**Introduzione**

Ogni linguaggio verrà utilizzato per un uso specifico, elencato di seguito e mostrato graficamente in un diagramma a blocchi:

**C++ (riquadro in blu):**

Il linguaggio in questione verrà utilizzato per “simulare” i veicoli di sharing e un’infrastruttura per raccogliere i pacchetti e renderli disponibili ai livelli sottostanti.

Nello schema i riquadri con su scritto “Veicolo 1” a “Veicolo n”, vogliono rappresentare il veicolo reale che produce dati telemetrici durante ogni suo utilizzo. Di norma questi manderebbero queste informazioni sulla rete internet tramite LoraWan, 2G, 5G o altre tecnologie wireless; per semplificare il progetto questi saranno dei singoli programmi C++ che manderanno dei dati pre-generati da veicoli reali, con una piccola varianza randomica per creare delle differenze nei grafici. Ogni veicolo è in grado di memorizzare tutti i dati dei sensori su una memoria persistente, in caso il gateway sia temporaneamente irraggiungibile (come le scatole nere negli aeroplani).

I veicoli manderanno questi dati al “Gateway per veicoli” in formato RAW, ovvero una concatenazione ordinata di byte, attraverso dei pacchetti UDP. Il gateway si preoccuperà di raggrupparne quanti più dati possibili in attesa che lo scraper python li richieda. In caso di attesa prolungata, il gateway può temporaneamente salvare i dati in memoria permanente, per poi rileggerli quando lo scraper si risveglia (il Gateway può notificare allo Scraper che sono disponibili nuovi pacchetti, in caso quest’ultimo non ne richieda da x secondi).

**Python (riquadro in verde):**

Il linguaggio in questione verrà utilizzato per decifrare i pacchetti RAW dei veicoli, in modo da renderli leggibili per l’essere umano.

Il “Server Veicoli” si occuperà di raccogliere tutti i dati che il gateway mette a disposizione e dividerli tra gli y elaboratori.

Gli elaboratori avranno a disposizione una funzione comune per poter decodificare i pacchetti, una sorta di stele di rosetta, in modo che ogni elaboratore sia indipendente l’uno dall’altro. La scelta di usare più elaboratori è legata al fatto che un singolo veicolo può produrre centinaia di pacchetti al secondo, e ogni pacchetto contiene più di 4~6 valori indipendenti tra di loro (sono 6 record distinti in un file csv o sql).

Tutti gli elaboratori manderanno i risultati dei pacchetti decodificati al “convogliatore result”, il quale si preoccuperà di inviare i dati ai “worker .csv” e al “worker SQL”.

Il primo si preoccuperà di salvare in modo persistente i dati elaborati di ogni veicolo, ognuno dei quali avrà la sua cartella e i suoi file divisi per giorno. A tale scopo si useranno per l’appunto file csv.

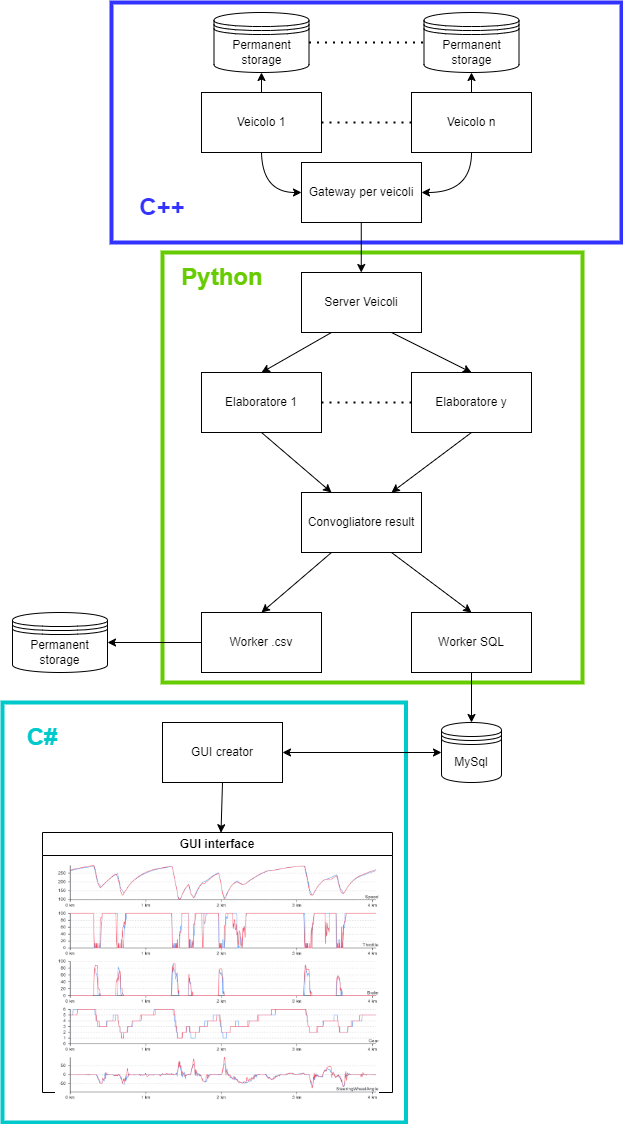
Il secondo si preoccuperà di creare e inviare le query MySQL a un database, in modo da rendere più semplice l’analisi dei dati con computer diversi.

**C# (riquadro in azzurro):**

Il linguaggio in questione verrà utilizzato per visualizzare i dati telemetrici del singolo veicolo o un quadro generale del parco veicoli.

Ciò è fatto tramite una GUI che permette all’utente di scegliere i dati di proprio interesse così da visualizzare solo i dati di proprio interesse.

Sarà possibile visualizzare medie, massimi e minimi dei valori di un singolo veicolo in un intervallo di tempo definito dall’utente, oppure consentire la visualizzazione dei dati in tempo reale."



**Descrizione pragmatica applicativo**

1. **Modulo veicolo(1,n) (C++):**Vi è un modulo veicolo, di fatto runnato n volte che:   
    a) deve creare un file csv proprio su cui inserisce dati  
    b) deve accedere all’area critica (modulo gateway) e inviare i propri dati.
2. **Modulo Gateway per veicoli (C++):** Il gateway tramite socket invia allo scraper gateway in python i dati.
3. **Modulo Server veicoli (Python):** Lo scraper prende i dati inviati dalla socket e inoltra i dati di ogni macchina a uno dei processi elaboratori disponibili.
4. **Modulo Elaboratore (1,n) (Python):**L’elaboratore, runnato n volte elabora i dati di una singola macchina arrivati e accedere all’area critica per poterli inoltrare al modulo convogliatore result
5. **Modulo convogliatore risultati (Python):** Il convogliatore dei risultati quando riceve dei dati li invia al worker csv e al worker MySQL. Dal punto di vista pratico crea un’istanza delle classi worker MySQL e worker csv che inseriscono i dati nei due formati.
6. **Modulo worker csv (Python):** Il worker csv creare un file con id unico relativo ai dati di quel mezzo.
7. **Modulo worker MySQL (Python):** Il worker MySQL inserisce nel database relazione i dati del veicolo.
8. **Modulo GUI(C#):** Il modulo GUI di C# mostra all’utente una basilare interfaccia grafica in cui l’utente sceglie cosa visualizzare. Il modulo prende i dati dal database in base alla richiesta utente e crea un’interfaccia specializzata che aggiorni costantemente i dati cosi che se vengono inseriti altri dati che rispettano i vincoli selezionati dall’utente, essi saranno aggiunti al grafico.

**Modulo C++**

**Immagine che contiene testo, diagramma, schermata, Carattere

Descrizione generata automaticamente**Il modulo c++ viene utilizzato per simulare il servizio di telemetria remota, dove ogni veicolo condivide lo stato di alcuni dei suoi sistemi a bordo, con un endpoint (Gateway per veicoli). La soluzione scelta per simulare questo comportamento, è quella di creare un unico applicativo c++ che, tramite l’utilizzo di classi, oggetti e thread, gestisca ogni elemento dello schema in blu.

Immagine che contiene cerchio, disegno, schizzo, diagramma

Descrizione generata automaticamente

Per comprendere meglio l’albero dei processi è possibile osservare il grafico di sopra. Il main ha il solo compito di creare e avviare il thread gateway. Questo, una volta inizializzato l’ambiente di lavoro, si occuperà di creare 6 thread di tipo Veicolo (quelli che produrranno i dati canbus).  
Preambolo per le descrizioni successive: con il termine pacchetto viene inteso un pacchetto dati dello standard automotive canbus, utilizzante gli stessi campi dettati dallo standard con l’aggiunta di una variabile per indicare il tempo (timestamp).

Immagine che contiene testo, schermata, diagramma, Carattere

Descrizione generata automaticamenteLa scelta di utilizzare un thread per gestire l’arrivo dei messaggi dei veicoli è legata alla possibilità di espansione dell’applicazione. Qualora si volesse ampliare il parco dei veicoli da monitorare, è possibile aumentare il numero di endpoint (thread gateway) per l’accumulo dei dati. L’operazione non comporterebbe una consistente riscrittura del codice sorgente, ma solo delle piccole modifiche irrisorie.  
Come prima cosa, il thread gateway prepara l’ambiente di lavoro, inizializzando i buffer di ricezione, i meccanismi di ricezione e invio dati. Successivamente avvia i thread veicolo dandogli un indicazione su “chi sono” (un identificativo) e dove devono condividere i loro dati di utilizzo.

*Qualora l’applicazione venga utilizzata nel mondo reale (e non simulato), non sarebbe presente “creazione dei thread veicoli” ma “apertura server ascolto veicoli”. In questa situazione il gateway non comunicherebbe ai veicoli il loro ID, ma sono loro che lo comunicano.*

Gli ultimi 3 blocchi descrivono il suo comportamento ciclico:

* Controlla se nei buffer sono presenti dei pacchetti. Qualora ce ne siano li legge (e li rimuove). Queste operazioni vengono effettuate con il meccanismo della mutua esclusione dei mutex.
* Formatta i pacchetti in una stringa di testo json
* Invia la stringa di testo formattata al modulo python (mediante l’utilizzo di una socket UDP

Immagine che contiene testo, schermata, diagramma, Carattere

Descrizione generata automaticamente

Ogni thread veicolo basa le sue operazioni su una classe comune, *Vehicle*. Questa contiene tutte le funzioni per permettere la generazione dei pacchetti canbus e l’invio di questi al gateway. La struttura di questi pacchetti è descritta nel foglio di calcolo excel “legenda pacchetti canbus”.  
Nonostante i thread condividano le stesse funzioni, quindi si comportano tutti allo stesso modo, i valori contenuti nei pacchetti non saranno mai uguali (è improbabile che due veicoli generino due pacchetti con lo stesso identico valore di tensione). La generazione dei valori si basa su una base di un veicolo reale, ma con l’aggiunta di una “varianza casuale”.

La prima operazione del thread veicolo, successivamente alla preparazione del suo ambiente e buffer interno, è quella di generare i pacchetti canbus. Questi verranno generati secondo una determinata frequenza (periodo di attesa di generazione definito staticamente). Successivamente alla generazione, i pacchetti verranno memorizzati in un buffer locale, pronti per essere inviati.  
Al termine della routine di generazione, il thread accede al buffer locale alla ricerca di pacchetti. Nel caso li trovi, questo li scriverà nel buffer del gateway (utilizzando il concetto di mutua esclusione di quest’area critica, mediante il meccanismo dei mutex).

**Modulo Python**

Il modulo Python ha come funzione principale quella di recuperare i dati inviati dal modulo C++ nella socket e dopo aver distribuito i dati a più thread in parallelo (gli elaboratori), essi vengono elaborati e inseriti in un database relazionale SQL e in uno storage locale tramite files csv associati ad un veicolo.

La condivisione dei dati tra processi viene effettuata tramite l’utilizzo di un oggetto Manager, che permette di creare e condividere facilmente liste di mutex e liste di liste.

Nel dettaglio si può dividere il modulo Python completo nei seguenti sottomoduli e strutture:

1. **Server Veicoli:** Modulo che si occupa di prendere i dati dalla socket e distribuire i dati in modalità round robin ai thread elaboratore i dati.
2. I dati vengono inseriti in una coda della classe elaboratore, che poi li prende dalla coda e li elabora. Di conseguenza è necessario usare un meccanismo di accesso al dato così da evitare una race condition con eventuale perdita o corruzione dei dati. Pertanto l’elaboratore e il server veicoli sono in competizione per il lock della coda dell’elaboratore.
3. **Elaboratore:** Prova a prendere dati dalla propria coda e se presenti prova a prenderli e li elabora. Dopo averli elaborati sono inviati all’area critica, risorsa a cui concorrono e dunque provano ad accedere tutti gli elaboratori per scrivere i dati una volta elaborati.
4. **Convogliatore Result:** Modulo che in while true prova ad accedere all’area critica per verificare se contenga dati inseriti dagli elaboratori. Se dopo aver acquisito il lock dovessero esserci dati, essi verranno presi (ergo eliminati dalla coda) e inseriti nel file csv e nel database relazionale MySQL.
5. **WorkerSql:** Classe che inserisce un dato nel database.
6. **WorkerCsv:** Classe che inserisce nel file Csv associato ad un veicolo i dati. Se tale file non esiste perché i dati sono di un nuovo veicolo non ancora presente nello storage, allora il file sarà creato e nominato in base all’identificativo del veicolo in questione.

**Modulo C#**

Il Modulo C# si occupa di prendere i dati dal database relazione MySql e mostrarli graficamente all’utente con gli eventuali cambiamenti in tempo reale.

Per poter eseguire il modulo è necessaria l’installazione tramite NuGet MySQLData e LiveCharts.

Dopo aver eseguito la soluzione GUI l’utente avrà davanti un primo form (**FirstPage.cs**) che gli da il benvenuto e gli fa scegliere se proseguire o chiudere l’applicazione. Una volta selezionata l’opzione di proseguire avrà davanti un nuovo form (**VisualizzaVeicolo.cs**) dove potrà scegliere di quale veicolo visualizzare i dati, la finestra temporale per cui vuole visualizzare i dati, la lista dei campi di cui può visualizzare i dati e infine ogni quanti secondi vuole che i dati vengano aggiornati nel grafico.

La lista dei veicoli tra i quali l’utente può scegliere di visualizzare i dati è presa dal database MySql, naturalmente tali sono inseriti dal modulo Python.

Una volta selezionati veicolo, campo, finestra temporale e refresh rate del grafico, se sono presenti per tale veicolo dati visualizzabili si passerà all’ultimo form (**GraficoLive.cs**) contente il grafico real time.

Il grafico, un carthesian Chart creato usando la libreria WinForms è costruito dinamicamente tramite un campo values, una classe ChartValues contente Observable Points, di fatto costituiti da 2 doubles.

A questo punto si fa una query al database e si prendono i dati inserendoli nel campo values cosi che il grafico venga inizializzato con i dati. Di questi dati si fanno l’operazione di media, calcolo massimo e calcolo del minimo e verranno visualizzati a schermo sotto il grafico cartesiano. Se tali dati sono superflui per l’utente vi è un apposito button da cliccare per nasconderne la visualizzazione.

Dopo la prima inizializzazione tramite un evento timer con frequenza pari a 1/refresh rate(scelto dall’utente) viene chiamata una funzione chiamata Update\_grafico che di fatto prende i dati nuovamente dal database e ricrea il grafico, cosi che, se nel frattempo sono stati inseriti nuovi dati verranno visualizzati dall’utente.

Unica piccola nota stonata del grafico è che essendo il sistema basato sui timestamp è stato necessario convertirli in double per essere aggiunti al grafico, pertanto un timestamp come 2024-13-12 08:00:00 verrà mostrato come 20.241.312.080.000. Cosi sacrificando un po' di leggibilità istantanea si riesce a dare un’idea dei tempi all’utente e soprattutto graficare correttamente i dati in formato (timestamp=>asse x, valore=>asse y).