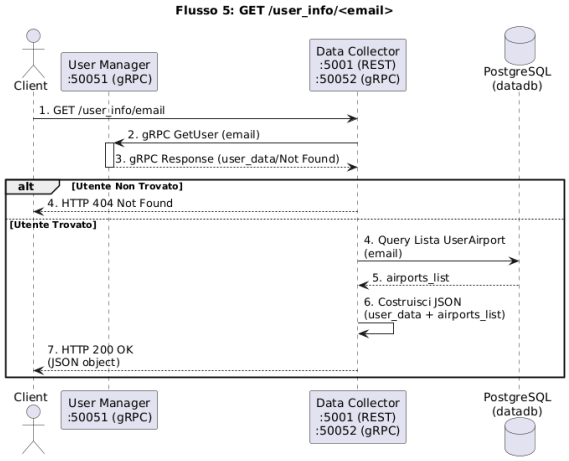

Figura 6: Diagramma di sequenza GET/user_info/<email>

- **GET /airports/<airport_code>/last_flight**: restituisce l'ultimo volo registrato per un dato aeroporto. Supporta un parametro query opzionale "direction" ('arrival' o 'departure') per filtrare i risultati.

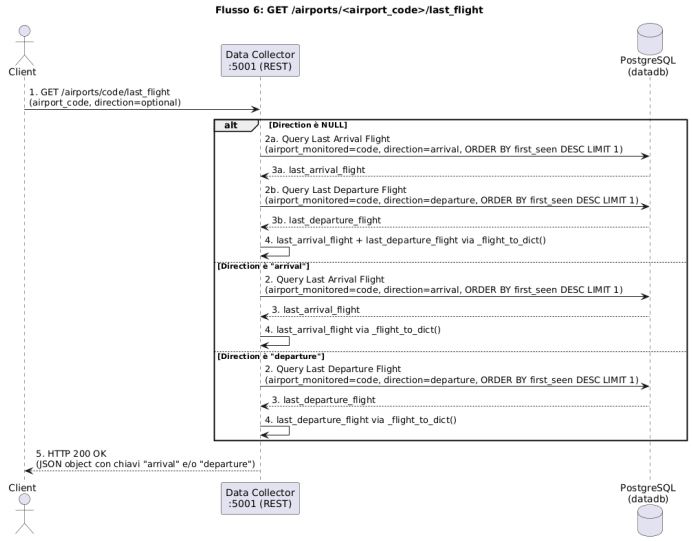

Figura 7: Diagramma di sequenza GET/airports/<airport_code>/last_flight

- **GET /airports/<airport_code>/average_flights**: calcola la media di voli per giorno per un aeroporto, in una finestra temporale configurabile (parametro "days", default 7). Supporta filtro opzionale per direzione.

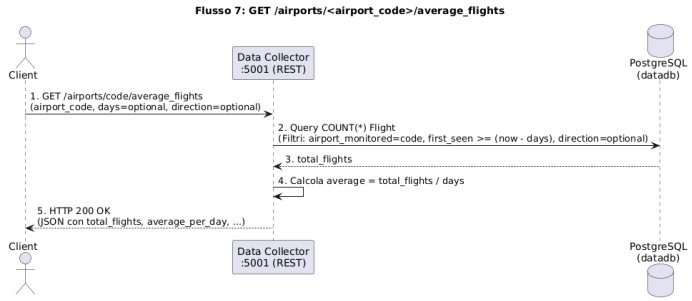

Figura 8: Diagramma di sequenza GET /airports/<airport_code>/average_flights

- **GET /airports/<airport_code>/busiest_hour**: identifica l'ora UTC con il maggior numero di voli per un aeroporto. Estrae l'ora da tutti i timestamp registrati e utilizza una struttura Counter per determinare l'ora più frequente.

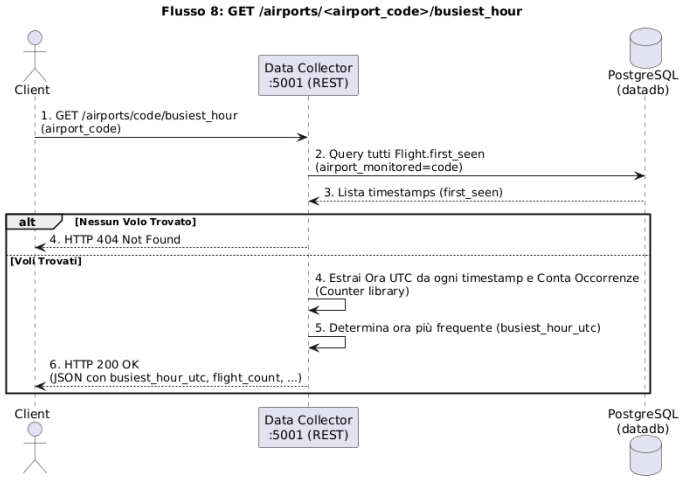

Figura 9: Diagramma di sequenza GET/airports/<airport_code>/busiest_hour

2.3.3 Raccolta Dati e Deduplicazione

La funzione `collect_flights()` in `services.py` implementa la logica di raccolta rispettando la seguente procedura:

1. Determina la lista degli aeroporti da monitorare: se non vengono specificati nel parametro "target_airports", interroga il database per recuperare tutti gli aeroporti con almeno un utente interessato.
2. Calcola la finestra temporale: ultimi 12 ore, escludendo le ultime 2 ore per garantire la disponibilità dei dati.
3. Per ogni aeroporto, esegue due richieste all'API: una per arrivals e una per departures.
4. Implementa la deduplicazione: prima di inserire un volo nel database, controlla che non esista già un volo con lo stesso `icao24`, `first_seen`, `airport_monitored` e `direction`. Questo previene l'accumulo di duplicati se la raccolta viene eseguita più volte.
5. Salva i dati nel database e genera statistiche sull'operazione (numero di aeroporti elaborati, voli aggiunti, errori).

2.3.4 Comunicazione gRPC

Il microservizio implementa il servizio `CollectorService` definito nel file `user_manager.proto`:

- **CleanupUser(email)**: richiesta ricevuta da user-manager quando un utente viene eliminato. Il metodo:

  1. Recupera tutti gli aeroporti di interesse dell'utente.
  2. Per ogni aeroporto, verifica se altri utenti lo hanno sottoscritto.
  3. Se nessun altro utente è interessato, elimina tutti i voli associati a quell'aeroporto.
  4. Elimina tutte le righe da `user_airports` relative all'utente.
  5. Ritorna un messaggio di successo con il conteggio degli elementi rimossi.

- **RemoveAirportInterest(email, airport_code)**: richiesta ricevuta da user-manager quando un utente rimuove un aeroporto dalle sue preferenze. Il metodo:

  1. Valida che l'interesse esista nel database.
  2. Elimina la riga corrispondente da `user_airports`.
  3. Verifica se altri utenti sono interessati allo stesso aeroporto.
  4. Se non ci sono altri utenti, elimina tutti i voli associati.
  5. Ritorna il numero di voli eliminati.

3 Scelte Tecnologiche e Implementazione

3.1 Canale gRPC bidirezionale

La scelta di implementare un canale gRPC bidirezionale tra i due microservizi è stata una decisione progettuale deliberata, giustificata dai seguenti aspetti:

Un'architettura puramente unidirezionale da data-collector verso user-manager si sarebbe limitata al data-collector che interroga user-manager per validare gli utenti (tramite `CheckUser()` e `GetUser()`). Tuttavia, questa soluzione non avrebbe consentito il flusso inverso di notifiche critiche.

Quando user-manager effettua operazioni di modifica (eliminazione di utenti o rimozione di preferenze aeroportuali), è necessario notificare data-collector affinché esegua la sincronizzazione: eliminare i voli associati e pulire la tabella `user_airports`.

3.2 Politiche di comunicazione

Il sistema implementa la semantica **at-most-once** per garantire che ogni operazione venga elaborata al massimo una volta, prevenendo gli effetti indesiderati di retry accidentali o duplicati di rete.

Il meccanismo si basa su un identificatore univoco `message_id` fornito dal client in ogni richiesta POST a `/addUser`. Il flusso è il seguente:

1. Il client genera un `message_id` (tipicamente un UUID) e lo include nel payload JSON.
2. user-manager interroga la tabella `message_ids` in userdb per verificare se il `message_id` è stato già elaborato.
3. Se il record esiste e non è scaduto (TTL = 5 minuti), il servizio restituisce immediatamente la risposta memorizzata (`response_data` e `response_status`) senza rielaborare l'operazione. Questo garantisce idempotenza: invocazioni ripetute della stessa richiesta producono il medesimo risultato.
4. Se il record non esiste o è scaduto, l'operazione viene elaborata, il risultato viene salvato nella cache con timestamp di scadenza, e la risposta viene ritornata al client.
5. Un job di background eseguito ogni 5 minuti ripulisce i record scaduti, evitando la crescita indefinita della tabella di cache.

Questa implementazione protegge da scenari comuni di fallimento della rete (timeouts, disconnessioni) dove il client, non ricevendo una risposta entro il tempo limite, reinvia la richiesta. La cache di idempotenza assicura che il secondo invio non produca effetti secondari (duplicate user creation, conflitti di vincoli).