UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche, Informatiche e Matematiche

Corso di laurea in Informatica

Riprogettazione e ottimizzazione di software per l'interfaccia uomo-macchina nell'ambito dell'automazione industriale

Relatore:
Giacomo Cabri

Candidato:
Enrico Marras

Anno Accademico 2022-2023

Indice

Introduzione

In questo elaborato si vuole descrivere l'attività di sviluppo effettuata durante il tirocinio curriculare, mettendo in evidenza tutto il processo che ha portato alla produzione di una soluzione conforme alle necessità evidenziate.

Il tirocinio in questione si è focalizzato sullo sviluppo di una **Human Machine Interface**, in breve HMI, ovvero un software che permette a un utente di comunicare con una macchina, un programma o un sistema, attraverso un'interfaccia grafica[1], comunemente utilizzato in ambito industriale.

L'automazione industriale si occupa dell'impiego coordinato di soluzioni tecnologiche allo scopo di ridurre la necessità dell'intervento umano[2], specialmente per quanto riguarda operazioni ripetitive, complesse o pericolose.

Nonostante questo campo abbia avuto una significativa evoluzione grazie a metodologie come l'industria 4.0, durante il tirocinio si è constatato come alcuni di questi settori siano rimasti più legati a un paradigma di lavoro antiquato, che predilige una maggiore dipendenza dal lavoro manuale e una carenza di tracciabilità e/o sicurezze.

Per questo motivo, tramite l'analisi delle problematiche e delle necessità del caso di studio, sono state adottate tecnologie e paradigmi moderni che hanno permesso il miglioramento dell'efficienza operativa anche in contesti precedentemente identificati come critici.

Nel primo capitolo verrà contestualizzata la situazione preesistente, fornendo informazioni sulle caratteristiche software, hardware e sul protocollo del prodotto originale.

Il secondo capitolo sarà finalizzato alla fase di analisi, in cui verranno evidenziati e analizzati i vari problemi riscontrati.

Con il terzo capitolo si passerà alla fase di progettazione, nella quale si mostreranno le principali soluzioni impiegate sia dal punto di vista delle tecnologie scelte, che dall'effettiva riprogettazione della logica interna.

Successivamente, nel quarto capitolo verrà analizzata l'effettiva implementazione delle funzionalità presentate nel terzo capitolo, e infine, nel quinto capitolo, verranno analizzate le opportunità di ottimizzazione relative all'implementazione.

---

[1] What is an HMI, Copadata, consultato il 28 Settembre 2023, https://www.copadata.com/en/product/zenon-software-platform-for-industrial-automation-energy-automation/visualization-control/what-is-hmi/

[2] Automazione Industriale, Treccani, consultato il 29 Settembre 2023, https://www.treccani.it/enciclopedia/automazione-industriale_%28Enciclopedia-Italiana%29/

Capitolo 1 – Contesto di sviluppo

Il tirocinio è stato svolto presso la DOT S.n.c., una realtà del territorio modenese che da quasi trent'anni si occupa principalmente di progettazione e produzione di prodotti a servizio dell'industria dell'automazione.

Durante il tirocinio è stato preso in analisi "MultiBench", un prodotto consolidato per la gestione di una HMI attualmente utilizzata in più macchinari e sviluppato in Visual Basic 6.0.

È importante notare come questo software sia completamente funzionante secondo le necessità per le quali è stato originariamente progettato, e non mostri problematiche evidenti che ne comprometterebbero il conseguimento delle mansioni.

Siccome MultiBench presenta più versioni, la fase di analisi si è concentrata principalmente sulla versione installata sul macchinario a disposizione durante il periodo di tirocinio, ovvero la prima sviluppata.

1.1 Caratteristiche Hardware

Il macchinario oggetto di analisi è una stazione di assemblaggio dei componenti di motori, che permette di svolgere diversi compiti al fine di personalizzare la fase dell'assemblaggio in base ai requisiti tecnici, il tutto comandabile da un operatore.

Le funzionalità sopra citate vengono comandate da quattro controllori programmabili per l'azionamento di motori, due inverter e uno stepper. Essi hanno il compito di monitorare e salvare nei propri registri interni stati riportati dai sensori del macchinario e, all'evenienza, anche di pilotare le sue componenti.

Si noti come l'attività di riprogettazione si è focalizzata esclusivamente sulla parte software che si interfaccia direttamente con i controllori, e pertanto non sono state necessarie modifiche sulla loro programmazione in quanto completamente funzionanti.

Tra i principali componenti pilotabili dai controllori vi sono quattro assi, ovvero gli effettivi protagonisti dell'attività di assemblaggio dei motori.

Per consentire un collegamento tra i vari controllori al **livello fisico** dello *stack* ISO/OSI[3], è stato usato lo standard per le comunicazioni seriali **RS-485** a **due fili**. Questo implica il suo utilizzo in modalità **half-duplex**, la quale prevede che le comunicazioni possano

[3] ISO/IEC 7498-1:1994, Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model, 7.7 Physical Layer

viaggiare indipendentemente dalla direzione, ma che solo un dispositivo alla volta possa trasmettere informazioni.

In accordanza a quanto consigliato dallo standard RS-485[4], i vari controllori, referenziati con il nome di "*Drive*", "*Inverter*" e "*Stepper*", sono interconnessi attraverso una *daisy-chain* o più comunemente chiamata struttura a bus (figura 1).

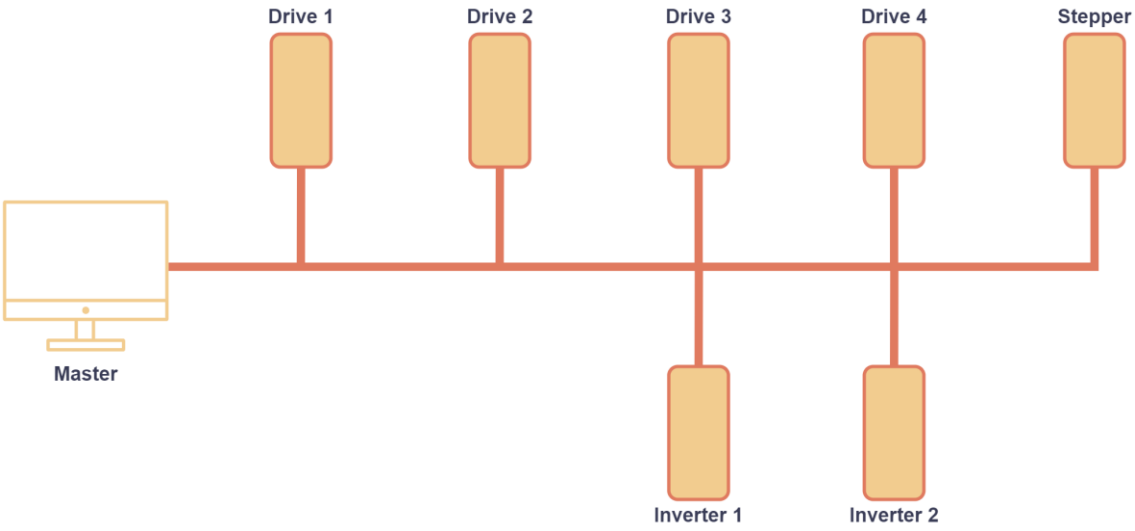

Figura 1 – Struttura delle connessioni tra i vari dispositivi

1.2 Architettura di comunicazione

Per quanto riguarda la comunicazione, i vari controllori sono stati predisposti per lavorare in un'architettura di tipo **master-slave**, nella quale è sempre presente:

- Un **master.**
- Uno o più **slave**.

In questo tipo di architettura, "ogni scambio di informazioni è originato dal master, il quale invia […] sul bus una particolare richiesta"[5].

Gli slave "sono normalmente in ricezione e ascoltano le richieste del master. Solo lo specifico slave interrogato cattura le informazioni inviate dal master […] e risponde inviando a sua volta le proprie informazioni sulla rete"[6].

In questo tipo di predisposizione, il PC che esegue l'HMI prende il ruolo di *master*, mentre tutti gli altri controllori saranno gli *slave*.

[4] The RS-485 Design Guide, TI, consultato il 10 Settembre 2023, pagina 1-2, https://www.ti.com/lit/an/slla272d/slla272d.pdf
[5] Protocollo Modbus su RS485 – Introduzione, Overdigit, consultato il 9 settembre 2023, pagina 3, https://www.overdigit.com/data/Blog/RS485-Modbus/Protocollo%20Modbus%20su%20RS485.pdf
[6] What is the OSI Model?, Forcepoint, consultato il 9 Settembre 2023, https://www.forcepoint.com/cyber-edu/osi-model/

1.3 Caratteristiche Software

Tra le principali funzionalità di MultiBench per questo macchinario sono presenti:

- Gestione dell'autenticazione su diversi profili utente in base alla tipologia di utilizzatore (ospite, operaio, manutentore, …).
- Continuo monitoraggio degli input/output del sistema e della comunicazione software-hardware.
- Visualizzazione del log eventi e feedback degli allarmi in tempo reale.
- Attuazione di movimenti manuali comandati da una figura umana.
- Configurazione dei parametri globali di riferimento.
- Creazione, salvataggio, caricamento ed esecuzione di programmi automatici di lavoro.
- Traduzione del testo delle componenti grafiche in più lingue.

1.4 Protocollo di comunicazione

Al fine di gestire tutte le comunicazioni tra i vari controllori è stato adottato lo standard comunicativo del protocollo **Modbus**. Seppur sia stato pubblicato originariamente nel 1979, rimane ad oggi uno dei protocolli di comunicazione più usati per connettere dispositivi elettronici industriali.

Il protocollo Modbus ha il compito di gestire le comunicazioni su più *layer* dello *stack* ISO/OSI.

Al livello ***Data Link***[7], prevede tutte le specifiche relative allo scambio dei *frame*[8] (sequenze di byte) tra un dispositivo e l'altro, le quali comprendono:

- Invio dei dati sul bus.
- Controllo delle temporizzazioni.
- Controllo degli errori mediante *checksum*.

Al livello **Applicativo**, si occupa della codifica delle possibili richieste del *master* e le relative risposte degli slave all'interno dei *frame*.

Questo livello permette inoltre di interagire con le varie applicazioni in esecuzione sui dispositivi.

[7] Introduction to Modbus Serial and Modbus TCP, Ccontrols, pagina 1, consultato il 29 Settembre 2023, https://www.ccontrols.com/pdf/Extv9n5.pdf

[8] Unità informativa del livello 2 dello stack ISO/OSI.

1.4.1 Modbus RTU

Modbus è disponibile in tante varianti comunicative, tra cui *RTU*, *ASCII* e *TCP/IP*.

Per questo progetto è stata utilizzata la modalità **RTU** o *Remote Terminal Unit*[9].

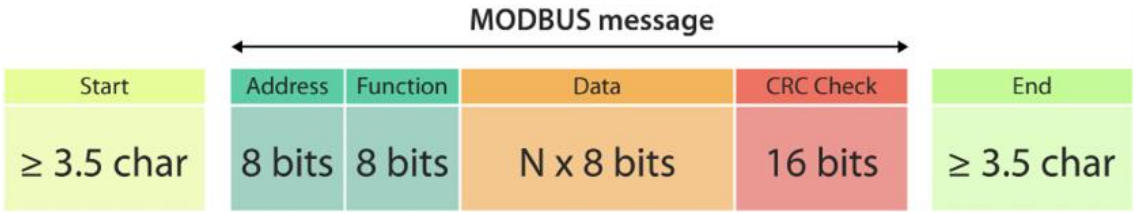

Figura 2 – Struttra del frame Modbus in modalità RTU

Come si evince dalla figura 2, il messaggio Modbus è suddiviso in diversi campi[10]:

- *Address* indica l'indirizzo dello slave destinatario del messaggio e può variare da 1 a 247. Il valore 0 è riservato per messaggi in broadcast dove non verranno effettuate risposte.
- *Function* permette al destinatario di capire quale azione dovrà svolgere e al mittente di capire quale azione è stata svolta dallo *slave*. In seguito verranno approfondite nel contesto di questo caso di studio.
- *Data* contiene informazioni addizionali che lo slave deve sapere per portare a termine la richiesta indicata tramite il *function code*.
- *CRC*, o *Cyclic Redundancy Check* è un codice di controllo effettuato su tutto il frame per verificare la sua integrità dopo la trasmissione. Viene inizialmente calcolato dal master e successivamente dallo slave destinatario e, qualora il valore *CRC* calcolato dallo slave differisca da quello indicato dal master, il frame verrà scartato.
- Analogamente alla logica degli *Internal Frame Gap*, si ricorre a utilizzare i campi *start* ed *end* per cercare di fornire più garanzie sulla corretta ricezione del pacchetto prima dell'invio del successivo.
- Opzionalmente, è possibile anche aggiungere un bit di parità alla fine del messaggio che rappresenta un'ulteriore garanzia sullo stato dell'integrità del *frame*.

[9] Modbus RTU communication guide, Virtual-serial-port, consultato il 29 settembre 2023, https://www.virtual-serial-port.org/articles/modbus-rtu-guide/

[10] Modbus Networking Guide, libelium, consultato il 29 settembre 2023, https://development.libelium.com/modbus_networking_guide/introduction

Per quanto concerne la codifica dei dati, Modbus utilizza una rappresentazione '*Big-Endian*' per indirizzi e per l'effettivo contenuto informativo (ad eccezione del controllo CRC, che usa una codifica '*Little-Endian*'). Questo implica che all'invio di una quantità numerica superiore a un singolo byte, verrà inviato per primo il byte più significativo[11].
È infine importante marcare che per garantire la corretta comunicazione tra i dispositivi è necessario assicurarsi che mittente e destinatario possano comunicare alla stessa velocità e che utilizzino la stessa struttura del messaggio Modbus.

[11] Unidrive M700 / M701 / M702 Guida dell'utente al controllo Versione numero: 2, 9.1.5 Codifica dei dati, p. 122

Capitolo 2 – Fase di analisi

Durante lo svolgimento del tirocinio, è stata posta molta importanza alla fase di progettazione in quanto è stato necessario individuare preventivamente le maggiori problematiche prima di passare all'effettiva implementazione della soluzione da applicare.

Nel corso di questo capitolo, verranno esaminate in dettaglio le principali criticità riscontrate e le limitazioni progettuali imposte.

2.1 Aggiornamento componenti grafiche

Tra le principali criticità d'uso identificate, una delle più significative ha riguardato la lentezza generale dell'interfaccia utente.

Questa ritardo si è presentato sia per le operazioni che richiedevano l'aggiornamento di elementi dell'interfaccia, ma anche, seppur in maniera minore, per tutte le interazioni tra l'utente e la macchina, dove vi era la necessità di cambiare lo stato interno di registri attraverso l'HMI.

L'esempio più rilevante per questa tematica riguarda la parte superiore dell'interfaccia di MultiBench mostrata nella figura 3, la quale presenta una serie di indicatori che riportano misure inerenti a stati interni degli assi.

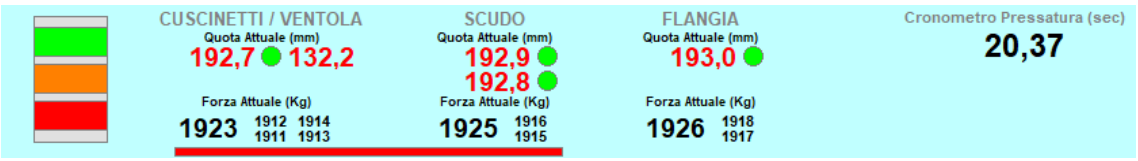

Figura 3 – Pannello superiore dell'interfaccia di MultiBench

Il ritardo dell'aggiornamento di queste misurazioni rappresenta un problema vista la loro importanza nelle scelte decisive dell'operatore, ma anche per il corretto funzionamento della macchina stessa.

Una delle principali cause di questo ritardo di aggiornamento è dovuta all'inefficienza del *polling* per il controllo dei vari *widget*, ovvero "all'attività di campionamento attivo degli stati di un dispositivo esterno da parte di un programma client come un'attività sincrona"[12].

[12] Polling (computer science), Wikipedia, consultato il 29 settembre 2023, https://en.wikipedia.org/wiki/Polling_(computer_science)

Seppur rappresenti una soluzione adatta per software prevedibili e di piccole dimensioni, il *polling* non permette di gestire efficacemente la potenza di calcolo[13], talvolta sprecata non potendola fornire su richiesta delle attività che ne hanno bisogno.

2.2 Codice obsoleto

Il codice sorgente di MultiBench si distanza notevolmente dai moderni paradigmi di programmazione in quanto, in primo luogo, non prende spunto da un *design pattern*, presentando una struttura talvolta monolitica e poco modulare.

Questa scarsa modularità la si può notare molto di frequente nelle parti di controllo del programma relative al *polling*, dove spesso molti stati di registri sono gestiti attraverso sequenze molto lunghe di *if-else* o *switch case*. Seguendo questo paradigma di scrittura, nel caso dell'estensione di una *feature*, implicherebbe un aumento significativo della complessità di comprensione e di manutenzione del codice esistente, con la possibilità di introdurre errori difficili da individuare.

Per questo motivo, risulta evidente la necessità di avere una logica costruttiva che consenta agilmente l'aggiunta di *widget* o di controlli di stato senza l'appesantimento eccessivo del codice.

Seppur supportato già dalla versione 4.0[14], al momento della scrittura del sorgente originale non è stato adottato un paradigma di programmazione a oggetti. Di conseguenza, MultiBench non presenta alcuna delle caratteristiche principali della *OOP*[15], tra cui si ricordano:

- Uso e divisione concettuale in classi.
- Polimorfismo.
- Ereditarietà.
- Incapsulamento.

Al posto di organizzare le funzionalità in classi con metodi e proprietà correlate, è stato originariamente utilizzato un approccio più procedurale che tendeva a complicare la comprensione tra le diverse parti del sistema.

Inoltre, l'assenza di ereditarietà e polimorfismo limitava la capacità di estendere e personalizzare le componenti grafiche per adattarle alle proprie esigenze.

[13] Polling, Teach-ict, consultato il 30 settembre 2023, https://teach-ict.com/2016/A_Level_Computing/OCR_H446/1_2_software/121_operating_systems/interrupts/miniweb/pg2.php

[14] Programming Microsoft Visual Basic 6, Chapter 6 – Classes and Objects, Visualbasicbooks, consultato il 25 settembre 2023, https://www.visualbasicbooks.com/progVB6samplepg1.html

[15] Object-oriented programming

In aggiunta a quelli che verranno trattati nel capitolo 2.3, la mancanza di incapsulamento poteva comportare problemi di sicurezza e la possibilità di accessi non autorizzati ad attributi sensibili.

2.3 Sicurezza

Dal punto di vista meccanico, il macchinario preso in oggetto presenta molteplici accorgimenti sul punto di vista della sicurezza, come l'uso di una barriera protettiva, molteplici interruttori di abilitazione, e vari sistemi secondari che forniscono garanzie anche in caso di fallimento dei primari.

Evitando di focalizzarsi in questa sede sulle caratteristiche fisiche o sulle vulnerabilità delle più vecchie versioni di Visual Basic, sarebbe il caso di porre l'attenzione sull'assenza di occultazione dei dati sensibili degli utenti da parte di MultiBench.

Poiché è stato necessario avere un sistema che garantisse l'utilizzo di specifiche funzionalità a precisi utenti, era stato originariamente creato un file contenente le loro informazioni necessarie.

Il file in questione era completamente visibile in chiaro, esponendo i dati degli utenti a potenziali rischi di accesso non autorizzato e in maggior modo inadatto a un paradigma di progetto sicuro per future interazioni con l'industria 4.0.

2.4 Problematiche secondarie

Oltre a quelle precedentemente elencate, sono state identificate altre problematiche, sicuramente minori, ma che potrebbero comunque fornire spazio di miglioramento. Tra queste:

- Carenza di una documentazione esaustiva e talvolta precisa, che ha rappresentato una problematica per la manutenzione del codice, nonché una complessità maggiore nella riprogettazione.
- Programma originale legato unicamente all'ambiente Windows, in quanto utilizzatore del *runtime environment* necessario all'esecuzione di codice Visual Basic.
- Assenza di un vero paradigma di programmazione per *multithreading* che avrebbe permesso di sfruttare più efficientemente l'hardware messo a disposizione.
- Uso di formati inadatti per i file di riferimento per la traduzione delle componenti grafiche, dove ogni stringa è referenziata solo dall'indice di riga.

2.5 Vincoli di sviluppo

Nel corso del progetto, sono stati imposti determinati vincoli che hanno influenzato le scelte e le direzioni prese nello sviluppo del sistema.

Anche se non si trattava di un obbligo progettuale formale, è stato necessario rispettare una limitazione temporale ben definita per l'intero sviluppo del software. La durata del progetto doveva essere rigorosamente contenuta entro il periodo di tirocinio, che corrispondeva a un totale di 375 ore.

Nonostante in fase progettuale si fosse discusso dell'idea di svincolare il prodotto da un preciso sistema operativo, l'azienda ha ritenuto più opportuno continuare lo sviluppo mirato alla distribuzione su ambiente Windows, in quanto più familiare e più adatto alle loro necessità. Ciononostante, è rimasto un obiettivo importante da tenere in considerazione nella scelta delle tecnologie impiegate, per non limitare future adozioni di altri sistemi operativi.

In aggiunta all'ambiente Windows, il prodotto finale avrebbe dovuto disporre anche di un *installer* che semplificasse l'installazione del software e che fornisse l'accesso al programma da un singolo file eseguibile (*.exe*).

Infine, è stato vincolante evitare l'utilizzo di *framework* o librerie che richiedessero l'acquisto di licenze per uso professionale, ed è anche stato ritenuto fondamentale utilizzare tecnologie consolidate, per minimizzare la possibilità che il loro supporto cessasse nell'immediato futuro.

Capitolo 3 – Fase di progettazione

In questo capitolo verranno discusse in primo luogo le tecnologie scelte e successivamente, seppur a un livello più astratto dell'effettiva implementazione, le soluzioni metodologiche usate per risolvere i problemi precedentemente discussi.

3.1 Tecnologie impiegate

Visto il contesto di sviluppo in parte di basso livello, si era inizialmente valutato il linguaggio C++, in quanto avrebbe garantito un'ottima gestione specifica della comunicazione, delle risorse e avrebbe permesso lo sviluppo non vincolato a un sistema operativo. Tuttavia, durante le prime prove, è stato notato come l'organizzazione e l'utilizzo di diverse librerie grafiche di C++ nell'ambiente Windows fosse complesso e poco agevole. Per questo motivo l'attenzione si è spostata sul linguaggio Python.

Seppur più lento per via della sua natura, Python avrebbe consentito una gestione più semplificata delle librerie grafiche e, grazie alla sua maggiore astrazione, avrebbe permesso di completare con più garanzie le funzionalità precedentemente discusse entro il periodo di tirocinio.

Nativamente, Python non fornisce alcun supporto diretto per la comunicazione con il protocollo Modbus, tuttavia sono presenti molti moduli esterni alla libreria standard.

La scelta per questo ambito è ricaduta su MinimalModbus[16], un modulo *open-source* che implementa il protocollo Modbus e permette la comunicazione tra un computer (*master*) e gli strumenti (*slaves*)[17].

Questo modulo fornisce diverse *API*[18] di basso livello che consentono una vasta gamma di operazioni di comunicazione, incluso il controllo dei parametri ad essi associati.

Infine, MinimalModbus soddisfa i vincoli di sviluppo posti dall'azienda poiché pubblicato attraverso la licenza Apache 2.0 che, nel rispetto delle sue linee guida, permette l'uso commerciale e la distribuzione[19].

Per quanto riguarda la libreria grafica è stata scelta Tkinter, un *framework* attualmente incluso nella libreria standard di Python di relativamente facile utilizzo.

[16] MinimalModbus, Github, consultato il 3 ottobre 2023, https://github.com/pyhys/minimalmodbus/
[17] Features, MinimalModbus, consultato il 3 ottobre 2023, https://minimalmodbus.readthedocs.io/en/stable/readme.html
[18] Application Program Interface
[19] Apache License, Version 2.0, Apache, consultato il 3 ottobre 2023, https://www.apache.org/licenses/LICENSE-2.0/

La scelta è ricaduta su Tkinter principalmente per via della sua stabilità, in quanto è una libreria grafica ampiamente utilizzata, leggera, documentata e già presente da diversi anni.

Benché Tkinter possa apparire datato dal punto di vista dell'aspetto grafico, ha comunque soddisfatto i requisiti del progetto, poiché non erano imposti vincoli riguardanti l'aspetto estetico dell'interfaccia utente.

In linea con MinimalModbus, Tkinter offre la possibilità d'utilizzo commerciale e la distribuzione dei prodotti che ne fanno uso come dipendenza, in quanto rilasciato con la licenza Tcl/Tk[20].

3.2 Priorità comunicative

Come menzionato nel capitolo 2.1, durante la fase di analisi era stata evidenziata la necessità di migliorare la responsività generale del programma, con particolare riferimento alle misurazioni indicate nella figura 3. Prendendo spunto dalla pagina "DIAGNOSI I/O #1" in figura 4, una delle interfacce di controllo più numerosa dal punto di vista delle trasmissioni, è stata scelta una logica di comunicazione con priorità.

Figura 4 – Interfaccia "DIAGNOSI I/O #1 di MultiBench

La scelta è stata possibile per via di una minore necessità di aggiornamento di tutti i *widget* contenuti nella pagina che, in contrapposizione alle misurazioni presenti nel pannello superiore della figura 3, non avrebbero richiesto necessariamente un continuo aggiornamento tempestivo.

[20] Tcl/Tk License Terms, Tcl.tk, consultato il 4 ottobre 2023, https://www.tcl.tk/software/tcltk/license.html

Questo significa che al posto di effettuare continue misurazioni su tutti i registri interessati dai *widget*, si sarebbero potuti effettuare più spesso campionamenti su quelli più "importanti" e più di rado su quelli secondari.
Conseguentemente a questa decisione, sono state stabilite due tipologie di priorità comunicative:

- **Alta priorità**: per tutte le comunicazioni inerenti a *widget* che necessitano di essere aggiornati tempestivamente, e per tutti i cambi di stato che l'utente decide di apportare ai registri interni degli slave per mezzo di scritture.
- **Bassa priorità**: per tutte le comunicazioni inerenti a *widget* che non necessitano di un aggiornamento tempestivo, e possono essere aggiornati in un secondo momento rispetto alle comunicazioni ad alta priorità, senza pregiudicare il funzionamento del macchinario.

3.3 Task

Al fine di organizzare al meglio ogni singola comunicazione, e quindi anche le priorità definite nel capitolo 3.2, è stata creata un'unità informativa composta da tutti i parametri riguardanti una singola trasmissione, il Task.
Internamente al Task, saranno quindi contenuti:

- **Priorità**: alta o bassa.
- **Porta**: identificatore della porta seriale (ad esempio "COM1", "COM2", … , "COM256" su Windows).
- **Slave-id**: Identificatore univoco dello slave.
- **Codice funzione**: codice dell'operazione Modbus da eseguire.
- **Indirizzo del registro**: indirizzo iniziale dell'operazione del Task.
- **Bytesize**: lunghezza del messaggio.
- **Baud-rate**: determina la velocità della comunicazione sul canale trasmissivo.
- **Bit di stop**: numero di bit di stop alla fine di ogni messaggio Modbus.
- **Bit di parità**: usato per determinare se i dati trasmessi sono stati ricevuti correttamente.
- **Timeout**: secondi dopo i quali il destinatario della comunicazione è considerato irraggiungibile.
- **Periodicità**: indicatore booleano per differenziare i Task da eseguire una singola volta da quelli periodici.
- **Register count**: numero di registri sul quale operare.

Questo tipo di organizzazione per le comunicazioni consente di avere oggetti più facili da maneggiare, in quanto una volta costruiti contengono già tutte le informazioni necessarie a un determinato compito, e pertanto non devono essere istanziati a ogni uso. Inoltre, agevolano il processo di ottimizzazione in quanto diventa più facile costruire e gestire dei Task fatti appositamente per un dispositivo rispetto a un altro.
Si prenda come esempio la velocità trasmissiva. È noto che in molti contesti applicativi si tende a sfruttare al massimo la velocità di trasmissione disponibile per ridurre al minimo il tempo necessario alle comunicazioni. In tal senso, ci si aspetta che tutti i dispositivi operino al massimo delle loro capacità di trasmissione, tuttavia ciò non implica che si possa sempre comunicare allo stesso modo. È questo il caso dei due *Inverter*, che possono trasmettere al più con un *baud-rate* di 38400 bit/secondo[21], ben lontano dai 115200 bit/secondo[22] di tutti gli altri dispositivi.

3.4 Coda di priorità

Ora che si è definita un'unità informativa contenente i compiti da eseguire, è necessario definire una struttura dati che permetta la loro gestione. A questo scopo è stata scelta una coda di priorità, la quale ha i seguenti compiti:

- Permettere l'inserimento dei vari Task, tra i quali:
  - Task **periodici** con **priorità normale**
  - Task **periodici** ad **alta priorità**
  - Task **non periodici** ad **alta priorità**
- Gestire l'accodamento dei vari Task periodici che sono stati eseguiti o alternativamente, la rimozione dalla coda dei Task non periodici eseguiti.

Si noti come ai fini del progetto, non è stato ritenuto fondamentale discriminare anche tra i Task non periodici a bassa priorità, in quanto non rappresentano una tipologia di compito necessario alle necessità progettuali.
Al fine di permettere a Task diversi di eseguire più o meno frequentemente si definiscono i seguenti puntatori della coda:

- Un **puntatore all'ultimo elemento della coda,** o fondo, per gli inserimenti dei Task periodici con priorità normale.

[21] Commander SK Advanced User Guide – Issue Number: 8, consultato il 5 ottobre 2023, https://www.nidec-netherlands.nl/media/2125-frequentieregelaars-commander-sk-advanced-user-guide-en-iss10-0472-0001-10.pdf
[22] Unidrive M700 / M701 / M702 Guida dell'utente al controllo Versione numero: 2, 9.1.5 Codifica dei dati, pagina 121

- Un **puntatore al primo elemento della coda** o cima.
- Un **puntatore al secondo elemento con priorità normale a partire dalla cima della coda**. Nel caso in cui non ce ne siano o ce ne sia solo uno, punterà al fondo della coda. Quest'ultimo puntatore è utilizzato per gli inserimenti dei Task periodici ad alta priorità.

La priorità di esecuzione dei singoli Task è determinata dalla loro posizione all'interno della coda. In particolare, i Task più vicini alla cima nella coda saranno eseguiti prima di quelli più lontani.

Inoltre, l'effettiva priorità di un Task non garantisce alcun vantaggio di esecuzione tutt'ora che il compito è stato inserito nella coda, bensì viene utilizzato per **fornire una posizione più o meno vantaggiosa al momento del reinserimento**.

Nella figura 5 si può osservare un generico stato della coda e dei suoi puntatori.

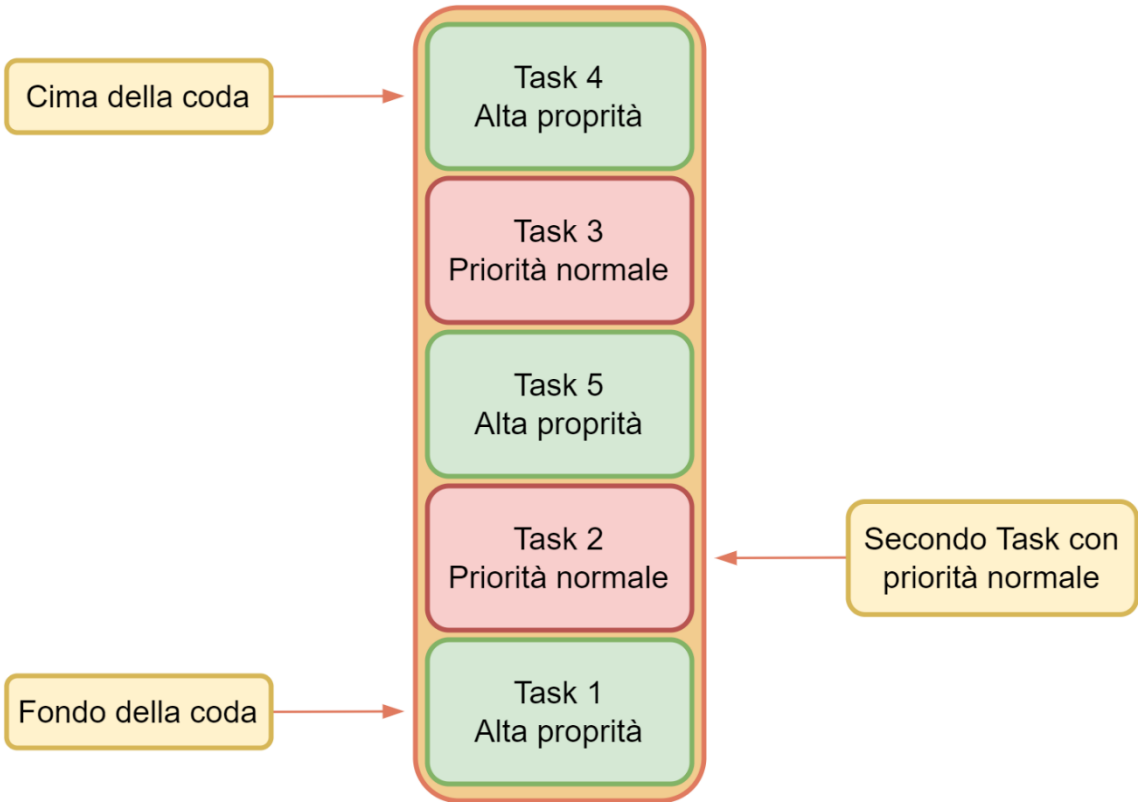

Figura 5 - Generico stato della coda e dei suoi riferimenti

Nei capitoli 3.4.1 e 3.4.2 verranno analizzate le metodologie di inserimento e di vita di ogni tipologia di Task, utilizzando per brevità la notazione $\{Task\}:\{Priorità\}$, con $Task \in \mathbb{N}$ e $Priorità \in \{\alpha, \beta, \gamma\}$, rispettivamente per i Task periodici ad alta priorità, periodici a priorità normale e non periodici ad alta priorità.

3.4.1 Comportamento dei Task periodici

I Task periodici a priorità normale seguono due regole:

- In fase di inserimento verranno accodati, ovvero verranno inseriti nella posizione più lontana dalla cima.
- Dopo la loro esecuzione, verranno accodati in fondo alla coda.

Il comportamento differisce per i Task periodici ad alta priorità, i quali:

- Al momento dell'inserimento prenderanno la posizione del secondo Task con priorità normale dalla cima della coda.
- Dopo la loro esecuzione, verranno nuovamente inseriti all'interno della coda, nella posizione del secondo Task con priorità normale più vicino alla cima.

Come esempio, si consideri la figura 6, nella quale sono stati delineati i seguenti stati:

- 6.1: Stato iniziale della coda.
- 6.2: Inserimento nella coda del Task periodico 4 a priorità normale in ultima posizione.
- 6.3: Inserimento nella coda del Task periodico 5 ad alta priorità nella posizione occupata dal task 3.
- 6.4: Esecuzione del Task periodico 1 ad alta priorità e conseguente reinserimento in coda nella posizione occupata dal Task 3.
- 6.5: Esecuzione del Task periodico 2 a priorità normale e conseguente reinserimento nel fondo della coda.
- 6.6: Esecuzione del Task periodico 5 ad alta priorità e conseguente reinserimento in coda nella posizione occupata dal Task 4.

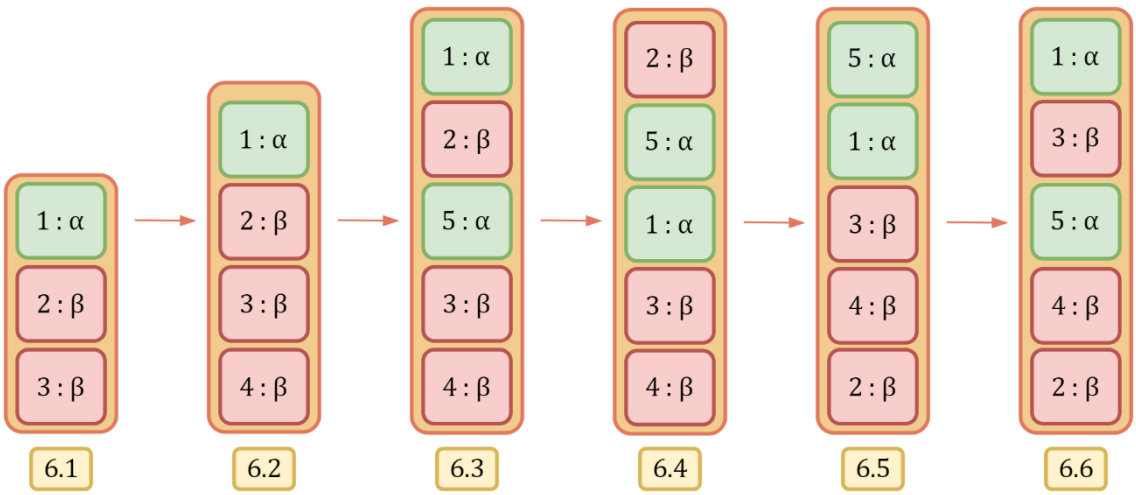

Figura 6 - Inserimento, esecuzione e reinserimento di Task periodici

Si noti come il puntatore di reinserimento per i Task ad alta priorità sia stato scelto appositamente in coincidenza con la posizione del secondo Task a priorità normale dalla cima della coda, per via dei vincoli progettuali di reattività. Tuttavia, questa scelta non preclude la possibilità di valutare eventuali bilanciamenti, in cui le varie priorità di Task competono in minor modo per la loro esecuzione più o meno frequente.
Questa caratteristica permette inoltre di gestire la velocità di convergenza della coda in uno stato "standard", nel quale i Task con priorità più alta si troveranno genericamente più vicini alla cima della coda di quanto siano i Task a priorità minore.
Definendo il **periodo** come il ciclo completo di esecuzione dei Task che riportano la coda allo stato iniziale, in cui tutti i Task sono stati completati almeno una volta, è possibile ottenere informazioni su quali e quanti Task sono stati eseguiti più o meno spesso.
Prendendo come stato di partenza il 6.3 in figura 6, a completamento del periodo si otterrebbero i seguenti risultati:

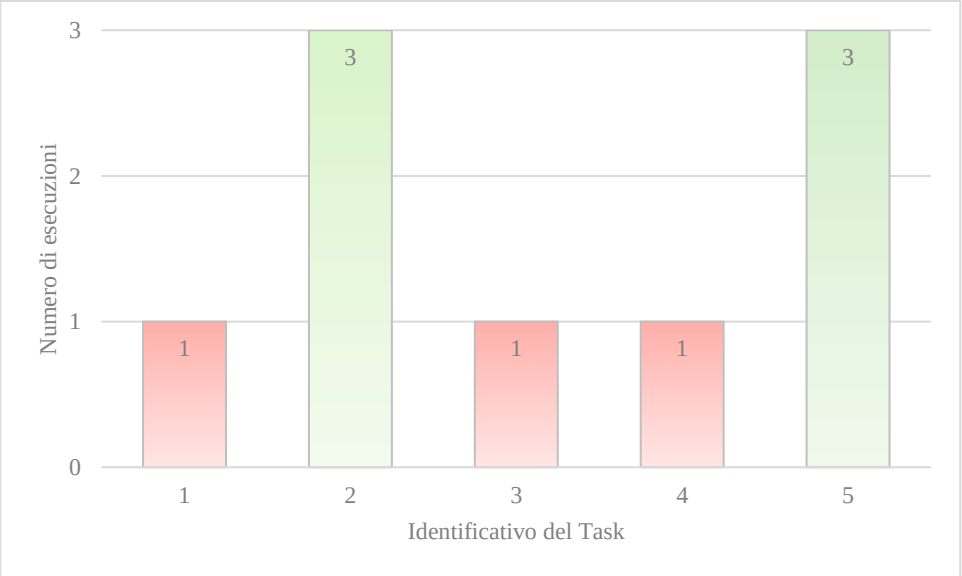

Figura 7 - Totale esecuzioni dei Task dallo stato 6.2 della figura 6

3.4.2 Comportamento di Task non periodici ad alta priorità

Si può pensare ai Task non periodici o Task immediati come quei compiti chiesti dall'utente in un particolare momento che richiedono una e una sola modifica di uno stato interno, come la pressione di un pulsante e il conseguente cambiamento di un valore in uno o più registri. Come suggerito dal nome e in contrapposizione alla controparte periodica, un Task non periodico non viene reinserito nella coda dopo la sua esecuzione. Il loro inserimento iniziale nella coda differisce dai Task periodici, in quanto per motivi di reattività dovranno essere inseriti nel punto più alto possibile della coda.

Per motivazioni legate all'implementazione che verranno ulteriormente elaborate nel capitolo 4, si è preferito un inserimento dei Task immediati nella seconda posizione dalla cima della coda.

L'inserimento differisce inoltre per via del numero di altri Task immediati interni alla coda al momento dell'inserimento. Se la coda contiene n Task non periodici, l'inserimento di un nuovo Task non periodico dovrà avvenire dopo tutti gli altri Task immediati già presenti, ma comunque prima di tutti quelli periodici.

Come esempio, si consideri la figura 8 e i seguenti stati:

- 8.1: Stato iniziale della coda.
- 8.2: Inserimento nella coda del Task immediato 4 nella posizione occupata dal Task 2.
- 8.3: Esecuzione del Task periodico 1 ad alta priorità e conseguente reinserimento in coda nella posizione occupata dal Task 3.
- 8.4: Esecuzione del Task immediato 4 e conseguente rimozione del medesimo dalla coda.
- 8.5: Inserimento nella coda del Task immediato 5 nella posizione occupata dal Task 1.
- 8.6: Inserimento nella coda del Task immediato 6 nella posizione occupata dal Task 1.

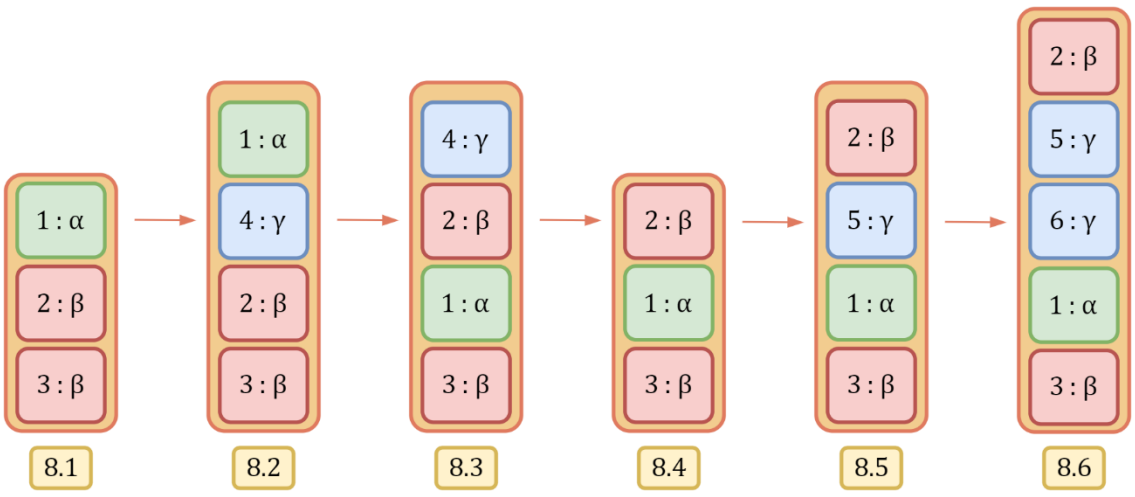

Figura 8 - Inserimento, esecuzione e rimozione di Task non periodici

Capitolo 4 – Strategie implementative

Nel corso di questo capitolo, verranno esaminate le strategie e le tecniche utilizzate per concretizzare i punti delineati in fase di analisi e di progettazione, focalizzandosi sui dettagli di implementazione più rilevanti.

4.1 Design pattern

Con l'aumentare della dimensione e della complessità dei software, emerge la necessità di adottare approcci strutturati ed efficienti per la progettazione e lo sviluppo. Uno di questi è l'utilizzo dei design pattern. Nell'ambito della progettazione di software, un design pattern[23] è una soluzione generale e ripetibile a un problema comune e ricorrente, e il loro utilizzo comporta miglioramenti del codice su diversi aspetti, tra cui:

- Manutenibilità
- Riusabilità
- Comprensibilità
- Scalabilità

Nelle prossime sezioni, verrà evidenziato per quali classi sono stati sfruttati i pattern.

4.2 Classe TaskHandler

La classe *TaskHandler* implementa la coda di priorità discussa nel capitolo 3.4.

Per questa classe si è scelta una struttura minimale con pochi e semplici metodi, dal momento che rappresenta una delle fondamenta di tutto il processo comunicativo ed è estremamente importante che sia il quanto più possibile affidabile.

Al suo interno, sono presenti diversi metodi primitivi, tra cui:

- *get_high_priority_requeuer_index*: per il recupero dell'indice di inserimento in coda per i Task ad alta priorità.
- *append_task*: per l'inserimento nella coda dei Task periodici.
- *insert_task*: per l'inserimento di un generico Task in un preciso punto della coda.
- *requeue_task*: per il reinserimento nella coda dei Task periodici, con discriminazione in base alla priorità.
- *pop_task*: per la rimozione del primo Task dalla coda.
- *insert_immediate_task*: per l'inserimento di un Task immediato nella coda.

[23] Design Patterns: Elements of Reusable Object-Oriented Software, 1994, 1.1 What Is a Design Pattern?, p. 20-22

Nell'effettiva implementazione, al momento dell'inizializzazione della coda e dei Task periodici, non si è ricorso al loro inserimento selettivo in punti precisi della coda in base alla priorità. Questo, perché si sfrutta la caratteristica delineata nel capitolo 3.4.1, che permette alla coda di convergere autonomamente a uno stato "standard" indipendentemente dalla situazione di partenza.

4.2.1 Gestione dei Task immediati

Come precedentemente anticipato nel capitolo 3.4.2, i Task immediati dovranno essere inseriti nella coda di priorità in una posizione più vicina possibile alla cima. L'inserimento andrà eseguito tenendo conto di tutti gli altri Task non periodici già presenti e in particolare, qualora la coda non sia vuota, senza mai inserire un Task immediato in cima. La motivazione di questa scelta implementativa è una misura di prevenzione per comportamenti anomali a tempo di esecuzione.

Si consideri lo stato iniziale 9.1 della coda in figura 9, nel quale ha inizio l'esecuzione del Task 1, che occupa la posizione più prioritaria nella coda. Siccome il Task 1 è di tipo periodico, dovrà essere reinserito nella coda al termine della sua esecuzione.

Si consideri ora il caso in cui un Task immediato 3 venga inserito in cima alla coda. Se il Task immediato 3 è stato inserito nel momento nel quale il Task 1 aveva iniziato la sua esecuzione ma non si era ancora concluso, al momento della sua effettiva conclusione verrà rimosso il Task immediato 3 e non il Task 1. Portando dunque alla mancata esecuzione del Task immediato 3 e una doppia esecuzione del Task 1.

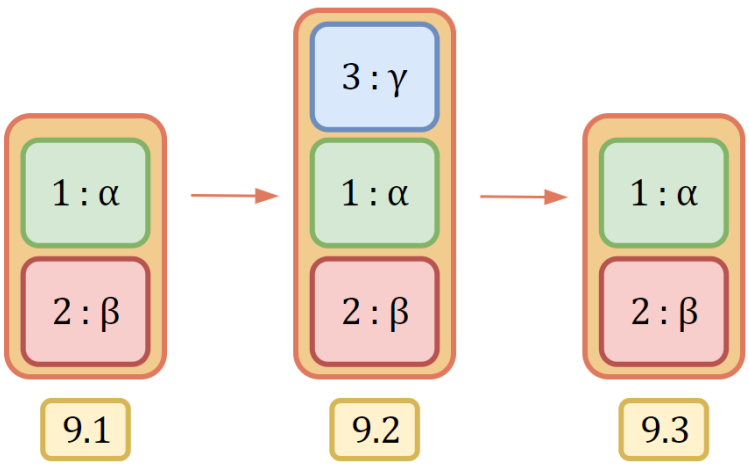

Figura 9 - Esempio di anomalia nell'esecuzione di Task immediati

4.3 Classe UserInterface e Task

Al fine di comprendere al meglio il funzionamento delle classi *UserInterface* e *Task* è necessario esaminarle contemporaneamente, in quanto intrinsecamente connesse.

La classe *UserInterface*, è il nucleo di controllo centrale dell'applicazione e si occupa di:

- Recuperare i dati di configurazione.
- Controllare, o generare in caso non esistano, le cartelle e i file relativi al database e agli utenti.
- Istanziare tutti i Task.
- Istanziare tutte le code di priorità inserendo al loro interno i Task opportuni.
- Allocare le varie strutture dati di supporto alle letture dei registri.
- Allocare, e renderizzare tutti i *widget* nella finestra grafica.
- Gestire l'avvio dei thread.
- Regolare e gestire l'esecuzione dei Task della coda.
- Controllare e avviare a richiesta le funzionalità attivabili dall'utente.

Nei controllori, al fine di risparmiare i non abbondanti registri, si sceglie spesso di utilizzare un singolo bit come interruttore di controllo per una funzionalità interne. Questo pone una difficoltà aggiuntiva allo sviluppo, in quanto nell'interfaccia grafica, più *widget* potrebbero dover interrogare lo stesso registro. Conseguentemente, senza una logica efficiente, si occuperebbe ripetutamente il mezzo comunicativo in intervalli di tempo molto ravvicinati, creando rallentamenti generali e interrogando il registro più volte del necessario.

In aggiunta, un ulteriore livello di complessità è dato dal fatto che ogni singolo *widget* deve avere un comportamento unico in base allo stato della comunicazione, e talvolta anche in base al valore ricevuto in fase di comunicazione.

È quindi necessario un meccanismo che consenta di definire comportamenti personalizzati per ciascun *widget*, evitando interrogazioni multiple dello stesso registro per diversi elementi grafici e che sia al contempo modulare e facilmente estensibile.

La soluzione impiegata si sviluppa inizialmente nel costruttore della classe *UserInterface*. Qui, vengono create nuove strutture dati di supporto che, in aggiunta ai parametri specificati nel capitolo 3.3, verranno fornite ai Task al momento della loro creazione:

- *object_references*: dizionario di riferimenti ai *widget* che necessitano un aggiornamento, nel quale ogni elemento grafico è identificato da una chiave corrispondente all'indirizzo del registro da cui deve ottenere le informazioni per aggiornarsi.

- *success*: dizionario di funzioni che delineano le azioni da svolgere in caso la comunicazione abbia successo, dove ogni funzione è identificata da una chiave corrispondente al riferimento del widget che dovrà aggiornare.
- *failure*: dizionario di funzioni che delineano le azioni da svolgere in caso di fallimento della comunicazione, dove ogni funzione è identificata da una chiave corrispondente al riferimento del widget che dovrà aggiornare.

Per passare funzioni all'interno dei dizionari *success* e *failure* sono stati usati dei riferimenti, e nello specifico è stato utilizzata la funzione *partial*[24] del modulo *functools*. *partial* permette di creare nuove funzioni parzialmente complete, ovvero consente di generare riferimenti a funzioni potenzialmente incomplete ma con alcuni argomenti fissati in modo predefinito.

Nel contesto delle componenti grafiche, questa scelta ci consente di distinguere tra azioni predefinite che devono essere eseguite in ogni aggiornamento, e altre azioni che dipendono da stati ed eventi che si verificano durante l'esecuzione del programma.

Come esempio, si consideri il semaforo dell'interfaccia "DIAGNOSI I/O #1" presente nella figura 3.

Nel caso in cui la lettura dei dati abbia successo, è necessario assegnare un comportamento specifico per ogni componente del semaforo, ossia le tre luci. A questo scopo è stata usata la funzione della figura 10 *success_traffic_light*, che permette di gestire singolarmente tutte e tre le componenti del semaforo senza la necessità di dichiarare una funzione per ognuna.

```
def success_traffic_light(self, canvas, on_color, off_color, mask_bit, value):
    '''
    Sets the background of the canvas to on_color if the bit corresponding
    to the mask_bit is 1, otherwise sets it to off_color
    '''
    if (value & pow(2, mask_bit)) == 0:
        canvas.configure(background=off_color)
    else:
        canvas.configure(background=on_color)
```

Figura 10 – Funzione per l'aggiornamento delle luci del semaforo

Definito il comportamento, è possibile inserire all'interno del dizionari *success* e *failure* le componenti grafiche e i loro comportamenti predefiniti.

[24] functools — Higher-order functions and operations on callable objects, Python Docs, consultato il 5 ottobre 2023, https://docs.python.org/3/library/functools.html

In figura 10, si mostrano i riferimenti alle funzioni per le componenti del semaforo nel dizionario *success* dell'interfaccia "DIAGNOSI I/O #1".

```
self.diagnosi1_success_reading = {
    self.green_traffic_light: functools.partial(self.success_traffic_light,
                                                self.green_traffic_light,
                                                colors.MEDIUM_GREEN,
                                                'darkgreen',
                                                1),

    self.red_traffic_light: functools.partial(self.success_traffic_light,
                                              self.red_traffic_light,
                                              'red',
                                              'darkred',
                                              2),

    self.orange_traffic_light: functools.partial(self.success_traffic_light,
                                                 self.orange_traffic_light,
                                                 'orange',
                                                 colors.DARKORANGE,
                                                 3),
```

Figura 11 - Dizionario dei comportamenti delineati in caso di successo della comunicazione del semaforo dell'interfaccia "DIAGNOSI I/O #1"

Si noti come il parametro *value* della funzione *success_traffic_light* sia stato volutamente omesso nelle definizioni delle funzioni parziali del dizionario *success*, in quanto risultato della comunicazione che avverrà solamente a tempo di esecuzione. Al contrario, nel caso in cui la comunicazione fallisca, il parametro *value* da passare alla specifica funzione parziale del dizionario *failure* sarà uguale a *None* già a "*compile time*".

Questo tipo di soluzione permette anche di unificare il modo in cui le componenti grafiche vengono aggiornate, perché non sempre i *widget* del *framework* Tkinter utilizzano le stesse modalità di aggiornamento in modo uniforme, come ad esempio la funzione *config* per alcuni elementi grafici e la funzione *configure* per altri. Così facendo, attraverso la separazione delle definizioni dei comportamenti dalla logica di esecuzione, si assicura che il componente responsabile dell'aggiornamento non debba identificare quale metodo utilizzare in base al tipo di *widget* da aggiornare.

Durante l'esecuzione dei compiti si evidenzia un nuovo problema, legato al fatto che non tutti i Task comunicano allo stesso modo. In altre parole, avendo compiti potenzialmente diversi, potrebbero non usare gli stessi codici funzione Modbus, che in questo contesto applicativo vorrebbe dire dover ricorrere a diverse *API* del modulo MinimalModbus.

Un Task potrebbe infatti dover leggere un registro tramite la funzione *read_register*, mentre un altro potrebbe dover scrivere un registro tramite la funzione *write_register*. Similmente, un Task potrebbe dover leggere solo un registro tramite *read_register*, mentre un altro potrebbe dover leggere più registri per mezzo di *read_registers*.
Per risolvere questo ulteriore problema si è preso spunto dal pattern *Command*:

> "*The Command pattern suggests that GUI objects shouldn't send […] requests directly. Instead, […] should extract all of the request details, such as the object being called, the name of the method and the list of arguments into a separate command class with a single method that triggers this request.*
> *Command objects serve as links between various GUI and business logic objects. From now on, the GUI object doesn't need to know what business logic object will receive the request and how it'll be processed. The GUI object just triggers the command, which handles all the details*."[25]

La classe *Task* si ispira a questo design pattern e implementa l'omonima unità informativa descritta nel capitolo 3.3. Oltre a raccogliere in un unico oggetto tutte le informazioni relative a una trasmissione, implementa anche il metodo "*trigger*" descritto dal pattern *Command* per azionare l'evento di comunicazione stesso, ovvero il metodo *communicate*.
Tuttavia, va notato che internamente alla classe *Task* non è presente una definizione esplicita per un metodo di nome "*communicate"* secondo la sintassi standard del linguaggio Python, bensì viene dinamicamente generato all'interno del costruttore della classe. In particolare, il *function code* e *register count* passati al costruttore della classe *Task*, verranno utilizzati come parametro per distinguere il compito della funzione *communicate*.
Per conseguire questo obiettivo è stata usata la funzione *built-in setattr*, la quale consente la creazione, o la modifica degli attributi di un oggetto dopo la sua creazione[26].
In altre parole, questo ci consente di generare dinamicamente un attributo che referenzi un una precisa funzione, il tutto a tempo di esecuzione.

[25] Command, Refactoring Guru, consultato il 5 ottobre 2023, https://refactoring.guru/design-patterns/command
[26] Built-in Functions, Python Docs, consultato il 7 ottobre 2023, https://docs.python.org/3/library/functions.html#setattr

Per ottenere una comprensione più dettagliata di questo concetto, si consideri l'esempio applicativo in figura 12.

```
# Based on the function code and on the register count, the communicate function will behave differently:
match self.function_code:
    case modbus_functions.READ_HOLDING_REGISTERS:
        if self.register_count == 1:
            setattr(self,
                    'communicate',
                    types.MethodType(lambda self: self.instrument.read_register(self.register_address),
                                     self))
        else:
            setattr(self,
                    'communicate',
                    types.MethodType(lambda self: self.instrument.read_registers(self.register_address,
                                                                                 self.register_count),
                                     self))
```

Figura 12 - Creazione dinamica del metodo "communicate" in base alle esigenze comunicative del Task

In questo esempio, il primo match-case utilizza la funzione del protocollo Modbus dal *function code* 3 "*Read Holding Registers"*, che indica una lettura di registri interni[27].
Il successivo *statement if-else* discrimina i casi in cui il numero di registri da leggere sia uno o più di uno. Stabilita la funzione Modbus e il numero di registri da leggere si può procedere con la generazione del metodo *communicate*, dove:

- *Setattr* imposta l'attributo *communicate* dell'oggetto corrente (*self*) come un nuovo metodo.
- Il metodo sarà creato attraverso l'incapsulamento di una *lambda function*, ovvero una funzione anonima[28], in un oggetto di tipo metodo. Al suo interno, si verifica l'effettiva chiamata all'*API* del modulo MinimalModbus con i relativi parametri.

Questo approccio permette di risolvere un'altra problematica architetturale riguardante lo slave *Stepper*, il quale indirizza i suoi registri partendo da 0 e non da 1 come tutti gli altri, tramite la creazione di uno specifico metodo *communicate* con un offset specializzato per i suoi Task.
A questo punto, lo step rimanente consiste nell'aggiornare le componenti grafiche, e per farlo è necessario fornire le strutture dati di riferimento *object_references*, *success*, e *failure* generate nel costruttore della classe *UserInterface*, ai vari Task.

[27] Modbus Functions, Schneider Electric, consultato il 7 ottobre 2023, https://product-help.schneider-electric.com/ED/ES_Power/NT-NW_Modbus_IEC_Guide/EDMS/DOCA0054EN/DOCA0054xx/Master_NS_Modbus_Protocol/Master_NS_Modbus_Protocol-4.htm
[28] Lambdas, Python Docs, consultato il 7 ottobre 2023, https://docs.python.org/3/reference/expressions.html#grammar-token-python-grammar-lambda_expr

La funzione *update_objects* della classe *Task* si occupa dell'aggiornamento delle componenti grafiche legate a una comunicazione, eseguendo opportunamente le funzioni dei dizionari *success* e *failure* in base all'esito della trasmissione. Durante l'esecuzione di *update_objects*, è necessario distinguere i registri che richiedono l'aggiornamento di una sola componente grafica rispetto a quelli che ne richiedono più di una. Una soluzione consiste nell'aggregare tutti gli oggetti che richiedono l'aggiornamento in base a dati presenti nello stesso registro all'interno di una lista, come ad nell'esempio in figura 13.

```
self.diagnosi1_high_priority_widget_reference_table_sweep1 = {
    1801: [self.cuscinetti_ventola_quota_status_canvas, self.scudo_quota_status_canvas1,
           self.scudo_quota_status_canvas2, self.flangia_quota_status_canvas,
           id(self.drive1_single_register_values[1801]), self.diagnosi1_ud1_modbus_status_list[3],
           self.diagnosi1_ud2_modbus_status_list[3], self.diagnosi1_ud3_modbus_status_list[3],
           self.diagnosi1_ud4_modbus_status_list[3], self.diagnosi1_inv1_modbus_status_list[3],
           self.diagnosi1_inv2_modbus_status_list[3]],
}
```

Figura 13 - Widget da aggiornare per mezzo delle informazioni provenienti dal registro 1801 del Drive 1

All'interno della funzione *update_objects* si verificherà la presenza di questa lista, e in tal caso, verranno invocate le opportune funzioni per tutti gli oggetti al suo interno, come illustrato nella figura 14.

```
if status:
    if self.object_references:
        for key, value in self.object_references.items(): # Key: number, value: reference to object
            # If the communication is successful, the object/s will be updated
            #  with the value/s from the reading cache
            if not isinstance(value, list):
                self.success[value](value=self.reading_cache[key])

            else:
                for list_element in value:
                    self.success[list_element](value=self.reading_cache[key])
else:
    # If the communication is unsuccessful, the object/s will be updated
    # with the failure dict and no value will be passed to the function
    if self.object_references:
        for key, value in self.object_references.items(): # Key: number, value: reference to object
            if not isinstance(value, list):
                self.failure[value]()
            else:
                for list_element in value:
                    self.failure[list_element]()
```

Figura 14 – Gestione dell'aggiornamento dei widget della funzione update_objects

Riassumendo, questo approccio permette alla classe *UserInterface* di non preoccuparsi di che tipo di comunicazione è assegnata a un determinato Task, bensì solamente del suo avvio.

4.4 Classe Translator

La classe *Translator* implementa un modulo finalizzato alla traduzione di tutte le componenti grafiche che presentano del testo, a tempo di esecuzione.

Come in MultiBench, la traduzione viene effettuata per mezzo di file esterni che contengono tutte le stringhe di testo dei *widget* per una lingua destinazione. La classe *Translator* si differenzia in questo senso dall'implementazione originale, utilizzando file di formato *json* per la loro facilità di gestione e modificabilità. Questo risulta importante in vista di future estensioni del programma con altre componenti grafiche, poiché nell'implementazione originale ogni *widget* era referenziato dal numero di riga, e l'aggiunta o la rimozione nei file di traduzione di una sola riga avrebbe causato lo sfasamento di tutti i riferimenti testuali dei *widget*. Con il formato *json*, questo problema non si presenta in quanto ogni stringa è identificata da una chiave univoca.

Per quanto concerne le traduzioni, la classe *Translator* è in grado di gestirne autonomamente due tipi:

- Traduzioni statiche: nelle quali un testo nella lingua di partenza è semplicemente tradotto nella lingua destinazione.
- Traduzioni dinamiche: nelle quali le opzioni di traduzione di un elemento nella lingua destinazione sono molteplici in base a vari stati interni del programma.

A tempo di inizializzazione, l'oggetto della classe *Translator* si occupa di verificare la presenza di eventuali file di traduzione nell'apposito percorso del programma, e fornisce un metodo *translate*, che ha il compito di tradurre tutte le componenti grafiche del programma nella lingua selezionata come argomento, qualora essa sia disponibile.

4.5 Classe ToggleButton

Uno dei vantaggi nell'uso della *OOP* è la possibilità di estendere classi per minimizzare la scrittura di codice, e per personalizzare il funzionamento di componenti già esistenti.

È questo il caso della classe *ToggleButton*, la quale estende la classe *Button* del *framework* Tkinter. Con un *toggle button* si intende un elemento grafico della famiglia dei bottoni che mantiene il suo stato fino a una suo nuovo azionamento. Questo elemento era assente dalla lista dei *widget* forniti da Tkinter, ma necessario per ricreare varie funzionalità.

Per il suo sviluppo è stato necessario definire in primo luogo i due stati del bottone, i quali possono variare per contenuto testuale, colore di sfondo, immagine e altre caratteristiche di stile del bottone stesso. Questi due stati sono contenuti rispettivamente nei dizionari *on_config* e *off_config*.

Alla pressione di un pulsante della classe *ToggleButton*, verrà azionata la funzione *toggle*, che si occupa di impostare il nuovo dizionario che verrà caricato dalla funzione *config_button*. Nella figura 15 si mostra l'implementazione di questi due metodi.

```
def toggle(self, *args):
    '''Toggle the button between states'''
    if self['state'] == 'normal':
        if self.toggled:
            self.config_dict = self.off_config
        else:
            self.config_dict = self.on_config
        self.toggled = not self.toggled
        self.config_button()
```

```
def config_button(self):
    '''Load the parameters of the current state'''
    for key in self.config_dict:
        self[key] = self.config_dict[key]
        if key == 'image':
            self.image = self.config_dict[key]
```

Figura 15 - Implementazione dei metodi toggle e config_button della classe "ToggleButton"

4.6 Gestione degli utenti

Essendo la gestione di più utenti una caratteristica importante nel contesto del programma, si è scelto un paradigma di salvataggio dei dati più strutturato rispetto al salvataggio su file testuale originario. Questa esigenza richiedeva tuttavia un miglioramento in termini di sicurezza.

In primo luogo, per immagazzinare i dati si è scelto il *DBMS*[29] sqlite, perché minimale e adatto per basi di dati di piccole dimensioni. I dati di ogni utente sono contenuti in una tabella *users*, con al suo interno i seguenti attributi:

- *id*: intero identificativo come chiave primaria.
- *username*: stringa di testo per identificare il nome dell'utente.
- *authorization_level*: intero indicante il livello di autorizzazione dell'utente.
- *password_hash*: campo blob utilizzato per memorizzare l'*hash* della password.
- *salt*: campo blob per memorizzare un valore di *salt*.

Per garantire un livello di sicurezza maggiore rispetto all'implementazione originale di MultiBench, le password non saranno salvate in chiaro all'interno del database, bensì verranno memorizzati i loro *hash*. Con *hash* o *digest* si indica l'output prodotto dalle funzioni *hash*, caratterizzate dalla capacità di generare una lunghezza di output fissa, indipendentemente dalla dimensione dell'input[30].

Per eseguire l'*hashing* è stata usata la funzione *SHA-256* in combinazione con la funzione crittografica *pbkdf2_hmac* del modulo *hashlib*.

[29] Database Management System
[30] Cryptography Hash functions, tutorialspoint, consultato il 9 ottobre 2023, https://www.tutorialspoint.com/cryptography/cryptography_hash_functions.htm

SHA-256 è una funzione progettata per prendere un input e produrre un output fissato di 256 bit.

La funzione di derivazione di chiavi crittografiche *pbkdf2_hmac* permette di aggiungere un ulteriore livello di sicurezza applicando il *key-stretching*, che consiste nell'applicazione ripetuta di una funzione di *hashing* alla password e al *salt*[31].

Con il valore noto come *salt* si intende una sequenza di byte pseudo-randomica da anteporre alla stringa della password stessa[32].

Completata la procedura di creazione degli utenti, i dati sensibili non saranno più salvati in chiaro e lo stato della tabella *users* del database potrebbe rispecchiare quello della figura 16.

id	username	authorization_level	password_hash	salt
1	BASIC GUEST	2	\\9 a 3 q x& go L U3%j	w]d > a
2	BASIC OPERATOR	2	*9 k0 CB팽 D lo] U~s |	*
3	ADVANCED OPERATOR	1	=N 6| ",yr +* * v'y	! *я C
4	SERVICE OPERATOR	0	06%@d An '$ M :	6 `QF 4 /ª 5m

Figura 16 - Esempio della tabella utenti del database

La procedura di autenticazione verrà svolta in maniera simile, e avrà successo solo se l'*hash* dell'unione della password fornita nel tentativo di login con il *salt* dell'utente richiesto corrisponda a quello salvato nella base di dati.

[31] PBKDF2, Purpose and operation, Wikipedia, consultato il 9 ottobre 2023, https://en.wikipedia.org/wiki/PBKDF2

[32] Encryption vs. Hashing vs. Salting - What's the Difference?, PingIdentity, consultato il 9 ottobre 2023, https://www.pingidentity.com/en/resources/blog/post/encryption-vs-hashing-vs-salting.html

Capitolo 5 – Ottimizzazione

In questo ultimo capitolo si analizzeranno le motivazioni che hanno portato ad applicare determinate strategie di ottimizzazione, soffermandosi sulla loro implementazione e sui benefici forniti.

5.1 Interazione con multipli registri

Molto spesso, i dati necessari al funzionamento del programma contenuti nei registri dei controllori, si trovano in zone di memoria contigue. Ciò permette di sfruttare il **principio di località spaziale**, che dice:

> "*If a particular storage location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future.*"[33]

Questo principio, associato alla possibilità di effettuare multiple letture tramite il protocollo Modbus, consente di ridurre in modo significativo il numero complessivo di comunicazioni.

Nell'implementazione di MinimalModbus è presente la possibilità di effettuare letture e scritture multiple di registri tramite le funzioni *read_registers* e *write_registers*, che permettono di interagire con al più 123 registri per comunicazione.

Per confrontare le prestazioni di una singola comunicazione ripetuta rispetto a una eseguita su multipli registri, si consideri l'analisi effettuata sulle rispettive funzioni in figura 17 con un *baud-rate* di 115200.

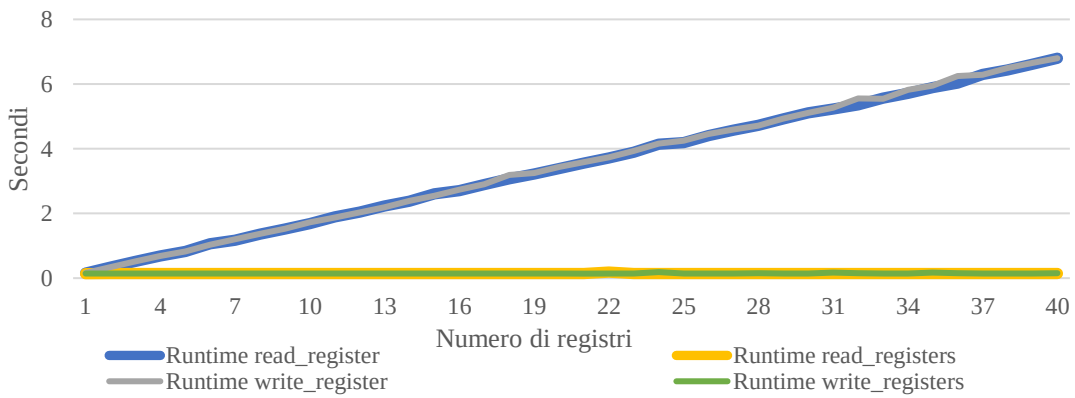

Figura 17 - Analisi lettura singoli registri iterativa rispetto a lettura di molteplici registri

[33] Locality of reference, Types of locality, Wikipedia, consultato il 14 ottobre 2023, https://en.wikipedia.org/wiki/Locality_of_reference

Le funzioni *read_register* e *write_register*, presentano un tempo di computazione dalla crescita lineare direttamente proporzionale al numero di registri letti, mentre le funzioni *read_registers* e *write_registers* sembrano soffrire meno di questo fenomeno.
Anche nel caso della lettura del massimo consentito di 123 registri illustrata in figura 18, non si notano aumenti significativi rispetto alle letture di pochi registri con la funzione *read_registers*.

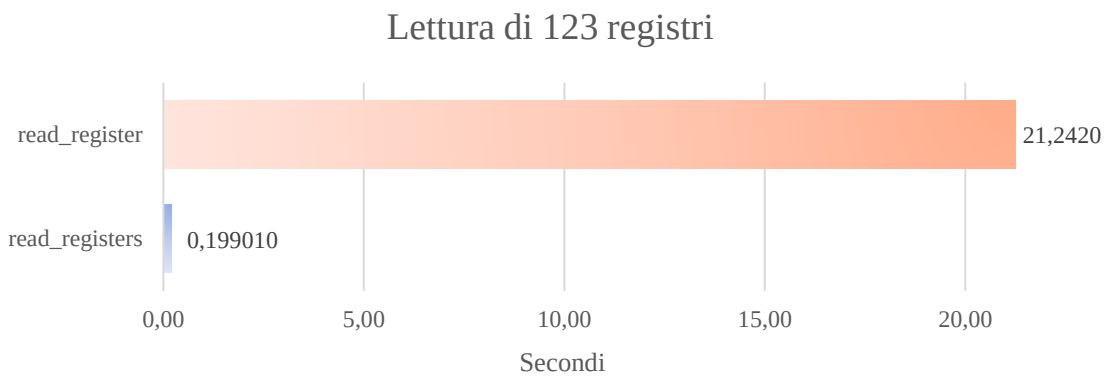

Figura 18 - Confronto tra letture singole ripetute e letture multiple con il massimo numero di registri consentiti

Risulta quindi evidente che la operazioni su più registri, quando possibile, siano sempre più convenienti di quelle ripetute su singoli registri.

5.2 Caching

Nella maggioranza dei casi, le letture di diversi registri implicano l'uso di tutti e soli i dati acquisiti nel momento dell'esecuzione del Task per l'aggiornamento delle componenti grafiche, ma non è sempre questo il caso. Infatti, alcuni *widget* necessitano di più dati presenti in diversi registri per essere aggiornati. Per ovviare a questa problematica è necessario effettuare *caching* dello stato dei registri, al fine di consentire un accesso in un secondo momento a questi valori da parte delle funzioni di aggiornamento interessate.
A livello implementativo sono stati creati dei dizionari di misurazioni, referenziati da una chiave corrispondente all'indirizzo del registro e un campo valore occupato da un'istanza della classe *Value*.
La classe *Value* è un semplice *wrapper* di un attributo intero che è stato utilizzato per evitare comportamenti anomali in fase di *caching* di questi valori. Siccome le letture di questi registri sono soggette allo stato della comunicazione stessa, sono state aggiunti nei rispettivi dizionari *success* e *failure*.

Dal momento che i valori memorizzati nei registri potrebbero essere uguali, si sarebbe potuto incappare in problematiche con CPython, l'interprete di Python utilizzato in questo progetto.

Infatti, nella fase iniziale si erano salvati questi valori di *cache* come dei semplici interi ricevuti dall'esecuzione delle funzioni *communicate*. Dal momento che la chiave contenuta nel dizionari *success* o *failure* corrisponde all'**identificatore del riferimento della variabile contenente il valore della *cache* dei registri**, nel caso due registri avessero avuto lo stesso valore, l'*entry* del dizionario aggiornata per ultima avrebbe sovrascritto la prima. Questo accade perché l'interprete CPython gestisce i numeri di piccole dimensioni tramite l'*Integer Interning*, una tecnica di ottimizzazione mirata a ridurre il consumo di memoria e al miglioramento delle prestazioni. Viene attuata evitando di allocare per ogni variabile di tipo intero di piccole dimensioni (da -5 a 256) un nuovo oggetto in memoria, bensì facendo puntare il riferimento della variabile in questione a una zona di memoria con valore preallocato dell'intero corrispondente[34].

In altre parole, questo potrebbe causare una sovrascrittura delle *entry* dei dizionari *success* e *failure* nel caso in cui due registri letti condividano lo stesso valore. Per ovviare a questo problema è stata creata la classe *Value*, che forza l'interprete ad allocare un oggetto in una nuova area di memoria, rendendo così l'identificatore dell'oggetto usato come chiave, univoco all'interno del programma.

Per quanto riguarda l'ottimizzazione, questo approccio di *caching* permette di ridurre il numero totale delle comunicazioni, e quindi di migliorare le prestazioni generali.

5.3 Problemi nella lettura di più registri

Le caratteristiche architetturali di alcuni *slave* non permettono sempre la lettura di più registri in un'unica comunicazione.

Nel caso degli *slave* Drive 1, 2, 3 e 4, il modulo coprocessore aggiuntivo utilizzato per comunicare attraverso il protocollo Modbus, non è in grado di supportare un numero di registri contigui superiori a **40 per letture o scritture a 16 bit**[35].

Lo stesso vincolo architetturale vale per gli Inverter, e l'unico *slave* esente da questa limitazione è lo Stepper.

[34] Integer Interning in Python (Optimization), CodeSanar, consultato il 15 ottobre 2023, https://www.codesansar.com/python-programming/integer-interning.htm

[35] Unidrive M700 / M701 / M702 Guida dell'utente al controllo Versione numero: 2, 9.1.6 Codici funzione, pagina 123

Nonostante il rispetto del limite superiore di registri per le letture e le scritture, non comunque è garantito che le comunicazioni con multipli registri vadano a buon fine. Infatti, molti registri interni dei controllori non sono accessibili, e in caso di forzata interrogazione ritornerebbero un codice di eccezione segnalando un errore nella lettura.
Nello specifico, questa problematica si può verificare anche se all'interno del *range* di interrogazione con più registri, si include un registro non accessibile.
Seppur durante il periodo di sviluppo non sia stato possibile approfondire questa tematica, è stata gestita con successo mediante la mappatura di tutti i registri interrogabili dei vari controllori. Attraverso l'implementazione di semplici script, si è proceduto all'interrogazione di tutti i registri degli *slave*, distinguendo quelli che potevano essere interrogati da quelli che non lo erano.
Di fatto, per tutti gli *slave* all'infuori dello Stepper, è stato possibile unire le letture dei registri in gruppi, a condizione che la distanza tra di essi fosse al più di 40 indirizzi e che tra gli estremi dell'intervallo di interrogazione non ci fossero registri non accessibili. A livello implementativo, questi gruppi sono stati gestiti tramite dizionari dal nome di *sweep*.
Per costruzione e separazione delle tipologie di Task all'interno dei dizionari degli *sweep*, è stato necessario differenziarli in base alla priorità dei propri compiti.
A titolo di esempio, si può considerare il dizionario in figura 19 specifico per il secondo *sweep* ad alta priorità per l'interfaccia "DIAGNOSI I/O #1".

```
self.diagnosi1_high_priority_widget_reference_table_sweep2 = {
    1911: self.cuscinetti_ventola_forza_min_label1,
    1912: self.cuscinetti_ventola_forza_max_label1,
    1913: self.cuscinetti_ventola_forza_min_label2,
    1914: self.cuscinetti_ventola_forza_max_label2,
    1915: self.scudo_forza_min_label,
    1916: self.scudo_forza_max_label,
    1917: self.flangia_forza_min_label,
    1918: self.flangia_forza_max_label,
    1923: self.cuscinetti_ventola_forza_label,
    1925: self.scudo_forza_label,
    1926: self.flangia_forza_label,
    1927: [self.cuscinetti_ventola_quota_measure1, self.cuscinetti_ventola_quota_measure2],
    1928: self.scudo_quota_label2,
    1929: self.scudo_quota_label1,
    1930: self.flangia_quota_label,
}
```

Figura 19 - Dizionario del secondo sweep ad alta priorità per l'interfaccia DIAGNOSI I/O #1

5.4 Ottimizzazione delle risposte

È comune tra i protocolli di comunicazione l'implementazione del *timeout*, una misura che indica il limite massimo di tempo che il mittente ha a disposizione per ricevere una risposta dal destinatario prima di considerarlo irraggiungibile.

Idealmente, in questo tipo di applicazione si preferirebbe scegliere dei *timeout* più piccoli possibile che allo stesso tempo non vadano a impattare le performance. Questo per fare in modo che l'applicazione appaia il più reattiva possibile anche nel caso di mancate comunicazioni.

Per questo caso di studio non si è analizzata una particolare instabilità del mezzo trasmissivo e pertanto, partendo da un valore medio per trasmissione di circa 0.20 secondi, si è scelto di aggiungere un errore del 25%, portando quindi il *timeout* per trasmissione a 0.25 secondi. Ciononostante, questo non vincola in alcun modo da future implementazioni di *timeout* adattivi basati sulla salute del mezzo trasmissivo.

Un'altra caratteristica da prendere in considerazione è il comportamento da adottare in caso di mancata risposta. Se si seleziona un *timeout* adeguato che copra la maggior parte delle trasmissioni con piccole variazioni di prestazioni, è probabile che le rimanenti siano mancate risposte dovute alla temporanea irraggiungibilità del destinatario. Pertanto, per evitare l'esecuzione dei compiti relativi a uno *slave* di cui una trasmissione ha raggiunto il limite del *timeout*, si potrebbero scegliere di ignorare una volta tutti i Task di quel preciso *slave* contenuti nella coda. Questa strategia permette di concentrarsi sulle trasmissioni degli *slave* di cui si è ricevuta un'effettiva risposta, tralasciando temporaneamente quelli legati ai destinatari considerati come irraggiungibili.

Per l'implementazione di questa logica è inizialmente necessario ottenere un resoconto dello stato attuale della coda. Per questa finalità viene usata la funzione *get_number_of_task_per_slave* della classe *UserInterface*, che si occupa di popolare un dizionario contenente il numero di Task che fanno attualmente parte della coda, differenziati per *slave*. Durante l'esecuzione, una struttura dati ausiliaria monitora il numero di Task degli *slave* irraggiungibili che vengono ignorati. Quando un Task di un destinatario da ignorare raggiunge la cima della coda, se il numero dei suoi Task ignorati è inferiore al suo totale dei Task nella coda, esso viene reinserito nella coda senza essere eseguito, e il contatore dei Task ignorati per quello slave viene incrementato.

I Task per un destinatario verranno nuovamente eseguiti quando il contatore di quelli ignorati raggiungerà il numero dei Task totali dello slave nella coda, con conseguente azzeramento del contatore.

In figura 20 si mostra un esempio applicativo di quanto spiegato in questo capitolo. Per semplicità verranno considerati solo Task periodici della stessa priorità, utilizzando la notazione $\{Task\}:\{Slave\}$, con $Task, Slave \in \mathbb{N}$ per i seguenti stati:

- 20.1: Tentativo di esecuzione del Task 1.
- 20.2: *Timeout* del Task 1, con successivo reinserimento del Task nella coda e incremento del contatore dei Task ignorati per lo *slave* 1.
- 20.3: Reinserimento del Task 2 nella coda, in quanto il contatore dei Task ignorati dello *slave* 1 è minore al numero dei Task dello stesso *slave* presenti nella coda. Successivo incremento dei Task ignorati per lo slave 1.
- 20.4: Azzeramento del contatore dei Task ignorati per lo slave 1, esecuzione e reinserimento del Task 3.

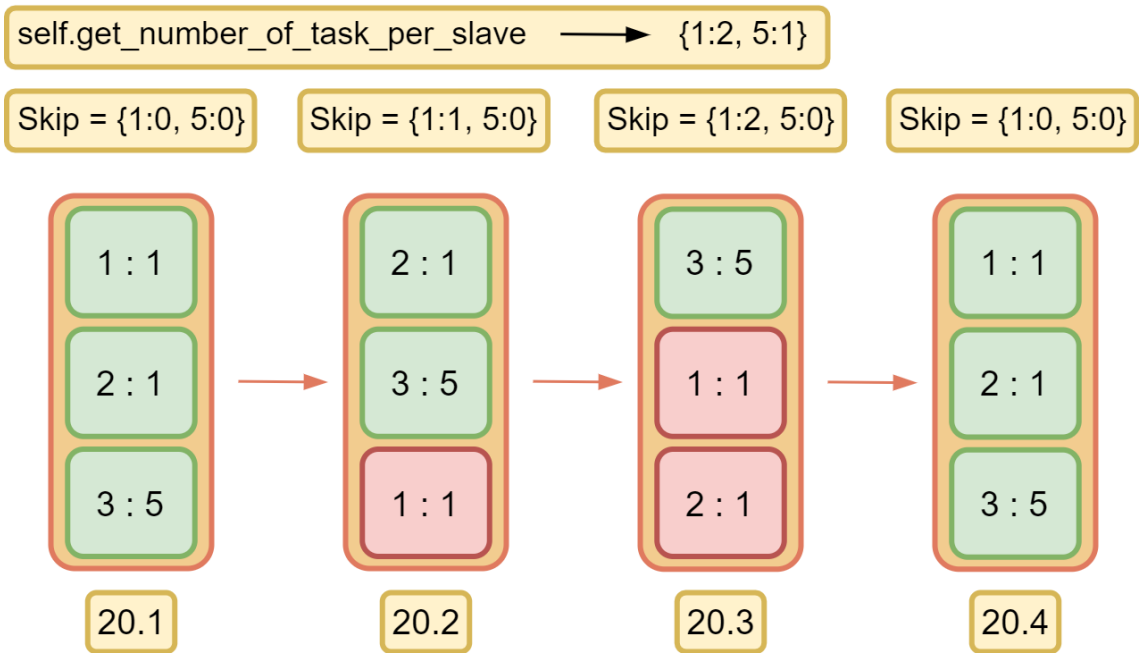

Figura 20 - Esempio di Task di uno slave ignorati dopo un timeout

Questo semplice accorgimento permette al programma di controllo di adattarsi automaticamente alla disponibilità dei destinatari delle comunicazioni. Ad esempio, se uno degli slave fosse volutamente disabilitato, non verrebbe eseguito più di un tentativo di comunicazione per quello slave a ogni periodo, aumentando la frequenza dei campionamenti delle altre attività e, di conseguenza, la reattività generale del programma.

5.5 Multithreading

Durante le prime fasi di sviluppo, il programma è stato testato senza sfruttare l'esecuzione parallela dei compiti. Tuttavia, con l'aumento della complessità e del carico di lavoro, questo paradigma di sviluppo ha mostrato evidenti limiti, che si traducevano in prestazioni insufficienti.

Uno dei principali problemi riscontrati era legato al ciclo principale o "*mainloop*" della finestra grafica del *framework* Tkinter, responsabile della gestione degli eventi. Quando era necessario eseguire un compito che richiedeva una quantità percettibile di tempo, il *thread* principale veniva bloccato fino al termine dell'esecuzione di tale compito. Questo comportava una notevole perdita di reattività e rallentamenti generali nell'interfaccia utente.

Per superare questa problematica e migliorare le prestazioni è stato introdotto il paradigma *multithreading* attraverso il modulo *threading* incluso nella libreria standard di Python. Questo approccio ha permesso di sfruttare in miglior modo le risorse hardware, specialmente nell'esecuzione di compiti che richiedevano aggiornamenti costanti.

La funzione *work_on_queue* è stata una dei maggiori beneficiari dell'uso del *multithreading*. Essa ha il compito di avviare e gestire tutte le fasi di esecuzione dei Task presenti nella coda. In base al successo o al fallimento, alla mancanza o alla malformazione delle risposte, la funzione prende decisioni operative in merito all'aggiornamento grafico, gestendo infine il reinserimento o l'eliminazione dei compiti.

Il multithreading è stato impiegato in altre aree del programma, tra cui:

- Controllo degli allarmi.
- Aggiornamento del grafico sulla salute delle comunicazioni e delle percentuali di risposta.
- Aggiornamento in tempo reale dell'orario.
- Salvataggio automatico dello stato del programma.

5.6 Vincoli di ottimizzazione

Oltre ai vincoli precedentemente menzionati nel capitolo 5.3, relativi al numero massimo di registri interrogabili e alla presenza di registri non accessibili, un altro fattore costruttivo che ha limitato l'ottimizzazione è stata la presenza di un sistema di comunicazione *half-duplex*. Nello specifico, non potendo inviare e ricevere messaggi nello stesso momento, il mezzo comunicativo può essere usato al più per una trasmissione

alla volta. Questo ha reso difficile l'adozione di approcci asincroni che avrebbero potuto migliorare significativamente le prestazioni del sistema.
Inoltre, anche in presenza di un mezzo comunicativo *full-duplex,* ad esempio quello dello standard per le comunicazioni seriali RS-485 a quattro fili, la struttura della connessione a *bus* non offrirebbe garanzie sulla gestione delle collisioni, rendendo il sistema potenzialmente più inaffidabile di quello *half-duplex* in un'ottica asincrona.

5.7 Comportamenti selettivi

Il programma originale MultiBench e la riprogettazione di questo caso di studio sono state organizzate graficamente tramite un menu di selezione dell'interfaccia. Nello specifico di Tkinter si è utilizzato il *widget notebook,* che permette di creare diverse interfacce selezionabili da un menu.
Ognuna di queste interfacce contiene *widget* con funzionalità diverse all'infuori del pannello superiore che, essendo non incluso nel *notebook,* è sempre presente indipendentemente dal menu selezionato. Siccome le varie interfacce non condividono compiti, per migliorare ulteriormente le prestazioni si potrebbe evitare di eseguire tutti quei Task che non sono relativi all'interfaccia correntemente selezionata. Applicativamente, quando un utente decide di voler cambiare interfaccia del *notebook,* sarebbe necessario cambiare i Task della coda per adattarli alle necessità dell'interfaccia selezionata.
Inizialmente si era pensato di rimuovere tutti i Task dalla coda e inserire quelli dell'interfaccia richiesta, tuttavia questo approccio è risultato molto problematico per due aspetti.
In primo luogo, la creazione del singolo Task ha un costo computazionale relativamente alto e messa nell'ottica di decine di Task, diventerebbe presto inadatto a un sistema che necessità di responsività.
In secondo luogo, la rimozione e l'inserimento di altri Task vanificherebbe lo stato in cui la coda era prima di cambiare interfaccia. In altre parole, se un utente che naviga tra le interfacce richiede una precisa scheda del *notebook* visualizzata precedentemente, i Task ad essa associati ripartirebbero dal primo in ordine di inserimento, anziché proseguire dall'ultimo Task che era stato eseguito prima del cambio del menu. Nonostante sia un problema risolvibile utilizzando strutture dati di supporto per memorizzare l'ordine dei Task al momento della rimozione dalla coda, aumenta la complessità generale della logica applicata.

Una soluzione più elegante alla problematica consiste nell'utilizzare più code di priorità, e nello specifico nell'uso di una coda per ogni interfaccia del *notebook*.

All'interno di ogni coda dell'interfaccia verranno inseriti i Task legati a quella precisa pagina del *notebook*, ed essi non verranno mai rimossi anche quando un'altra interfaccia verrà richiesta da un utente. Un selettore sarà usato come riferimento per selezionare la coda interessata dall'attuale interfaccia da utilizzare. Così facendo non è necessario attendere che i Task vengano allocati e reinseriti nella coda ogni qualvolta l'utente cambi menu, e permette di sospendere e riprendere l'esecuzione dei Task quando necessario.

Per mantenere costantemente allocate a tempo di esecuzione più code di priorità è necessario mantenere almeno un riferimento per ognuna. Questo perché l'interprete CPython utilizza un particolare metodo di *garbage collection* chiamato *reference counting*.

Viene definito come *reference count* il contatore, presente in ogni oggetto istanziato dall'interprete, pari al numero di riferimenti che puntano a quell'oggetto. Il ciclo di *garbage collection* dealloca tutti gli elementi che hanno un *reference count* pari a zero[36]. Mantenendo quindi un riferimento attivo per ogni coda, si possono mantenere in memoria tutte le code di priorità senza che vengano deallocate.

A livello implementativo, qualora una pagina del notebook venga selezionata, il riferimento all'attuale coda da utilizzare viene assegnato al selettore *self.priority_queue*, come illustrato in figura 21.

```
def notebook_tab_change(self, event):
    '''Changes the priority queue and the slave task counter depending on the selected tab.'''

    current_tab = self.notebook.index(self.notebook.select())

    match current_tab:
        case 0: # Work program menu
            self.priority_queue = self.programma_lavoro_queue
            self.slave_task_counter = self.programma_lavoro_slave_task_counter

        case 1: # Manual movements
            self.priority_queue = self.movimenti_queue
            self.slave_task_counter = self.movimenti_slave_task_counter
```

Figura 21 - Assegnamento del selettore della coda per le rispettive code delle prime due interfacce

[36] Garbage collector design, Python Developer's Guide, consultato il 16 ottobre 2023, https://devguide.python.org/internals/garbage-collector

Tramite questo approccio, si è preferito un maggior consumo in termini di memoria e un incremento nel tempo di avvio del programma per allocare preventivamente tutti i Task, al fine di ottenere prestazioni significativamente migliori a *runtime*.

5.8 Risultati dell'ottimizzazione

Per mezzo della nuova struttura di controllo del programma e delle tecniche descritte in questo capitolo, è stato ottenuto un miglioramento sostanziale della reattività generale. Per misurare questo miglioramento sono stati campionati i tempi di risposta dell'aggiornamento visivo dello stesso *widget* del pannello superiore di entrambi i programmi. In figura 22 si mostrano i tempi di risposta ottenuti dalle misurazioni.

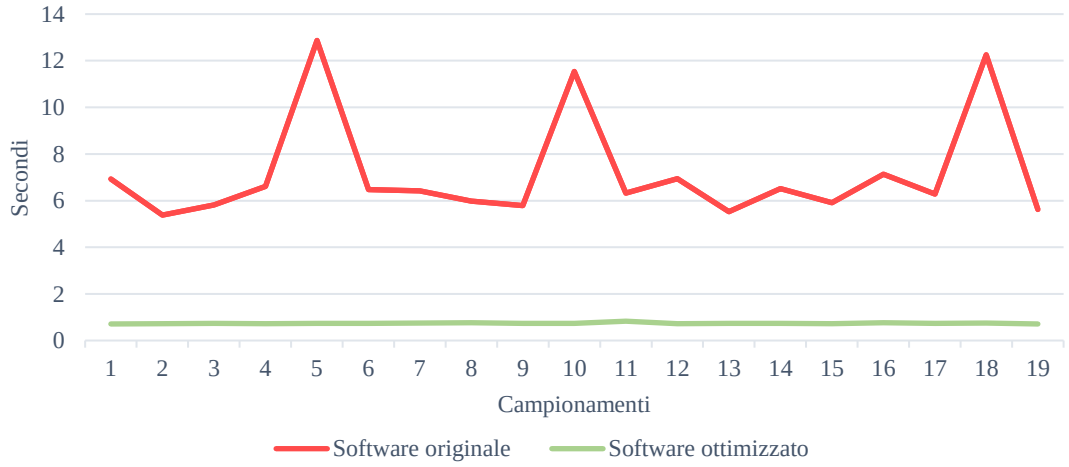

Figura 22 - Campionamenti della reattività tra i due programmi

I campionamenti del software originale evidenziano l'instabilità degli aggiornamenti grafici gestiti dai cicli di *polling*. Al contrario, nella versione riprogettata, si osserva un notevole miglioramento della stabilità. Ignorando il rumore causato dai picchi delle misurazioni in figura 22, possibilmente dovuti a un hardware non prestazionale, si ottengono i seguenti valori medi di aggiornamento mostrati in figura 23.

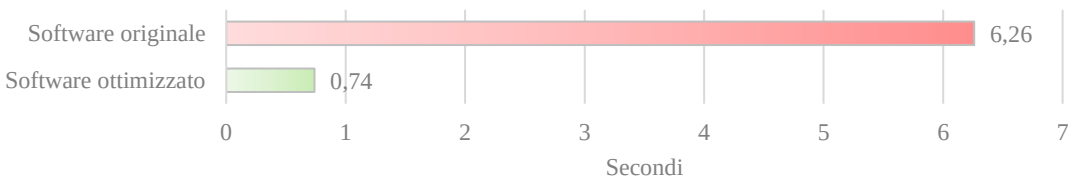

Figura 23 - Valori medi di aggiornamento dello stesso widget tra i due programmi

Conclusioni

Questo caso di studio ha offerto l’opportunità di mettere in pratica metodologie e tecniche di sviluppo nel contesto della progettazione del software.

Nella fase iniziale di analisi, sono state identificate le varie problematiche del software originario, evidenziando le maggiori criticità su cui focalizzarsi.

Durante la fasi di progettazione e sviluppo è stato possibile ricercare e sperimentare diverse soluzioni, per trovare i miglior compromessi che rispettassero i vincoli di progetto.

Infine, nella fase di ottimizzazione sono state apportate modifiche applicative mirate al miglioramento delle prestazioni generali, che hanno contribuito all’aumento sostanziale della responsività e conseguentemente della usabilità.

Il prodotto finale, non rappresenta soltanto una soluzione attua a migliorare le problamatiche evidenziate, bensì un punto di partenza per ulteriori evoluzioni e adattamenti che permetterà di adattarsi alle esigenze dei clienti.

Tra le future implementazioni ed estensioni da tenere in considerazione, potrebbe essere di particolare interesse ricercare soluzioni per rimuovere gli attuali vincoli, specialmente quelli di tipo hardware, che potrebbero permettere ulteriori miglioramenti dal punto di vista prestazionale.

Complessivamente, questo caso di studio mette in evidenza l'importanza di un approccio strutturato per affrontare le sfide nel campo della progettazione del software e mostra come il processo di progettazione sia essenziale per il raggiungimento degli obiettivi.

Appendice

Di seguito, verranno mostrate le principali interfacce dei menu del software originale MultiBench e le rispettive versioni riprogettate.

Figura 24 - Pannello superiore di MultiBench

Figura 25 - Pannello superiore riprogettato

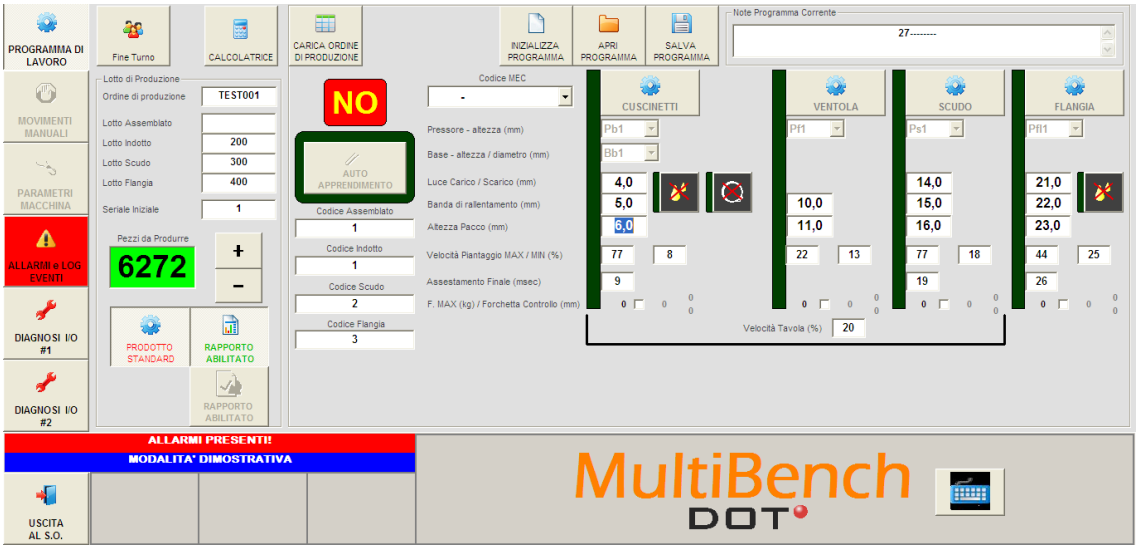

Figura 26 - Interfaccia "PROGRAMMA DI LAVORO" di MultiBench

Figura 27 - Interfaccia "PROGRAMMA DI LAVORO" riprogettata

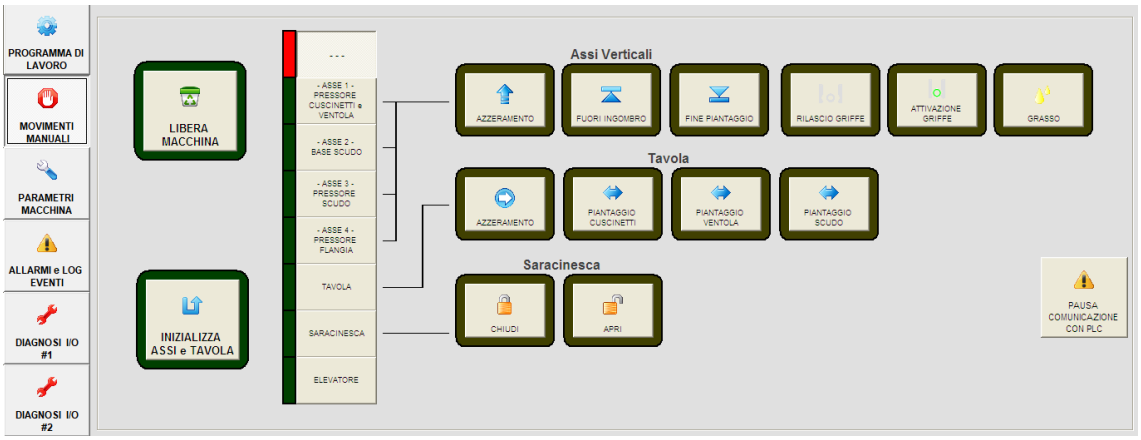

Figura 28 - Interfaccia "MOVIMENTI MANUALI" di MultiBench

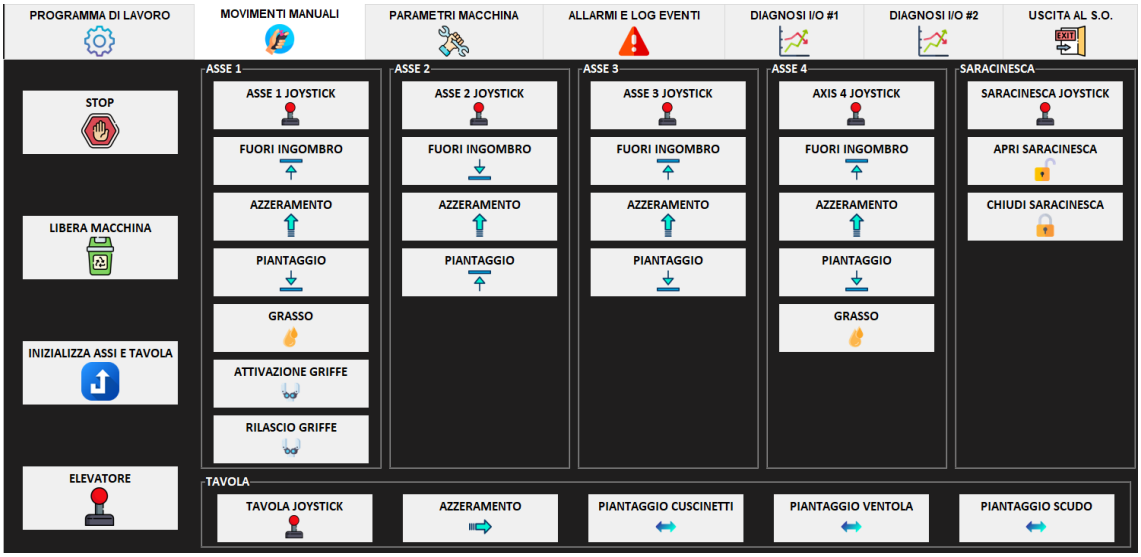

Figura 29 - Interfaccia "MOVIMENTI MANUALI" riprogettata

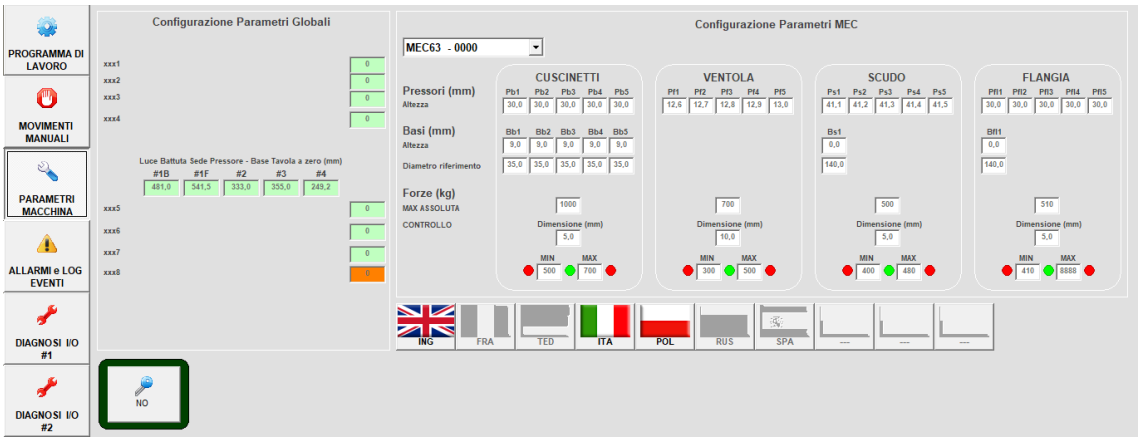

Figura 30 - Interfaccia "PARAMETRI MACCHINA" di MultiBench

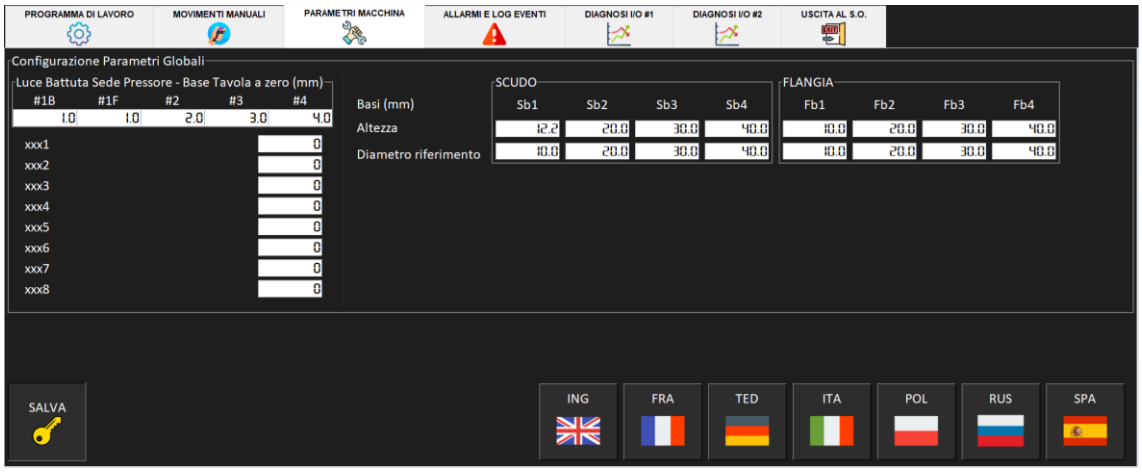

Figura 31 - Interfaccia "MOVIMENTI MANUALI" riprogettata

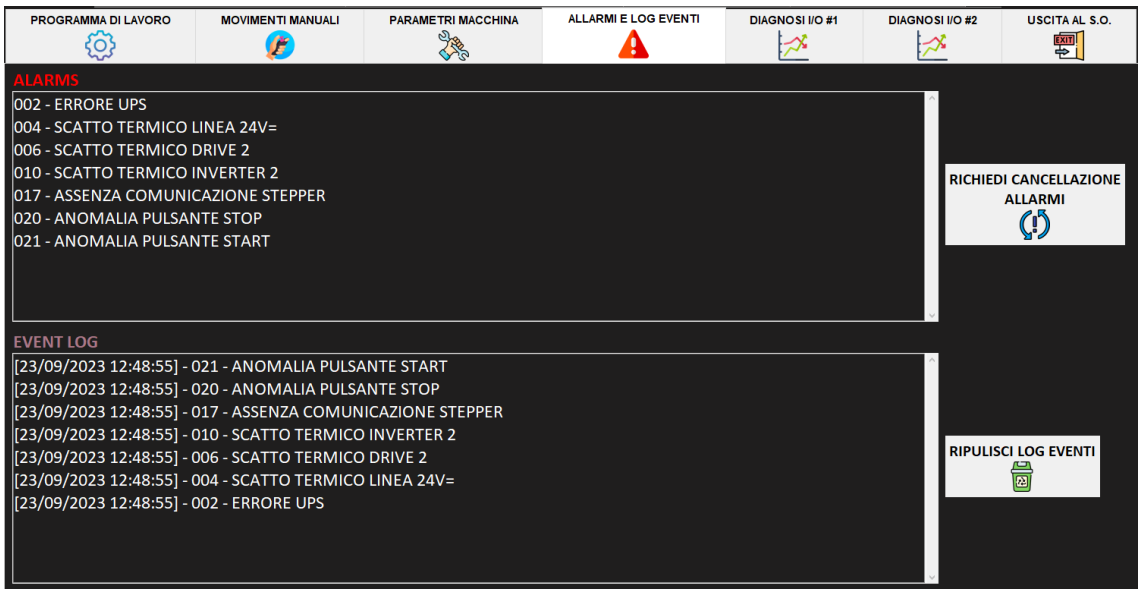

Figura 32 - Interfaccia "ALLARMI E LOG EVENTI" riprogettata

Figura 33 - Interfaccia "DIAGNOSI I/O #1" di MultiBench

Figura 34 - Interfaccia "DIAGNOSI I/O #1" riprogettata

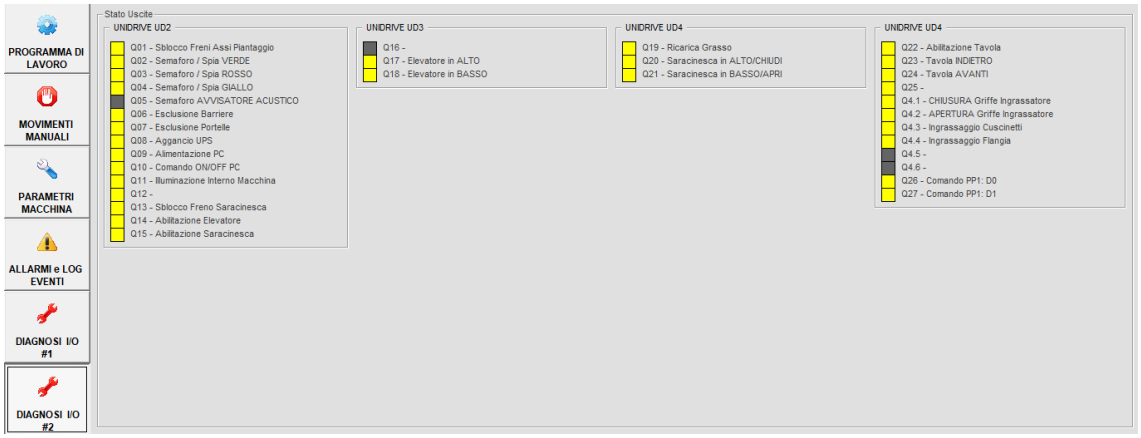

Figura 35 - Interfaccia "DIAGNOSI I/O #2" di MultiBench

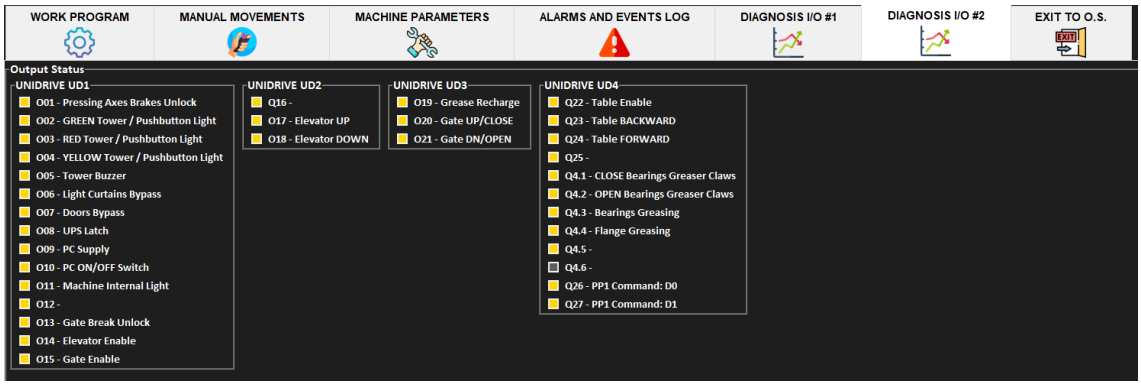

Figura 36 - Interfaccia "DIAGNOSI I/O #2" riprogettata

Bibliografia

- ISO/IEC 7498-1:1994, Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model, 7.7 Physical Layer.
- Unidrive M700 / M701 / M702 Guida dell'utente al controllo Versione numero: 2, 9.1.5, p. 121-123.
- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, 1994, 1.1 What Is a Design Pattern?, p. 20-22

Sitografia

- What is an HMI, Copadata, consultato il 28 Settembre 2023, https://www.copadata.com/en/product/zenon-software-platform-for-industrial-automation-energyautomation/visualization-control/what-is-hmi/
- Automazione Industriale, Treccani, consultato il 29 Settembre 2023, https://www.treccani.it/enciclopedia/automazione-industriale_%28Enciclopedia-Italiana%29/
- Introduction to Modbus Serial and Modbus TCP, Ccontrols, pagina 1, consultato il 29 Settembre 2023, https://www.ccontrols.com/pdf/Extv9n5.pdf
- Modbus RTU communication guide, Virtual-serial-port, consultato il 29 settembre 2023, https://www.virtual-serial-port.org/articles/modbus-rtu-guide/
- Modbus Networking Guide, libelium, consultato il 29 settembre 2023, https://development.libelium.com/modbus_networking_guide/introduction
- Polling (computer science), Wikipedia, consultato il 29 settembre 2023, https://en.wikipedia.org/wiki/Polling_(computer_science)
- MinimalModbus, Github, consultato il 3 ottobre 2023, https://github.com/pyhys/minimalmodbus/
- Features, MinimalModbus, consultato il 3 ottobre 2023, https://minimalmodbus.readthedocs.io/en/stable/readme.html
- Apache License, Version 2.0, Apache, consultato il 3 ottobre 2023, https://www.apache.org/licenses/LICENSE-2.0/
- Tcl/Tk License Terms, Tcl.tk, consultato il 4 ottobre 2023, https://www.tcl.tk/software/tcltk/license.html

- Commander SK Advanced User Guide – Issue Number: 8, consultato il 5 ottobre 2023, https://www.nidec-netherlands.nl/media/2125-frequentieregelaars-commander-sk-advanced-user-guideen-iss10-0472-0001-10.pdf
- functools — Higher-order functions and operations on callable objects, Python Docs, consultato il 5 ottobre 2023, https://docs.python.org/3/library/functools.html
- Command, Refactoring Guru, consultato il 5 ottobre 2023, https://refactoring.guru/design-patterns/command
- Built-in Functions, Python Docs, consultato il 7 ottobre 2023, https://docs.python.org/3/library/functions.html#setattr
- Modbus Functions, Schneider Electric, consultato il 7 ottobre 2023, https://product-help.schneider-electric.com/ED/ES_Power/NTNW_Modbus_IEC_Guide/EDMS/DOCA0054EN/DOCA0054xx/Master_NS_Modbus_Protocol/Master_NS_Modbus_Protocol-4.htm
- Lambdas, Python Docs, consultato il 7 ottobre 2023, https://docs.python.org/3/reference/expressions.html#grammar-token-python-grammar-lambda_expr
- Cryptography Hash functions, tutorialspoint, consultato il 9 ottobre 2023, https://www.tutorialspoint.com/cryptography/cryptography_hash_functions.htm
- PBKDF2, Purpose and operation, Wikipedia, consultato il 9 ottobre 2023, https://en.wikipedia.org/wiki/PBKDF2
- Encryption vs. Hashing vs. Salting - What's the Difference?, PingIdentity, consultato il 9 ottobre 2023, https://www.pingidentity.com/en/resources/blog/post/encryption-vs-hashing-vs-salting.html
- Locality of reference, Types of locality, Wikipedia, consultato il 14 ottobre 2023, https://en.wikipedia.org/wiki/Locality_of_reference
- Integer Interning in Python (Optimization), CodeSanar, consultato il 15 ottobre 2023, https://www.codesansar.com/python-programming/integer-interning.htm
- Garbage collector design, Python Developer’s Guide, consultato il 16 ottobre 2023, https://devguide.python.org/internals/garbage-collector