UNIVERSITÀ DEGLI STUDI DI CATANIA

Corso di Laurea Magistrale in Ingegneria Informatica LM-32

Relazione Homework 2

Corso di Distributed Systems and Big Data

Anno Accademico 2025-2026

Studente:
Riccardo Maria Villaggio
Matricola: 1000084767

Indice

1 Introduzione

Il presente documento descrive l'evoluzione architetturale e funzionale del sistema di monitoraggio del traffico aereo sviluppato inizialmente per il primo homework del corso di Distributed Systems and Big Data. Mentre la prima iterazione si concentrava sui fondamenti dei microservizi e sulla comunicazione tramite gRPC, l'obiettivo di questo secondo progetto è stato quello di introdurre componenti e pattern avanzati per migliorare notevolmente la scalabilità, la sicurezza, la fault tolerance e le performance del sistema.
In particolare, sono state introdotte diverse modifiche, quali un *API Gateway* per la gestione centralizzata del traffico e l'isolamento dell'accesso ai microservizi, divenendo l'unico punto di accesso al sistema, un modulo *Circuit Breaker* per la tolleranza ai guasti esterni circa le chiamate a OpenSky Network e un broker (*Apache Kafka*) per l'introduzione di una nuova feature di alerting personalizzato via email basata su soglie definite dagli utenti sui dati dei voli raccolti circa i loro aeroporti di interesse.
Insieme a ciò, sono stati apportati miglioramenti significativi alla robustezza operativa (tra cui politiche di *Graceful Shutdown*) e alla topologia di rete, segmentata per garantire il massimo isolamento dei componenti, nonché ai componenti di business già esistenti per adattarsi alle nuove funzionalità e requisiti e alle prestazioni complessive del sistema.

1.1 Obiettivi dell'Evoluzione

L'estensione del sistema è stata guidata dai seguenti requisiti architetturali e funzionali:

- **API Gateway**: Abbandono dell'esposizione diretta dei microservizi. L'introduzione di **NGINX** come *Reverse Proxy* fornisce un unico punto di ingresso sicuro (HTTPS), gestisce traffico cifrato tramite SSL e nasconde la complessità della rete interna ai client.
- **Broker (Apache Kafka)**: Integrazione di **Apache Kafka** per implementare una pipeline di elaborazione reattiva del sistema. Il sistema non si limita più a salvare i dati passivamente, ma reagisce in tempo reale all'arrivo di nuovi voli, innescando processi di analisi e notifica in modo asincrono senza bloccare il flusso di raccolta dati.
- **Resilienza e Fault Tolerance**: Implementazione del pattern **Circuit Breaker** nel client verso OpenSky Network. Questo protegge il sistema da cascate di errori e latenze eccessive in caso di indisponibilità delle API esterne, garantendo una degradazione controllata delle funzionalità. Inoltre, è stata implementata una gestione avanzata dei segnali di sistema (`SIGTERM`/`SIGINT`) per garantire la chiusura ordinata (*Graceful Shutdown*) di tutti i servizi, prevenendo la perdita di dati in transito o la corruzione degli stati.
- **Alerting Personalizzato**: Nuova funzionalità di business che permette agli utenti di definire soglie personalizzate (`high_value`, `low_value`) sul numero di voli. Il superamento di queste soglie genera notifiche via email, gestite da moduli dedicati.

1.2 Aggiornamento dello Stack Tecnologico

Allo stack tecnologico originale (Python, Flask, gRPC, MySQL, Docker) sono stati integrati nuovi componenti per supportare le funzionalità avanzate:

- **NGINX**: Utilizzato come API Gateway e, in possibili estensioni future, come Load Balancer per gestire la scalabilità orizzontale.
- **Apache Kafka (KRaft mode)**: Piattaforma di messaging distribuita per la gestione asincrona degli eventi tra i microservizi.
- **kafka-python**: Libreria client per l'interazione Producer/Consumer con il broker Kafka.
- **smtplib & ssl**: Moduli Python standard per la gestione sicura dell'invio di email tramite protocollo SMTP.
- **Pattern Circuit Breaker**: Implementazione custom in Python per la gestione dei failure verso servizi terzi.

2 Nuova Architettura del Sistema

L'architettura del sistema è stata profondamente rivista rispetto alla versione precedente (HW1). Oltre all'introduzione di nuovi componenti funzionali, l'infrastruttura è stata riprogettata per supportare requisiti di sicurezza, comunicazione, fault tolerance, scalabilità e performance. Il sistema risultante è riportato in Figura 1, la quale illustra la nuova architettura del sistema, evidenziando i componenti principali, le reti di comunicazione e i flussi di dati tra i vari servizi.

2.1 Componenti Aggiuntivi

1. **NGINX (API Gateway)**: Agisce come *Reverse Proxy*, ma è stato già predisposto a estensioni future per agire anche come Load Balancer (si faccia riferimento alla configurazione in `nginx/nginx.conf`). Espone verso l'host solo le porte 80 (HTTP, con redirect verso 443) e 443 (HTTPS). I microservizi backend (*User Manager* e *Data Collector*) non sono più raggiungibili direttamente dall'esterno, riducendo drasticamente l'esposizione del sistema verso potenziali attacchi.

2. **Apache Kafka (KRaft Mode)**: È stato configurato in modalità *KRaft* (senza Zookeeper), fungendo da backbone per il disaccoppiamento tra la raccolta dati e la logica di notifica.

3. **Alert System**: Un nuovo microservizio *stateless* (Consumer/Producer) che elabora lo stream di dati proveniente dal topic `to-alert-system`. Verifica se i dati raccolti superano le soglie (`high_value`, `low_value`) definite dagli utenti e, in caso positivo, produce un evento di notifica sul topic `to-notifier`.

4. **Alert Notifier System**: Un worker dedicato (Consumer) che ascolta il topic `to-notifier`. È responsabile dell'invio effettivo delle email tramite server SMTP (Gmail). Questo disaccoppia la logica di business dai tempi di latenza dell'invio email.

2.2 Topologia di Rete e Sicurezza

Nella nuova topologia è stata adottata una segmentazione di rete a tre livelli, implementata tramite bridge network di Docker isolate:

- **Persistence Layer (`user-net`, `data-net`)**: Due reti private dedicate esclusivamente alla comunicazione tra i microservizi e i rispettivi database. Questo impedisce che componenti esterni (come il Gateway o i consumer Kafka) possano accedere direttamente ai dati persistenti.

- **Messaging Layer (`kafka-net`)**: Una rete dedicata al traffico di messaggistica tra il Data Collector (modulo interno *Scheduler*), il broker Kafka e i servizi di alerting. L'isolamento di questa rete evita che il traffico massivo dei log di Kafka interferisca con le altre comunicazioni.

- **Service Layer (`shared-net`)**: La rete condivisa che permette all'API Gateway di instradare le richieste HTTP verso i microservizi e mantiene la comunicazione sincrona gRPC interna (es. verifica esistenza utente tra Data Collector e User Manager).

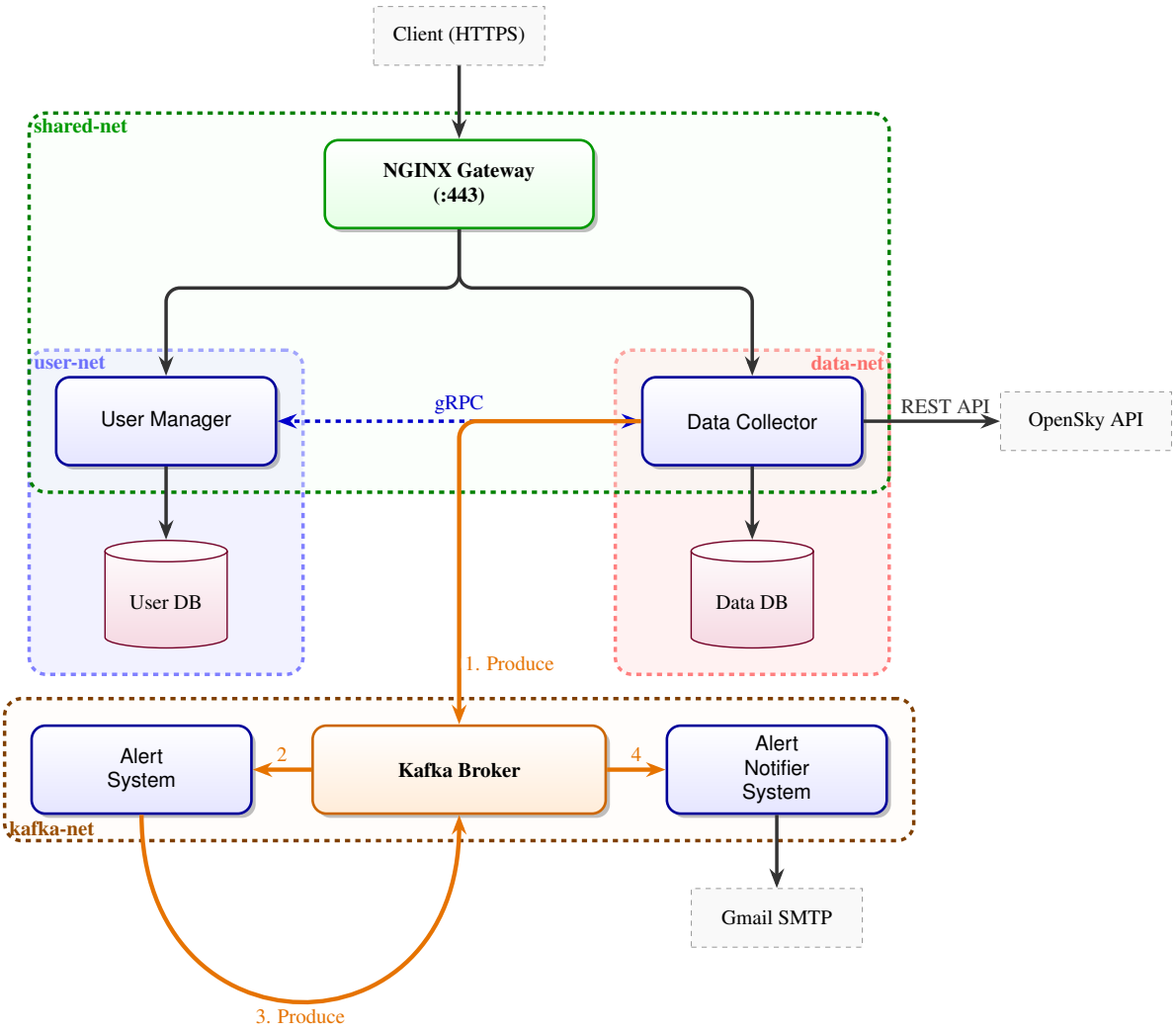

Figura 1: Architettura del sistema e flussi dati

2.3 Dettagli Implementativi e Motivazioni Architetturali

Per massimizzare la sicurezza, l'efficienza e la continuità operativa, sono state adottate specifiche configurazioni a livello di container (visibili nel `docker-compose.yml`):

- **Politica di Riavvio Automatica**: In continuità con la progettazione della prima iterazione, anche adesso tutti i servizi (microservizi, database e broker) sono stati configurati con la direttiva `restart: unless-stopped`. Questa configurazione è cruciale, specialmente in un'architettura a eventi: se un consumatore (es. *Alert System*) dovesse terminare inaspettatamente o se il Broker Kafka dovesse riavviarsi, Docker provvederà a ripristinare automaticamente i container. In combinazione con i volumi persistenti, questo garantisce che il sistema converga autonomamente verso uno stato consistente senza necessità di intervento manuale.

- **Esposizione Controllata delle Porte (`expose` vs `ports`)**: A differenza del primo homework, i microservizi *User Manager* e *Data Collector* non utilizzano più la direttiva `ports` per mappare le porte sull'host. È stata utilizzata, invece, la direttiva `expose`, che rende le porte (5000, 5001, 50051, 50052) accessibili esclusivamente all'interno delle reti Docker virtuali. L'unico punto di contatto con il mondo esterno rimane la porta 443 di Nginx, forzando così tutto il traffico a passare attraverso i controlli di sicurezza del Gateway. Nei database, le porte sono ancora accessibili dall'host per facilitare operazioni di manutenzione e debugging, in futuro anche queste potranno essere rimosse per aumentare ulteriormente la sicurezza.

- **Configurazione Kafka KRaft (Zookeeper-less)**: Il broker Kafka è stato configurato per operare in modalità *KRaft*, eliminando la dipendenza dal servizio Zookeeper. Questa scelta architetturale moderna semplifica il deployment (un container in meno da gestire) e riduce il consumo di risorse. Il broker agisce sia come controller che come data node (ruolo `broker,controller`), ideale per un ambiente di sviluppo distribuito leggero ma funzionalmente completo.

- **Strategia di Load Balancing (Predisposizione alla Scalabilità)**: Sebbene l'attuale deployment utilizzi una singola replica per servizio, la configurazione di Nginx è stata progettata per supportare nativamente la scalabilità orizzontale. Nel blocco `upstream`, è stato mantenuto l'algoritmo di default *Round Robin*, sufficiente per l'attuale topologia. Tuttavia, come documentato nella configurazione, la strategia per un futuro ambiente multi-replica prevede l'adozione di `least_conn`. Poiché il sistema è progettato per essere *stateless* (con lo stato delegato interamente ai database), non è necessario utilizzare meccanismi di *session stickiness* (come `ip_hash`); l'algoritmo `least_conn` risulterebbe quindi ottimale per bilanciare il carico verso l'istanza con meno connessioni attive, specialmente in presenza di richieste a latenza variabile.

3 Modifiche, Ottimizzazioni e Problematiche Risolte

In questa sezione si descrivono le principali modifiche implementate nei microservizi esistenti, insieme alle motivazioni e ottimizzazioni apportate per migliorare le performance e la stabilità del sistema, nonché le problematiche incontrate durante lo sviluppo e le relative soluzioni adottate. In particolare, durante la fase di testing sotto carico (*Stress Test*) e di avvio concorrente dei container (*Cold Start*), sono emerse criticità legate alla gestione della concorrenza sul database e alla stabilità del gruppo di consumer Kafka.

3.1 Configurazione Kafka

Poiché l'architettura prevede un cluster Kafka composto da un singolo Broker, è stato necessario sovrascrivere le configurazioni di default (tipicamente pensate per cluster multi-nodo) per garantire la corretta operatività. In particolare, sono stati impostati i seguenti parametri per evitare che il broker fallisse nella creazione dei topic interni o rifiutasse le transazioni per mancanza di repliche:

- `KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1`
- `KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1`
- `KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1`

Inoltre, è stato adottato un topic a partizione singola. Tale scelta è giustificata dalla natura degli eventi trattati nelle specifiche del progetto: le notifiche relative ai voli sono eventi atomicamente indipendenti e privi di correlazione causale tra utenti diversi. Di conseguenza, non è richiesto un ordinamento globale rigoroso tra i messaggi, rendendo superfluo in questa fase l'uso di chiavi di partizionamento complesse o la gestione di partizioni multiple per il bilanciamento del carico.
In particolare, l'utilizzo del multithreading nel *Data Collector* (Producer) non è correlato strettamente alla topologia del broker, ma risponde alla necessità di parallelizzare le chiamate I/O-bound verso l'API REST di OpenSky, al fine di massimizzare il throughput di raccolta dati.
Dunque, l'adozione di un topic a più partizioni non apporterebbe benefici significativi nel contesto attuale, ma rappresenterebbe il naturale punto di evoluzione qualora il sistema dovesse richiedere in futuro una scalabilità orizzontale dei consumatori, dovendo, a conti fatti, assegnare a ciascun consumer del gruppo una partizione dedicata per garantire l'elaborazione parallela e bilanciata del carico.

3.2 Gestione della Concorrenza nel Data Collector

Il servizio *Data Collector* continua ad utilizzare un approccio multithread (`ThreadPoolExecutor` con 5 worker) per parallelizzare il recupero dati da OpenSky. L'uso iniziale dei parametri di default ha causato errori di *InterfaceError* dovuti alla condivisione non sicura delle sessioni SQLAlchemy.
Per ovviare a questo problema, sono state implementate le seguenti strategie:

1. **Disaccoppiamento (DTO Pattern)**: È stata implementata una serializzazione immediata (`.to_dict()`) nel thread principale. Ai worker vengono passati dizionari Python puri (copie in memoria) invece di oggetti ORM "vivi", eliminando i conflitti di sessione.

2. **Impostazioni custom del connection pool**: Il pool di connessioni è stato configurato esplicitamente per gestire il carico concorrente:

```
app.config['SQLALCHEMY_ENGINE_OPTIONS'] = {
    'pool_size': 20,       # Numero di connessioni mantenute aperte nel pool
    'max_overflow': 10,    # Connessioni extra creabili durante i picchi di carico
    'pool_recycle': 1800, # Riciclo connessioni ogni 30 minuti per evitare timeout
    'pool_pre_ping': True,# Verifica salute connessione prima dell'uso
    'pool_timeout': 30     # Timeout attesa connessione libera
}

```

Listing 1: Configurazione SQLAlchemy Engine

3. **Gestione Deadlock MySQL**: Poiché l'operazione di Bulk Upsert su chiavi contigue può generare Gap Locks in MySQL, viene catturato l'errore 1213 (DEADLOCK) e applicata una strategia di Retry con Backoff Casuale. Questo desincronizza i thread concorrenti, permettendo al database di serializzare le transazioni in conflitto senza perdere dati e senza bloccare l'intero sistema.

```
if "deadlock" in error_str or "1213" in error_str:
    if attempt < max_retries - 1:
        (backoff)
        sleep_time = random.uniform(0.5, 2.0)
        time.sleep(sleep_time)
        continue

```

Listing 2: Gestione Deadlock MySQL

Inoltre, l'accesso concorrente al Kafka Producer è gestito nativamente dalla thread-safety della libreria client, che serializza i messaggi in un buffer interno (accumulator) prima dell'invio. Questa architettura garantisce un elevato throughput senza introdurre race conditions, delegando la complessità della sincronizzazione di rete al driver Kafka e mantenendo il codice applicativo snello e scalabile.

3.3 Ulteriori migliorie di UX, robustezza e performance nello Scheduler

Sono state implementate diverse ottimizzazioni specifiche per il modulo Scheduler del Data Collector:

- **Locking e Trigger Immediato**: Per migliorare la UX, quando un utente aggiunge un interesse per un aeroporto non monitorato, il sistema attiva immediatamente una raccolta dati (bypassando l'attesa del ciclo periodico).

```
     # Check preliminare degli aeroporti monitorati
    is_new_airport = db.session.execute(
        db.select(func.count()).select_from(UserInterest).filter_by(
    airport_icao=airport_icao)
    ).scalar() == 0

    # ... salvataggio interesse ...

    if is_new_airport:
        # Creiamo un thread separato per raccogliere i dati
    immediatamente sul nuovo aeroporto
        threading.Thread(target=scheduler.collect_single_airport, args
    =(airport_icao,)).start()

```

Listing 3: Trigger Raccolta Immediata

Per evitare race condition tra questo trigger manuale e lo scheduler periodico (fenomeno raro, ma possibile), viene utilizzato un set di lock in memoria per evitare che lo stesso aeroporto venga processato contemporaneamente (`self.processing_airports`).

```
def _acquire_lock(self, icao):
    with self.processing_lock:
        if icao in self.processing_airports:
            return False #Lock gia' acquisito
        self.processing_airports.add(icao)
        return True

def collect_single_airport(self, target_icao):
    # Tenta di acquisire il lock prima di procedere
    if not self._acquire_lock(target_icao):
        print(f"SKIP {target_icao}: Gia' in fase di aggiornamento.")
        return

    try:
        # ... logica di raccolta dati ...
    finally:
        # Rilascio garantito del lock alla fine
        self._release_lock(target_icao)

```

Listing 4: Gestione Lock per Aeroporto

- **Controllo Producer**: Prima di ogni invio, viene verificata l'esistenza dell'istanza del producer per evitare crash in caso di disconnessione. Inoltre, è stato implementato un metodo robusto per la connessione a Kafka, che gestisce i tentativi di riconnessione in modo thread-safe utilizzando un lock dedicato (`self.kafka_lock`).:

```python
def _connect_kafka(self):
    try:
        kafka_bootstrap_servers = os.getenv('KAFKA_BOOTSTRAP_SERVERS', 'kafka:9092')
        self.kafka_producer = KafkaProducer(
            bootstrap_servers=kafka_bootstrap_servers,
            value_serializer=lambda v: json.dumps(v).encode('utf-8'),
            acks='all',
            retries=5,
            linger_ms=50,
            batch_size=32768,
            retry_backoff_ms=1000
        )
        print(f"Kafka Producer connesso a {kafka_bootstrap_servers}", flush=True)
    except Exception as e:
        print(f"Errore connessione Kafka: {e}", flush=True)
        self.kafka_producer = None

# Esempio di controllo prima dell'utilizzo
if not self.kafka_producer:
    with self.kafka_lock:
        if not self.kafka_producer:
            self._connect_kafka()

```

Listing 5: Connessione Kafka e Controllo Producer

3.4 Configurazione dei Consumer Kafka (Alert System e Notifier)

Il sistema gestisce due consumer con profili opposti: l'*Alert System* (CPU-bound e veloce) e l'*Alert Notifier* (I/O-bound, lento a causa delle chiamate SMTP). Durante l'invio massivo di notifiche, la latenza di rete SMTP causava il superamento del timeout di elaborazione, portando a continui ed indesiderati *Consumer Rebalancing*.
La configurazione adottata per mitigare il problema prevede:

- **Batch Size Ridotto**: Impostando `max_poll_records=10` (default 500), il tempo di elaborazione per batch è sceso drasticamente, garantendo che il ciclo di `poll()` termini ben prima del timeout dovuto al parametro `max.poll.interval.ms` (default 300.000ms).
- **Session Timeout Aumentato**: Il parametro `session_timeout_ms` è stato portato a 30.000ms (30s) per tollerare i picchi di latenza infrastrutturale.
- **Heartbeat Interval**: Impostato a 10.000ms (1/3 del session timeout, come raccomandato dalla documentazione ufficiale) per mantenere viva la sessione anche durante elaborazioni intense.

- **Auto Commit Disabilitato**: `enable_auto_commit=False` per garantire il controllo transazionale completo (il commit dell'offset avviene solo dopo l'elaborazione o l'invio email con successo).

Nota sul Cold Start: È stato osservato che il meccanismo di *heartbeat* della libreria `kafka-python` è attivo solo quando il consumatore è nello stato `STABLE`. Durante la fase di inizializzazione, eventuali latenze non innescano l'espulsione, poiché il client non è ancora entrato nel gruppo.

3.5 Ottimizzazione dei Producer Kafka

Il sistema utilizza due configurazioni distinte per i Producer, ottimizzate per i diversi casi d'uso: *Massimo Throughput* (Scheduler) e *Bassa Latenza* (Alert System).

3.5.1 Scheduler (Data Collector)

Lo Scheduler produce raffiche di dati, generate dall'elevato numero di voli scaricati periodicamente. Per massimizzare il throughput e ridurre l'overhead di rete, la configurazione privilegia il batching:

- `linger_ms=50`: Il producer attende fino a 50ms per accumulare più messaggi possibili prima di inviarli.
- `batch_size=32768`: Dimensione del buffer aumentata a 32KB per ridurre il numero di richieste verso il broker.
- `acks='all'`: Garantisce la massima durabilità attendendo la conferma di scrittura (compatibilmente con il singolo nodo attivo).
- `retries=5`: Numero di tentativi in caso di fallimenti temporanei di rete.

3.5.2 Alert System

L'*Alert System* agisce come producer quando deve inoltrare le notifiche elaborate al componente *Notifier*. In questo scenario, la priorità è la reattività (l'utente deve ricevere l'alert tempestivamente), non l'efficienza del batch.

- `linger_ms=5`: Il tempo di attesa è ridotto al minimo (5ms) per inviare il messaggio quasi istantaneamente, mantenendo un minimo di aggregazione solo in caso di picchi simultanei.
- `batch_size=16384`: Dimensione standard (16KB), sufficiente per i payload JSON delle notifiche.
- `acks='all'` e `retries=5`: Mantiene la garanzia di consegna, critica per non perdere le notifiche agli utenti.

3.6 Sicurezza e Logging (Nginx e User Manager)

L'introduzione di Nginx come reverse proxy ha permesso di centralizzare la gestione della sicurezza, permettendo l'utilizzo di SSL per ottenere maggiore protezione delle richieste e il reindirizzamento forzato del traffico HTTP verso HTTPS.

```
server {
    listen 80;
    server_name localhost;
    # Redirect forzato di tutto il traffico HTTP su HTTPS
    return 301 https://$host$request_uri;
}

server {
    listen 443 ssl;
    server_name localhost;

    # Configurazione Certificati SSL (Self-Signed per sviluppo)
    ssl_certificate /etc/nginx/ssl/nginx-selfsigned.crt;
    ssl_certificate_key /etc/nginx/ssl/nginx-selfsigned.key;
    ssl_protocols TLSv1.2 TLSv1.3;

    # Esempio di routing verso i microservizi
    location /users {
        proxy_pass http://user_manager_backend;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }

    location /interests {
        proxy_pass http://data_collector_backend;
        # ... headers ...
    }
}
```

Listing 6: Configurazione SSL e Redirect in Nginx

Poiché tutte le richieste arrivano al backend dall'IP del container Nginx, è stato configurato l'header `X-Real-IP` per preservare l'indirizzo originale del client. Nel microservizio *User Manager*, l'IP viene recuperato prioritariamente da questo header:

```
def get_client_id():
    # Proviamo a prendere il vero IP dal header X-Real-IP
    ip = request.headers.get('X-Real-IP')

    if not ip:
        # Fallback
        ip = request.remote_addr or "unknown"
    return ip
```

Listing 7: Recupero IP Client

3.7 Gestione Segnali e Graceful Shutdown

Tutti i componenti del sistema (Scheduler, Consumer, Producer) implementano una gestione corretta dei segnali di terminazione (`SIGINT`, `SIGTERM`) per garantire la chiusura ordinata delle risorse (connessioni DB, socket Kafka) e prevenire la perdita di dati.

```
def handle_sigterm(*args):
    raise KeyboardInterrupt

signal.signal(signal.SIGINT, handle_sigterm)
signal.signal(signal.SIGTERM, handle_sigterm)
# ...
finally:
    if producer:
        producer.close() # Flush pending messages
```

Listing 8: Gestione Segnali

3.8 Circuit Breaker e Strategia di Fault Tolerance

Per garantire la fault tolerance del sistema a fronte di malfunzionamenti delle API esterne (OpenSky Network), è stato implementato il pattern **Circuit Breaker**. Questo componente agisce da intermediario in tutte le chiamate di rete, proteggendo il sistema da fallimenti a cascata.
La logica del Circuit Breaker implementata prevede i tre stati canonici:

- **CLOSED**: Stato normale, le richieste vengono eseguite. Se una richiesta ha successo, il contatore dei fallimenti viene resettato immediatamente (strategia *Consecutive Failures*), per evitare che errori sporadici accumulati nel tempo portino all'apertura accidentale del circuito.
- **OPEN**: Al raggiungimento di una soglia di fallimenti consecutivi (configurata a 3), il circuito si apre. Le richieste successive vengono bloccate immediatamente (*Fail-Fast*) sollevando un'eccezione, senza tentare la connessione di rete, per un periodo di recovery (60 secondi). La scelta di usare 3 come numero di soglia è legata al fatto che: per un singolo aeroporto, il Data Collector effettua due chiamate (una per le partenze e una per gli arrivi); supponendo che ci sia un problema solo per un aeroporto, scegliere 3 permette di tollerare un fallimento completo di un aeroporto senza aprire il circuito. Viceversa, significa che effettivamente ci sono problemi generalizzati con l'API esterna.
- **HALF-OPEN**: Trascorso il timeout, il sistema consente il passaggio di una singola richiesta di prova. Se questa ha successo, il circuito si chiude e il servizio viene considerato ripristinato; in caso contrario, torna nello stato OPEN.

3.8.1 Gestione della Concorrenza e Locking

Un aspetto implementativo critico, che differenzia questa soluzione da implementazioni più semplici, riguarda la gestione del **Locking**. È stato scelto deliberatamente di utilizzare un lock *fine-grained*, acquisendolo solo per la lettura e la scrittura dello stato interno, ma rilasciandolo durante l'esecuzione effettiva della funzione remota (`func`).

Mantenere il lock durante l'intera durata della chiamata di rete (che può durare diversi secondi in caso di latenza o timeout) avrebbe causato un fenomeno di **Head-of-Line Blocking**, serializzando di fatto le richieste di tutti i thread concorrenti e riducendo drasticamente il *throughput* del sistema. Rilasciando il lock, i thread non si bloccano a vicenda durante l'attesa di I/O, massimizzando il parallelismo.

```
class CircuitBreaker:
    def call(self, func, *args, **kwargs):
        # 1. Controllo stato (SOTTO LOCK)
        with self.lock:
            if self.state == 'OPEN':
                if time.time() - self.last_failure_time > self.
    recovery_timeout:
                    self.state = 'HALF_OPEN'
                else:
                    raise CircuitBreakerOpenException("CircuitBreaker is
    OPEN")

        try:
            # 2. Esecuzione Chiamata (SENZA LOCK per evitare blocking I/O)
            result = func(*args, **kwargs)

            # 3. Aggiornamento Successo (SOTTO LOCK)
            with self.lock:
                if self.state == 'HALF_OPEN' or self.state == 'CLOSED':
                    self.state = 'CLOSED'
                    self.failures = 0
            return result

        except Exception as e:
            # 4. Gestione Fallimento (SOTTO LOCK)
            with self.lock:
                self.failures += 1
                self.last_failure_time = time.time()
                if self.failures >= self.failure_threshold:
                    self.state = 'OPEN'
            raise e
```

Listing 9: Implementazione Circuit Breaker con Lock Fine-Grained

3.8.2 Integrazione con il Client HTTP: Il Wrapper Selettivo

Il Circuit Breaker è stato progettato per essere agnostico rispetto al protocollo, intercettando generiche eccezioni (`Exception`). Tuttavia, la libreria `requests` non solleva eccezioni per codici di errore HTTP come 500 o 429.
Per colmare questo divario, è stato implementato un metodo wrapper (`_make_http_call`) all'interno del client `OpenSkyClient`. Questo metodo è responsabile di trasformare i codici di stato "infrastrutturali" in eccezioni, permettendo al Circuit Breaker di reagire correttamente, discriminando però tra errori fatali ed errori applicativi:

- **Errori Infrastrutturali (5xx, Timeout) e Rate Limit (429):** Vengono trasformati in eccezioni. Questi indicano un problema del server o un sovraccarico, quindi il Circuit Breaker deve intervenire e aprirsi.
- **Errori Logici (404 Not Found, 401 Unauthorized):** Non vengono trasformati in eccezioni. Il server funziona correttamente ma la richiesta non ha prodotto dati o richiede autenticazione. In questi casi il Circuit Breaker rimane **CLOSED**, delegando la gestione (es. refresh del token o ritorno di lista vuota) alla logica di business del chiamante.

```python
class OpenSkyClient:
    def __init__(self):
        self.cb = CircuitBreaker(failure_threshold=3, recovery_timeout=60)

    def _make_http_call(self, method, url, **kwargs):
        """
        Esegue la richiesta e solleva eccezioni solo per errori critici,
        permettendo al Circuit Breaker di intercettarli.
        """
        response = method(url, **kwargs)
        if 500 <= response.status_code < 600:
            raise Exception(f"Server Error: {response.status_code}")
        if response.status_code == 429:
            raise Exception("Rate Limit Exceeded")
        return response

    def get_departures(self, airport_icao, ...):
        try:
            # Il CB avvolge il wrapper, non direttamente la requests.get
            response = self.cb.call(self._make_http_call, requests.get, url
    , ...)

            # Gestione logica applicativa (il CB qui e' rimasto CHIUSO)
            if response.status_code == 200:
                return response.json()
            elif response.status_code == 404:
                return [] # Aeroporto valido ma senza voli
            elif response.status_code == 401:
                self.token = None # Trigger rinnovo token al prossimo giro
                return self.get_departures(...)

        except CircuitBreakerOpenException:
            print(f"CircuitBreaker OPEN: Saltata richiesta per {
    airport_icao}")
            return []
        except Exception:
            return [] # Fallback su errore critico
```

Listing 10: Wrapper HTTP e utilizzo nel Client

3.9 Modifiche al Modello Dati e Diagrammi di Sequenza

Per evitare ridondanza, in questo documento vengono riportati solo i dettagli della struttura dati `UserInterest`, unica ad essere stata modificata rispetto alla versione precedente del sistema. Per la definizione completa delle altre tabelle (es. `User`, `FlightData`), si rimanda alla documentazione tecnica del primo homework. Allo stesso modo, gli endpoint (che non hanno subito modifiche) non sono stati riportati, mentre i diagrammi di sequenza sono mostrati solo per le operazioni che hanno subito modifiche rilevanti. Si osservi che adesso tutte le operazioni avvengono tramite l'API Gateway (NGINX), che instrada le richieste ai microservizi appropriati in maniera trasparente al Client, ma per semplicità grafica si omette questo dettaglio.
Il modello dati degli interessi (`UserInterest`) è stato esteso per includere due nuovi campi opzionali e vincoli di integrità a livello di database, come mostrato nella Tabella 1.

Tabella 1: Struttura aggiornata della tabella `user_interests`

Colonna	Tipo	Vincoli	Descrizione
id	INTEGER	PK, AUTO_INCREMENT	Identificativo univoco
user_email	VARCHAR(255)	NOT NULL	Email utente
airport_icao	VARCHAR(10)	NOT NULL	Codice ICAO aeroporto
high_value	INTEGER	NULLABLE	Soglia superiore per alert
low_value	INTEGER	NULLABLE	Soglia inferiore per alert
created_at	DATETIME	DEFAULT NOW()	Timestamp di creazione
UNIQUE(user_email, airport_icao) - Constraint per unicità			
CHECK(high_value IS NULL) OR (low_value IS NULL) OR (high_value > low_value) - Constraint di validità soglie			

3.9.1 Diagramma di Sequenza: Aggiunta Interesse

La Figura 2 illustra il flusso di aggiunta di un nuovo interesse. Rispetto alla versione precedente, il payload della richiesta POST include ora i campi opzionali `high_value` e `low_value`. Per semplicità grafica, come anticipato, il componente infrastrutturale NGINX è stato omesso dal diagramma di sequenza, in quanto agisce come proxy trasparente tra il Client e i Microservizi.

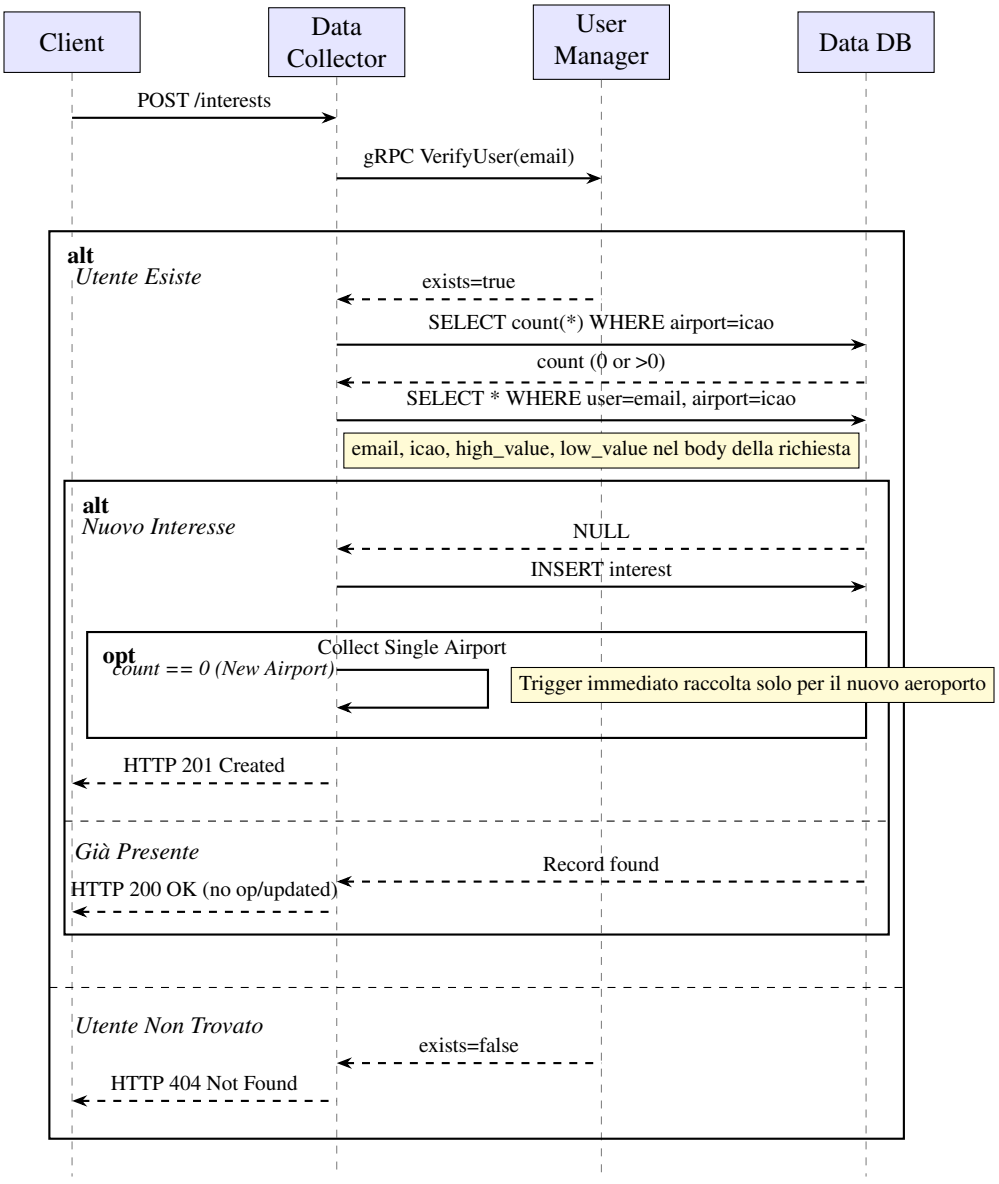

Figura 2: Diagramma di sequenza: Aggiunta Interesse

3.9.2 Diagramma di Sequenza: Messaging Kafka

Il flusso di notifica implementato tramite Kafka è illustrato nella Figura 3. Il processo si articola nei seguenti passaggi:

1. Il **Data Collector**, alla fine di ogni ciclo di raccolta, dopo aver salvato i dati nel database, pubblica un messaggio sul topic *to-alert-system*, contenente le informazioni sui voli recuperati.
2. L'**Alert System** consuma il messaggio, recupera le soglie dell'utente e valuta se generare un alert controllando se, per ogni profilo, il valore del numero di voli per un dato aeroporto supera le soglie definite. Per inviare un alert, pubblica un messaggio sul topic *to-notifier*.
3. L'**Alert Notifier System** invia l'email all'utente con i parametri che sono stati forniti nel messaggio consumato.

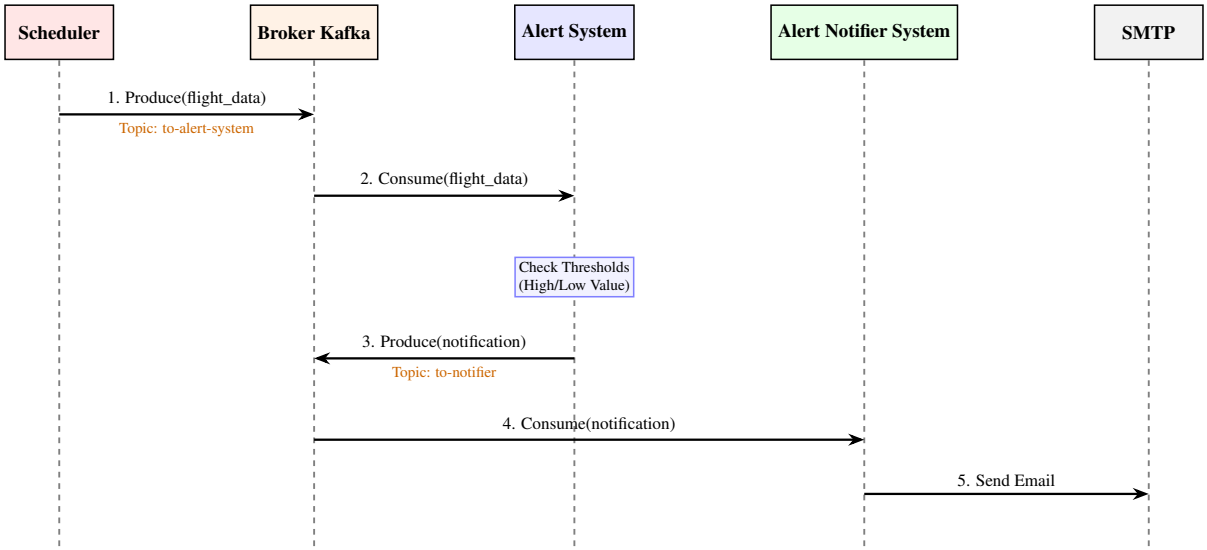

Figura 3: Diagramma di sequenza dettagliato del flusso di notifica Kafka

3.10 Configurazione SMTP

Il sistema supporta l'invio reale di email tramite configurazione SMTP. Le credenziali e i parametri del server sono iniettati come variabili d'ambiente nel container `alert-notifier-system`, permettendo di configurare qualsiasi provider email (es. Gmail) senza modificare il codice sorgente.

```yaml
    # Alert Notifier System
    alert-notifier-system:
        # ...
        environment:
            KAFKA_BOOTSTRAP_SERVERS: kafka:9092
            SMTP_SERVER: smtp.gmail.com
            SMTP_PORT: 465
            SMTP_USER: ${SMTP_USER}
            SMTP_PASSWORD: ${SMTP_PASSWORD}
            SENDER_EMAIL: ${SENDER_EMAIL}
        # ...
```

Listing 11: Configurazione SMTP in docker-compose.yml

Il microservizio *Alert Notifier* utilizza la libreria standard `smtplib` con supporto SSL per stabilire una connessione sicura e inviare le email. Di seguito è riportato l'estratto del codice che gestisce l'invio:

```python
import smtplib
import ssl
from email.message import EmailMessage

# Recupero configurazione da variabili d'ambiente
SMTP_SERVER = os.getenv('SMTP_SERVER', 'smtp.gmail.com')
SMTP_PORT = int(os.getenv('SMTP_PORT', 465))
SMTP_USER = os.getenv('SMTP_USER')
SMTP_PASSWORD = os.getenv('SMTP_PASSWORD')

def send_email(recipient_email, subject, body):
    context = ssl.create_default_context()

    msg = EmailMessage()
    msg.set_content(body)
    msg['Subject'] = subject
    msg['From'] = SMTP_USER
    msg['To'] = recipient_email

    try:
        with smtplib.SMTP_SSL(SMTP_SERVER, SMTP_PORT, context=context) as
    server:
            server.login(SMTP_USER, SMTP_PASSWORD)
            server.send_message(msg)
        print(f"EMAIL INVIATA con successo a: {recipient_email}")
        return True
    except Exception as e:
        print(f"ERRORE INVIO EMAIL: {e}")
        raise e
```

Listing 12: Invio Email con smtplib e SSL

4 Conclusioni e Sviluppi Futuri

L'evoluzione del sistema, grazie a tutte le nuove soluzioni introdotte, ha permesso di incrementare significativamente la robustezza e la scalabilità complessiva. In particolare, il fine tuning dei parametri di rete e di messaggistica (batching, timeouts, gestione sessioni) ha reso il sistema stabile anche in condizioni di *Stress Test* e *Cold Start*, eliminando le criticità tipiche della gestione concorrente delle risorse.

4.1 Possibili Sviluppi Futuri

Sebbene l'architettura attuale soddisfi pienamente i requisiti funzionali e non funzionali, ulteriori evoluzioni potrebbero includere:

- **Orchestrazione con Kubernetes**: Migrazione da Docker Compose a un cluster Kubernetes (K8s) per gestire nativamente l'autoscaling dei pod (es. scalare i consumer *Alert Notifier*).
- **Osservabilità Distribuita**: Integrazione di uno stack di monitoraggio (es. Prometheus e Grafana) e tracciamento distribuito (es. Jaeger o OpenTelemetry) per visualizzare in tempo reale il flusso delle richieste attraverso i microservizi e la latenza dei messaggi Kafka.