Sistema di acquisizione telemetria da veicoli da corsa

Nicolas Farabegoli
nicolas.farabegoli@studio.unibo.it

Luglio 2021

Uno dei maggiori problemi che riguarda la telemetria dei veicoli da corsa è che quest'ultima può essere osservata solo a gara conclusa. Risulterebbe utile poter osservare e analizzare la telemetria in tempo reale in modo che possano essere considerate strategie oppure in caso di problemi possano essere prese contromisure di sicurezza. Il sistema ha l'obiettivo di collezionare i dati provenienti dalla telemetria dei veicoli e renderli disponibili in tempo reale ai tecnici del team. Il sistema prevede poi un sistema per la comunicazione diretta con il pilota mediante messaggi che vengono mostrati sul dashboard del veicolo. Il sistema deve fornire un punto di accesso per la federazione in modo che possa verificare che la competizione si svolga senza irregolarità e nel pieno rispetto delle regole di sicurezza. Vista la riservatezza dei dati che il sistema elabora, sono adottate forme di autenticazione e autorizzazione affinché utenti non autorizzati non siano in grado di accedervi.

Indice

1 Obiettivi/Requisiti

Il sistema ambisce a migliorare la gestione della telemetria (in tempo reale) dei veicoli da corsa, oltre a individuare e segnalare condizioni di pericolo dei veicoli e la comunicazione con il pilota.

1.1 Obiettivi

Il progetto ha quattro obiettivi principali:

- Acquisire la telemetria in tempo reale dei veicoli da corsa
- Memorizzare i dati della telemetria in modo da poterli consultare in futuro
- Possibilità di configurare trigger di allarmi che avvisino l'utente qualora si verifichino condizioni critiche del veicolo
- Garantire accesso alla federazione affinché quest'ultima possa verificare il regolare svolgimento del gran premio.

1.2 Terminologia

Di seguito sono riportate le terminologie di dominio che verranno utilizzate durante la relazione:

- **Amministratore di sistema**: rappresenta la figura preposta per amministrare la parte informatica del sistema, quindi la creazione e gestione degli utenti.
- **Tecnici di pista**: rappresenta la figura preposta a interagire attivamente con il sistema. Questa figura è la parte più attiva nel sistema.
- **Team**: rappresenta il proprietario del sistema. Il team è formato da più *tecnici di pista* e uno o più *amministratori di sistema*
- **Federazione**: è una figura esterna all'organizzazione del team ma deve poter interagire con una parte del sistema in modo indipendente
- **Telemetria**: con questo termine si intendono tutte le informazioni che il veicolo è in grado di produrre e inviare al sistema. Un esempio di telemetria sono i valori di temperatura motore, velocità, giri motore, ecc.

1.3 Casi d'uso

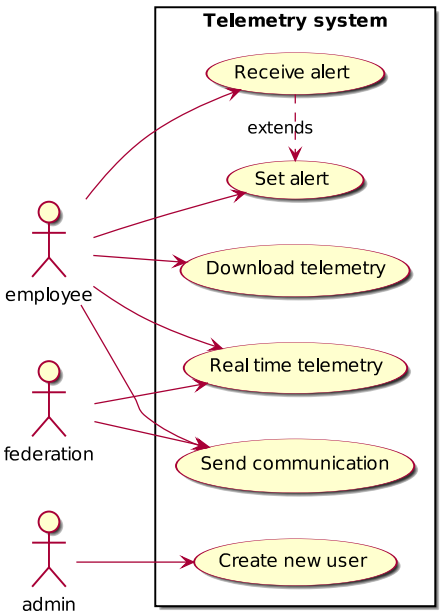

Figura 1: Casi d'uso del sistema.

In figura 1 viene mostrato uno use case diagram per indicare i vari casi d'uso del sistema. Nello specifico vengono mostrate le principali operazioni come l'osservazione della telemetria, impostare allarmi, inviare comunicazioni ecc.

1.4 Scenarios

I soggetti del sistema sono: *team* (proprietario del sistema), *federazione*, *amministratore del sistema* e *tecnici di pista*.

I possibili scenari sono:

- La federazione **FIA** vuole osservare i dati della telemetria dei veicoli in gara. La federazione impone che se vengono utilizzati sistemi che interagiscono col il veicolo, tali sistemi devono poter essere aperti alla federazione per poter controllare il corretto svolgimento della competizione. La federazione richiede all'amministratore di sistema le credenziali per potervi accedere. Una volta acceduti al sistema la federazione, mediante API, verifica che i dati di telemetria prodotti dal veicolo durante la corsa siano regolari e non vi siano anomalie.

- Si vuole aggiungere una nuova *federazione* al sistema per poter partecipare a una nuova competizione. *L'amministratore del sistema*, mediante apposita API, crea il nuovo utente assegnandogli i permessi specifici per identificare l'utente come "federazione".

- Nel team viene assunto un nuovo *tecnico di pista* e gli si vuole fornire accesso al sistema per poter operare durante il week-end di gara. *L'amministratore di sistema* provvede alla creazione del nuovo utente assegnandogli i permessi per identificare l'utente come "tecnico di pista"
- Il *tecnico di pista* Franco Garavalli vuole osservare la telemetria del veicolo del proprio team con nome *Alpha Leonis*. Franco mediante le apposite API si mette in ascolto della telemetria specificando l'id del veicolo.
- Il *tecnico di pista* Mario Chiaravalli vuole essere avvisato se i giri del motore superano i 12000 rpm. Mario, mediante le apposite API, è in grado di registrare una nuova regola che consenta di inviare un alert qualora si verifichi la condizione impostata nella regola stessa appena inserita.

2 Analisi dei requisiti

2.1 Requisiti utente

Si identificano come utenti del sistema i componenti del team e la federazione, questi si aspettano dal sistema:

- Un punto di accesso per poter osservare i dati (sia per la federazione che per gli utenti del team)
- La possibilità di osservare lo storico della telemetria
- La possibilità di un'analisi in tempo reale della telemetria
- La possibilità di comunicazione con il pilota
- La possibilità di impostare soglie di allarme, che se superate dai dati provenienti dalla telemetria, generano avvisi agli utenti del team

2.2 Requisiti funzionali

Dal punto di vista delle funzionalità, ci si aspetta che il sistema:

- Il sistema sia in grado di prevedere la ricezione dei principali parametri prodotti dalla telemetria del veicolo
  - RPM, velocità, temperatura, ecc.
- Sia in grado di rilevare condizioni di pericolo (ad esempio temperatura alta, RPM fuori soglia, ecc.) e segnalarle
- Dare la possibilità di configurare le soglie di cui al punto precedente
- Fornisca la possibilità di scaricare i dati storici della telemetria

2.3 Requisiti non funzionali

- Disponibilità: il sistema deve garantire forte disponibilità poiché non si può verificare lo scenario in cui il veicolo invia i dati ma il sistema non li percepisce e non li immagazzina
- Sicurezza: data la natura dei dati e la loro riservatezza occorre prestare molta attenzione agli aspetti di security per proteggere il sistema da eventuali attacchi o da utenti non autorizzati.
- Rendere il sistema più estendibile possibile in modo tale che ulteriori funzionalità possano essere implementate facilmente.
- Facilità di integrazione con sistemi già esistenti e non

2.4 Requisiti implementativi

Di seguito viene riportato un elenco delle tecnologie che verranno impiegate nello sviluppo del sistema:

- Architettura a micro-servizi del sistema
- Utilizzo di un MOM (Message Oriented Middleware) per la comunicazione tra micro-servizi (RabbitMQ)
- Utilizzo di MQTT per la comunicazione bidirezionale tra veicolo e sistema
- Utilizzo di Python come linguaggio per l'implementazione del sistema
- Utilizzo di Docker per la containerizzazione del sistema

3 Design

3.1 Struttura

Il sistema prevede diverse funzionalità come l'acquisizione della telemetria da parte del veicolo, la sua memorizzazione, la relativa analisi per osservare eventuali anomalie e la gestione degli utenti per interagire con il sistema. Dal momento che diverse sono le funzionalità del sistema, si è pensato che una soluzione distribuita fosse ideale per implementarlo; nello specifico un'architettura a microservizi si adatta piuttosto bene allo scopo.

L'architettura a microservizi consente di sviluppare il sistema in modo tale da aderire ai requisiti illustrati nelle sezioni precedenti. Adottare tale architettura significa rendere sin da subito il sistema facilmente estendibile e integrabile con nuove funzionalità, oltre a rendere i componenti tra loro debolmente accoppiati, punto fondamentale per i sistemi distribuiti organizzati a microservizi. Infine questa architettura consente di poter implementare i vari servizi con tecnologie e linguaggi differenti senza che gli altri servizi ne risentano in termini di prestazioni e di comunicazioni.

Per quanto riguarda la messa in produzione di un sistema basato su microservizi, risulta piuttosto semplice farne il deployment ma soprattutto è vantaggioso quando si devono fare azioni correttive mirate su un microservizio: non è necessario che venga nuovamente eseguito il deploy dell'intero sistema ma solo del servizio coinvolto. Inoltre questa architettura consente di scalare molto facilmente, ad esempio mediante replicazione.

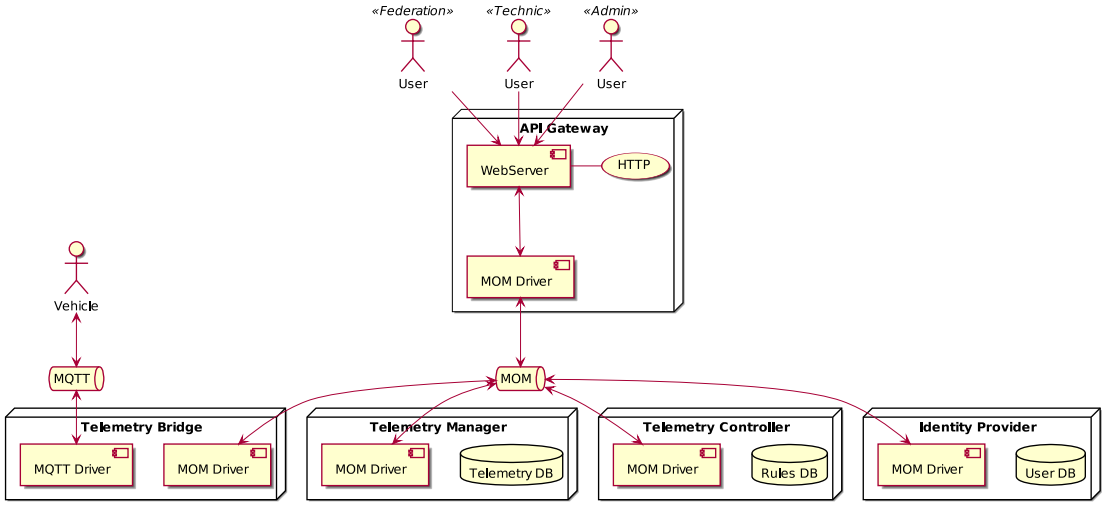

Figura 2: Architettura a microservizi del sistema con relative macro-interazioni tra i servizi.

In figura 2 si mostra l'architettura del sistema, nello specifico vengono mostrati i servizi coinvolti e le interazioni che hanno tra loro per lo scambio di informazioni.

L'**API Gateway** rappresenta l'entry-point del sistema: espone web API con cui gli utenti possono interagire sfruttando le funzionalità del sistema. Il **Telemetry Manager** ha come obiettivo quello di acquisire e memorizzare la telemetria dei veicoli. **Telemetry Controller** è il microservizio preposto ad osservare la telemetria e verificare che non ci siano regole (specificate dagli utenti) che vadano in conflitto con i dati appena ricevuti dalla telemetria, in tal caso segnala agli utenti mediante un alert che è stata violata una regola. **Telemetry Bridge** è il microservizio preposto ad acquisire tramite MQTT tutti i dati di telemetria dei veicoli e propagarli nel sistema. Infine **Identity Provider** ha la funzione di registrare e autenticare gli utenti.

3.1.1 Struttura interna dei microservizi

Ogni microservizio possiede una struttura interna comune che viene poi estesa e integrata con i moduli (classi) specifici per il funzionamento del servizio. Ogni microservizio ha come unico mezzo di comunicazione verso gli altri microservizi il **MOM**, quindi come mostrato in figura 3 abbiamo una classe astratta che rappresenta l'interfaccia del componente, tale interfaccia è poi istanziata in una classe che ne implementa la logica. La classe `MOM` rappresenta invece il mezzo di comunicazione ed è costituita da tanti `Component`. Ogni `Component` a sua volta può fare uso di altre entità come ad esempio un repository per interagire con le operazioni del DB, qualora tale componente avesse necessità di operare con una base dati.

Tale diagramma delle classi è ricorrente in tutti i microservizi presenti nel sistema.

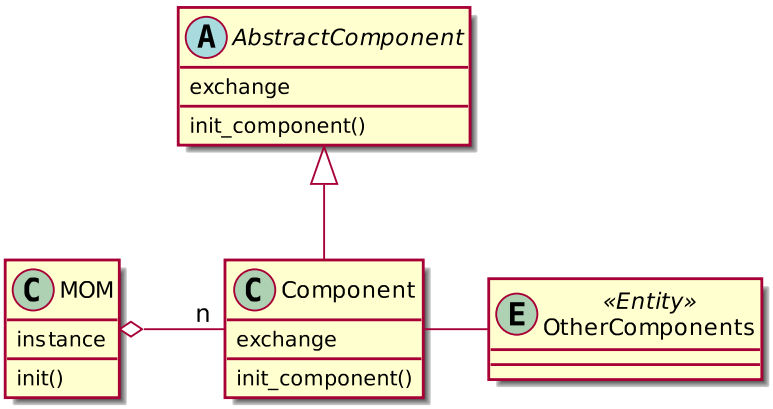

Figura 3: Architettura comune ai microservizi del sistema.

3.1.2 Entrypoint del sistema: API Gateway

Il quesito che viene naturale porsi quando si intende utilizzare il sistema è: come fanno i diversi client (implementati eventualmente con tecnologie differenti come webapp, applicazione Android/IOS, ecc.) ad interagire con il sistema e accedere ai vari servizi offerti? La granularità delle API offerte dai microservizi è spesso e volentieri differente dalle esigenze dell'utente; tipicamente i microservizi forniscono molte API semplici, il

che significa che il client necessita di interagire con più servizi per ottenere ciò che desidera. Per sopperire a questo, la soluzione è creare un'**API Gateway** che rappresenta l'unico entrypoint del sistema in cui i diversi client interagiscono. Nell'API Gateway si specificano tutte le operazioni che si vogliono offrire ed è poi suo compito gestire le richieste provenienti dai client e a sua volta ridireziona tale richiesta al microservizio (o più microservizi) corretto. Utilizzando questo pattern è possibile definire API specifiche per il tipo di dispositivo che fa richieste, ad esempio possono essere definite API differenti sulla base che a richiedere le risorse sia una web-app piuttosto che un app mobile.

L'utilizzo di un API Gateway ha i seguenti benefici:

- Isolare i client da problemi di determinazione della locazione dei microservizi
- Fornire API complete e ottimali per ogni tipo di client (non coperto dal progetto)
- Semplificare la logica del client spostando la necessità di fare chiamate multiple ai vari microservizi direttamente dentro al gateway
- Esporre un interfaccia standard "web friendly" che traduce internamente le richieste nei protocolli utilizzati, in questo caso in richieste per il MOM

Adottare questo pattern implica qualche "effetto collaterale", ad esempio si incrementa la complessità del sistema introducendo l'API Gateway; un altro aspetto da considerare è l'introduzione inevitabile di overhead dovuto ad ulteriori richieste che a sua volta il gateway deve fare. Nonostante gli svantaggi appena citati si è reputato utile adottare questo pattern in quanto i vantaggi offerti ne giustificano l'utilizzo.

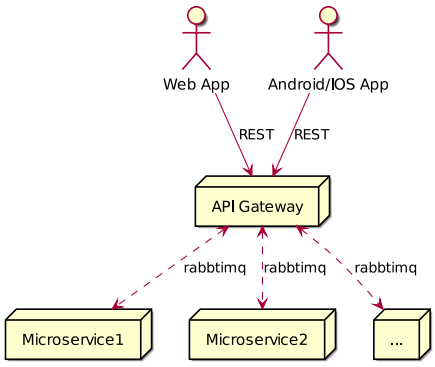

Figura 4: API Gateway pattern.

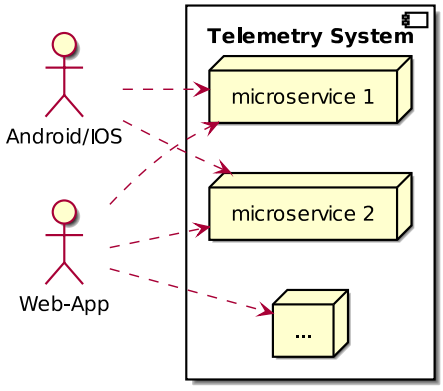

Figura 5: Direct access to microservices.

In figura 4 viene mostrato l'interazione dei client con il gateway, mentre in figura 5 si mostrano le interazioni dei client direttamente con i microservizi.

3.2 Comportamento

In questa sezione viene esaminato il funzionamento di ogni microservizio mostrando diagrammi di sequenza che illustrano i comportamenti a fronte della ricezione dei messaggi.

3.2.1 tvs-bridge

Questo microservizio si occupa di osservare la telemetria proveniente dal veicolo e propagarla al resto del sistema. Inoltre questo microservizio gestisce l'invio dei messaggi (creati dai tecnici) verso il veicolo.

In figura 6 viene mostrato il diagramma di sequenza che illustra come viene gestito un nuovo messaggio di telemetria proveniente dal veicolo.

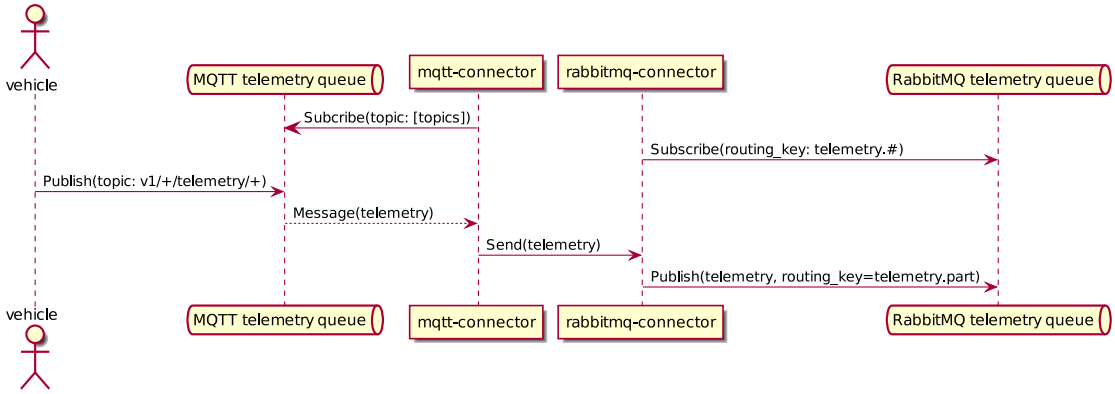

Figura 6: Diagramma di sequenza che mostra come il microservizio `tvs-bridge` gestisce nuovi messaggi di telemetria.

All'avvio del microservizio il componente `mqtt-connector` sottoscrive al broker MQTT tutti i topic a cui è interessato; allo stesso modo il componente `rabbitmq-connector` si sottoscrive alla coda destinata a gestire la telemetria. Una volta che i due componenti hanno effettuato la sottoscrizione è possibile ricevere la telemetria da parte del veicolo. Appena arriva un nuovo messaggio di telemetria, questo viene intercettato dal componente `mqtt-connector` che non fa altro che trasmetterlo al componente `rabbitmq-connector`, il quale lo invia nella coda associata.

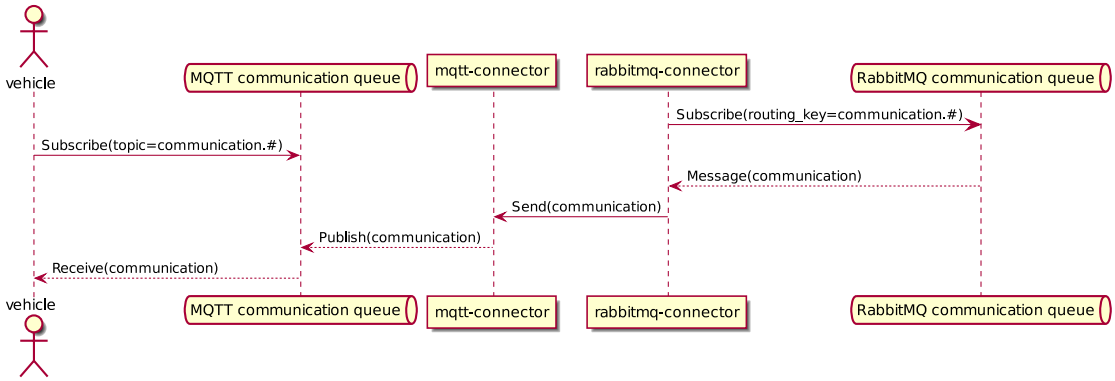

Figura 7: Diagramma di sequenza che mostra come il microservizio `tvs-bridge` gestisce le comunicazione verso il veicolo.

In figura 7 è rappresentato il diagramma di sequenza che mostra le interazioni tra i componenti del microservizio per propagare la comunicazione. Quando l'utente decide di inviare una comunicazione a un veicolo, tale messaggio viene selettivamente inviato al microservizio `tvs-bridge`. Una volta de-serializzato il messaggio, si verifica che la struttura sia compatibile con quella attesa e se il controllo ha successo viene propagato il messaggio..

3.2.2 tvs-telemetry-manager

L'obiettivo di questo microservizio è quello di osservare la telemetria di qualsiasi veicolo e memorizzarla nel DB. Inoltre è possibile interagire con il microservizio facendo richiesta dello storico della telemetria fornendo alcuni semplici filtri per ricevere selettivamente un solo sottoinsieme.

In figura 8 viene mostrato il diagramma di sequenza che illustra le fasi per il salvataggio della telemetria su DB. Inizialmente il microservizio si sottoscrive alla coda preposta alla comunicazione della telemetria. Non appena viene pubblicato un nuovo oggetto contenete la telemetria, la coda lo propaga al componente `rabbitmq-connector`, il quale procede a salvarla su DB.

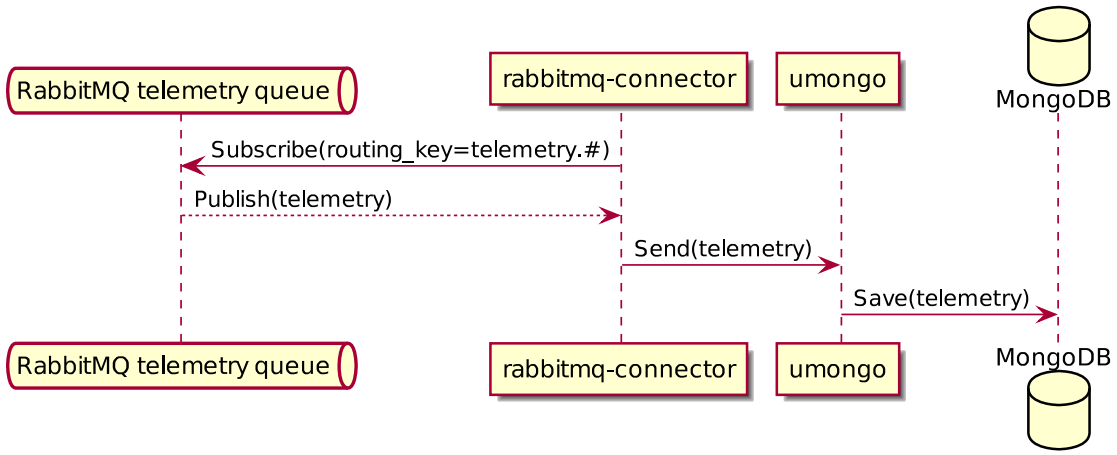

Figura 8: Diagramma di sequenza che mostra come il microservizio `tvs-telemetry-manager` memorizza la telemetria proveniente dai veicoli.

In figura 9 viene mostrato il diagramma di sequenza che mostra le operazioni per effettuare una query al DB per ottenere lo storico della telemetria.

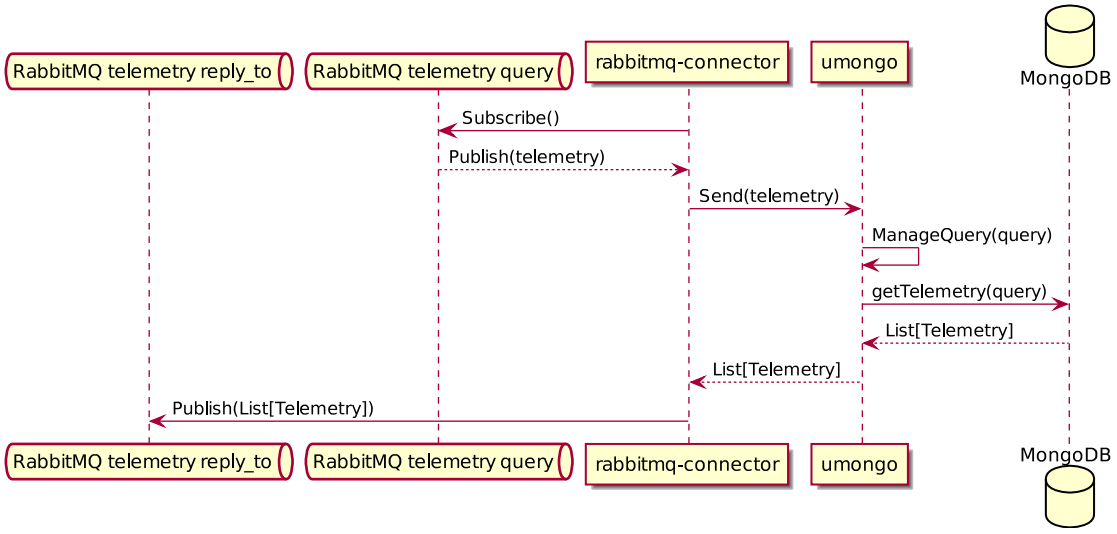

Figura 9: Diagramma di sequenza che mostra le interazioni per ottenere lo storico della telemetria.

3.2.3 tvs-telemetry-controller

Il controllo degli alert è effettuato da questo microservizio. Mediante l'osservazione della telemetria si verifica che i valori contenuti in essa non superino i valori impostati nelle

regole di telemetria. Nel caso si verificasse tale situazione il servizio emette un alert per avvisare gli utenti del sistema.

In figura 10 si mostra come il microservizio, per ogni dato di telemetria pubblicato, controlla se è definita una regola associata a quella telemetria e nel caso esista viene verificato che non violi tale regola. Se la regola viene violata viene generato un alert che avvisa gli utenti autorizzati del sistema. Si precisa che il diagramma di sequenza mostra solamente il caso in cui viene trovata una regola associata alla telemetria e quest'ultima viola tale regola; in tutti gli altri casi non viene generato alcun messaggio in quanto si assume che il veicolo non abbia problemi.

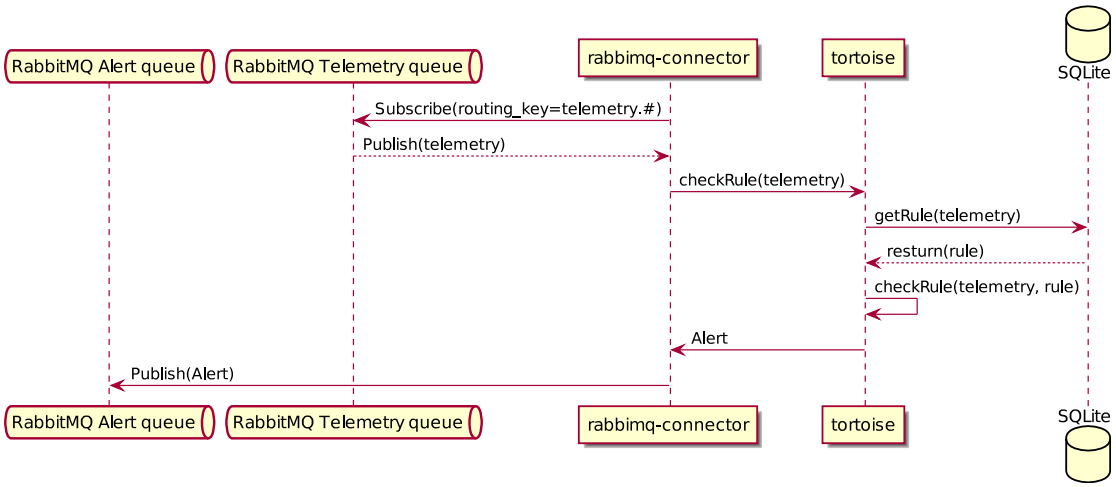

Figura 10: Diagramma di sequenza che mostra come viene controllata la telemetria e in caso di alert come viene propagato nel sistema.

In figura 11 viene mostrato il diagramma di sequenza che mostra come viene memorizzata una nuova regola.

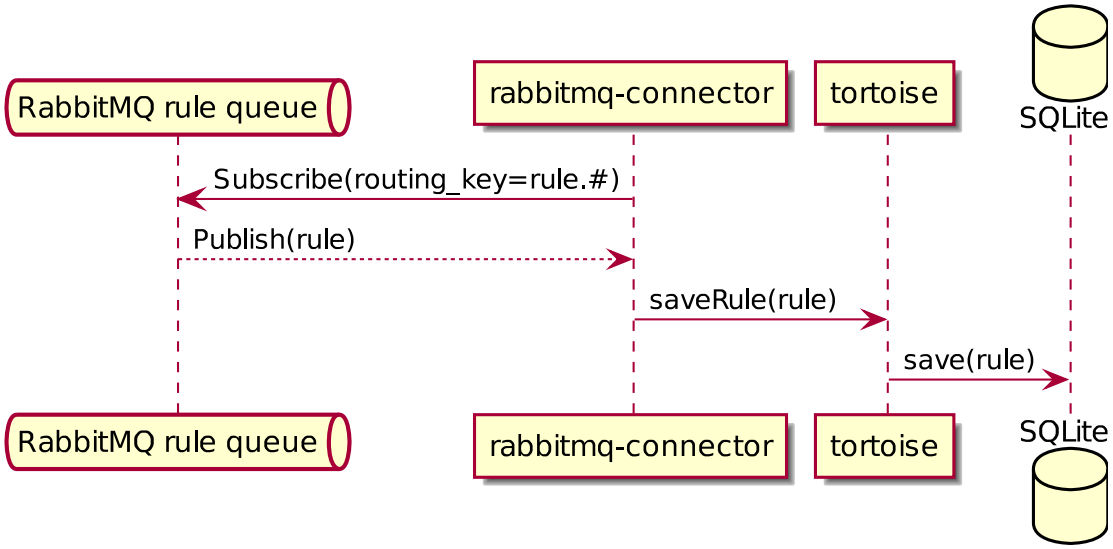

Figura 11: Diagramma di sequenza che mostra il salvataggio di una nuova regola per segnalare un eventuale alert.

3.2.4 tvs-identity

La gestione degli utenti del sistema è demandata a questo microservizio, il quale ha il compito di creare un nuovo utente e autenticare i diversi utenti del sistema. Di fatto è strutturato come identity provider per le restati componenti del sistema.

In figura 12 viene mostrato il diagramma di sequenza che mostra le fasi di creazione di un nuovo utente. Il componente `rabbitmq-connector` si mette in ascolto di un nuovo messaggio. Appena sopraggiunge un nuovo messaggio, viene decodificato verificando che abbia una struttura valida, in caso affermativo viene salvato il nuovo utente avendo cura di fare prima salt+Hash della password. Una volta salvato l'utente nel DB, viene inviato alla coda di risposta l'oggetto contenete l'utente appena creato a conferma della creazione dello stesso.

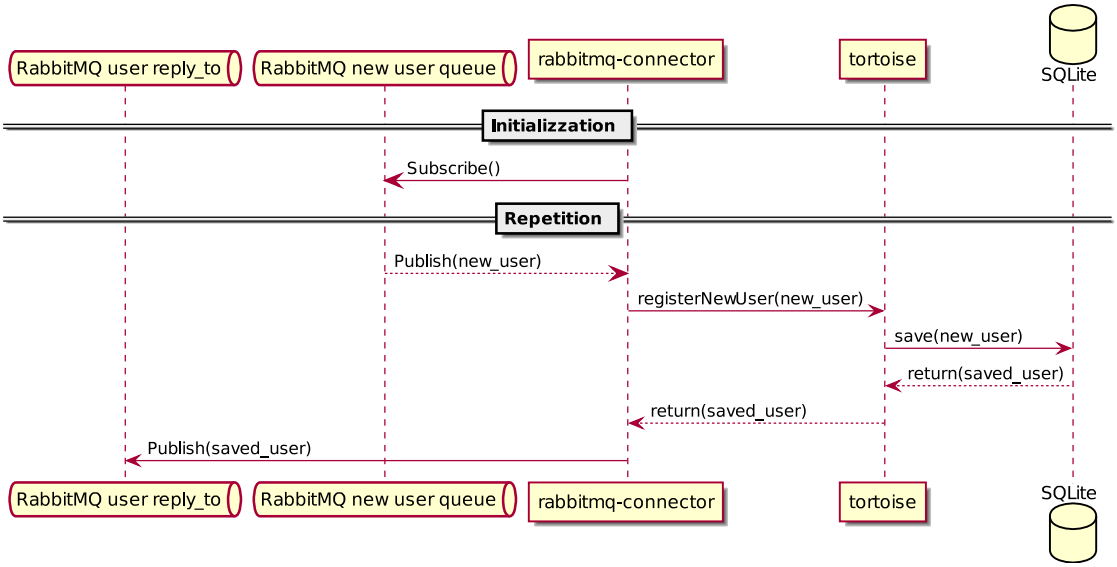

Figura 12: Diagramma di sequenza che mostra l'operazione di registrazione di un nuovo utente nel sistema.

In figura 13 viene mostrato il diagramma di sequenza che mostra l'autenticazione di un utente al sistema. Per ogni richiesta viene effettuata l'interrogazione al DB per ottenere i dati dell'utente che si vuole autenticare. Viene quindi verificato il token di autenticazione: nel caso sia valido viene inviato un messaggio che riporta l'esito positivo dell'autenticazione; diversamente viene propagato un messaggio di errore.

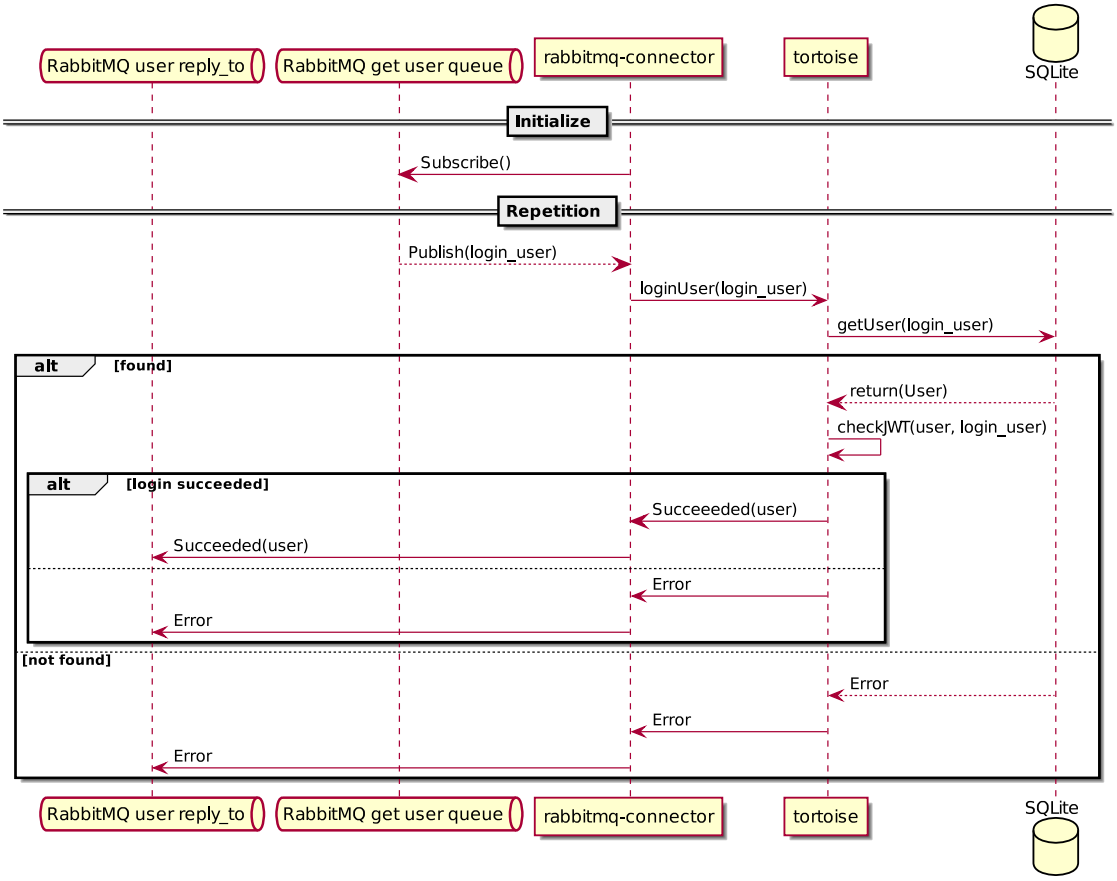

Figura 13: Diagramma di sequenza che mostra l'autenticazione dell'utente.

3.2.5 tvs-gateway

Questo microservizio è di fatto l'entrypoint del sistema, mediante il quale vengono esposte API per interagire con il sistema; `tvs-gateway` rappresenta la parte più complessa e quella che opera con tutti gli altri microservizi. I servizi offerti si dividono in due categorie:

- API per lo storico dei dati: questa rappresenta la parte "statica", ovvero la parte in cui è l'utente ad iniziare l'interazione.
- Dati in real-time: questa è la parte "dinamica" nella quale il server attivamente comunica i dati di telemetria ed eventuali alert.

Per quanto riguarda la prima, queste operazioni vengono offerte esponendo API, mentre per la seconda il servizio è offerto mediante websocket.

3.3 Interazioni

In questa seziona verranno analizzate le diverse interazioni che i microservizi hanno tra loro per ottemperare le funzionalità del sistema.

La telemetria che i veicoli producono è l'elemento principale che il sistema deve gestire, a tal proposito in figura 14 viene mostrato un diagramma di sequenza che mostra come i microservizi interagiscono tra loro e con l'utente a fronte della ricezione di un nuovo dato di telemetria. Appena un veicolo pubblica un nuovo messaggio contenente la telemetria, i microservizi `tvs-telemetry-manager` e `tvs-telemetry-controller` lo ricevono in modo parallelo e asincrono; rispettivamente viene memorizzata la telemetria e controllato che un eventuale regola per quel valore non sia violato. A seguito del salvataggio della telemetria viene propagato tale valore all'utente attraverso il gateway. Qualora una qualsiasi regola venisse violata, il servizio dedicato provvede ad emettere un messaggio di alert all'utente.

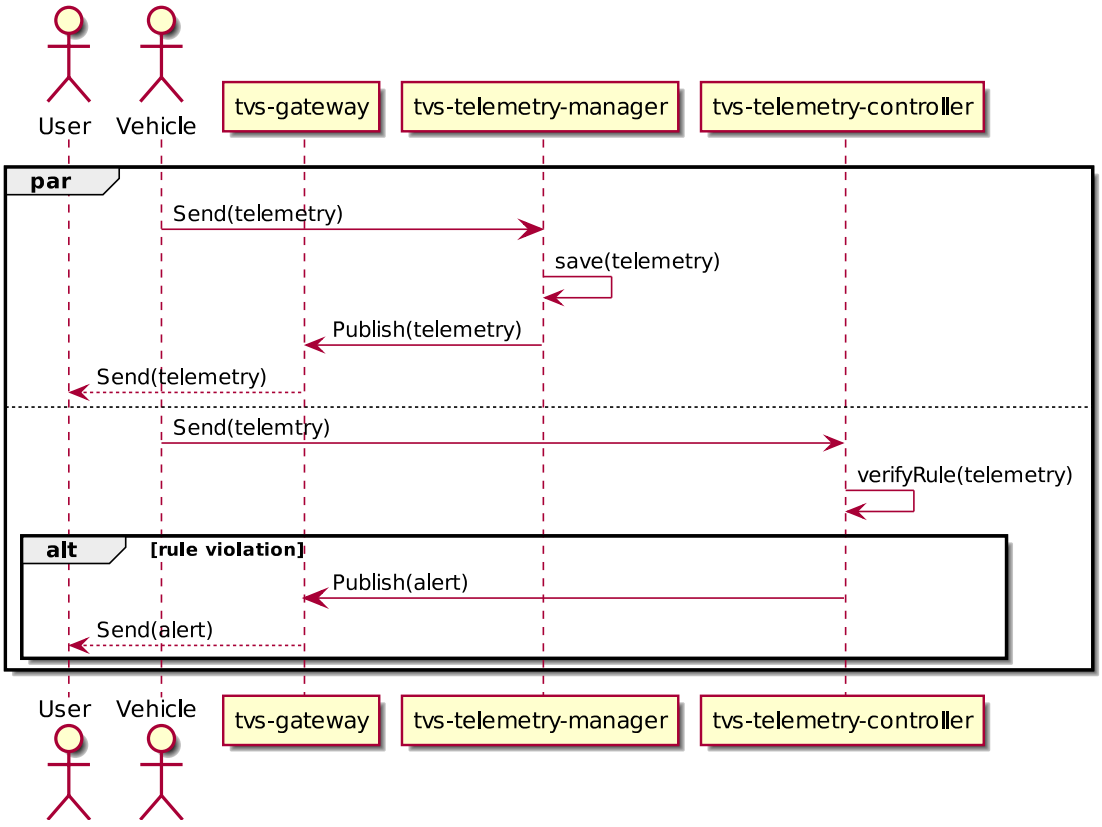

Figura 14: Diagramma di sequenza che mostra le iterazioni che i microservizi hanno a fronte della ricezione di un nuovo dato di telemetria.

In figura 15 viene mostrato il diagramma di sequenza che riporta il funzionamento della comunicazione di un utente verso un veicolo. L'utente, mediante il gateway, invia

il messaggio che vuole fare recapitare al veicolo. Il gateway elabora il messaggio e lo propaga al microservizio `tvs-bridge` il quale, verificato che sia un messaggio valido, lo invia al veicolo.

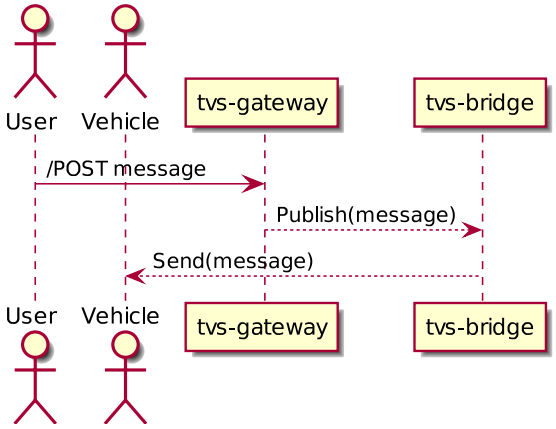

Figura 15: Diagramma di sequenza che mostra l'invio di un messaggio da parte di un utente ad un veicolo.

Una delle funzionalità più importanti che il sistema mette a disposizione è la possibilità di fornire all'utente lo storico della telemetria di un dato veicolo. In figura 16 viene mostrata l'interazione tra il gateway (unico servizio esposto direttamente all'utente) e il microservizio preposto alla memorizzazione della telemetria. Nello specifico l'utente effettua una chiamata all'API relativa alla telemetria, il gateway elabora la richiesta e la pubblica sulla coda. Il microservizio `tvs-telemetry-manager` riceve la richiesta e fornisce la risposta con i dati di telemetria al gateway che risponde all'utente con i dati ricevuti.

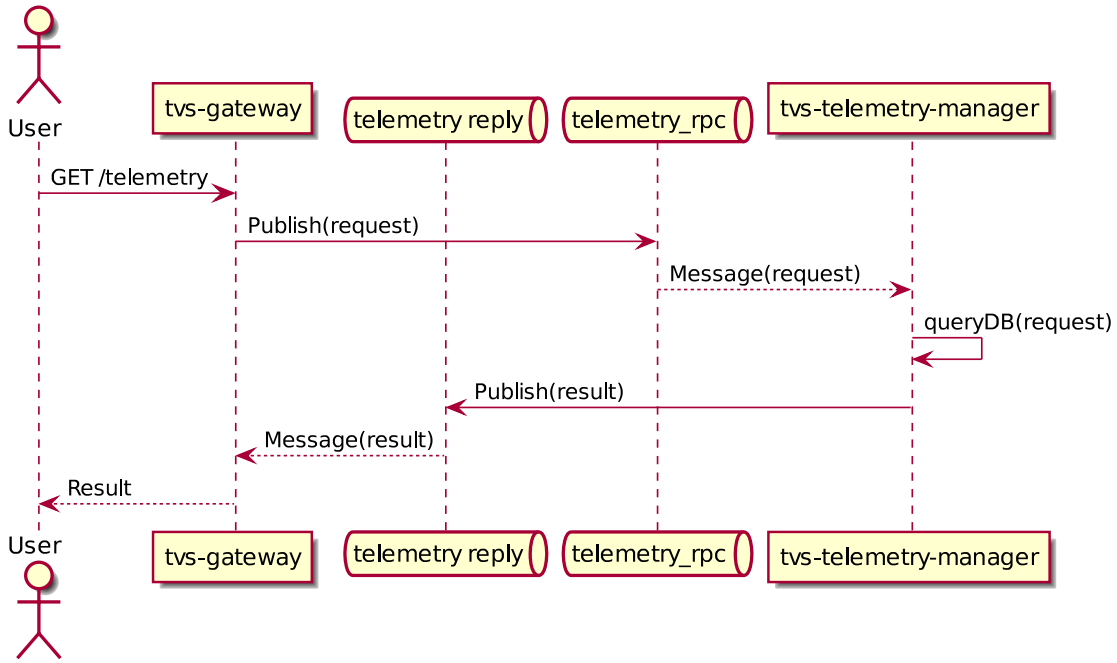

Figura 16: Diagramma di sequenza che mostra l'interazione del gateway con il microservizio preposto alla memorizzazione della telemetria.

Altre funzionalità che il sistema mette a disposizione come: segnalazione di allarmi e autenticazione dell'utente non vengono riportati per non appesantire la lettura poiché seguono lo stesso pattern di comunicazione appena mostrato in figura 16.

4 Tecnologie usate

In questa sezione si esaminano le principali tecnologie utilizzate nel progetto, illustrando le caratteristiche e i vantaggi che hanno nell'implementazione del progetto.

4.1 RabbitMQ

RabbitMQ è un **message broker** open-source che utilizza lo standard AMQP (Advanced Message Queuing Protocol) e non solo. È caratterizzato dal fatto che garantisce alta disponibilità, fault-tolerance e scalabilità, oltre a fornire alte prestazioni nella gestione concorrente di operazioni.

RabbitMQ è un middleware che consente a diversi servizi di un'applicazione di comunicare tra loro senza preoccuparsi di perdite di messaggi in quanto fornisce diversi livelli di QoS (Quality of Service). Fornisce inoltre un routing efficiente dei messaggi garantendo così disaccoppiamento tra i servizi.

Tra i vari casi d'uso di RabbitMQ troviamo le architetture basate su microservizi dove RabbitMQ rappresenta una soluzione efficacie ed efficiente per la comunicazione

tra i servizi del sistema. Quando progettiamo applicazioni a microservizi emergono principalmente due problematiche come il service discovery e il drop o la duplicazione dei messaggi nella rete; RabbitMQ può aiutare in entrambi i casi utilizzando il concetto di coda come mezzo di trasporto. I servizi possono pubblicare o consumare i messaggi dalle code disaccoppiando di fatto i servizi che stanno comunicando. Inoltre risolve il problema che se un servizio non è temporaneamente disponibile, allora RabbitMQ conserva il messaggio (al contrario di HTTP) e lo invia quando il servizio torna disponibile senza quindi perdite di comunicazioni. Anche la service discovery è altrettanto semplice: basta solo sapere dove è il broker di RabbitMQ e il nome della coda su cui operare. In questo caso il nome della coda funge da "indirizzo" del servizio. Consumare i messaggi da una coda risulta essere scalabile in quanto più consumer contemporaneamente possono ricevere i messaggi che vengono propagati nella coda e meccanismi di load balancing garantiscono che il carico sia distribuito equamente.

RabbitMQ risulta particolarmente utile in tutti quei contesti in cui le architetture sono fluide e i microservizi cambiano poiché fornisce un meccanismo di routing dei messaggi estremamente efficacie. Le logiche di routing sono implementate in base al tipo di exchange che può essere dinamicamente creato dall'applicazione quando ne ha necessità. I servizi specificano un insieme di code su cui vogliono ricevere i messaggi ed effettuano un binding tra exchange e code, specificando un pattern associato alla chiave che colui che invia il messaggio utilizza quando pubblica un messaggio (la chiave è una sorta di metadato che serve all'exchange per fare il routing del messaggio alle code giuste).

RabbitMQ fornisce quattro tipologie di exchange che cercano di coprire i principali casi d'uso per lo scambio di messaggi:

- **Direct exchange**: il messaggio viene consegnato a tutte le code il cui bindind key è uguale al routing key riportato nel messaggio. Se si effettua un binding della coda con binding key uguale al nome della coda stessa, allora è come se si effettuasse una comunicazione one-to-one. Si può recapitare il messaggio a più code direttamente specificando la stessa binding key per diverse code.

- **Topic exchange**: il messaggio viene consegnato a tutte le code dove la wildcard del binding key fa match con la routing key del messaggio. Se ad esempio il binding key è `log.*.error` allora se tutti i messaggi che fanno match con quella espressione regolare (come `log.ui.error` oppure `log.payment.error`) vengono recapitati alle code corrispondenti.

- **Fanout exchange**: tutti i messaggi vengono propagati alle code in cui è presente un binding con questo tipo di exchange. In questo modo non è necessario a livello di applicazione definire logiche per il broadcast dei messaggi, basta solo fare il binding delle code con questo tipo di exchange e tutti i messaggi verranno consegnati. Non è necessario specificare alcun binding key in poiché verrebbe ignorato.

- **Headers exchange**: questo exchange si basa sulla struttura del messaggio AMQP ed è in grado di effettuare routing complessi basandosi sull'header del messaggio (e non solo).

4.2 MQTT

MQTT è un protocollo di tipo publish/subscribe particolarmente leggero e con una occupazione di banda minima. A differenza di HTTP che è un protocollo di tipo request/response, MQTT è di tipo "event driven" e quindi i messaggi sono inviati in modo asincrono al client. Questo tipo di architettura disaccoppia i client da tutti gli altri rendendola scalabile senza introdurre dipendenze tra i client.

Alla base di MQTT vi sono due entità principali: **MQTT broker** ed **MQTT clients**. Il broker è responsabile di consegnare il messaggio di un client mittente ad uno o più client ricevitori. Un client pubblica il messaggio al broker e i client che vogliono ricevere il messaggio si sottoscrivono al broker ricevendo così il messaggio. Ogni messaggio MQTT include un *topic*: il client che invia il messaggio specifica un topic e tutti gli altri client che vogliono ricevere quel messaggio devono sottoscriversi al broker con lo stesso topic. Per aumentare il controllo e la granularità della ricezione dei messaggi, quando un client si sottoscrive al broker può specificare un topic contenente un pattern (un'espressione regolare) che se fa match con il topic presente nel messaggio allora quest'ultimo viene recapitato al client, diversamente non verrà recapitato.

Il broker è anche in grado di fare buffering dei messaggi: se un messaggio non può essere recapitato al client poiché quest'ultimo non è connesso, allora il messaggio viene tenuto memorizzato nel broker fino a quando il client non ritorna disponibile. Questo meccanismo risulta particolarmente utile quando si hanno dispositivi che hanno una connessione alla rete non affidabile. Per supportare la consegna affidabile dei messaggi MQTT supporta tre tipi di QoS (Quality of Service): 0, 1 e 2. Il QoS è un accordo tra il mittente e il destinatario che specifica il livello di garanzia con il quale il messaggio verrà consegnato. Vengono ora mostrate le caratteristiche dei tre tipi di QoS in MQTT:

- **QoS 0 - at most once**: questo è il livello minimo, garantisce una consegna del messaggio a "best effort", quindi non viene data nessuna garanzia sulla consegna del messaggio. Il ricevitore non spedisce nessun ACK di conferma e il messaggio non viene memorizzato dal mittente per una eventuale ritrasmissione. Questo livello è anche chiamato "fire and forget" e fornisce lo stesso livello di affidabilità del protocollo TCP.

- **QoS 1 - at least once**: questo livello garantisce che almeno una volta il messaggio viene recapitato al destinatario. Il mittente memorizza il messaggio fino a quando non riceve un `PUBACK` dal destinatario che assicura la ricezione del messaggio. È possibile che il messaggio venga recapitato più di una volta.

- **QoS 2 - exaclty once**: è il livello di servizio più alto in MQTT. Ci garantisce che ogni singolo messaggio è stato recapitato una sola volta ai destinatari corretti. Questo livello è il più lento ma più sicuro tra i tre. Questo livello è implementato con un "four-part handshake" tra mittente e destinatario.

4.2.1 Mosquitto

Eclipse Mosquitto è un broker MQTT open-source che implementa il protocollo MQTT nelle versioni 5.0, 3.1.1, 3.1. È caratterizzato per la sua leggerezza ed è adatto in tutti i contesti d'uso dai dispositivi a basso consumo fino a server ad alte prestazioni.

4.3 Docker

Docker rappresenta la soluzione migliore per effettuare il deployment di un sistema. Grazie a Docker l'applicazione viene eseguita all'interno di un container, il quale fornisce al sistema tutte le risorse di cui ha bisogno per essere eseguita. Questo approccio ha il vantaggio che possiamo effettuare il deployment del sistema senza preoccuparci di installare dipendenze esterne o preparare il sistema oppure preoccuparci della compatibilità delle risorse su cui vogliamo eseguire l'applicazione.

Nell'ambito dei microservizi è diventato lo standard "de-facto" per il deployment. Tra i maggiori vantaggi nell'utilizzo di docker in sistemi distribuiti e a microservizi vi è l'isolamento dell'ambiente di esecuzione di ogni servizio, quindi è possibile prevedere diversi ambienti di esecuzione indipendenti tra loro. Altro vantaggio è la scalabilità: scalare un microservizio mediante docker è semplice e immediato, oltre a portare tutti i vantaggi di una replicazione per aumentare la disponibilità o bilanciare il carico, ad esempio.

5 Dettagli implementativi

L'intero sistema è stato scritto utilizzando Python come linguaggio di riferimento. Dal momento che principalmente l'intero sistema e, nello specifico, ogni microservizio effettua operazioni IO-bound, si è deciso di basarsi sulla libreria `asyncio` [2] fornita direttamente da Python.

La libreria `asyncio` consente di scrivere codice concorrente utilizzando il meccanismo **async/await**. Nella documentazione viene riportato:

> "asyncio is used as a foundation for multiple Python asynchronous frameworks that provide high-performance network and web-servers, database connection libraries, distributed task queues, etc."

A seguito di quanto riportato si sono identificate tutte le varie librerie utili a implementare il sistema basate su `asyncio`. Questo ha permesso di scrivere e implementare in modo semplice e particolarmente leggibile i microservizi, senza rinunciare ad aspetti di prestazioni (che la libreria `asycio` garantisce).

A supporto ulteriore della scelta di utilizzare questa libreria si osserva più avanti nella documentazione la seguente affermazione:

> "asyncio is often a perfect fit for IO-bound and high-level structured network code."

Vista la natura del progetto e le operazioni principali che devono essere eseguite, questa libreria rappresenta la scelta migliore.

Di seguito vengono riportate le librerie utilizzate per le implementazioni dei microservizi:

- `aio-pika` [1]: libreria utilizzata per interfacciarsi a RabbitMQ. Questa libreria espone API totalmente asincrone basate su asyncio.
- `paho-mqtt` [5]: libreria utilizzata per utilizzare MQTT. Questa libreria non è basata su asyncio, ma con alcuni accorgimenti è stato possibile integrarla con API asincrone.
- `fastapi` [3]: libreria impiegata per realizzare REST API. Questa libreria si basa su asyncio e promette prestazioni elevate.
- `tortoise` [7]: libreria impiegata per interfacciarsi a DB di tipo SQLite. Libreria ORM basata su asyncio.
- `Motor` [4]: libreria utilizzata per interfacciarsi a MongoDB. Libreria ODM basata su asyncio.

5.1 Replica set in MongoDB

Tra i vari requisiti di progetto vi è quello della disponibilità: vogliamo che il sistema sia sempre disponibile e che non abbia interruzioni. La parte di memorizzazione della telemetria rappresenta la parte più critica del sistema in quanto non è ammissibile che i veicoli producano la telemetria senza che questa venga memorizzata. Per fronteggiare questa problematica si è ricorso ai replica set di MongoDB [6]. Un replica set in MongoDB non è altro che un insieme di processi `mongod` che mantengono gli stessi dati. Replica set fornisce ridondanza e replicazione dei dati, garantendo così alti livelli di fault tolerance. In questa modalità viene definito un nodo primario, il quale riceverà tutte le operazioni di scrittura. Tutti i nodi definiti come secondari operano in modo tale da riflettere il contenuto del dataset del nodo primario. Se il nodo primario per qualche ragione non dovesse essere non disponibile, allora un nodo secondario inizia un processo di elezione per eleggere un nuovo nodo primario. In questo modo garantiamo che se un nodo diventa non disponibile abbiamo la garanzia del fatto che i dati che vogliamo memorizzare verranno memorizzati senza interruzioni. Nel caso specifico del progetto è stata scelta una topologia chiamata "three-member replica set" che consiste in un nodo primario e due nodi secondari.

5.2 Autenticazione utenti

La parte di autenticazione utenti ha richiesto una certa disamina in fase di progettazione: l'autenticazione al sistema è una parte critica del sistema stesso in quanto utenti non autorizzati non possono aver accesso ai dati della telemetria e operare configurazioni.

A tal proposito si è cercato di identificare quale fosse il meccanismo di autenticazione e autorizzazione migliore in base al dominio applicativo e alle tecnologie in gioco.

Nei sistemi distribuiti la classica autenticazione basata su cookies potrebbe risultare critica: tutte le istanze che richiedono l'autenticazione di un utente devono coordinarsi e sincronizzarsi per essere aggiornati con le sessioni dei vari utenti. Seppur questo non rappresenti un particolare problema, risulta tedioso ed "error prone" adottare questo approccio in un sistema distribuito.

Una possibile soluzione è data dal meccanismo di autenticazione **JWT** (Json Web Token). Questo meccanismo basa il suo funzionamento sul generare un token dipendente da un seme (segreto) che viene inviato all'utente a seguito dell'autenticazione. Con tale token l'utente è in grado di autenticarsi al sistema senza dover far uso di cookies oppure inserire ogni volta username e password. Questo meccanismo risolve il problema della sincronizzazione tra parti poiché basta solamente che le varie istanze condividano solo il seme e la decodifica del token è univoca, ciò significa che non serve coordinazione tra eventuali parti del sistema per autenticare un utente.

Nel caso del sistema progettato, si è scelto di adottare un pattern ricorrente: utilizzare un microservizio preposto a gestire l'autenticazione degli utenti del sistema. Allo stato attuale viene gestita una sola istanza di tale microservizio, ma nel caso si voglia scalare nel numero delle istanze, queste non necessiterebbero di sincronizzarsi tra loro poiché il meccanismo offerto da JWT non lo richiede. Questo è un grosso vantaggio sia in termini di complessità del codice sia di overhead dovuto alla sincronizzazione tra le parti

5.3 Documentazione REST API

Il sistema espone REST API per poter interagire con esso. La documentazione di tali API è stata generata mediate `Swagger` adottando lo standard `OpenAPI`. La documentazione delle API è disponibile visitando `http://localhost:8000/docs`. Per generare tale documentazione occorre avviare il sistema mediante il comando: `docker-compose up -d` e poi è possibile accedere alla documentazione all'indirizzo sopra citato.

6 Validazione

Per la validazione del codice ci si è basati sui test scritti per ogni microservizio. Ove possibile il codice è stato sviluppato mediante la tecnica *Test Driven Development* in cui sono stati sviluppati una serie di unit test che vanno a verificare i principali scenari d'uso. Dove questo approccio non avesse consentito di scrivere test significativi oppure dove non fosse stato possibile in quanto si richiedevano comunicazioni con altri microservizi, allora la validazione è stata eseguita utilizzando le API del gateway verificando che i comportamenti e le risposte fossero congrue con il comportamento desiderato.

É stata configurata la CI/CD di gitlab per eseguire, ad ogni commit, dei controlli sulla qualità del codice. In particolare sono stati definiti due stage: il primo verifica mediante `pylint` che la formattazione dei sorgenti sia adeguata alle regole prefissate per il progetto; il secondo invece manda in esecuzione gli unit test mediante il tool `pytest`.

7 Testing

Per ogni microservizio sono stati definiti (dove possibile) dei test che non fossero banali. É stato utilizzato `pytest` come framework per il testing. Mediante la CI/CD di Gitlab è stato possibile eseguire gli unit test di ogni microservizio in maniera totalmente automatizzata ad ogni commit, in questo modo risulta piuttosto facile individuare eventuali errori nel codice.

Tra i test più rilevanti troviamo: la verifica della funzionalità di login e creazione di un utente. In questi test si sono andati a testare tutti gli scenari che possono accadere in fase di registrazione e creazione di un utente. É stata definita una *fixture* che crea un DB fittizio e "fresco" per ogni esecuzione dei test, in questo modo abbiamo garanzia del fatto che operiamo ogni volta in un ambiente non "sporcato" dai test precedenti.

7.1 Sviluppi futuri nei test

Durante la fase di creazione dei test ci si è resi conto che molte delle funzionalità di un microservizio richiedono l'interazione con altri microservizi, per questo motivo è risultato molto difficile (talvolta impossibile) creare unit test significativi e complessi. Ricercando soluzioni che potessero sopperire a questo problema, si sono trovati i mock: approccio utilizzato negli unit test in cui si sostituiscono le dipendenze con oggetti che simulano i comportamenti reali del sistema. Con questo approccio si sostituirebbero le interazioni che il microservizio deve avere con gli altri, creando degli oggetti fittizi. Il modo in cui `pytest` gestisce i mock risulta particolarmente complesso e di non facile comprensione; anche la documentazione offerta risulta poco chiara e dispersiva. Per questi motivi, allo stato attuale, non è stato possibile farne uso nel progetto.

In uno sviluppo futuro l'utilizzo dei mock è fondamentale, per questo motivo sarà obiettivo futuro esaminare meglio la documentazione per poter utilizzarli massivamente nei test del progetto, coprendo così il maggior numero di funzionalità del sistema.

8 Esempi d'uso

Per testare la parte visualizzazione della telemetria e alert in tempo reale è possibile fare uso di uno script creato appositamente per simulare un veicolo che invia per un certo tempo la propria telemetria al sistema. Prima di procedere con l'osservazione della telemetria è opportuno seguire i passi seguenti:

- Autenticarsi al sistema mediante l'API `localhost:8000/token` fornendo username e password (un utente presente nel sistema è username=mario@rossi.it e password=mariorossi). Verrà data come risposta un token.
- Collegarsi ora alle websocket per la telemetria e gli alert: un possibile strumento per poter testare le websocket è presente all'indirizzo `https://www.websocket.org/echo.html`. Nella casella *location* inserire la seguente stringa: `ws://localhost:8000/ws/telemetry?token={TOKEN}`, dove {`TOKEN`} è il token acquisito al punto precedente, questo server per autenticare la websocket.

- Se la connessione è andata a buon fine ogni volta che il veicolo emette un dato di telemetria, questo viene visualizzato nella console della websocket.
- Procedere in modo analogo per collegarsi alla websocket per gli alert, semplicemente sostituire `telemetry` con `alert`. Ovviamente le due websocket sono indipendenti e possono essere utilizzate simultaneamente.
- A questo punto è possibile eseguire lo script e osservare nella websocket della telemetria, i dati che provengono dal veicolo; nella websocket degli alert invece verranno visualizzati gli eventuali allarmi.

Per maggiori dettagli su come simulare la telemetria di un veicolo si faccia riferimento al `README.md` nel repository `https://gitlab.com/pika-lab/courses/ds/projects/ds-project-farabegoli-ay-20-21`

Per quanto riguarda l'utilizzo della parte di API si faccia riferimento alla documentazione swagger illustrata precedentemente; nella documentazione vengono forniti degli esempi d'uso delle API.

9 Conclusioni

È stato implementato un sistema di acquisizione della telemetria di veicoli da corsa. Il sistema prevede la memorizzazione della telemetria, la gestione di allarmi e la comunicazione con il pilota. Il sistema è stato realizzato con architettura a microservizi e un MOM per la coordinazione dei microservizi stessi.

9.1 Lavori futuri

Attualmente la gestione degli allarmi non prevede la loro memorizzazione. Una possibile implementazione futura potrebbe essere quella di creare un nuovo microservizio che si occupa solamente della memorizzazione degli alert dei vari veicoli, predisponendo quindi una controparte REST per poter interagire con tale microservizio.

Allo stato attuale non è presente una parte di frontend per semplificare le operazioni agli utenti del sistema, si prevede quindi in futuro di realizzare un frontend che racchiuda tutte le funzionalità del sistema in un interfaccia semplice e intuitiva.

9.2 Cosa abbiamo imparato

Lo sviluppo di questo progetto ha permesso di prendere confidenza con diverse tecnologie e strumenti applicati in un contesto distribuito. Architettando il sistema a microservizi ha consentito di esaminare vantaggi e svantaggi di tale architettura, portando a studiare pattern ricorrenti tipici di questo modello cercando di portarli nel contesto di questo progetto. Si è preso confidenza con nuove tecnologie come MQTT e RabbitMQ; nel caso di MQTT si è studiato il modello di comunicazione e lo si è reputato adatto per comunicare i dati di telemetria. RabbitMQ ha permesso una comunicazione tra i microservizi piuttosto agevole e semplice ma allo stesso tempo robusta e che soddisfa i requisiti del

progetto. Durante l'implementazione del sistema si è studiato nel dettaglio il funzionamento di RabbitMQ e i suoi possibili scenari d'uso e modalità operative consentendo di individuare quali fossero le migliori implementazioni da utilizzare in base al tipo di comunicazione necessaria. Si è presa confidenza con la programmazione asincrona offerta dalla libreria `asyncio` che ha permesso di realizzare un codice facilmente leggibile ma allo stesso tempo performante. Infine si è studiato il funzionamento di swagger+OpenAPI per la generazione e documentazione di API REST standard, scrivendo API chiare e ben documentate. Grazie a questo progetto è stato possibile approfondire il funzionamento di diverse tecnologia attualmente largamente utilizzate in sistemi distribuiti moderni cimentandosi nell'integrazione tra queste in modo semplice ed efficace.

Riferimenti bibliografici

[1] aio-pika library reference. `https://aio-pika.readthedocs.io/en/latest/`. Visitato: 12/01/2021.

[2] asyncio library reference. `https://docs.python.org/3/library/asyncio.html`. Visitato: 12/01/2021.

[3] fastapi library reference. `https://fastapi.tiangolo.com/`. Visitato: 12/01/2021.

[4] motor library reference. `https://motor.readthedocs.io/en/stable/tutorial-asyncio.html`. Visitato: 12/01/2021.

[5] paho-mqtt library reference. `http://www.eclipse.org/paho/`. Visitato: 12/01/2021.

[6] replica-set riferimento. `https://docs.mongodb.com/manual/replication/`. Visitato: 18/06/2021.

[7] tortoise library reference. `https://tortoise-orm.readthedocs.io/en/latest/`. Visitato: 12/01/2021.