



SOFTWARE ENGINEERING FOR EMBEDDED SYSTEMS

Distributed Railway Interlocking Systems

Damerini Jacopo

Matricola: 7125740

E-Mail: jacopo.damerini@stud.unifi.it

Filino Alessandro

Matricola: 7125518

E-Mail: alessandro.filino@stud.unifi.it

Macaluso Alberto

Matricola: 7115684

E-Mail: alberto.macaluso@stud.unifi.it

Data Consegna: 02/10/2023

Anno Accademico 2022/23

INDICE

1 Introduzione	4
1.1 Descrizione interlocking distribuito	4
1.2 Requisiti architetturali del singolo nodo	4
1.3 Descrizione taskset	4
1.3.1 Task a	4
1.3.2 Task b	5
1.3.3 Task c	7
1.3.4 Task d	7
1.3.5 Task e	7
1.3.6 Task f	7
1.4 Descrizione funzionamento Two-Phase Commit Protocol	7
1.5 Descrizione protocollo di comunicazione	9
1.6 Vincoli temporali e deadline	10
1.7 Fasi costitutive del processo di sviluppo	11
2 Modellazione	13
2.1 Introduzione	13
2.2 Transizioni di switch e di preselection	13
2.3 Rappresentazione dei messaggi	14
2.4 Condivisione della memoria	15
2.5 Task non rappresentati nella rete	16
2.6 Utilizzo delle risorse	17
2.7 Disegno Rete	17
3 Codice	20
3.1 Modulo kernel	20
3.1.1 initTask (Task a)	20
3.1.2 controlTask (Task b)	22
3.1.3 wifiTask (Task c)	23
3.1.4 posiTask (Task d)	25
3.1.5 diagTask (Task e)	25
3.1.6 logTask (Task f)	26
3.2 sendLogTask	26
3.2.1 destructorTask	26
3.2.2 timerTask	27
3.2.3 gpio	27
3.2.4 dkm	27
3.3 Utility e scripts di appoggio	27
3.3.1 host.py	27
3.3.2 launch_host.sh	28

3.3.3	log.py	28
3.3.4	logParser.py	28
3.3.5	launch_nodes.sh	29
4	Testing	30
4.1	Requisiti di architettura per il sistema distribuito	30
4.2	Requisiti per il prototipo di prova	30
4.3	Test di sistema	30
5	Risultati	32
5.1	Misurazione dei tempi	32
5.2	Errori nella misurazione	33
5.3	Unità di tempo nel sistema	33
5.4	Analisi della rete con Oris	34
5.5	Analisi delle deadline e dei runtimes	35
6	Setup	39
6.1	Setup hardware	39
6.1.1	Configurazione remota	39
6.1.2	Configurazione locale	40
6.1.3	Led e bottone	41
6.2	Setup software	42
6.3	Utility WorkBench	45
6.3.1	Wrtool	46
6.3.2	Wrdbg	46
6.3.3	connect.sh	46
6.3.4	Guida utilizzo	47
6.3.5	Esempio d'uso	48
7	Sviluppi futuri	50
7.1	Dislocazione geografica	50
7.2	Espansione della rete	50
7.3	Simulatore	51
7.4	Distribuire la conoscenza	51
7.5	Itinerari dinamici	51
7.6	Priority nelle prenotazioni	52
7.7	Migrazione a Rust	53

1 Introduzione

1.1 Descrizione interlocking distribuito

Nell'ambito della segnalazione ferroviaria, un interlocking (traducibile in italiano come Apparato Centrale) è l'apparato di segnalazione che previene movimenti di treni in conflitto tra loro, al fine di garantire la sicurezza della circolazione. Tipicamente tale sistema è centralizzato e comanda segnali e scambi in accordo con le regole di sicurezza codificate nelle normative in vigore.

In questo progetto è stato considerato invece un sistema distribuito, in cui gli elementi del tracciato ferroviario come binari e scambi hanno capacità di computazione che usano non solo per controllare sensori ed attuatori, ma anche per funzionalità di sicurezza e gestione del traffico ferroviario. In questo modo, questa rete distribuita è in grado di esercitare la funzione di interlocking senza l'ausilio di alcun elemento di computazione centralizzato (se non per un monitoraggio del funzionamento complessivo).

1.2 Requisiti architetturali del singolo nodo

Nella sezione seguente vengono riassunti i requisiti prototipali richiesti da ogni singolo nodo per riuscire a formare un'applicazione distribuita con funzionalità descritte nel seguente articolo [1]. I requisiti sono qui di seguito elencati:

1. Il nodo deve essere implementato su Raspberry e VxWorks.
2. Il nodo deve essere implementato come un'applicazione multitasking.
3. Il nodo deve eseguire i task descritti nei paragrafi successivi (vedi [1.3](#)).
4. La comunicazione tra task avviene mediante i meccanismi forniti da VxWorks.
5. I sensori di occupazione sono simulati da interruttori binari.
6. Gli attuatori (motori degli scambi) sono simulati da led.
7. Il comportamento fisico di uno scambio è simulato dal "Task di simulazione posizionamento di uno scambio".

1.3 Descrizione taskset

Nella sezione seguente vengono presentati tutti i task di cui è richiesta l'implementazione nei requisiti presentati per un singolo nodo.

1.3.1 Task a

Il *task inizializzazione e configurazione (a)* si attiva all'accensione della scheda ed ha come scopo quello di ricevere attraverso il collegamento di comunicazione la configurazione del sistema, consistente nell'elenco di itinerari che interessano il nodo e l'informazione se il nodo è relativo ad un circuito di binario o ad uno scambio. Con itinerario si intende una sequenza lineare finita di nodi e per i nodi della sequenza di tipo scambio l'itinerario definisce la posizione richiesta allo scambio. Le informazioni presenti da fornire per ogni nodo di un itinerario sono ID dell'itinerario, indirizzo del precedente nodo dell'itinerario, indirizzo del seguente nodo dell'itinerario,

posizione del nodo nell’itinerario (First, Middle, Last) e se il nodo è uno scambio, la posizione richiesta per quell’itinerario.

1.3.2 Task b

Il *task di lettura sensori, esecuzione della logica di controllo, comando attuatori (b)* risulta essere uno dei task di maggior importanza all’interno del taskset. Il suo compito è quello di simulare la lettura dei sensori ed attuatore e processare i messaggi scambiati tra i nodi per implementare un algoritmo two-phase commit per stabilire un consenso tra nodi sulla possibilità di riservare un itinerario. Per fare tutto questo la logica di esecuzione compie ad ogni ciclo un passo della macchina a stati (vedi 1 e 2). Questo task ha inoltre il compito di avviare il positioningTask e far passare il nodo in uno stato fail-safe in caso di malfunzionamenti.

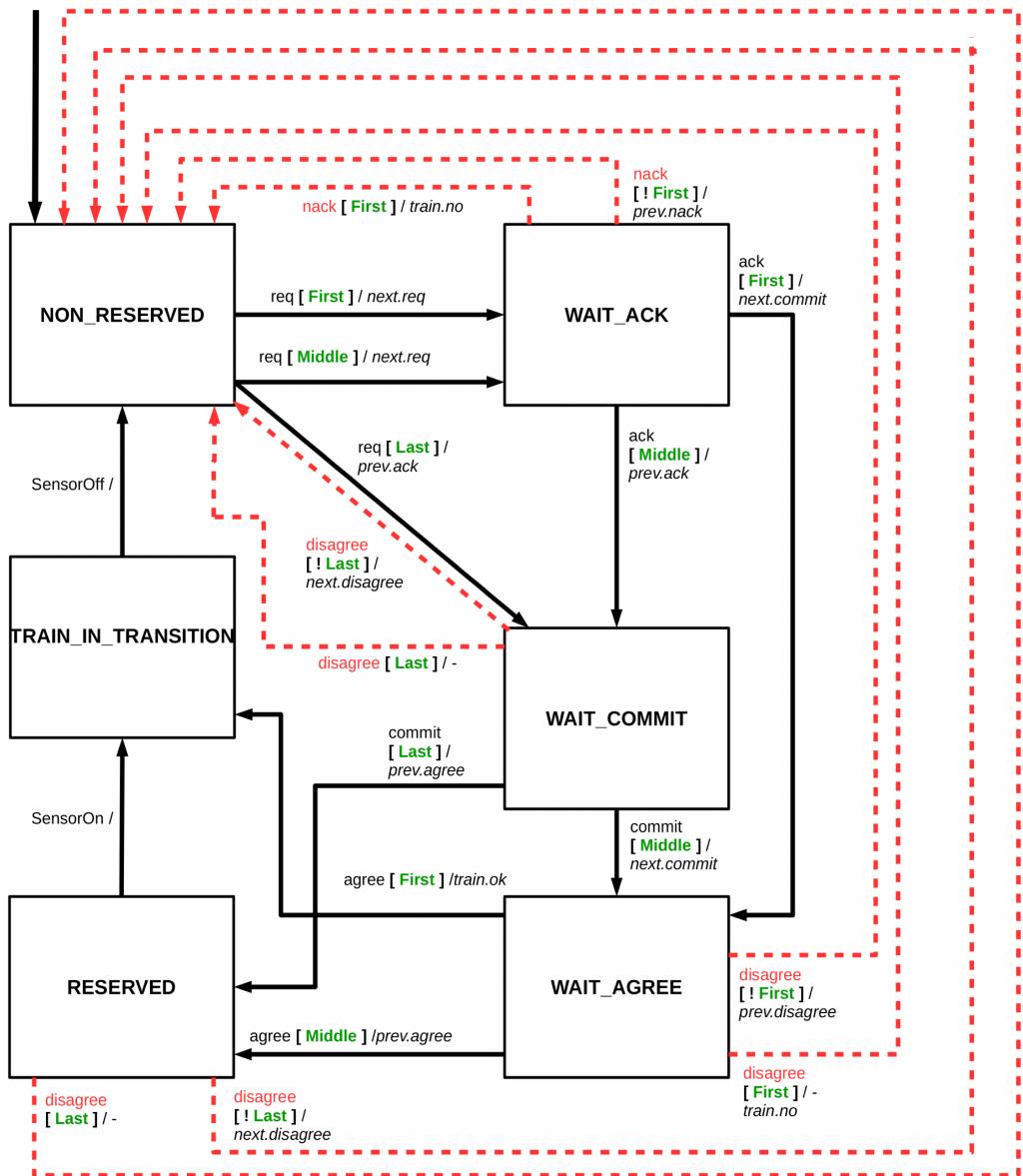
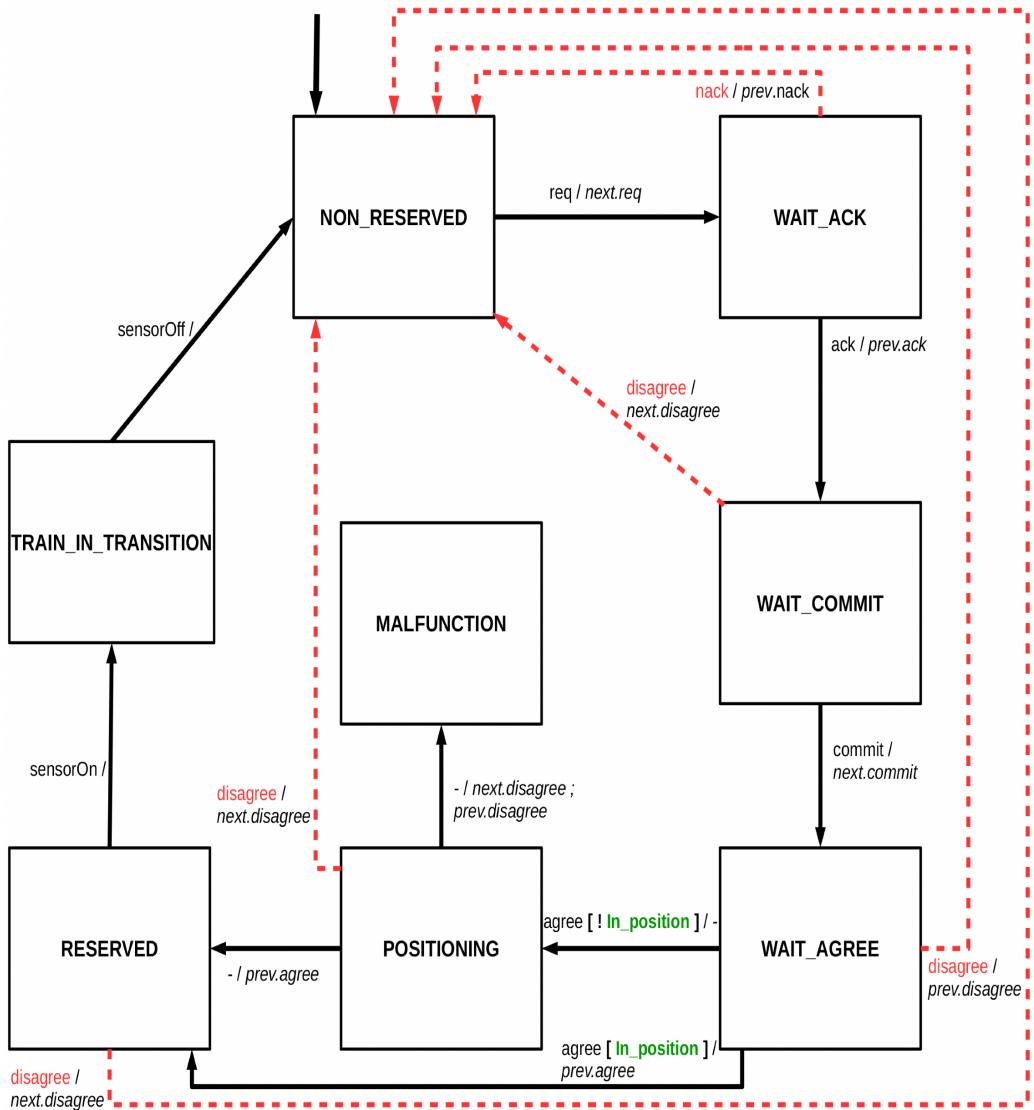


Figure 1: Macchina a stati circuito di binario



In_position := current_position = requested_position

Figure 2: Macchina a stati scambio

Legenda macchine a stati:

- Relativi a scambio di messaggi
 - *req* : messaggio di richiesta itinerario in arrivo
 - *ack, agree, disagree, commit*: messaggi del protocollo two phase commit
 - *next.XX*: invio del messaggio XX al prossimo elemento dell'itinerario
 - *prev.YY*: invio del messaggio YY al precedente elemento dell'itinerario
 - Tutti i messaggi sono da intendersi come *msg(id)*, cioè contengono l'identificatore di itinerario di riferimento
 - Tutti i messaggi contengono lo stesso id del messaggio req la cui ricezione fa passare la macchina dallo stato NON_RESERVED allo stato ACK
- Relativi a sensori

- *sensorOn*, *sensorOff*: sono rispettivamente gli eventi di occupazione e liberazione del nodo e sono simulati dalla commutazione di un interruttore.

Nota: le figure sono prese dal seguente paper [3], quindi i dettagli non sempre coincidono esattamente con quanto espresso nei requisiti.

1.3.3 Task c

Il *task di ricezione e invio messaggi* (*c*) ha lo scopo di ricevere i messaggi provenienti dall' host o da altri nodi sfruttando le interfacce di rete presenti sulla scheda. Ha inoltre l'incarico di inviare, sempre verso host o altri nodi, i messaggi necessari alla corretta esecuzione dei task in esecuzione nel nodo stesso. **Nota:** Nel nostro sistema ci siamo riferiti al task c indistintamente con i nomi *commTask* e *wifiTask*.

1.3.4 Task d

Il *task di simulazione posizionamento di uno scambio* (*d*) ha lo scopo di simulare il comportamento del deviatoio. L'esecuzione di questo task infatti va ad incrementare la variabile che indica la posizione simulata che passa da un minimo (detto posizione normale o diritta) ad un massimo (detto posizione rovescia o deviata) o viceversa. Il task dovrà inoltre poter essere interrotto a simulare un fallimento del motore che pilota il deviatoio, il quale rimarrà in uno stato indeterminato (né normale, né rovescio).

1.3.5 Task e

Il *task di diagnosi* (*e*) viene eseguito durante gli intervalli di idle e consiste in un algoritmo di diagnosi delle comunicazioni tra nodi vicini. Se durante la diagnosi si verifica un timeout su una comunicazione, il task manda il nodo in uno stato fail-safe e invia una notifica all'host.

1.3.6 Task f

Il *task di log* (*f*) ha lo scopo di memorizzare, con il relativo timestamp, ogni evento di accettazione o di rifiuto di una richiesta e ogni evento di occupazione del nodo. Inoltre in caso di richiesta del log da parte dell'host dell'host dovrà inviare la lista degli eventi memorizzati.

1.4 Descrizione funzionamento Two-Phase Commit Protocol

Come illustrato in [3], il proposito fondamentale del protocollo è garantire che due treni non finiscano mai in una situazione pericolosa in cui entrambi occupano la stessa traccia. Questa situazione viene evitata richiedendo che i treni prenotino il loro percorso prima di attraversarlo. Per prenotare un percorso, il treno deve ottenere il consenso da tutte le componenti del tracciato che costituiscono il percorso.

Nel sistema descritto, la rete è composta da elementi di tracciato, costituiti da segmenti lineari e segmenti di scambio, tutti dotati di sensori e apparati che ne permettano la comunicazione (cablati o wireless).

Il concetto ruota intorno alla prenotazione dei percorsi da parte dei treni attraverso un protocollo di commit a due fasi che cerca di ottenere un consenso tra gli elementi che compongono un dato percorso. Inoltre è previsto un meccanismo di rilascio sequenziale per liberare gli elementi prenotati mentre vengono attraversati dal treno. Un protocollo di commit a due fasi è un algoritmo di consenso distribuito che coordina un insieme di processi

che partecipano tutti alla stessa transazione distribuita. Il protocollo aiuta a determinare se eseguire o annullare una determinata transazione in un insieme di processi distribuiti. Nel nostro caso, i processi corrispondono ai nodi nella rete ferroviaria e la transazione concordata è la prenotazione di un percorso per un determinato treno. Il protocollo richiede un coordinatore assegnato responsabile di avviare il commit (ossia il treno) e prevede due fasi di scambio di messaggi, ognuna composta da un messaggio di richiesta e un messaggio di risposta comunicato al coordinatore. In entrambi i casi i messaggi sono inviati in modo sequenziale a tutti i nodi facenti parte del percorso dato.

Un messaggio di richiesta viene propagato attraverso tutti i nodi partecipanti, e quando raggiunge l'ultimo nodo, questo inizierà un messaggio di risposta che verrà propagato nuovamente attraverso tutti i nodi partecipanti.

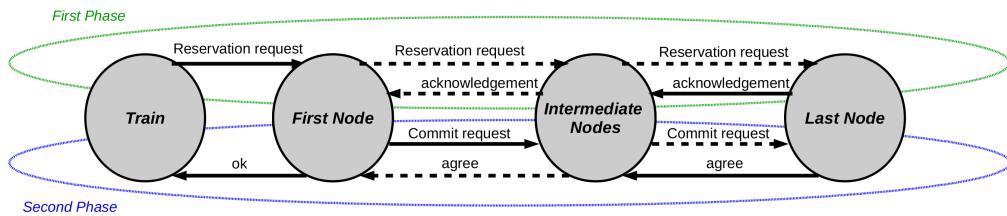


Figure 3: Schema del two-phase commit protocol [3]

La prima fase, definita "fase di votazione", inizia con il coordinatore che invia un messaggio di query che viene poi propagato attraverso la rete di nodi. Il messaggio di query iniziale conterrà i dettagli e i dati di ciò che deve essere eseguito. Se tutti i nodi sono d'accordo con la query, l'ultimo nodo inizierà un messaggio di conferma da propagare attraverso la rete. Se, tuttavia, uno dei nodi non è d'accordo con la query, il nodo in disaccordo invierà un messaggio di negazione da propagare nuovamente al coordinatore.

In questa fase, un nodo potrebbe scegliere di non essere d'accordo se rileva la presenza di un treno sul tratto di binario, che non è il treno che ha avviato la richiesta di prenotazione, oppure potrebbe dissentire semplicemente perché il nodo è già coinvolto in una richiesta di prenotazione avviata da un altro treno.

La seconda fase, definita "fase di completamento", è di solito avviata dal coordinatore dopo aver ricevuto il messaggio di conferma dalla prima fase. Nel nostro caso tuttavia, i nodi sono collegati in modo sequenziale e il treno coordinatore non ha nulla di nuovo da aggiungere alla comunicazione. Pertanto, il nodo immediatamente successivo al treno nella catena di comunicazione agirà come coordinatore.

Quando il primo nodo nella catena di comunicazione riceve un messaggio di conferma dalla prima fase, inizierà immediatamente una richiesta di commit, che informa i nodi di eseguire la transazione in attesa. Man mano che ogni nodo riceve la richiesta di commit, si preparerà ad eseguire la transazione. Se un nodo non soddisfa le condizioni per eseguire la transazione, invierà un messaggio di disaccordo che verrà propagato a tutti i nodi coinvolti nella transazione. Al ricevimento di un messaggio di disaccordo, i nodi che hanno già eseguito la transazione cercheranno di tornare allo stato precedente, mentre i nodi che devono ancora eseguire il commit abbandoneranno la transazione in sospeso. Il messaggio di commit farà sì che i deviatoi sul percorso siano posizionati secondo quanto richiesto. Un evento di disaccordo causerà semplicemente l'abbandono della prenotazione in sospeso da parte dei nodi, tornando a uno stato non prenotato che indica la disponibilità per nuove richieste.

Quando l'ultimo nodo del sistema riceve il messaggio di commit, inizierà un messaggio di accordo da propagare

nuovamente attraverso i nodi. Quando il primo nodo, immediatamente successivo al treno, riceve il messaggio di accordo, invierà un messaggio di OK al treno.

Questa implementazione di un interlocking distribuito porta con sé una serie di vantaggi e svantaggi. Ne illustriamo alcuni di seguito:

Vantaggi

- Migliorata affidabilità: se un nodo fallisce, gli altri nodi possono continuare a funzionare, riducendo il rischio di un completo fallimento del sistema e migliorando l'affidabilità complessiva.
- Ridondanza: la ridondanza può essere implementata nel sistema avendo duplicati o triplicati nodi per funzioni critiche, fornendo capacità di backup in caso di guasti.
- Tempi di risposta più veloci: la distribuzione dell'elaborazione su più nodi può portare a tempi di risposta più rapidi per operazioni ferroviarie critiche.
- Facilità di aggiornamenti: l'aggiornamento o la sostituzione dei singoli componenti può essere più semplice rispetto al trattare con un sistema monolitico.
- Scalabilità
- Riduzione dei tempi di manutenzione

Svantaggi

- Costi iniziali più elevati: implementare un sistema distribuito di interlocking richiede generalmente hardware e software più sofisticati, con conseguenti costi iniziali maggiori rispetto ai sistemi centralizzati tradizionali.
- Rischi per la sicurezza: con più punti di accesso e comunicazione, i sistemi distribuiti possono presentare più criticità
- Dipendenza da reti di comunicazione: i sistemi distribuiti dipendono pesantemente dalle reti di comunicazione per la comunicazione tra i nodi. Guasti o interruzioni della rete potrebbero compromettere il funzionamento del sistema.
- Maggiore complessità

1.5 Descrizione protocollo di comunicazione

Nel corso dello sviluppo si è reso necessario creare un'insieme di comandi che permettessero la comunicazione tra i singoli nodi. Per semplicità di implementazione e maggiore interpretabilità è stato scelto di utilizzare delle stringhe scambiate attraverso utilizzo di socket di comunicazione tra nodo ricevente e nodo destinatario. In particolare è stato utilizzato TCP basato su IPv4. Poiché diverse tipologie di messaggi avevano bisogno di un numero differente di campi aggiuntivi è stato utilizzato il simbolo punto e virgola (;) per andare a separare le diverse aree che componevano un singolo messaggio. Infine potendo arrivare più messaggi è stato necessario usare il simbolo punto (.) per dividere i diversi messaggi in arrivo. I messaggi possono essere divisi in due categorie:

- Messaggi di controllo
 - **REQ;n;m** : indica una richiesta di prenotazione da parte dell'host **n** per l'itinerario **m**
 - **ACK;n;m** : indica un messaggio di conferma di ricezione della richiesta
 - **COMMIT;n;m** : indica che il nodo è disponibile a riservare il suo percorso
 - **AGREE;n;m** : indica che il nodo è d'accordo a riservare il nodo e nel caso di scambio ha già eseguito il posizionamento
 - **NOT_OK** : indica un generico messaggio di disaccordo durante una qualunque fase del protocollo, che riporta il nodo nello stato **NOT_RESERVED**
 - **SENSOR_ON/OFF** : utilizzati per simulare in modalità remota la pressione dei bottoni che indicano la presenza o meno di un treno
- Messaggi di diagnostica
 - **PING_REQ** : indica che il nodo da cui si è ricevuto il messaggio è in diagnostica e attende risposta sullo stato di funzionamento del nodo ricevente
 - **PING_ACK** : indica che il nodo che ha inviato questo messaggio è in funzione e in grado di rispondere

1.6 Vincoli temporali e deadline

Un punto essenziale nello sviluppo del progetto è stato andare ad individuare e quantificare eventuali vincoli temporali da rispettare per garantire il corretto funzionamento dell'interlocking. Infatti poiché il sistema è legato a componenti fisiche come sensori ed attuatori è risultato necessario individuare il tempo massimo di esecuzione di alcune azioni. Mentre alcuni di questi vincoli erano presenti in modo esplicito nel documento dell'elaborato altri si sono dovuti stimare utilizzando valori di riferimento tipici nell'ambito ferroviario. In particolare:

- Da specifiche si ha che la procedura di posizionamento del binario attiva un motore che impiega 3 secondi per completare l'operazione,
- Il tempo necessario a riservare un tratto di linea (ovvero ad eseguire completamente un two phase commit protocol) può essere stimato intorno ad 1 secondo, nel caso tutto vada a buon fine e non ci sia bisogno di ritrasmissione massicce,
- Il tempo che intercorre da quando un treno riserva un tratto di linea a quando effettivamente ci passa (ovvero il tempo che ci si aspetta che passi da quando un nodo è nello stato **RESERVED** a quando si ha l'arrivo del treno con **sensorOn**) è assai variabile in quanto dipende dalla velocità del treno. Abbiamo a grandi linee due casi:
 - Il mezzo potrebbe trovarsi fermo ad un segnale che protegge l'itinerario, in attesa della conferma di disponibilità del tratto successivo. In questo caso avrebbe da percorrere lo spazio che separa il segnale dal track circuit di cui si considera il sensore, potenzialmente più di 1 km. Partendo da fermo si ha che un treno percorre una tale distanza ad una media tra 40 km/h e 100km/h, ma tale velocità potrebbe essere ancora più bassa. Questo comporta nel caso più stringente un intervallo di circa $\frac{1 \text{ km}}{100 \text{ km/h}} = 36$ secondi,

- Il mezzo può raggiungere il segnale a velocità ancora superiore, ma questo solo nel caso il segnale sia diventato verde con largo anticipo. In particolare poiché dovrà essere garantita la possibilità di arresto nel caso il segnale non indichi in tempo la disponibilità del tratto il verde dovrà scattare quando il treno si trova ad una distanza di almeno 2 km. Quindi ipotizzando anche una velocità doppia, trascorreranno sempre circa $\frac{2 \text{ km}}{200 \text{ km/h}} = 36$ secondi prima che il treno raggiunga il sensore.
- Il tempo necessario a percorrere completamente un tratto di linea (ovvero per far passare da sensorOn a sensorOff) dipende anche esso dalla velocità del treno e dalla lunghezza di quel particolare circuito di binario o scambio. In particolare analizzando i deviatoi maggiormente utilizzati nell'ambito ferroviario italiano [4] è possibile stimare un range di tempi di percorrenza del ramo deviato, tra questi il valore minimo è circa 2.2 secondi. Stimando una velocità di percorrenza di 300 km/h per il ramo di corretto tracciato e una lunghezza dello scambio di 100 m si ottiene invece un tempo di percorrenza di circa $\frac{0.1 \text{ km}}{300 \text{ km/h}} = 1.2$ secondi. Infine supponendo un circuito di binario di soli 100 m e utilizzando la velocità indicata precedentemente si ottiene ancora il valore di 1.2 secondi. E' necessario quindi che il sistema riesca a verificare e gestire il passaggio di un treno in un tempo minore di questo appena indicato e ciò è garantito dal fatto che il task di controllo, che contiene al suo interno questo ruolo, rispetti tale deadline (vedi 5.5).

Type	Radius [m]	Tan	Lenght [m]	Speed [km/h]	Time [s]
S 50 UNI	170	0.12	23.99	30	2.879
S 50 UNI	245	0.1	30.29	30	3.635
S 60 UNI	170	0.12	25.08	30	3.010
S 60 UNI	250	0.12	29.84	30	3.581
S 60 UNI	400	0.074	39.08	60	2.345
S 60 UNI	400	0.094	38.02	60	2.281
S 60 UNI	1200	0.04	73.67	100	2.652
S 60 UNI	1200	0.055	69	100	2.484
S 60 UNI	3000	0.034	109.83	160	2.471
S 60 UNI	3000	0.033	132	160	2.970
S 60 UNI	6000	0.015	180	220	2.945

Figure 4: Deviatoi maggiormente usati nelle ferrovie italiane

- Il tempo che intercorre tra la prenotazione di uno stesso itinerario nella stessa direzione può essere stimato superiore ad almeno 120 secondi. Invece per treni che percorrono l'itinerario in direzioni opposte la prenotazione può anche avvenire in contemporanea.

Utilizzando tutte queste informazioni è stato possibile creare delle deadline nel modello, da essere utilizzate in fasi successive per la verifica del rispetto dei tempi di esecuzione del software eseguito sulle schede raspberry.

1.7 Fasi costitutive del processo di sviluppo

Il progetto è stato svolto suddividendo le sue fasi di sviluppo in diverse parti:

- A partire dai requisiti richiesti, è stato inizialmente necessario creare un primo modello del sistema attraverso una Petri Net utilizzando Oris Tool 1.0 [2]

- Abbiamo successivamente individuato e quantificato a partire da stime dei tempi tipici nell'ambito ferroviario eventuali deadline da rispettare.
- Abbiamo proceduto poi a delineare un'architettura di rete e un'architettura hardware in grado di supportare le funzionalità necessarie.
- Si è proseguito in seguito a mettere a punto un protocollo di comunicazione in grado di rispettare le specifiche.
- Successivamente si è proceduto ad implementare le funzionalità richieste su schede Raspberry Pi 4B e sistema operativo real-time VxWorks 7. Tale implementazione è stata scritta in linguaggio C andando a caricare i moduli sviluppati direttamente nel kernel eseguito dalle schede.
- Una volta arrivati ad avere un prototipo del sistema abbiamo testato l'effettiva logica di esecuzione, andando ad estrarre i tempi con cui venivano eseguiti i diversi task.
- Grazie ai tempi d'esecuzione estrapolati abbiamo potuto verificare il rispetto della feasibility del nostro modello e abbiamo inoltre controllato anche il rispetto delle deadline proposte. Tutto questo è stato possibile utilizzando lo strumento Petri Net Simulator del Tool Oris.
- Infine abbiamo riflettuto e delineato quali potrebbero essere eventuali sviluppi futuri per portare avanti il progetto.

Nota: C'è da tenere presente che le fasi appena descritte sono state eseguite con una certa sequenzialità, tuttavia il presentarsi di nuove criticità ha portato a tornare a modificare quanto progettato in punti precedenti dello sviluppo. Questa continua procedura di miglioramento ha permesso di andare a sviluppare un sistema maggiormente performante e con ancora più funzionalità.

Il progetto è costituito nella sua interezza da circa 3900-4000 righe di codice suddivise in approssimativamente 35 file utilizzando linguaggi C, Python e Bash.

2 Modellazione

2.1 Introduzione

Nell'era moderna del trasporto ferroviario, l'efficienza, la sicurezza e la precisione rivestono un ruolo fondamentale per garantire il flusso continuo delle operazioni. Al fine di raggiungere questi obiettivi ambiziosi, è essenziale sviluppare sistemi di interlocking avanzati e sofisticati. L'obiettivo fondamentale del nostro progetto è stato quello di affrontare le sfide intrinseche legate alla sincronizzazione, alla temporizzazione e alla gestione delle comunicazioni in un sistema complesso come quello di interlocking distribuito in esame. Nel contesto di questo progetto, ci siamo concentrati su una modellazione innovativa di tale sistema basata su Preemptive Timed Petri Net (PTPN).

L'obiettivo primario di questo progetto è stato quello di creare un modello che consenta una rappresentazione accurata e dettagliata di un sistema ferroviario, includendo sia le componenti fisiche, come i treni, i binari e gli scambi, sia le componenti di comunicazione che svolgono un ruolo cruciale nell'orchestrazione delle operazioni. Questo approccio ha richiesto una profonda analisi e un'attenta progettazione, in quanto si è rivelato necessario affrontare sfide significative legate alla gestione dei tempi di firing delle transizioni, all'elaborazione dei messaggi e alla complessa interazione tra task di natura diversa.

Tra le sfide principali che abbiamo affrontato, spicca la gestione dei tempi di firing delle transizioni. Nel nostro sistema, i tempi di esecuzione delle varie transizioni differiscono di ordini di grandezza notevoli, a causa della rappresentazione dettagliata delle diverse componenti coinvolte. Ad esempio, i tempi associati al passaggio di treni o posizionamento degli scambi sono radicalmente diversi rispetto ai processi di comunicazione e ciò ha richiesto una meticolosa sincronizzazione per garantire il corretto funzionamento del sistema nel suo complesso. Un ulteriore elemento chiave di sfida è stata la volontà di rappresentare in modo estremamente accurato gli scambi di messaggi all'interno del sistema. Non ci siamo limitati a modellare la semplice richiesta iniziale di prenotazione di itinerari, ma abbiamo voluto descrivere finemente tutto il complesso flusso di messaggi scambiati tra i vari componenti del sistema. In questo modo è stato possibile comprendere meglio le interazioni tra i vari nodi e avere una rappresentazione assai più realistica e vicina al comportamento effettivo.

Infine, va sottolineata la complessità intrinseca dei task che compongono il nostro sistema. Si ha infatti che i compiti che l'interlocking deve eseguire sono molti e soprattutto si ha che questi task interagiscono profondamente tra loro, aggiungendo ulteriori livelli di complessità alla progettazione e all'implementazione del modello. In questo documento introduttivo, esploreremo nel dettaglio il processo di modellazione utilizzato per affrontare queste sfide e giungeremo a una comprensione approfondita dei risultati ottenuti. L'obiettivo finale è quello di fornire una soluzione innovativa e robusta per la gestione delle infrastrutture ferroviarie, apendo la strada a ulteriori sviluppi nel campo dei sistemi di controllo distribuito.

2.2 Transizioni di switch e di preselection

Nel contesto del nostro modello di sistema di interlocking distribuito basato sulla Preemptive Timed Petri Net (PTPN), vi è stata la necessità di affrontare situazioni in cui, durante l'esecuzione, si possono presentare diverse opzioni o percorsi possibili, ognuno dei quali rappresenta un comportamento differente del sistema. Per catturare questa varietà di possibili percorsi, abbiamo introdotto il concetto di "transizioni di switch".

Le transizioni di switch sono un meccanismo all'interno del modello che rappresenta il momento in cui, durante

l'esecuzione, il sistema ha la possibilità di prendere una decisione tra diverse opzioni. Queste transizioni sono organizzate in gruppi e sono rese "enabled" da un gettone posizionato in un posto comune. L'idea qui è che la presenza del gettone indica che il sistema è giunto a un punto in cui una scelta deve essere fatta, e quindi tutte le transizioni all'interno del gruppo diventano potenzialmente eseguibili.

Tuttavia, c'è un'importante restrizione: solo una delle transizioni all'interno del gruppo di switch può effettivamente essere eseguita. Questo elemento di causalità rappresenta il fatto che, quando si raggiunge un certo punto dell'esecuzione, si può decidere di seguire solo un particolare percorso rispetto agli altri e tale scelta effettiva è influenzata da altri fattori. In sostanza, le transizioni di switch catturano il concetto di biforcazione o di decisione all'interno del nostro modello e rappresentano situazioni in cui il sistema ha la flessibilità di adottare diverse azioni a seconda di alcune condizioni del momento.

Procedendo con lo sviluppo abbiamo osservato che assegnando stesse risorse e priorità a queste transizioni di switch, il sistema eseguiva un percorso troppo prevedibile e deterministico. In pratica, sempre la stessa transizione all'interno del gruppo di switch veniva eseguita, portando a un risultato costante e non casuale. Questo andava contro la nostra intenzione di catturare il reale processo decisionale del sistema in situazioni simili.

Abbiamo quindi notato che, all'interno del tool Oris 1.0, le transizioni di switch erano eseguite in un ordine basato sugli ID, e che la transizione con l'ID minore veniva sempre eseguita per prima. Questo, a sua volta, era collegato al modo in cui le transizioni venivano disposte nell'editor visuale, dove la transizione con l'ID più basso era la prima posizionata nel tool. Questo spiegava il comportamento deterministico che stavamo riscontrando. Per affrontare questo problema e reintrodurre la possibilità di eseguire una qualunque nelle transizioni di switch, abbiamo introdotto una "transizione di preselezione" prima di ogni transizione che componeva lo switch. Poiché queste transizioni, a differenza delle altre, non richiedevano l'uso di risorse si è venuta a creare un'interessante dinamica: le transizioni di preselezione fungevano da "innesco", rompendo l'ordine deterministico delle esecuzioni delle transizioni di switch.

In definitiva, la nostra strategia di introdurre le transizioni di preselezione unite a quelle di switch ci ha permesso di ottenere un comportamento in cui diversi percorsi d'esecuzione erano possibili. Questo ha reso il nostro modello più fedele alla realtà delle decisioni complesse che il sistema di interlocking deve affrontare.

2.3 Rappresentazione dei messaggi

Un punto essenziale del modello è stato rappresentare lo scambio di messaggi all'interno del sistema di interlocking. Per gran parte dello sviluppo era stato pensato di modellare un singolo messaggio come un gettone che si muoveva all'interno della rete. Questa scelta era stata fatta in quanto cercava di rispecchiare l'implementazione del codice, dove venivano usate le code di messaggi per rappresentare l'accumulo e lo scambio di messaggi. Quindi nella rete erano presenti particolari posti che andavano a simulare questo accumulo di messaggi ricevuti o da inviare. Questo approccio tuttavia ha portato dei problemi piuttosto rilevanti:

1. Poiché abbiamo modellato la rete con transizioni di switch (punti in cui il flusso di esecuzione può prendere percorsi diversi) se esiste uno di questi percorsi in cui il gettone della coda non viene consumato questo può portare all'accumulo.
2. Poiché i task sono ciclici è stato deciso di rappresentarli in Petri come una serie di transizioni che tornano nello stato idle (ovvero creano un loop). Tuttavia se c'è la possibilità di accumulare anche solo un gettone si ha che l'analizzatore non riuscirà a concludere l'analisi poiché il numero di gettoni tenderà ad infinito.

Un esempio di questo comportamento può essere osservato nella rete di seguito che riassume nel modo più semplificato possibile il funzionamento del nostro modello a quello stato di sviluppo. In questa rete abbiamo:

- t_0 è il generatore di messaggi
- t_3 è il consumatore di messaggi
- t_1 è un'altra transizione che porta ad un percorso in cui si fa un'azione che non consuma il messaggio

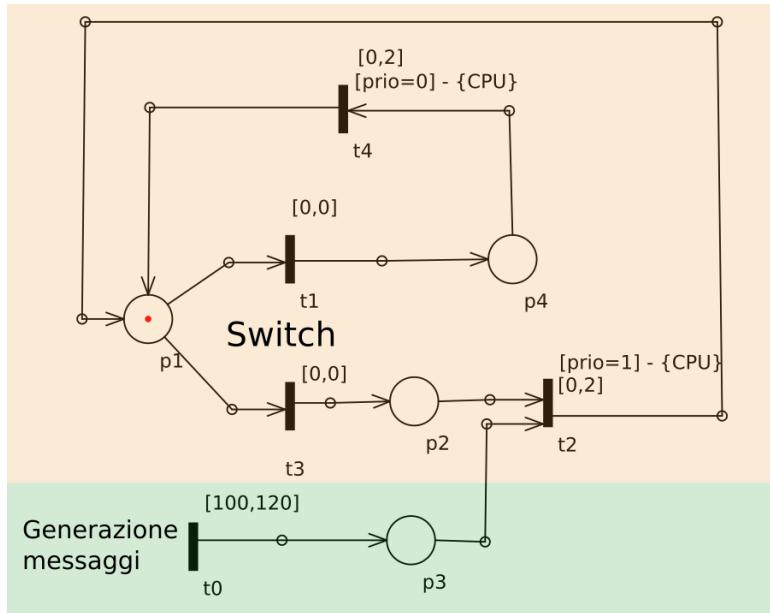


Figure 5: Rete semplificata in cui sono presenti in contemporanea gettoni per rappresentare messaggi e switch

In questa rete è presente tra le transizioni di switch una che non consuma un messaggio la quale porta al verificarsi del problema 1), inoltre poichè sono presenti cicli in cui si accumulano gettoni si ha la presenza anche del problema 2). Tutto ciò comporta un'esplosione dello spazio degli stati rendendo impossibile proseguire nell'analisi di tale sistema.

Studiando e testando la rete abbiamo capito che i problemi 1) e 2) si verificavano perché volevamo rappresentare in contemporanea sia l'astrazione delle code con un gettone visibile, sia l'astrazione degli switch di esecuzione. Poichè l'uso degli switch ha il grande vantaggio di rappresentare molto finemente quello che avviene nella rete, portando comunque ad una semplificazione delle transizioni necessarie a rappresentare il sistema, siamo arrivati a trovare il modo di utilizzare il costrutto degli switch per rappresentare anche l'esistenza delle code. Questo è stato possibile andando ad aggiungere ulteriori percorsi d'esecuzione all'interno degli switch. Inoltre questa scelta ha avuto anche il vantaggio di ricalcare maggiormente l'implementazione, creando una maggiore simmetria con il codice. Per questa serie di motivi abbiamo deciso di abbandonare la precedente rappresentazione delle code con i gettoni e optare per questa nuova soluzione che utilizzava solo gli switch.

2.4 Condivisione della memoria

Per garantire un'ottimale svolgimento delle operazione del sistema è stata necessaria un'orchestrazione dei task presenti. In particolare un punto critico del progetto è stato individuare un modo per consentire lo scambio di informazioni tra i vari task in esecuzione. Per questo scopo sono stati utilizzati all'interno del codice due

meccanismi diversi quali semafori a mutua esclusione e code di messaggi. Entrambi questi design sono forniti come primitive sviluppate da VxWorks e presenti nelle librerie `semMLib` e `msgQLib`. Per quanto riguarda le code è stata utilizzata l'opzione `MSG_Q_PRIORITY` che permette di impostare un accesso alla struttura basato sulla priorità del task che richiede l'accesso e per i semafori è stata utilizzata l'opzione `SEM_Q_PRIORITY` per avere un comportamento analogo. Per evitare inversioni di priorità è stato implementato un protocollo di accesso alle risorse condivise che consiste nell'innalzamento della priorità, acquisizione della risorsa, esecuzione dell'operazione, rilascio della risorsa e infine abbassamento della priorità. L'opzione `SEM_INVERSION_SAFE` permette un accesso senza inversioni di priorità, tuttavia per garantire maggiore flessibilità e possibilità di test abbiamo deciso di implementare manualmente un protocollo di priority inheritance. Tale protocollo assicura che un task che possiede una risorsa venga eseguito con la priorità del task a più alta priorità che può competere per accedere a quella risorsa. Una volta innalzata la priorità del task, questa rimane al livello superiore fino a quando tutti i semafori di mutua esclusione di cui il task è proprietario non vengono rilasciati; a quel punto il task torna alla sua priorità normale. In questo modo, il task "ereditante" è protetto dalla prelazione di qualsiasi task a priorità intermedia.

La rappresentazione all'interno della rete di Petri di un semaforo di mutua esclusione è stata piuttosto diretta ed è consistita nel creare un posto in cui è presente al più un gettone, tuttavia non potendo utilizzare enabling function e update function la modellazione delle code dei messaggi è stata più complicata. Per questo motivo per rappresentare l'accesso concorrente a tale struttura abbiamo usato una combinazione di posti a mutua esclusione, per indicare l'utilizzo della coda in tale momento da un task, e di transizioni di switch, per modellare la presenza/assenza di un messaggio in un momento preciso dell'esecuzione.

2.5 Task non rappresentati nella rete

All'interno della rete di Petri non vengono mostrati due task essenziali per il corretto funzionamento del sistema: non sono presenti infatti in modo esplicito il task di inizializzazione, il task di log e il task che gestisce i timer. I motivi per queste scelte progettuali sono i seguenti:

- Per quanto concerne l'initTask, è importante sottolineare che questo task è attivo esclusivamente durante la fase di accensione del sistema, rappresentando così una fase transitoria. Una volta completato il processo di setup, l'initTask termina la sua esecuzione. In altre parole, il suo ruolo nell'ambito dell'interlocking si limita a pochi secondi. Inoltre, è essenziale notare che tale task non interagisce direttamente né con i meccanismi di prenotazione di itinerari, né tanto meno con il funzionamento del sistema di controllo e sicurezza. Questi compiti verranno attivati successivamente, una volta che la fase di setup sarà conclusa con successo. Di conseguenza, la configurazione della rete implicitamente modella il comportamento del sistema una volta che la fase di setup è stata completata con successo. Nel caso si verifichi una futura necessità di modellare eventuali problemi durante la fase iniziale di setup, è possibile farlo in modo agile attraverso il seguente approccio: inizialmente, si rimuovono tutti i gettoni dalle posizioni corrispondenti nel modello proposto e successivamente si introduce un nuovo posto con un gettone che attiva uno switch a due transizioni. La prima transizione metterà un gettone in tutte le posizioni in cui è già presente un gettone" nel modello originale (ad esempio, posti di stato idle e semafori). La seconda transizione, al contrario, porterà il sistema in uno stato di fallimento. Poiché sono stati rimossi tutti i gettoni dalle posizioni in cui erano nel modello originale, ciò comporterà l'impossibilità di attivare ulteriori transizioni,

portando il sistema in uno stato non operativo. Questa strategia proposta consente, se ce ne sarà bisogno in futuro, di modellare efficacemente possibili problematiche nella fase iniziale di setup, dando al modello la flessibilità di riflettere scenari complessi e aspetti transitori del funzionamento del sistema.

- Per quanto riguarda il logTask e il timerTask si ha che una volta avviato il sistema essi si trovano per la maggior parte del tempo nello stato **SUSPEND** non occupando quindi nessun core della CPU. In particolare essi vengono utilizzati per loggare le transizioni degli altri task, di conseguenza potremmo includere il loro "peso" nel modello andando a considerare l'errore di misurazione dei tempi che introducono (più in dettaglio in: [5.2](#))

2.6 Utilizzo delle risorse

La scheda Raspberry Pi 4B è dotata di un processore ARM quad-core e lo scheduler di VxWorks è in grado di sfruttare appieno tutti questi core computazionali. Di default si ha che ogni task viene inizialmente assegnato in modo casuale ad uno dei core ed ogni volta che un task passa dallo stato **SUSPEND** allo stato **READY** si ha una nuova riassegnazione in base allo stato del sistema. Questo dinamicità è un meccanismo che permette di ottimizzare l'utilizzo di tutti i core, tuttavia non è modellabile in Oris in quanto in tale editor le risorse vengono assegnate in modo statico. Per questo motivo è stato deciso di riservare un singolo core per eseguire tutti i task rappresentati all'interno del sistema e questo è stato possibile grazie all'utilizzo della primitiva `taskCpuAffinitySet()`. In particolare è stato scelto il core 3 in quanto abbiamo notato che gli altri core venivano maggiormente impiegati già dai task di sistema necessari al funzionamento di VxWorks stesso. Questa scelta ha comunque permesso di rispettare con ampio margine le deadline imposte per i vari task del sistema, portando inoltre ulteriori vantaggi. In primo luogo si è riuscito a sfruttare maggiormente un singolo core, lasciando gli altri tre disponibili per sviluppi futuri di nuove funzionalità. Oltre a ciò l'utilizzo di un singolo core ha permesso di diminuire molto lo spazio degli stati (siamo infatti passati da circa 60mila stati a circa 30mila stati) in quanto il numero di combinazioni possibili utilizzando più risorse è sicuramente maggiore di quello ottenuto con l'impiego di una sola risorsa.

2.7 Disegno Rete

In questa rete vengono rappresentati quattro task diversi:

- ctrlTask: è il task presente nella parte alta della rete. Consiste di una prima serie di posti e transizioni che modellano la presenza/assenza di un treno sul binario, successivamente è presente la modellazione dello stato di esecuzione del diagTask e delle conseguenti scelte di esecuzione da parte del ctrlTask (avviare diagnostica, segnalare problemi di comunicazione con i nodi vicini, continuare l'esecuzione in attesa di completamento della diagnostica, etc...). Abbiamo poi una serie di posti e transizioni che implementano la lettura di un eventuale messaggio presente nella coda dei messaggi scambiati tra commTask e ctrlTask e una processazione di tale messaggio. Infine abbiamo una modifica dello stato del nodo (con eventuale attivazione del posiTTask) e la scrittura da parte del ctrlTask di un messaggio di risposta che verrà inviato dal commTask. Durante i possibili flussi di esecuzione si ha sempre un ritorno in idle a cui seguirà un'attesa prima del nuovo rilascio.
- commTask: è il task presente nella parte centrale della rete. Consiste di una prima serie di posti e

transizioni che modellano l’interazione con il task di diagnostica per controllare lo stato di avanzamento di tale procedura. Successivamente si ha la possibilità di processare un messaggio e renderlo disponibile per il ctrlTask, rispondere ad eventuali messaggi che non richiedono la supervisione del ctrlTask o semplicemente proseguire perchè non sono presenti messaggi. Le prime due azioni possono essere eseguite in modo ciclico, mentre l’esecuzione della terza avviene una sola volta. Il task prosegue poi a verificare se il ctrlTask ha inserito dei messaggi da inviare all’interno della coda di messaggi che permette la comunicazione tra i due task, nel caso sia presente un messaggio procede ad inviarlo prima di tornare in idle ed attendere il nuovo rilascio.

- diagTask: è il task presente nella parte in basso della rete. Consiste di una prima serie di posti e transizioni che modellano un’iniziale interazione con il commTask per segnalare l’avvio della diagnostica, un’attesa per permettere ai nodi vicini di rispondere alla diagnostica e una successiva nuova interazione con il commTask per verificare quanti nodi vicini sono attivi e segnalare la terminazione del tempo massimo di risposta. Successivamente si ha una comunicazione con il ctrlTask per indicare l’esito della diagnostica e in seguito a questo il diagTask termina.
- posiTTask: è il task presente nella parte centrale della rete, spostato leggermente a destra. Questo task viene avviato dal ctrlTask e rimane attivo per il tempo necessario a complementare lo spostamento del deviatoio. Completata tale operazione tale task segnala l’esito finale e riattiva il ctrlTask.

Nella figura sono evidenziati, oltre ai quattro task descritti, ulteriori posti e transizioni che hanno lo scopo di modellare azioni eseguite su zone di memoria condivise, come semafori, variabili globali e code di messaggi.

Distributed Railway Interlocking System PTPN - V 5.2

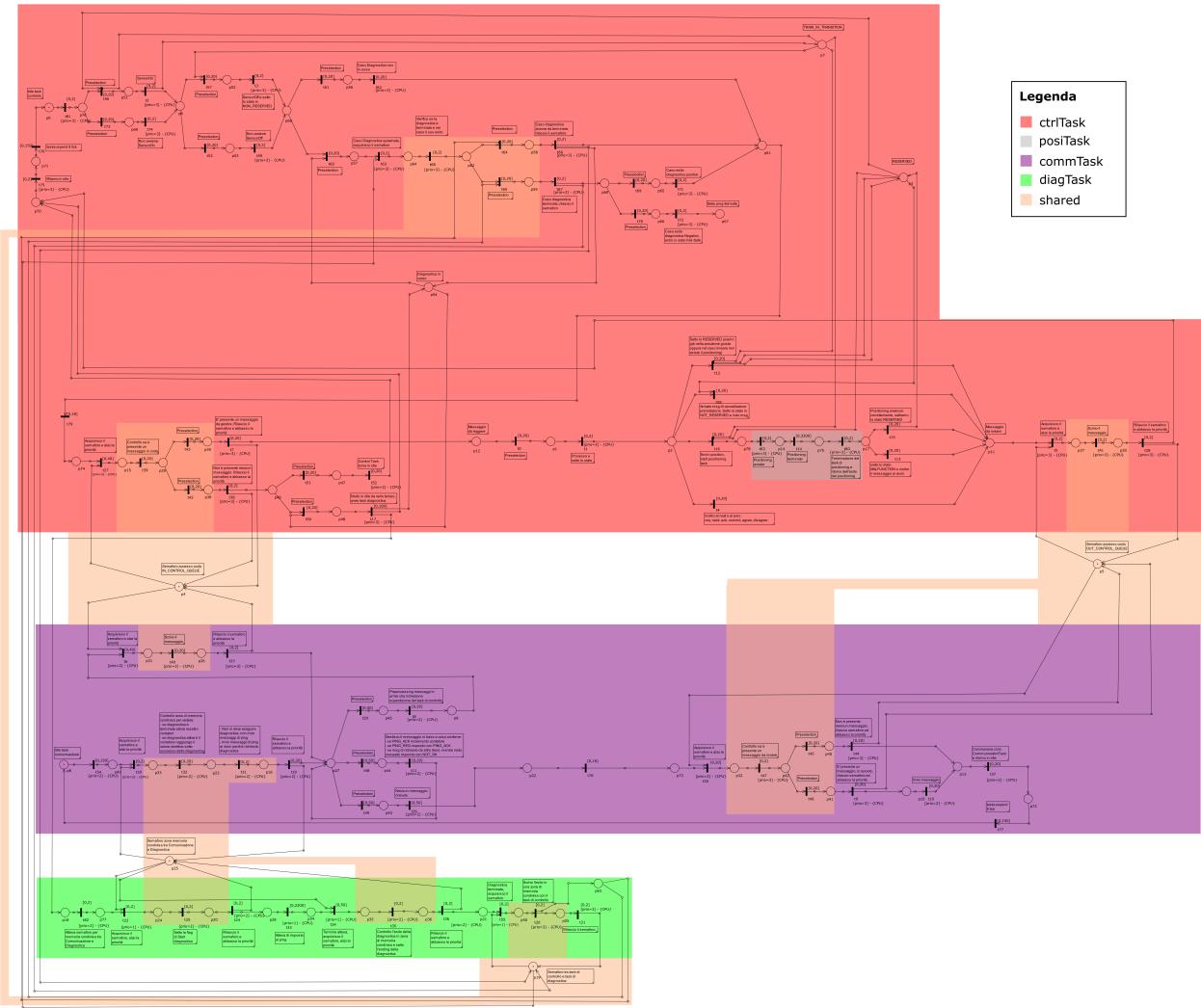


Figure 6: Disegno finale della rete modellata attraverso Oris

Nota: i latest firing time assegnati alle transizioni sono stati inseriti dopo aver analizzato i reali tempi impiegati dal sistema per completare tali azioni. Dai log d'esecuzione infatti è stato possibile estrarre una statistica dei tempi massimi d'esecuzione.

3 Codice

3.1 Modulo kernel

In questa sezione verrà descritto il codice che una volta compilato andrà a comporre il modulo kernel da caricare sui raspberry. I file header e i sorgenti si trovano rispettivamente in `/Interlocking_system/includes` e in `/Interlocking_system/src`.

3.1.1 initTask (Task a)

Come discusso nelle fasi iniziali del progetto per semplificare lo sviluppo di questo primo prototipo di sistema è stato deciso di optare su itinerari statici, ovvero composti da specifici nodi fissi e non modificabili successivamente. Inoltre si è scelto che tutte le informazioni su come sono organizzati gli itinerari sono mantenute solo dall'host (e quest'ultimo ha un indirizzo IP statico). Per questo motivo ogni nodo all'accensione ha bisogno per prima cosa di interagire con l'host, il quale fornisce informazioni su dove si trova il nodo nei vari itinerari, quali sono i codici identificativi e gli indirizzi IP dei nodi adiacenti. Per instaurare questa comunicazione viene aperto un socket tra l'host e ogni nodo.

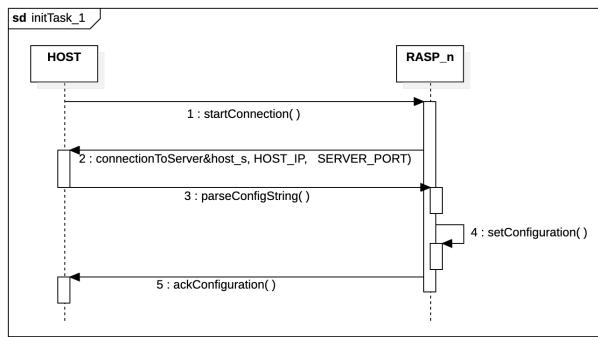


Figure 7: initTask fase preliminare

Finita la prima parte di configurazione si ha due successive fasi:

1. In una prima fase ogni nodo agisce come una sorta di "*client*" e cerca di aprire un socket verso tutti i nodi che sono suoi predecessori negli itinerari. Una volta riuscito ad aprire una connessione il nodo invia un messaggio per indicare la sua presenza e attende la risposta dell'altro nodo. Se tutto va a buon fine il nodo aggiunge questa connessione all'array di connessioni attive e procede a passare alla seconda fase.
2. Nella seconda fase ogni nodo agisce invece come una sorta di "*server*" e cerca di aprire un socket con i nodi che si trovano nella posizione successiva di ogni itinerario che interessa il nodo. Una volta confermata la connessione con tutti i nodi successivi si ha che il nodo notifica il completamento ai nodi precedenti. Questa procedura è leggermente diversa nel caso di nodi di terminazione: non avendo nodi successivi si ha che un nodo di terminazione può direttamente comunicare la terminazione della configurazione ai suoi predecessori.

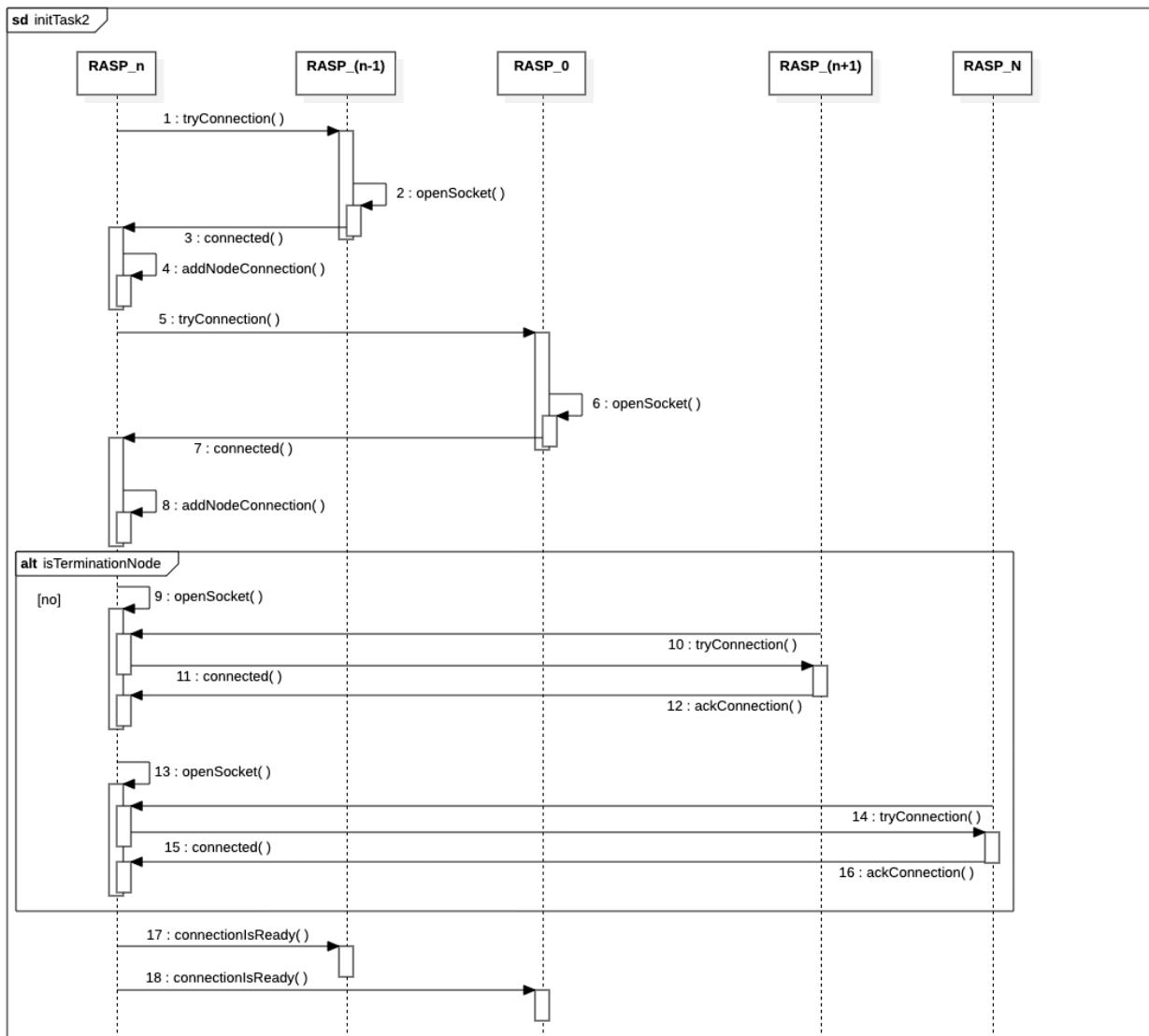


Figure 8: initTask fase successiva

Nota: per come è strutturata la configurazione si ha che i primi nodi a passare allo stato di "server" sono quelli vicini all'host che infatti si comporta come primo "server". A partire da questo primo livello di nodi la configurazione si propaga ai nodi direttamente successivi e così via fino ad arrivare ai nodi di terminazione. Questi nodi saranno invece i primi a informare i precedenti del successo della configurazione. Tale conferma di completamento avverrà "all'indietro" fino ad arrivare all'host.

Una volta completata la configurazione l'"initTask" avrà anche il compito essenziale di creare tutti i semafori e le code necessarie al funzionamento del sistema. Esso dovrà inoltre procedere a spawnare il task di comunicazione e quello di controllo prima di terminare.

3.1.2 controlTask (Task b)

Il task di controllo viene abilitato dall'initTask in seguito alla configurazione dei nodi ed è uno dei processi fondamentali tra quelli attivi nel Raspberry. Esso infatti, tra i vari compiti, ha come ruolo primario quello di comunicare con il wifiTask per analizzare e gestire i messaggi relativi al **two-phase-commit-protocol**.

Il suo diagramma di flusso è il seguente :

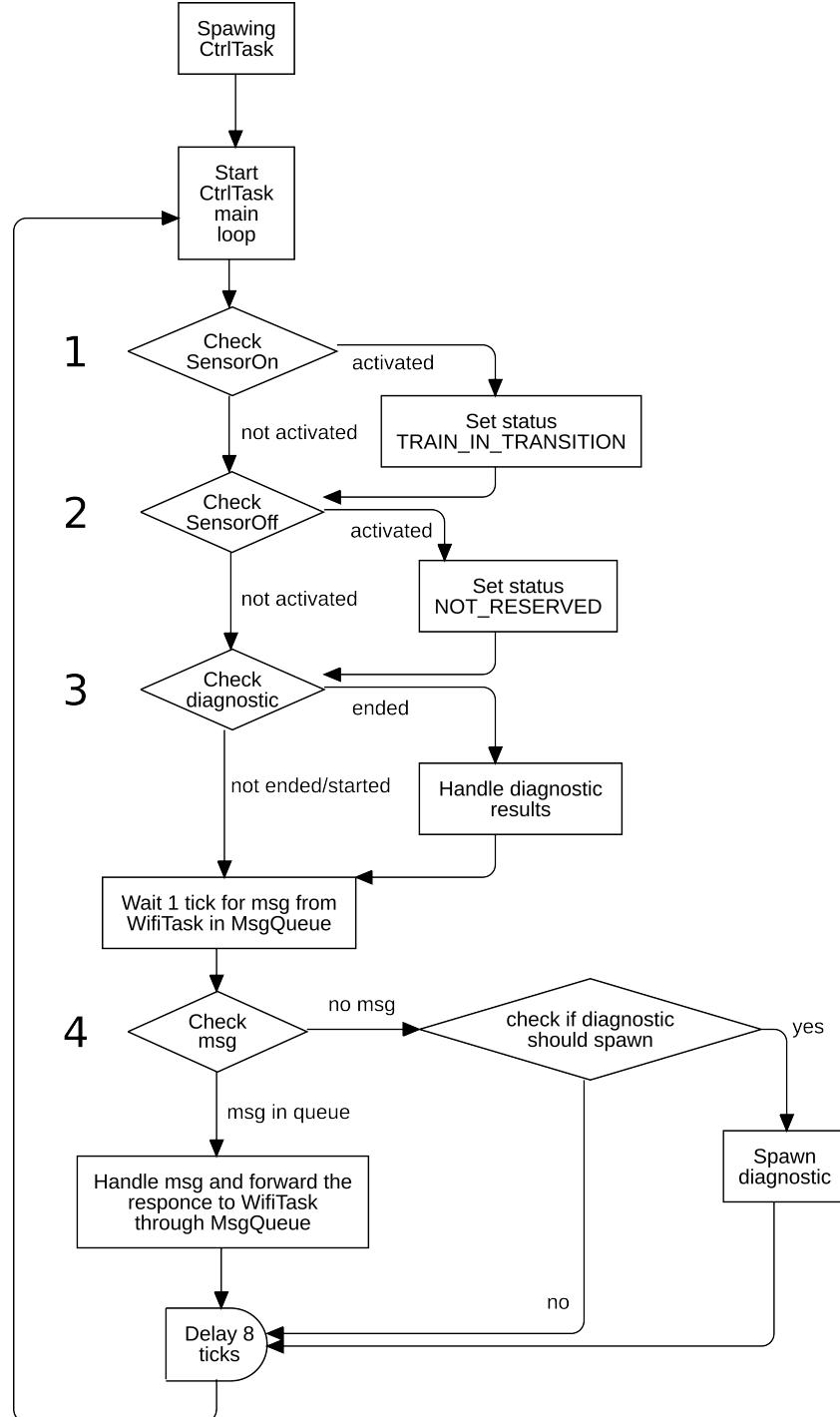


Figure 9: Control task flow diagram

Come è visibile in figura 9 si tratta di un task periodico le cui attività principali sono così descrivibili :

1. **Controllo del sensore per il SensorOn.**

2. **Controllo del sensore per il SensorOff.**

In entrambi i casi precedenti, in accordo con lo stato attuale del nodo all'interno del protocollo, avviene una lettura del valore di un bottone tramite l'utilizzo della libreria `vxbGpioLib.h`. Questa permette l'utilizzo delle interfacce gpio dei raspberry ed è stata conseguentemente impiegata anche per i led.

In alternativa al controllo fisico effettuato tramite bottoni è stato implementato un metodo basato sullo scambio di messaggi tra i nodi che permette di simulare i `SensorOn` e i `SensorOff`. Questo, unito a dei timer programmabili, ci ha dato la possibilità di testare in modo più dinamico e preciso le varie casistiche.

3. **Controllo dello stato della diagnostica.**

In questo caso, in seguito alla richiesta di occupazione di un semaforo, si controlla un'area di memoria condivisa con il task di diagnostica. Se essa è stata completata se ne controlla l'esito e si agisce di conseguenza, in caso contrario si procede normalmente.

4. **Controllo dei messaggi da taskWifi.**

In quest'ultimo blocco si verifica la presenza o meno di messaggi da gestire da parte del task wifi :

- **Nessun messaggio presente.**

Si controllano i timer per poter decidere se eseguire una nuova diagnostica (sempre se non è già in corso quella precedente, poiché in ogni istante nel sistema può essere attivo un solo task di diagnostica).

- **Messaggio presente.**

Viene analizzato opportunamente il messaggio. Se rispetta lo stato corrente del nodo e la fase del protocollo lo richiede, esso verrà inoltrato al nodo successivo o precedente nel percorso. In caso contrario verrà generato un messaggio di errore (`NOT_OK`) che sarà inviato sia al nodo successivo che precedente. Tutti i nodi gestiranno nello stesso modo questo tipo di messaggio in modo tale da garantire l'inoltro dell'errore su tutta la tratta.

Conclusa la gestione del messaggio il raspberry procederà ad aggiornare il proprio stato e ricomincerà il loop.

3.1.3 wifiTask (Task c)

Il wifiTask rappresenta il task di comunicazione descritto nelle specifiche e il suo ruolo principale è quello di scambiare messaggi tra i nodi della rete (definita in seguito all'initTask).

Il suo sorgente implementa diverse primitive (utilizzate anche da altri task) come la possibilità di instaurare connessioni tra raspberry, inviare e ricevere messaggi, inviare file di log all'host ecc...

Di seguito è illustrato il diagramma di flusso del task.

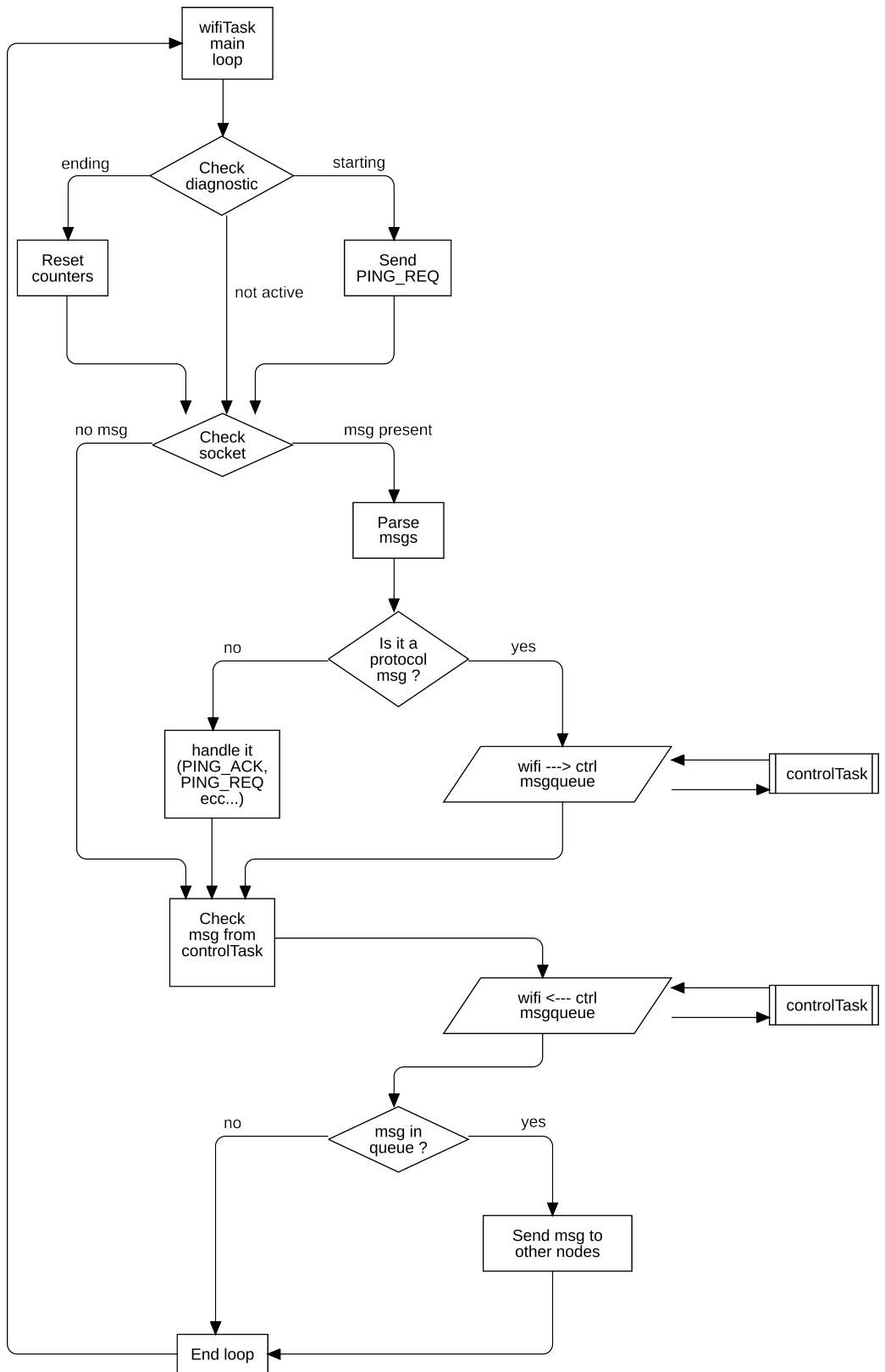


Figure 10: Wifi task flow diagram

Il funzionamento del wifiTask può essere riassunto come segue: durante il main loop il task verifica lo stato della diagnostica, gestisce i messaggi ricevuti da altri nodi e inoltra i messaggi da inviare da parte del task di controllo.

Poiché il processo accede a delle zone di memoria condivise con altri task è stata posta particolare attenzione alla loro gestione e sincronizzazione. Le primitive che sono state utilizzate per garantire ciò sono :

- **MsgQueue:**

Le code di messaggi sono un meccanismo di sincronizzazione che permette a un processo di inviare messaggi a un'altra entità all'interno del sistema operativo in modo asincrono, consentendo così la comunicazione e la condivisione di dati tra processi senza la necessità di utilizzare variabili globali o altre forme di condivisione di dati. Nel nostro caso sono state configurate in modo tale che la scrittura avvenga in modalità bloccante, mentre la lettura ha un timeout di 1 tick (dipendente dalla frequenza impostata).

Sono state impiegate due msgQueue:

- **IN_CONTROL_QUEUE** coda di messaggi da wifiTask a controlTask
- **OUT_CONTROL_QUEUE** coda di messaggi da controlTask a wifiTask

- **Semafori:**

I semafori sono un altro meccanismo di sincronizzazione utilizzato per gestire l'accesso concorrente a risorse condivise tra processi o thread.

Seppur non visibili nel diagramma, sono stati utilizzati alcuni semafori per accedere a delle variabili globali condivise, in particolare :

- **WIFI_CONTROL_SEM** semaforo per gestire race condition tra wifiTask e controlTask
- **WIFI_DIAG_SEM** semaforo per gestire race condition tra wifiTask e diagTask

Una riflessione più approfondita sulla memoria condivisa e sulle primitive di Vxworks usate è stata fatta qui:

[2.4](#)

3.1.4 posTask (Task d)

Il seguente task ha lo scopo di simulare l'azionamento del deviatoio quando necessario per completare la preparazione al passaggio di un treno. In particolare questo task viene spawnato all'occorrenza dal task di controllo. Il task di positioning procede ad aumentare una variabile intera da un valore minimo fino a raggiungere un valore massimo. Per simulare una certa aleatorietà si ha che il task controlla prima se un numero generato in modo casuale è inferiore ad una certa soglia: in quel caso segnala che il deviatoio ha avuto un problema, altrimenti procede a concludere con successo il movimento. In ogni caso la variabile che indica la posizione del deviatoio viene cambiata in accordo con come è andata la procedura e alla fine si ha comunque un resume del task di controllo, il quale procederà a controllare l'esito del posizionamento del deviatoio.

3.1.5 diagTask (Task e)

Il task di diagnostica viene spawnato dal controlTask in base ad un timer che viene resettato ogni volta che il nodo riceve un messaggio relativo al two phase commit protocol. Il suo ruolo è quello di verificare che tutti i nodi con cui è stata stabilita una connessione (seguendo la configurazione ricevuta in seguito all'initTask) siano attivi e in grado di rispondere. In caso contrario il diagTask lo comunica al controlTask che seguirà a far entrare il raspberry in uno stato di **PING_FAIL_SAFE**. Questo è definito dall'accensione di un led di colore rosso e dal rifiuto delle successive richieste di prenotazione del binario.

L'attività di invio delle richieste (**PING_REQ**) e il conteggio delle risposte (**PING_ACK**) viene delegata al wifiTask.

Ciò viene reso possibile tramite la struct `ping_status` che contiene al suo interno lo stato attuale della diagnostica. Il `diagTask`, una volta comunicato l'inizio della diagnostica al `wifiTask` tramite la variabile appena descritta, si sospende tramite la primitiva `taskDelay()` per un determinato numero di secondi. In seguito verifica se tutti i nodi hanno risposto e conclude comunicando il risultato al `controlTask`. Nel caso in cui arrivassero altre risposte oltre timeout, queste verrano dropgate, di conseguenza è necessario configurare adeguatamente il tempo di delay da impostare. Durante le nostre simulazioni abbiamo misurato un tempo medio di risposta da parte dei nodi di circa $200 \div 300$ ms, abbiamo dunque impostato il timeout a 2s garantendo un buon margine.

3.1.6 logTask (Task f)

Il seguente task ha lo scopo di fornire uno strumento per poter scrivere i log di esecuzione. Il task crea una coda in cui altri task possono aggiungere dei messaggi da salvare e procede a controllare periodicamente questa coda. Se sono presenti dei messaggi il task procede a scriverli in `/usr/log/log.txt`. Nel caso di crash inaspettati del sistema è possibile procedere a spegnere, riaccendere e connettersi ai raspberry per estrarre i log dell'esecuzione precedenti non ancora cancellati, inviando sulla schermata telnet il comando `cmd` e successivamente `more /usr/log/log.txt`.

Ogni riga del log è composta da:

- data e ora in cui è stato creato il messaggio,
- task che ha creato il messaggio,
- contenuto del messaggio, in particolare questo campo è utilizzato per indicare il numero di transizione.

3.2 sendLogTask

Questo task viene spawnato in seguito ad una richiesta telnet (eseguita dall'opzione `-t` dello script `workbench.sh` (6.3)) ed ha lo scopo di inviare i file di log all'host. Il task inizialmente provvederà a sospendere il `logTask` per il tempo necessario ad effettuare una copia locale dei log. Una volta completata esso effettuerà il resume del `logTask` e procederà ad inviare la copia all'host.

3.2.1 destructorTask

Questo task non è presente nelle specifiche ed è stato ideato da noi per permettere sia l'interruzione sicura dei processi attivi all'interno dei raspberry sia il rilascio delle risorse allocate dai vari task. Esso si basa sul costrutto dei segnali, già implementati in VxWorks in accordo con lo standard POSIX.

Ogni task al suo avvio specifica l'handler relativo ad uno specifico segnale (`SIGUSR1`) che rappresenterà il comando di distruzione del dato task. L'handler appena definito sarà responsabile di disallocare ogni struttura dati precedentemente inizializzata e di fare terminare il task. Il ruolo del task distruttore sarà quindi quello di inviare il segnale ad ogni task attivo ordinandone così la chiusura.

Il task viene generato dalla funzione `startDestructor` implementata in `dkm` (3.2.4) di conseguenza può essere attivato tramite shell telnet, in alternativa può essere utilizzata anche l'utilità `workbench.sh` (6.3).

3.2.2 timerTask

La creazione di questo task si è resa necessaria per poter misurare i tempi di esecuzione delle transizioni con una maggiore accuratezza. Il sorgente mette a disposizione la funzione `getTimeMicro()` che restituisce il tempo in μs trascorso dall'avvio del task (più un offset compreso tra 0 e $2^{32} \mu\text{s}$). Il ritardo medio ossia l'errore di misurazione introdotto dalla funzione è di circa $2 \div 3 \mu\text{s}$.

Una spiegazione più dettagliata del funzionamento del task è presente in: [5.1](#).

3.2.3 gpio

Il led posto su ogni raspberry è di tipo RGB e l'intensità dei tre canali di colore è esprimibile attraverso un segnale PWM da inviare ad ognuno dei tre pin. Poichè per la board utilizzata non è presente un supporto diretto di VxWorks per generare in modo efficiente un segnale PWM e visto che la creazione di un task addetto a tale scopo andava a creare un overhead di computazione utile solo in fase di demo e non in una reale applicazione abbiamo deciso di limitare i colori disponibili a 2^3 (8 colori distinti) andando ad usare solo tre segnali alto/basso. Essendo questo un numero sufficiente per esprimere gli stati in cui si trovava il sistema in un determinato momento, abbiamo quindi evitato di implementare una modulazione a larghezza d'impulso (PWM). All'interno del file `gpio.h` possiamo trovare le definizioni usate nel file `gpio.c`. Nello specifico per semplificare la dichiarazione dei vari colori, abbiamo definito una struct che espliciti quale dei tre canali (RED, GREEN e BLU) accendere per quello specifico colore (es. `YELLOW = {.R = true, .G = true, .B = false}`).

All'interno dello stesso file sono riportati i metodi per leggere il bottone connesso ad ogni raspberry e quelli per accendere e spegnere il led con il colore prescelto (vedi [sez. 6.1.3](#) per l'elenco completo degli stati).

3.2.4 dkm

Questo file ricopre il ruolo di entry point di alcuni processi quali quello di configurazione e il distruttore. In esso infatti sono implementate alcune funzioni che iniziano con "start" ed hanno lo scopo di generare dei task tramite la primitiva `taskSpawn` di VxWorks. Per esempio la funzione `startInit` genera l'`initTask` e da il via al processo di configurazione permettendo la successiva attivazione di tutti i task.

3.3 Utility e scripts di appoggio

Di seguito saranno descritti degli scripts bash o python utilizzati per automatizzare o simulare alcuni compiti. Questi sono stati eseguiti sul computer detto "host" collegato alla stessa rete locale dei raspberry.

3.3.1 host.py

Durante lo sviluppo del sistema si è reso necessario, nonostante non fosse indicato in modo esplicito nelle specifiche, andare a simulare anche il comportamento di un eventuale treno. Per questo motivo è stato creato un apposito script che riuscisse a permettere di eseguire tutte le funzioni richieste. Non avendo particolari vincoli e per velocizzare lo sviluppo e il debugging abbiamo optato per l'utilizzo di Python. Lo script è stato denominato `host.py`. Questa parte di codice ha più funzionalità:

- Durante la fase di inizializzazione l'host procede a connettersi con i nodi che sono posizionati all'inizio di ogni itinerario. In particolare l'host riceve da tali nodi l'informazione sull'identificativo di tale nodo

(denominato RASP_ID) e grazie a questo è in grado di creare una stringa di configurazione per tale nodo. Tale stringa contiene informazioni sulla data e ora attuale, il numero di nodi precedenti e successivi presenti e il numero di itinerari di cui fa parte il nodo. L'host procede poi ad inviare informazioni su RASP_ID e IP dei nodi precedenti e successivi e infine l'identificativo di ogni rotta, denominato route_id.

- Una volta completata la parte di configurazione iniziale l'host avvia due thread: un primo thread si occuperà di controllare la sintassi ed inviare i messaggi che l'utente inserisce da tastiera, il secondo thread invece ha lo scopo di controllare se i socket di comunicazione risultano ancora attivi ed inoltre mostra i messaggi ricevuti dai nodi. Se tali messaggi sono richieste di ping allora procede anche a rispondere, segnalando che l'host è ancora attivo e connesso.

Per semplificare la gestione degli itinerari disponibili e la conseguente creazione della stringa di configurazione necessaria per ogni nodo sono state sviluppate specifiche classi chiamate *Node*, *Route* e *Graph*. La prima classe consente di creare i nodi, la seconda classe si occupa di creare gli itinerari e ottenere informazioni su eventuali nodi presenti in uno specifico itinerario ed infine la terza classe ha il compito di far ottenere informazioni d'insieme di tutta la rete, come ad esempio quali itinerari interessano uno specifico nodo.

3.3.2 launch_host.sh

Questo script bash è un semplice wrapper di `host.py`. Esso esegue il parsing del file `\connect\build.config` e passa poi i risultati come argomenti a riga di comando. È stato realizzato principalmente per accedere con più facilità ad alcuni parametri di `host.py` e per compiere simulazioni diverse velocemente.

3.3.3 log.py

Attraverso lo script `log.py` posto in `host_script/` è possibile raccogliere i log provenienti dai vari nodi. L'host fungerà da server centralizzato ed ascolterà i vari messaggi provenienti dalla porta **6455**.

Per avviare lo script è necessario specificare due parametri:

- **HOST_IP**: L'indirizzo ip su cui ascoltare la connessione in arrivo
- **WD_PATH**: La directory iniziale in cui salvare i log ricevuti. Successivamente i file saranno quindi salvati in: `WD_PATH/execution_log_files/log_Indirizzo_nodo.txt` (uno per ogni nodo connesso)

Il funzionamento generale è molto semplice: finché i nodi connessi non terminano l'esecuzione, attraverso la funzione `receiveAndSaveLog` per ognuno di essi si riceve i vari messaggi inviati attraverso delle socket. Inizialmente vengono ricevute alcune informazioni sullo stato dei nodi e sulla dimensione dei log, in questo modo è possibile visualizzare un po' di feedback sullo stato attuale della procedura. In seguito alla comunicazione di inizio invio i messaggi vengono concatenati in un unico file che rappresenterà il log finale per l'i-esimo nodo.

3.3.4 logParser.py

Raccolti i log provenienti dai vari nodi come descritto nel capitolo 3.3.3, per poterli analizzare attraverso il tool **ORIS**, era necessario porre un layer intermedio che adattasse la struttura dei log prodotti dal nostro sistema con quella richiesta in input.

Lo script `logParser.py` ha proprio questo scopo:

- Riceve in input i log raccolti nella directory `connect/execution_log_files/` che saranno nella forma:

```

1 63428403    wifiTask      [t26] Non ci sono messaggi da gestire
2 63433382    wifiTask      [t78] attesa acquisizione msg completata
3 63433404    wifiTask      [t30] acquisisco semaforo per la coda

```

- Per ogni riga di ogni file log, attraverso le espressioni regolari si estraggono le informazioni riguardanti il nome della transizione ed il timestamp corrispondente.

In particolare l'espressione utilizzata è: `(\d+) .*\[(t\d+)\]` la quale:

- `(\d+)`: Ricerca nella riga i-esima del file log, il timestamp prodotto
- `.*\[(t\d+)\]`: Ricerca dopo quattro spazi, il nome della transizione corrispondente che sarà formata da un prefisso `t`, seguita da un numero intero.

- Successivamente viene quindi determinato il tempo di esecuzione effettivo richiesto da una transizione che sarà dato dalla differenza tra il timestamp della transizione corrente e la sua precedente.
- Infine viene quindi salvato nella directory `connect/host_script/importer_files` per ogni log un file contenente il modello richiesto pronto per essere importato su ORIS.

Il risultato prodotto dall'esecuzione di questo script sarà quindi una sequenza di transizioni del tipo `t_i` seguito dal tempo di esecuzione richiesto.

```

1 t26
2 0
3 t78
4 5
5 t30
6 0

```

3.3.5 launch_nodes.sh

Questo script permette l'avvio del modulo kernel caricato sui raspberry. Esso si basa sul file `\connect\build.config` per ottenere l'ip dell'host corrente e su `\connect\target.txt` per il numero e l'indirizzo ip dei raspberry attivi. Una volta lanciato si connette tramite telnet ai nodi ed esegue su ognuno la funzione `startInit` (implementata in [3.2.4](#)) passando come argomenti l'ID del nodo medesimo e l'ip dell'host. In questo modo ogni raspberry ottiene un ID univoco ed è in grado di instaurare una connessione verso l'host (conoscendone l'indirizzo).

4 Testing

4.1 Requisiti di architettura per il sistema distribuito

Per quanto riguarda la funzionalità da sviluppare per il sistema distribuito essa consiste nel creare un applicativo con le seguenti caratteristiche: dall’interfaccia utente dell’host deve essere possibile richiedere il passaggio per un itinerario inviando una richiesta al primo nodo di quella particolare rotta (si presuppone che l’host conosca il primo nodo di ogni itinerario). Tale nodo dovrà riuscire a ricevere tale messaggio dall’host e dovrà procedere a verificare se è possibile riservare l’itinerario richiesto dialogando con i successivi nodi nella rotta. Infine, concluso lo scambio di messaggi tra nodi, dovrà comunicare all’host l’esito negativo o positivo della richiesta.

4.2 Requisiti per il prototipo di prova

Il prototipo di prova utilizza la configurazione minima riportata in figura 11. In particolare sono presenti 5 nodi divisi in due itinerari ognuno dei quali ha un proprio circuito di binario iniziale e finale e condivide uno stesso nodo di scambio con l’altro itinerario. Entrambi i percorsi sono nella stessa direzione e questo porta a fare sì che se l’host invia la richiesta per uno dei due itinerari si avrà che l’altro risulterà indisponibile fino a quando il treno non avrà concluso il suo passaggio.

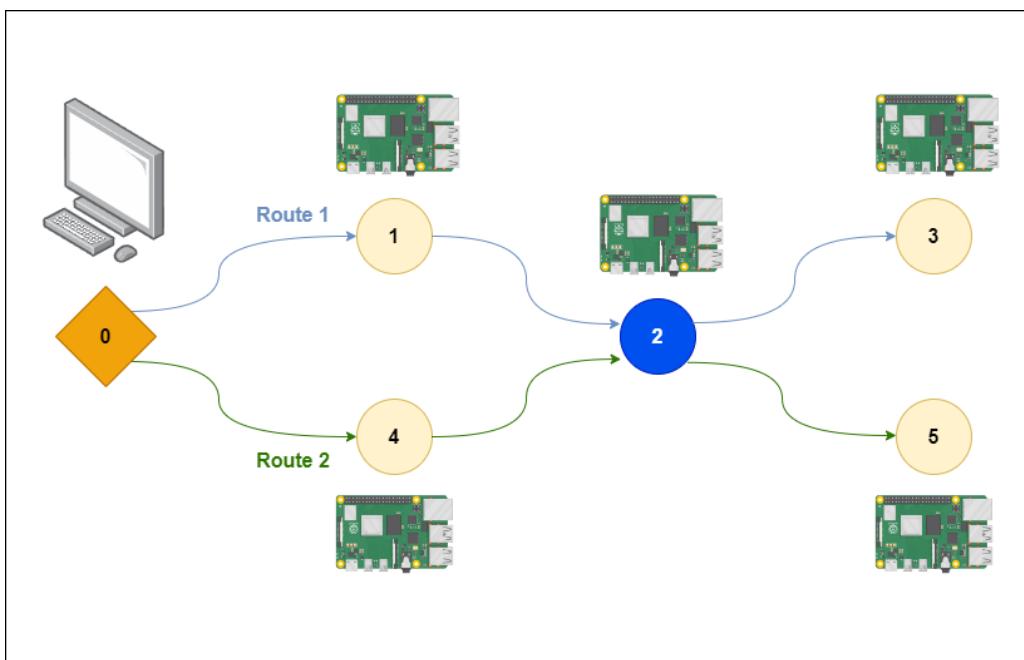


Figure 11: rappresentazione degli itinerari di prova

4.3 Test di sistema

Si è proceduto a verificare, con esito positivo, i seguenti casi di test:

- La richiesta di un itinerario va a buon fine, per entrambi gli itinerari, in caso né il circuito di binario né lo scambio siano occupati
- La richiesta di un itinerario non va a buon fine, per entrambi gli itinerari, in caso il circuito di binario o lo scambio, o entrambi, siano occupati

- L'occupazione in sequenza degli elementi di un itinerario richiesto, seguito dalla liberazione in sequenza degli stessi elementi, rende l'itinerario nuovamente disponibile
- La richiesta di un secondo itinerario mentre un primo itinerario è richiesto o riservato non va a buon fine
- L'occupazione non in sequenza degli elementi di un itinerario risulta impossibile per costruzione in quanto la rete di comunicazione tra nodi viene definita staticamente in base alla configurazione degli itinerari. Un nodo può quindi comunicare direttamente solo con i nodi immediatamente successivi o precedenti alla sua posizione nella rotta.
- La liberazione non in sequenza degli elementi di un itinerario occupato rende comunque l'itinerario disponibile quando tutti gli elementi sono liberi. Questo test non è eseguibile nella configurazione di simulazione con i messaggi **SENSOR_ON** e **SENSOR_OFF** in quanto essi vengono propagati in sequenza.
- La simulazione di un fallimento nel movimento dello scambio blocca il sistema (stato fail-safe). Questo stato viene identificato dall'accensione di un led di colore rosso e il sistema procederà a rifiutare ulteriori prenotazioni di tratte che includono quello specifico binario di scambio.

5 Risultati

5.1 Misurazione dei tempi

Per poter analizzare il sistema si è reso necessario essere in grado di misurare i tempi di esecuzione delle varie transizioni. Per questo motivo abbiamo studiato due diverse primitive:

- `clock_gettime(CLOCK_REALTIME, struct timespec *tp)` è la primitiva POSIX che restituisce una struttura con informazioni sui secondi e nanosecondi passati dall'epoch. Tuttavia questo tempo è basato sull'utilizzo di un timer impostato da VxWorks che di default ha un clock di 60Hz. Questa frequenza è modificabile, ma in un range assai limitato di valori e questo comporta l'impossibilità di riuscire a misurare ordini di grandezza ancora più piccoli dei millisecondi. Infatti, facendo un calcolo, si ha che a 60Hz un singolo tick corrisponde a $\frac{1}{60}s \simeq 16.7ms$, per questo motivo il valore restituito nel campo dei nanosecondi dalla funzione sarà dovuto solo ad approssimazioni numeriche e non reale precisione del timer in questo ordine di grandezza.
- `sysTimestamp()` è una primitiva di VxWorks che permette di andare direttamente a interagire con i registri del timer presente nella CPU BCM2711 del raspberry pi 4B. Questo timer è impostato alla frequenza di lavoro di 54 MHz e dispone di 32 bit di registro. Di conseguenza si raggiunge la capienza massima di memoria in $\frac{2^{32}-1}{54 \times 10^6}s \simeq 79.536s$. Superato questo valore il registro viene resettato e parte nuovamente a contare da zero. Non potendo direttamente accedere ai registri di comparazione del timer in quanto potenzialmente utilizzati dal sistema operativo VxWorks e non potendo usare le primitive implementate da VxWorks (perchè limitate ad un rate nell'ordine dei millisecondi) si è reso necessario creare un task addetto a monitorare lo stato del timer e dei suoi overflow. Il task salva in una variabile globale chiamata `totalCurrentTimeMicro` il tempo totale, espresso in microsecondi, trascorso a partire dal suo spawn, questo è calcolato come segue:

$$\text{overflowCounter} \times \text{TIME_TO_GET_OVERFLOW_MICRO} + \frac{\text{currentTimerRegValue}}{\text{TIMER_FREQ_MHZ}}$$

dove:

- **TIMER_FREQ_MHZ** = 54 è la frequenza del timer
- **TIME_TO_GET_OVERFLOW_MICRO** = $\frac{2^{32}-1}{54}$ indica il tempo in microsecondi necessario a raggiungere un overflow
- **overflowCounter** indica il numero di overflow avvenuti dall'avvio del timer
- **currentTimerRegValue** indica il valore attuale presente nei registri del timer

In particolare **overflowCounter** viene calcolato effettuando ripetutamente un controllo del registro a 32bit per verificare quando siamo vicini all'overflow. Per evitare una busy sleep, una volta avvenuto un overflow, il task viene sospeso per circa 76 s , in quanto ci aspettiamo che il registro raggiunga il massimo valore in 79 s e quindi siamo sicuri di non saltare nessun overflow.

Per avere più precisione nella misurazione degli eventi e delle transizioni abbiamo quindi scelto di utilizzare

`sysTimestamp()`, in particolare è stata definita la funzione getter `getTimeMicro()` che tramite un apposito semaforo restituisce il valore della variabile globale `totalCurrentTimeMicro` prima descritta.

5.2 Errori nella misurazione

Poiché il calcolo dei tempi è stato implementato tramite software e non tramite apposita strumentazione hardware, ci aspettiamo un certo margine di imprecisione. Inoltre le primitive da noi utilizzate non sono state studiate nell'ottica di essere il più efficienti possibili ed abbiamo quindi introdotto dei ritardi dovuti al loro utilizzo. In particolare per generare ogni messaggio di log ,tramite la funzione `logMessage()`, è necessario l'uso di alcune funzioni di formattazione delle stringhe che contengono però al loro interno numerose operazioni.

Abbiamo provato ad ottenere una stima della media dei ritardi introdotti rispettivamente dalle funzioni `getTimeMicro()` e `logMessage()` (la seconda contiene la prima al suo interno). Nel calcolo abbiamo eseguito in successione `getTimeMicro()` salvando le differenze di tempo (in microsecondi) tra una chiamata della funzione e la precedente, in seguito è stata applicata la stessa procedura con `logMessage()`. Abbiamo stimato un ritardo introdotto di circa $2 \div 3 \mu\text{s}$ per la prima e $20 \div 40 \mu\text{s}$ per la seconda (`logMessage()` varia maggiormente in quanto dipende da più parametri come la lunghezza del messaggio da loggare, ecc..).

5.3 Unità di tempo nel sistema

All'interno del sistema di interlocking distribuito avvengono diverse operazioni con tempi di esecuzione piuttosto eterogenei per quanto riguarda gli ordini di grandezza. In particolare possiamo individuare tre diversi gruppi:

- **Ordine di grandezza dei secondi:**

Sono transizioni dovute ad eventi fisici o ad attese protratte all'interno del sistema. Degli esempi di questa categoria sono i 2 secondi necessari per completare l'attesa dei ping da parte del task di diagnostica oppure i 3 secondi necessari per simulare il movimento del deviatoio di uno scambio da parte del task di positioning.

- **Ordine di grandezza dei millisecondi:**

Sono transizioni relative ad attese basate sulla frequenza di clock di Vxworks (60 Hz di default) di conseguenza nel nostro caso sono multipli di $16 \div 17$ millisecondi. Alcuni esempi sono le attese per la ricezione di un messaggio da parte dei socket, per la verifica della presenza di un messaggio nelle msgQueue oppure dei task delay applicati al termine dei loop del task di controllo o del task di comunicazione.

- **Ordine di grandezza dei nanosecondi:**

Sono transizioni che costituiscono la maggior parte delle operazioni eseguite dal sistema. A questo gruppo appartengono l'inizializzazione e la modifica di variabili, semplici operazioni aritmetiche, accesso ad array, ecc...

Nei log, che presentano una precisione dei microsecondi, queste operazioni non risultano comunque istantanee (eseguite con tempo 0) a causa dei ritardi introdotti dalla misurazione descritti prima (5.2).

Poiché non abbiamo la possibilità di misurare accuratamente le transizioni che avvengono nell'ordine dei nanosecondi e poiché non ci sono operazioni che avvengono nell'ordine dei microsecondi, la scelta dell'unità base per la modellazione della rete di Petri del sistema su Oris è ricaduta sui millisecondi. Questo ci ha permesso di restringere in buona parte gli intervalli in cui spaziano i tempi di firing delle transizioni. Da questa scelta e in

accordo con l'ampia differenza di ordini di grandezza tra i primi due gruppi di transizioni e il terzo, deriva il fatto che le operazioni eseguite nell'ordine dei nanosecondi vengono percepite come istantanee nella rete.

5.4 Analisi della rete con Oris

Il tool Oris 1.0 dispone di alcuni moduli che permettono di delineare e condurre certi tipi di analisi sulle reti di Petri, in particolare noi abbiamo utilizzato i seguenti strumenti:

- PetriNet Editor:

È un editor che permette di rappresentare graficamente diverse tipologie di reti di Petri, con la possibilità di assegnare priorità, risorse ecc ad ogni transizione. Nel nostro caso la tipologia di rete utilizzata è stata PTPN.

- TCAalyzer:

Una volta istanziata una rete di Petri tramite l'editor, è possibile analizzare ed esportare(in diversi formati) gli stati calcolati tramite il TCAalyzer.

```
Total time: 2.438
# trace bounds: [+INF,-INF]
## StateClassGraph Summary: ##
# stateClass: 34075
#####
Time end: Sun Oct 15 15:24:42 2023

Opening file: C:\Documents and Settings\admin\Desktop\Workspace interlocking v5_2\Workspace interlocking v5_2\Project1\TCAalyzer\output.txt
Writing file: C:\Documents and Settings\admin\Desktop\Workspace interlocking v5_2\Workspace interlocking v5_2\Project1\TCAalyzer\output.txt

Opening file: C:\Documents and Settings\admin\Desktop\Workspace interlocking v5_2\Workspace interlocking v5_2\Project1\TCAalyzer\output.xml
Writing file: C:\Documents and Settings\admin\Desktop\Workspace interlocking v5_2\Workspace interlocking v5_2\Project1\TCAalyzer\output.xml

Opening file: C:\Documents and Settings\admin\Desktop\Workspace interlocking v5_2\Workspace interlocking v5_2\Project1\TCAalyzer\output.graphml
Writing file: C:\Documents and Settings\admin\Desktop\Workspace interlocking v5_2\Workspace interlocking v5_2\Project1\TCAalyzer\output.graphml

Time exit: Sun Oct 15 15:24:51 2023

analysis terminated
Process terminated.
```

Figure 12: Analisi con TCAalyzer

- Importer Files:

È un plugin che permette di caricare molteplici tipi di files che verranno poi utilizzati dagli altri moduli.

Nel nostro caso è stato impiegato per caricare i file di testo per il modulo PetriNet simulator.

- PetriNet Simulator:

Questo strumento richiede le definizioni dei processi con le relative deadline e gli opportuni log contenti le transizioni ed i tempi di esecuzione, rispettivamente `task_set_def.txt` e `RTAI_relative_millisec_1.txt`.

Una volta caricati, esso permette di verificare la corrispondenza con la rete di Petri descritta tramite l'editor e che le deadline siano correttamente rispettate.

Come mostrato in figura: 12 il numero totale di stati relativi alla nostra rete è circa 34mila.

Il modello è stato strutturato in modo tale che venga utilizzata una singola risorsa (CPU) che corrisponde ad uno specifico core del raspberry, mentre le priorità sono state assegnate staticamente con questo criterio: `controlTask = 3`, `wifiTask = 2`, `diagTask = 1`. Entrambi i task che eseguono in loop (`controlTask` e `wifiTask`) alla fine di ogni ciclo effettuano un `taskDelay()`, in questo modo permettono anche ai task con priorità più bassa di eseguire. Sotto queste premesse ci si aspettava che l'ordine di esecuzione dei task e di conseguenza delle

transizioni fosse quasi predefinito e poco variabile portando ad avere un numero di stati molto più contenuto. Sempre in accordo con questa tesi sono le analisi condotte sui log tramite il PetriNet Simulator. Tutte le esecuzioni delle transizioni risultano feasible è in accordo con quanto specificato nel PetriNet Editor , ma la percentuale di copertura dello spazio degli stati risulta essere molto bassa ($1 \div 2\%$). Questo conferma il fatto che effettivamente le combinazioni di esecuzioni possibili delle transizioni in realtà sono molto meno di quelle calcolate dal TCAalyzer. La spiegazione di questo fatto è dovuta principalmente all'utilizzo degli switch.

Adottando il costrutto degli switch e in particolare delle transizioni di preselezione, siamo stati costretti ad impostare queste ultime come transizioni senza risorsa. Questo poiché, in caso di due transizioni abilitate con la stessa priorità e risorsa assegnata, Oris esegue sempre per prima la transizione con id associato più basso, senza considerare i tempi di firing. In questo modo tutte le transizioni di preselezione, non avendo risorsa associata (ne priorità), possono eseguire in competizione con tutte le transizioni della rete e questo porta ad un numero molto alto di combinazioni possibili che nella realtà non possono avvenire.

Poiché l'insieme degli stati realmente possibili è comunque contenuto nell'insieme totale e dati i vantaggi ottenuti dall'impiego degli switch (2.2) abbiamo comunque deciso di utilizzarli nel nostro modello.

5.5 Analisi delle deadline e dei runtimes

Per poter verificare le deadline dei task della rete abbiamo dovuto definirle nel file `task_set_def.txt` che, una volta caricato mediante il File Importer Box, ci ha permesso di convalidarle con il PetriNet Simulator.

Le deadline sono state assegnate come segue:

- Task di controllo: 500ms.
- Task di comunicazione: 500ms.
- Task di diagnostica: 2200ms.
- Task di posizionamento: 3200ms.

Logicamente non sono state descritte le deadline dei task che non sono stati modellati nella rete di Petri (Init-Task, LogTask ecc...).

Durante i nostri test i limiti sono sempre stati tutti rispettati, è necessario fare però una considerazione sul task di positioning. Esso a differenza degli altri task (come diagnostica e comunicazione) non esegue in parallelo al task di controllo, ma in sequenza. Il flusso del task di controllo dei binari di scambio infatti viene interrotto con l'inizio del task di positioning e ripreso al termine di esso. Questa scelta è stata fatta con lo scopo di non andare ad appesantire maggiormente la rete con altri semafori e switch e il conseguente incremento del numero di stati. Risulta comunque molto semplice rimodellare il task in modo che esegua in parallelo (seguendo per esempio l'implementazione del task diagnostica) e lo abbiamo considerato come un possibile sviluppo futuro. Il comportamento appena descritto comporta inevitabilmente il fatto che il task di controllo nei binari di scambio, durante i cicli in cui avviene il positioning, sfiora la propria deadline a causa dell'attesa del posizionamento. Non lo abbiamo considerato un problema in quanto , sotto le stringenti ipotesi di richieste di prenotazioni ripetute (tutte con necessità di riposizionamento del deviatoio), questo avviene circa l' $1 \div 1.5\%$ delle volte rispetto ai tutti i cicli del task di controllo. Inoltre anche in questi casi il task di comunicazione è stato progettato per gestire e rispondere in maniera autonoma a diversi messaggi.

Oltre alle analisi condotte tramite il tool Oris1, abbiamo cercato di rappresentare alcune statistiche basandoci direttamente sui log estratti dai raspberry. Tutte le analisi sono state condotte sotto le seguenti condizioni:

- 20 richieste di prenotazione di due tratte distinte effettuate in modo alterno, così da rendere sempre necessario il movimento del deviatoio.
- Tasso di fallimento del positioning task impostato allo 0%.
- Simulazione del passaggio dei treni in circa 2s per ogni binario.
- Avvio del task di diagnostica dopo 2s senza ricezione di messaggi.

I grafici riportati sono relativi ad un binario di scambio in modo da poter rappresentare anche i tempi del task di positioning e non si discostano comunque dai dati raccolti nei nodi lineari.

Come si può vedere dal grafico, i task rispettano con ampio margine le deadline descritte precedentemente.

Nota: Non sono stati rappresentati gli "outlier" del task di controllo in occasione del positioning (sarebbero sovrapposti ai bars del positioning).

Distribution of task termination times

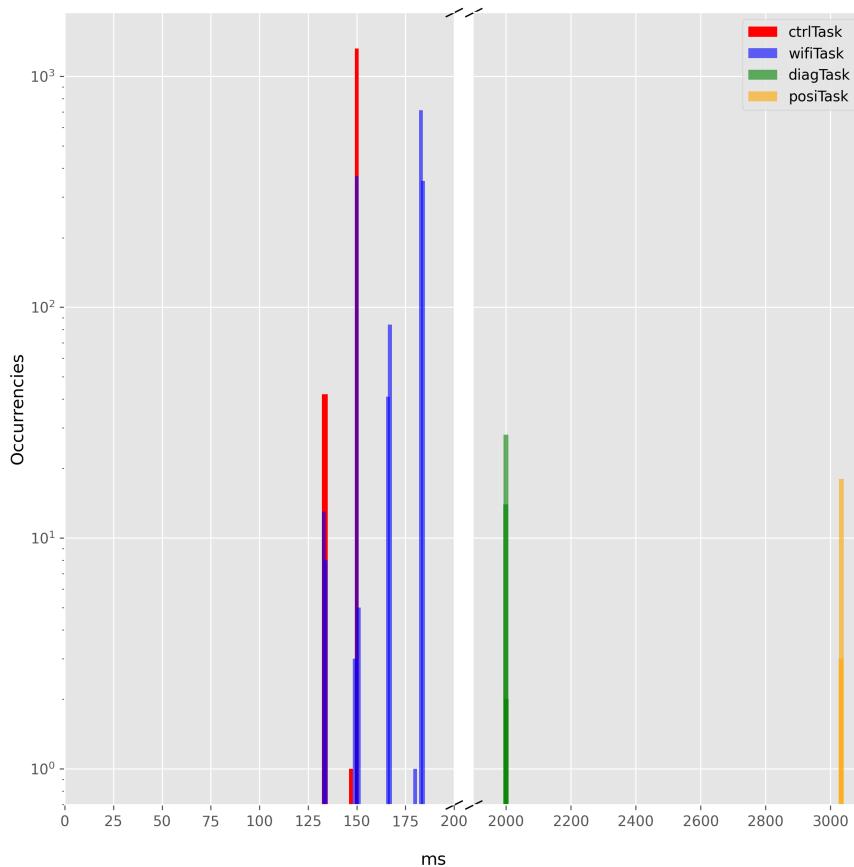
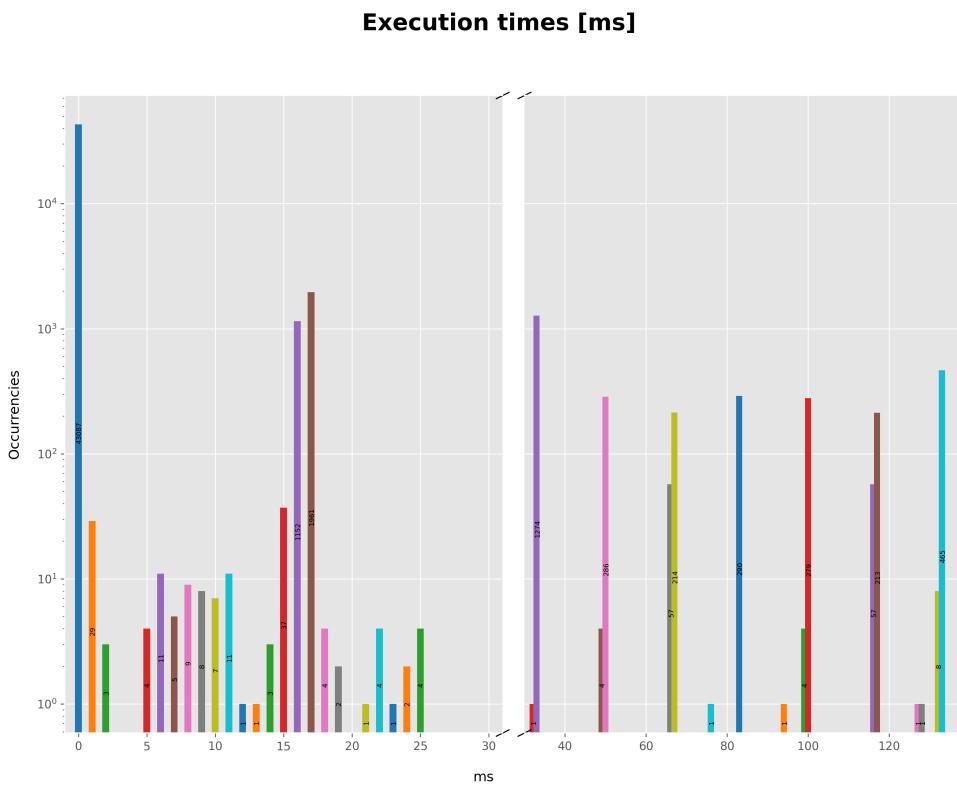


Figure 13: Distribuzione dei tempi di terminazione dei task

Nei grafici seguenti sono riportati i tempi d'esecuzione delle transizioni. Questi sono i tempi impiegati dal sistema per completare l'azione associata alla relativa transizione e differiscono dai tempi calcolati a partire dall'enable della transizione in quanto non considerano il tempo d'attesa necessario ad acquisire la risorsa da

parte del task. Si può notare che la maggior parte delle transizioni esegue in un tempo inferiore al millisecondo e per questo motivo il numero di transizioni che esegue in tempo zero è il maggiore. Inoltre le restanti transizioni si distribuiscono a multipli del clock di sistema, questo perchè queste transizioni sono legate a funzioni che generano un tempo di attesa basato sulla frequenza del sistema operativo (la spiegazione dettaglia si trova qui [5.1](#)). Si possono comunque notare pochi esempi di tempi che non rispettano questa distribuzione e queste irregolarità sono legate ad operazioni intensive come operazioni sulle stringhe, sui socket e da una variabilità data dall'esecuzione di task di sistema.

Nel primo grafico ([14](#)) sono rappresentati i tempi di esecuzione di tutte le transizioni raccolte dai log. Vediamo che nella parte di sinistra si ha una maggior variabilità proprio perchè qui sono rappresentate le operazioni intensive come operazioni sulle stringhe e sui socket. Sulla parte di destra invece si ha una forte regolarità, data dal fatto che le transizioni che impiegano tali tempi ad eseguire sono legate quasi esclusivamente a operazioni di attesa che risultano essere molto riproducibili.



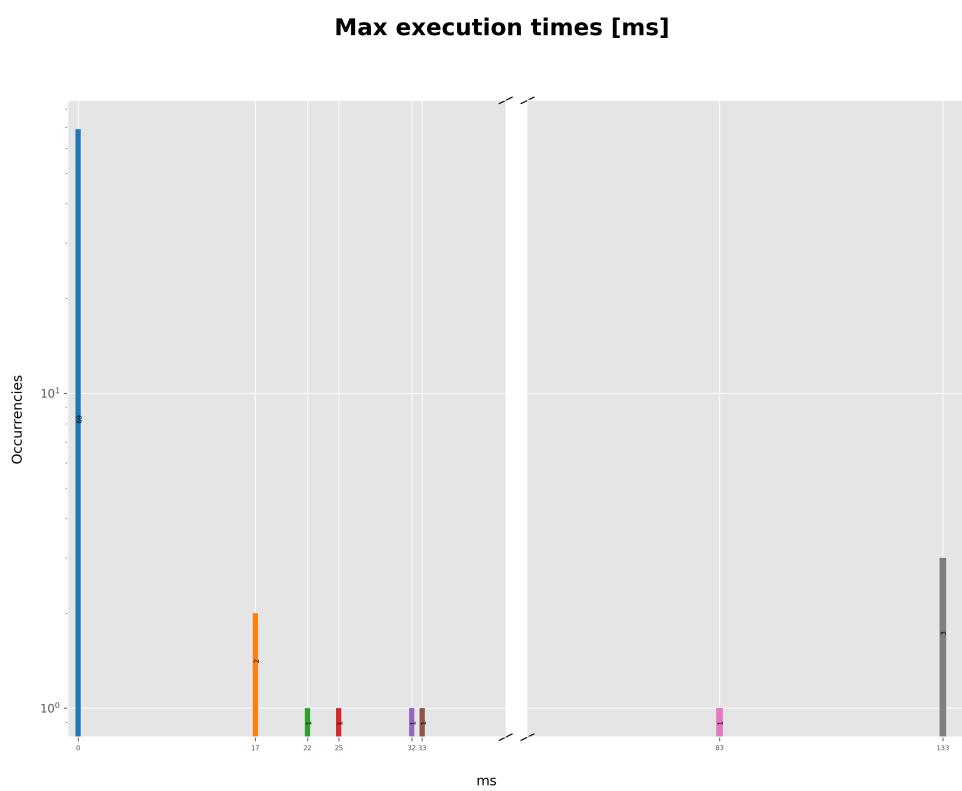


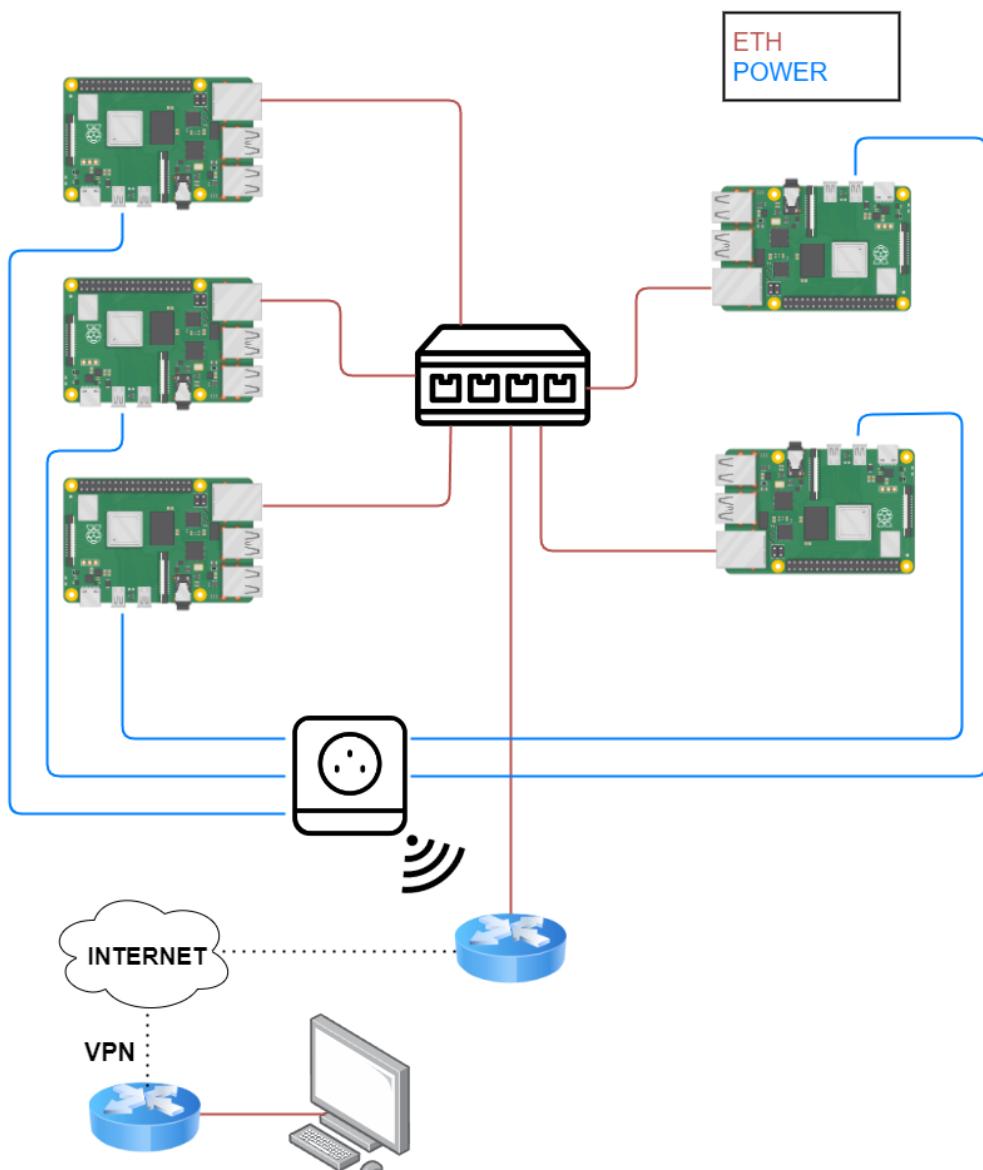
Figure 15: Distribuzione dei tempi massimi di esecuzione delle transizioni

6 Setup

6.1 Setup hardware

6.1.1 Configurazione remota

Per creare l'infrastruttura di rete abbiamo sfruttato la porta Ethernet presente su ogni raspberry e abbiamo collegato tale porta al router (creando una topologia a stella) e anche il dispositivo host (il computer che simula il treno) è stato collegato nella stessa rete. Dovendo lavorare da PC posizionati in luoghi geograficamente diversi abbiamo utilizzato una VPN per fare sì che i computer risultassero nella stessa rete, così da poter testare il codice più agevolmente. Abbiamo aggiunto anche una presa smart per poter spegnere e riaccendere i raspberry da remoto: questa aggiunta è stata necessaria poiché in caso di bug nell'esecuzione del codice i raspberry andavano in crash terminando la comunicazione e quindi togliendo la possibilità di eseguire un reboot da linea di comando.



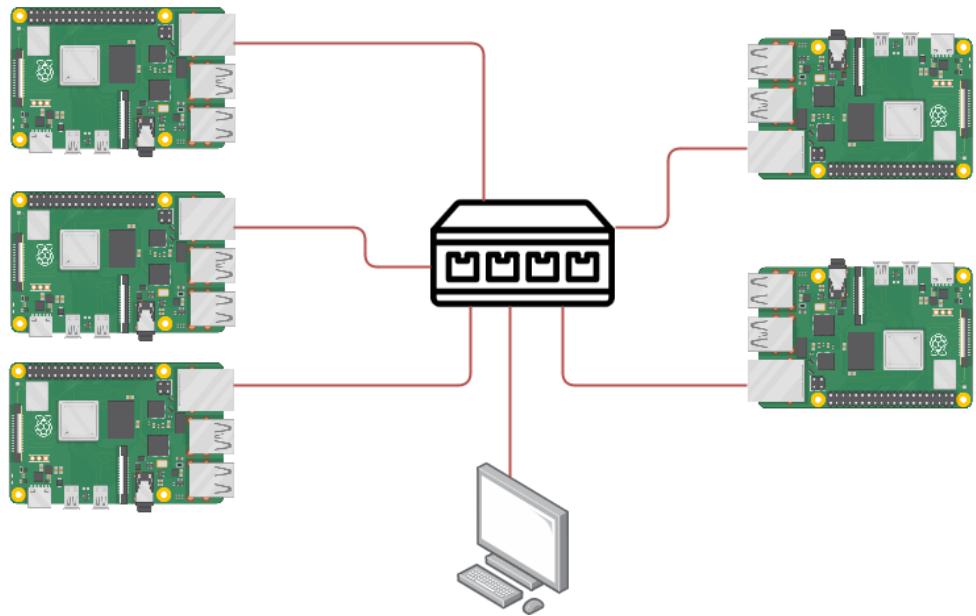
6.1.2 Configurazione locale

Il sistema può anche essere replicato in locale senza l'ausilio di sistemi di rete ponendosi come unico requisito quello di avere a disposizione un server DHCP interno.

Supponendo quindi di utilizzare un sistema Linux, possiamo installare al suo interno un server DHCP tramite il comando `sudo apt-get install isc-dhcp-server` ed inserire nel file di configurazione, posto in `/etc/dhcp/dhcpd.conf`, la configurazione di rete per i raspberry.

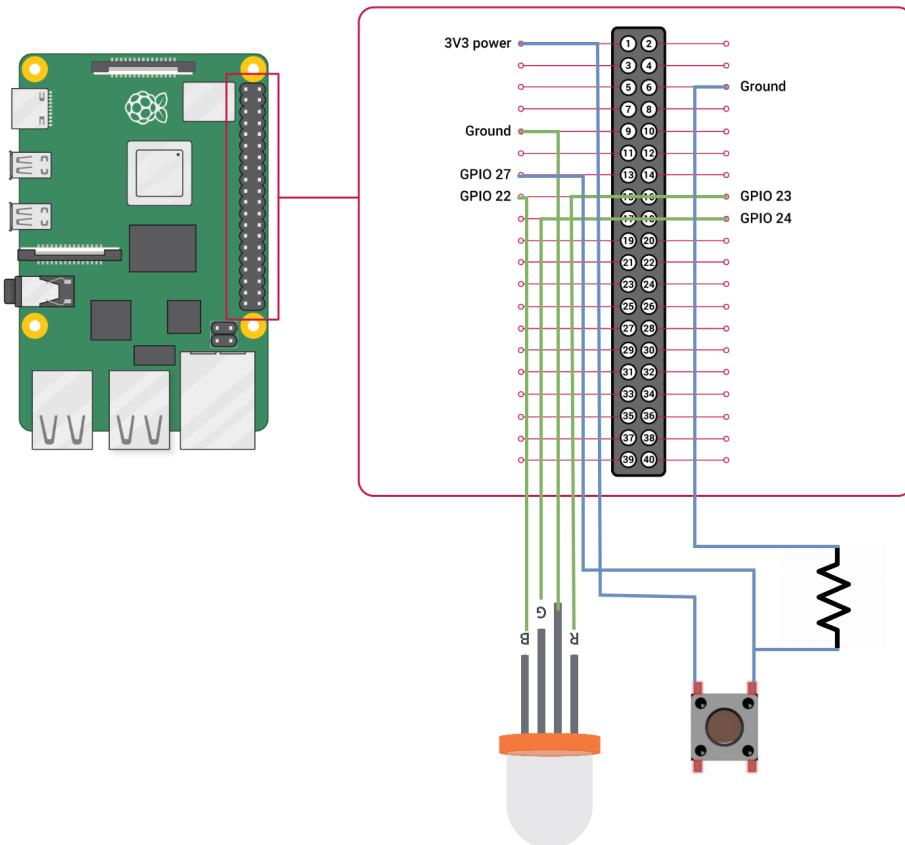
```
1 default-lease-time 600;
2 max-lease-time 7200;
3
4 ddns-update-style none;
5 authoritative;
6
7 subnet 192.168.1.0 netmask 255.255.255.0 {
8     range 192.168.1.201 192.168.1.215;
9     option routers 192.168.1.201; #Indirizzo dell'host del sistema
10 }
11
12 host rasp1 {
13     hardware ethernet dc:a6:32:40:23:61;
14     fixed-address 192.168.1.211;
15 }
16
17 host rasp2 {
18     hardware ethernet dc:a6:32:40:21:30;
19     fixed-address 192.168.1.212;
20 }
21
22 host rasp3 {
23     hardware ethernet dc:a6:32:40:22:8f;
24     fixed-address 192.168.1.213;
25 }
26
27 host rasp4 {
28     hardware ethernet dc:a6:32:40:23:2b;
29     fixed-address 192.168.1.214;
30 }
31
32 host rasp5 {
33     hardware ethernet dc:a6:32:40:21:8a;
34     fixed-address 192.168.1.215;
35 }
```

Assegniamo quindi all'interfaccia di rete del sistema Linux l'indirizzo corrisponde all'opzione `routers` (questo sarà anche l'indirizzo dell'host del sistema) ed inseriamo nel file `/etc/default/isc-dhcp-server` il nome dell'interfaccia nella variabile `INTERFACESv4`.



6.1.3 Led e bottone

Di seguito viene mostrata una rappresentazione schematica di come è stato collegato un led RGB ed un bottone ad ogni raspberry della rete.



Il bottone è stato utilizzato per rappresentare il passaggio di un treno da un binario ed è gestito all'interno della libreria gpio.

Il led invece permette una percezione visiva dello stato in cui si trova il sistema. In particolare:

- YELLOW: colore di inizializzazione che viene mantenuto fino al completamento dell'initTask. Lo possiamo notare nelle fase iniziale di accensione del sistema, in quanto tale fase richiede vari secondi per essere completata.
- BLUE: indica lo stato NOT_RESERVED, in cui il nodo è in attesa di ricevere richieste. Indica che il binario non è ancora stato riservato e quindi risulta libero e prenotabile.
- CYAN: indica uno dei possibili stati intermedi in cui si ha scambio di messaggi per completare la prenotazione dell'itinerario.
- WHITE: indica lo stato in cui è in corso il positioningTask.
- GREEN: indica lo stato RESERVED, quindi si ha avuto una prenotazione completata con successo.
- MAGENTA: indica lo stato TRAIN_IN_TRANSITION, quindi che il treno sta attualmente passando sul binario.
- RED: indica un qualunque possibile stato di fail presente nel sistema.

6.2 Setup software

In questa sezione andiamo ad illustrare una guida su come fare il setup del sistema, includendo i passi da seguire per rendere funzionante il software sviluppato:

- Il sistema è stato sviluppato utilizzando Ubuntu 22.04
- Sono stati utilizzati dei Raspberry Pi 4 eseguendo al loro interno VxWorks 7. Questo è stato possibile seguendo le istruzioni poste dal [sito ufficiale di Windriver](#).
- È necessaria l'installazione di Workbench 4 di VxWorks 7
- È necessario scaricare ed estrarre l'SDK per raspberry pi 4b dal [sito ufficiale di VxWorks](#). L'estrazione deve avvenire nella directory padre del progetto (ossia `parent_folder\distributed-railway-interlocking-system`). Durante lo sviluppo è stato utilizzato l'SDK versione 1.4
- Si procede a clonare la [repo di github](#)
- All'interno della cartella connect si deve rinominare il file `build_example.config` in `build.config` e aggiungere il percorso di installazione di WindRiver e l'indirizzo IP dell'host (computer da cui inviare i comandi che simula il treno):

```

1 [WindRiver_path] : "/PATH/TO/INSTALLATION/WindRiver",
2 [Host_ip] : "192.168.1.203"

```

- Nel caso si utilizzi VSCode, oltre ad installare le estensioni di utilità come “C/C++ Extension Pack”, si consiglia di aggiungere al file `c_cpp_properties.json` nella cartella `.vscode` i seguenti parametri, in modo che Intellisense mostri gli errori di sintassi:

```

1 {
2     "env": {
3         "WINDRIVER_SDK": "/PATH/TO/SDK/wrsdk-vxworks7-raspberrypi4b"

```

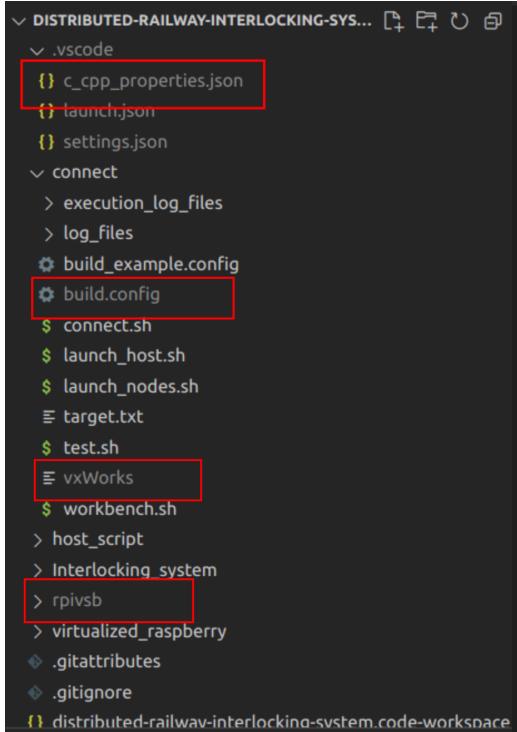
```

4     },
5     "configurations": [
6         {
7             "name": "VxWorks",
8             "includePath": [
9                 "${workspaceFolder}/Interlocking_system/**",
10                "${WINDRIVER_SDK}/vxsdk/sysroot/krnl/h/public/**",
11                "${WINDRIVER_SDK}/vxsdk/sysroot/usr/h/public/**",
12                "${WINDRIVER_SDK}/vxsdk/sysroot/usr/h/public/",
13                "${WINDRIVER_SDK}/vxsdk/sysroot/share/h/public/",
14                "${WINDRIVER_SDK}/vxsdk/sysroot/usr/h/**",
15                "${WINDRIVER_SDK}/vxsdk/sysroot/share/h/**"
16            ],
17            "defines": [
18                "VX_CPU_FAMILY=arm",
19                "CPU=_VX_ARMARCH8A",
20                "TOOL_FAMILY=llvm",
21                "TOOL_VERSION=llvm_2016_04",
22                "CC_VERSION=llvm_9_0_1_1",
23                "TOOL=llvm"
24            ],
25            "intelliSenseMode": "linux-clang-arm64"
26        }
27    ],
28    "version": 4
29 }

```

- Si procede a decomprimere il file **rpivsb.rar** in modo che il contenuto estratto sia all'interno della root del git (ovvero dentro **distributed-railway-interlocking-system**)
- Si procede a decomprimere il file **vxWorks.zip** in modo che il contenuto estratto sia all'interno della cartella **connect**

A questo punto la situazione dovrebbe essere la seguente:



Note:

- Nel file `connect/target.txt` vengono indicati gli indirizzi IP a cui sono presenti i diversi raspberry utilizzati:

```

DISTRIBUTED-RAILWAY-INTERLOCKING-SYS...
  connect > target.txt
  1 192.168.1.211
  2 192.168.1.212
  3 192.168.1.213
  4 192.168.1.214
  5 192.168.1.215
  6

```

- Il numero totale di raspberry presenti è un argomento da passare quando si lancia lo script python `host_script/host.py`. Per modificarlo in modo permanente si consiglia di modificare il file `connect/launch_host.sh` (default 5 raspberry) e ulteriori file, come ad esempio, `connect/workbench.sh`:

```

Edit Selection View Go Run Terminal Help
EXPLORER ...
DISTRIBUTED-RAILWAY-INTERLOCKING-SYS...
    .vscode
        c_cpp_properties.json
        launch.json
        settings.json
    connect
        execution_log_files
        log_files
        build_example.config
        build.config
        connect.sh
        launch_host.sh
        launch_nodes.sh
        target.txt
        test.sh
        vxWorks
$ workbench.sh
$ launch_host.sh
connect > $ launch_host.sh
1   host_ip=$( grep -Po '(?=<\[Host_ip\] : ")[^"]*' ./build.config )
2   python3 ../../host_script/host.py $host_ip

```

- Nel caso si utilizzi una presa smart TP-Link Tapo P100 è possibile integrare l'accensione e lo spegnimento dell'intero sistema direttamente da terminale, sfruttando un particolare comando che abbiamo integrato nello script `workbench.sh`. E' necessario però un setup del file `tapo.config` che andrà posizionato nella cartella `connect`. In tale file si dovrà aggiungere la mail e la password utilizzate per accedere all'applicativo TP-Link:

```

1 [USER]
2 user_id = account@email.com
3 password = your_password

```

6.3 Utility WorkBench

Vista la natura del progetto in cui sono presenti multipli target risultava piuttosto laborioso l'utilizzo dell'interfaccia grafica per compiere specifiche azioni. Per esempio, attraverso la GUI di WorkBench 4, è possibile connettersi e caricare i moduli su un singolo target alla volta. Per questo motivo abbiamo deciso di creare lo script `./workbench.sh` (presente nella directory `connect`) che ci ha consentito di interagire con i raspberry in modo più efficiente e maggiormente scalabile. Durante lo sviluppo del progetto è risultato uno strumento essenziale, inoltre ci è sembrato necessario in quanto se si volesse testare il sistema su un numero di raspberry ancora più grande l'uso dell'interfaccia grafica sarebbe sicuramente da escludere.

Lo script è composto da moduli (implementati tramite funzioni) che permettono di compiere numerose azioni sui raspberry, in particolare :

- Connessione ai target di debug (raspberry)
- Load e unload dei moduli sui target connessi
- Terminazione dei task attivi e rilascio delle risorse
- Apertura di connessioni telnet verso i target
- Compilazione dei moduli (In base alla configurazione precedentemente definita)
- Spegnimento, accensione e riavvio dei target
- Trasferimento dei log dai target verso l'host.

L'utility si basa sui file di configurazione `connect/target.txt` e `build.config` entrambi predisposti in fase di [Setup software](#).

Per poter consentire l'utilizzo di diverse shell interattive (illustrate più nel dettaglio in seguito) è stato progettato un meccanismo che sfrutta le named pipe di Ubuntu congiuntamente alle regex. In questo modo è stato possibile ridirigere e analizzare l'input e l'output delle shell in background, evitando così di dover creare multipli script e di riavviarli in ogni fase di load/unload dei moduli.

6.3.1 Wrtool

Wrtool è un utility a riga di comando sviluppata da Wind River e fornita nell'installazione di Vxworks (documentazione consultabile previa registrazione [qui](#)). Essa permette la creazione, la compilazione e la gestione dei progetti Vxworks. Sebbene sia possibile gestire il progetto interamente su Wrtool, in fase iniziale ci è risultato più comodo crearlo e configurarlo tramite l'interfaccia grafica di WorkBench 4. Nel nostro caso quindi ci siamo limitati ad utilizzarlo per effettuare la compilazione del progetto (funzione `build()` di `connect/workbench.sh`), in particolare gli step svolti sono stati:

- Definizione della directory di workspace (dove si trova il progetto precedentemente creato)
- Importazione del progetto
- Compilazione

Il risultato della compilazione viene rediretto verso il file `connect/workbench_log_files/build_log.txt`

6.3.2 Wrdbg

WRDBG è la shell di debug fornita da Wind River (il manuale è consultabile [qui](#)). Sebbene la sintassi dei comandi di WRDBG sia compatibile con GDB quando possibile, non è intesa come una shell GDB. Infatti poiché questa shell di debug è destinata alle connessioni ai target di debug di Wind River, aggiunge alcuni comandi che non sono disponibili con GDB (come i comandi di connessione al target). La shell di debug è disponibile sia dall'interfaccia utente del WorkBench sia da riga di comando.

Questo strumento ci ha permesso di :

- Connettersi ai raspberry in remoto tramite indirizzo IP (specificando il kernel file del target `connect/vxWorks` ottenuto in [Setup software](#)) .
- Effettuare il load e l'unload dei moduli parallelamente su diversi target.

Gli output della shell vengono interpretati in modo automatico dallo script `connect/workbench.sh` con l'utilizzo di alcune regex, ma restano comunque visibili in `connect/workbench_log_files/log_INDIRIZZO_IP_DEL_TARGET.txt`.

6.3.3 connect.sh

Il file `connect.sh` è trasparente all'utente in quanto viene utilizzato esclusivamente come script di appoggio da parte di `workbench.sh`. Esso è presente nella directory `connect/` e mette a disposizione alcune funzionalità utili per la connessione verso i nodi. Infatti, definiti gli indirizzi dei target all'interno del file `connect/target.txt`, attraverso questo script è possibile verificare che il nodo i-esimo sia connesso correttamente alla rete tramite dei ping e:

- In caso affermativo:
 - Viene aperta una shell di debug `wrdbg` in background nel nodo. Per fare questo, viene utilizzata una named pipe (denominata in `temp/fifo_IndirizzoTarget`) per consentire una comunicazione bidirezionale tra host e target e un file di log per restituire eventuali errori verificati durante il processo di vita del sistema
 - Viene inoltre aperta una shell Unix utile al programmatore per interfacciarsi manualmente tramite comandi telnet con il raspberry di riferimento
- In caso negativo: Viene riportato un codice di errore all'utente

Terminate le operazioni sui target, è stata inoltre prevista una funzione di `cleanup` che si occuperà di chiudere correttamente le named pipe per garantire l'integrità dei dati.

6.3.4 Guida utilizzo

Le funzionalità implementate in `./workbench.sh` sono utilizzabili eseguendo lo script da riga di comando con i seguenti argomenti:

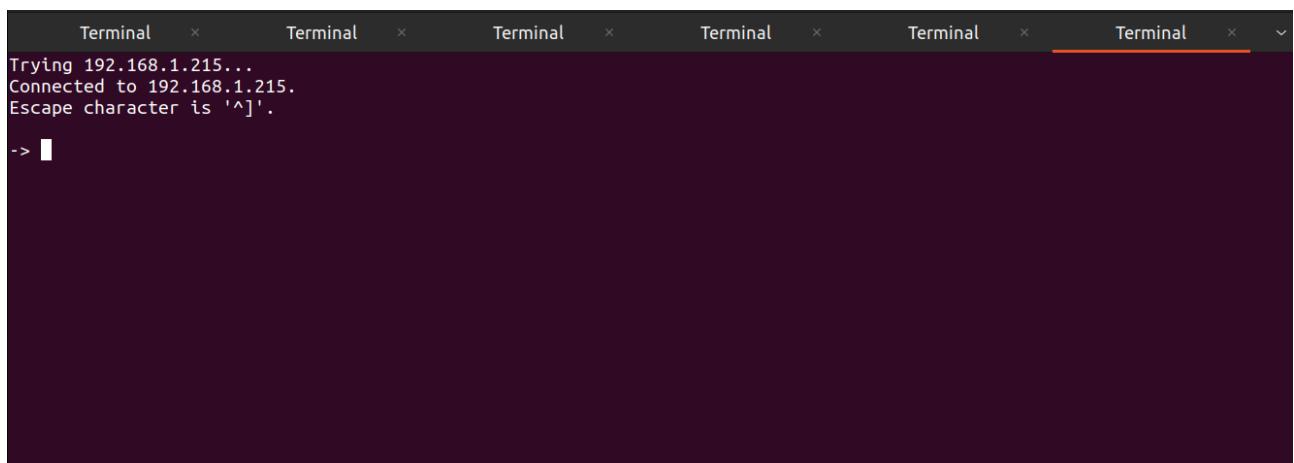
- **-b:** permette di sfruttare il comando `prj build` e quindi usare la toolchain impostata al momento della creazione del progetto su WorkBench 4 per compilare il progetto. I file che vengono compilati con questo comando sono quelli che si trovano all'interno delle cartelle `src` e `includes`. Durante la fase di compilazione eventuali errori e warning vengono salvati in `workbench_log_files/build_log.txt`
- **-c:** permette di sfruttare il comando `target connect` e quindi connettersi a tutti i target selezionati e aprire una connessione telnet per lo scambio di dati. Viene verificato anche che la connessione sia andata a buon fine entro un certo tempo definito da un timeout.
- **-l:** permettere di sfruttare il comando `module load` e quindi di andare a caricare sui raspberry i moduli precedentemente compilati. Prima del load vengono eseguiti ulteriori comandi per “sanificare” lo stato dei raspberry (ad esempio `task_delete` e `module unload`) ed evitare l'insorgere di problemi. Anche in questo caso si ha una verifica di esito positivo dell'operazione entro un timeout.
- **-u:** permette di rimuovere quei moduli precedentemente caricati nel sistema tramite il comando `module unload`. Inoltre viene prima verificato che sia stata instaurata una connessione con i targets ossia che esista una named pipe per ogni nodo. In caso negativo, viene richiamato il comando `connect` con **-c**
- **-o:** permette di lanciare uno script python che si mette in attesa di ricevere da tutti i raspberry i log di esecuzione e procede a salvarli in `connect/execution_log_files`.
- **-d:** effettua il task delete nei raspberry tramite il comando `startDestructor`.
- **-r:** permette di eseguire un reboot di tutti i targets.
- **-v:** esegue il riavvio della presa smart. Questo comando è utile dovendo lavorare in un sistema remoto a cui non è possibile accedere fisicamente. Vengono inoltre cancellate e ricreate le named pipe di comunicazione verso i nodi.

- **-t**: esegue in automatico un numero di richieste di prenotazioni pari all'argomento passato da linea di comando.
- **-s**: esegue lo spegnimento del sistema. Vengono quindi spenti i raspberry e rimosse eventuali named pipe create in precedenza dallo script `connect.sh`.
- **-h**: mostra un messaggio di help con una breve spiegazione delle funzionalità.

Nota: questi comandi possono essere chiamati in successione, ad esempio per compilare, connettersi e caricare i moduli si può lanciare `./workbench -bcl`.

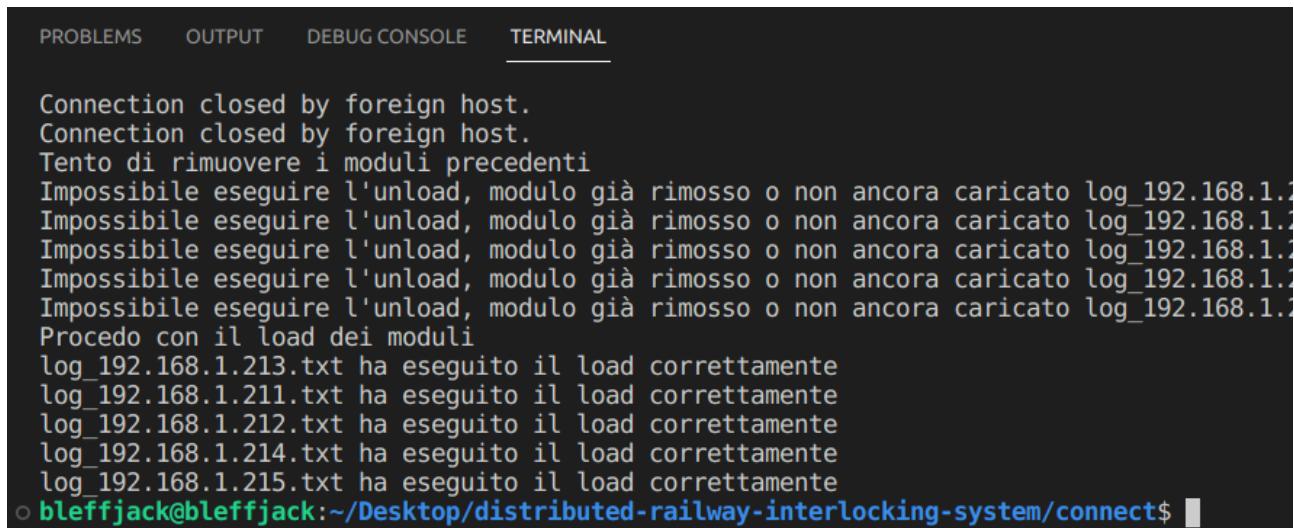
6.3.5 Esempio d'uso

Una volta cambiato directory e posizionati in `connect` lanciare `./workbench -bcl` per compilare, connettersi ai targets:



```
Trying 192.168.1.215...
Connected to 192.168.1.215.
Escape character is '^]'.
```

e, una volta completata l'apertura dei terminali, caricare i moduli:



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

Connection closed by foreign host.
Connection closed by foreign host.
Tento di rimuovere i moduli precedenti
Impossibile eseguire l'unload, modulo già rimosso o non ancora caricato log_192.168.1.
Impossibile eseguire l'unload, modulo già rimosso o non ancora caricato log_192.168.1.
Impossibile eseguire l'unload, modulo già rimosso o non ancora caricato log_192.168.1.
Impossibile eseguire l'unload, modulo già rimosso o non ancora caricato log_192.168.1.
Impossibile eseguire l'unload, modulo già rimosso o non ancora caricato log_192.168.1.
Procedo con il load dei moduli
log_192.168.1.213.txt ha eseguito il load correttamente
log_192.168.1.211.txt ha eseguito il load correttamente
log_192.168.1.212.txt ha eseguito il load correttamente
log_192.168.1.214.txt ha eseguito il load correttamente
log_192.168.1.215.txt ha eseguito il load correttamente
bleffjack@bleffjack:~/Desktop/distributed-railway-interlocking-system/connect$
```

A questo punto si eseguono in ordine gli script `launch_host.sh` e `launch_nodes.sh`.

Conclusa la fase di connessione e avvenuta la configurazione (visibile dall'apparizione di: "Messaggio da inviare ai nodi specifici:"), sarà possibile specificare i comandi da inviare da tastiera.

Nel nostro esempio e con la nostra configurazione, abbiamo chiesto di riservare la rotta 1 (il cui primo nodo ha

`id=1`) e tentato di prenotare una seconda rotta con un nodo in comune con la precedente che, giustamente, ci è stata negata.

Nota: per riservare una rotta abbiamo usato il comando `1 REQ;0;1`. dove con il primo numero specifichiamo l'id del nodo a cui inviare il messaggio, mentre la sintassi del secondo blocco è descritta in [1.5](#).

```
alessandro@MacBook-Pro-di-Alessandro:~/github/distributed-railway-interlocking-system/connect$ ./launch_host.sh
Connected by ('192.168.1.211', 53340)
[MSG] : RASP_ID : 1
Connected by ('192.168.1.212', 63596)
[MSG] : RASP_ID : 2
Connected by ('192.168.1.213', 57321)
[MSG] : RASP_ID : 3
Connected by ('192.168.1.214', 52455)
[MSG] : RASP_ID : 4
Connected by ('192.168.1.215', 60515)
[MSG] : RASP_ID : 5
[MSG] : Configurazione eseguita su RASP_ID : 1
[MSG] : Configurazione eseguita su RASP_ID : 2
[MSG] : Configurazione eseguita su RASP_ID : 3
[MSG] : Configurazione eseguita su RASP_ID : 4
[MSG] : Configurazione eseguita su RASP_ID : 5
Connected by ('192.168.1.214', 55187)
[MSG] : RASP_ID : 4
Connected by ('192.168.1.211', 51032)
[MSG] : RASP_ID : 1
[MSG] : RASP_ID : 4
[MSG] : RASP_ID : 1
Tutte le connessioni sono state stabilite
[T1] Reading and answer thread avviato
[T2] Send msg from keyboard thread avviato
(10:59:28)[T2] Messaggio da inviare ai nodi specifici:
1 REQ;0;1.
(10:59:48)[T2] Inviato REQ;0;1. al nodo 1 (IP: 192.168.1.211)
(10:59:53)[T1] Messaggio da 192.168.1.211 : TRAIN OK;0;1.
(10:59:58)[T2] Messaggio da inviare ai nodi specifici:
4 REQ;1;2.
(11:00:08)[T2] Inviato REQ;1;2. al nodo 4 (IP: 192.168.1.214)
(11:00:09)[T1] Messaggio da 192.168.1.214 : NOT_OK;1;2.
(11:00:18)[T2] Messaggio da inviare ai nodi specifici:
```

7 Sviluppi futuri

7.1 Dislocazione geografica

Al momento, i vari nodi sono interconnessi all'interno di una singola rete locale, in linea con la natura prototipale del progetto. Tuttavia, uno dei punti focali per il futuro è rappresentato dalla necessità di esaminare attentamente le performance del sistema quando le diverse schede saranno posizionate a distanze considerevoli l'una dall'altra. Questo passo è cruciale poiché l'espansione verso aree geografiche più ampie inevitabilmente comporterà un prolungamento dei tempi di comunicazione tra i nodi. Tale estensione avrà un impatto diretto sui tempi di esecuzione delle operazioni e sul rispetto delle rispettive deadline, richiedendo una riconsiderazione delle strategie di pianificazione e gestione delle attività.

La sfida più pressante in questo scenario è rappresentata dalla gestione delle comunicazioni tra i nodi distribuiti. L'adozione di distanze notevoli richiederà scelte decisive per quanto riguarda l'infrastruttura di rete. Ad esempio, si potrebbe considerare l'utilizzo di indirizzi IP pubblici statici o l'implementazione di una VPN per permettere lo scambio di messaggi tra le unità. La decisione in merito avrà un impatto rilevante sulla stabilità, l'affidabilità e la latenza delle comunicazioni, aspetti critici per garantire un funzionamento ideale del sistema. Un ulteriore elemento da considerare è l'esposizione delle schede a una rete pubblica. Ciò infatti introduce nuove sfide legate alla sicurezza della rete. Sarà necessario intraprendere un'analisi approfondita dei rischi di security legati alla connessione a reti pubbliche, valutando attentamente le potenziali vulnerabilità e i possibili scenari di attacco. Per mitigare tali rischi, potrebbe risultare indispensabile l'implementazione di soluzioni firewall avanzate, progettate appositamente per preservare l'integrità e la sicurezza dell'infrastruttura di interlocking.

7.2 Espansione della rete

L'attuale configurazione di prova rivela una natura estremamente minimale. Tuttavia, orientando lo sguardo verso il futuro, emerge un ambito di sviluppo di particolare rilievo: quello di esplorare le prestazioni del sistema su una scala di rete notevolmente più ampia e articolata.

In questa prospettiva, si prevede non solo l'incremento del numero di nodi che costituiscono l'architettura, ma anche un'affinazione complessiva attraverso la creazione di itinerari più intricati e complessi. Questa direzione introduce la possibilità di modellare scenari in cui numerosi host operano simultaneamente, muovendosi in direzioni opposte all'interno del sistema.

Nel valutare questo scenario, è essenziale dare un'attenzione particolare al monitoraggio dei tempi di esecuzione delle operazioni e alla verifica dell'aderenza alle scadenze previste. La complessità aggiuntiva introdotta consentirà di comprendere come vari il funzionamento complessivo del sistema e permetterà di individuare eventuali fasi critiche o colli di bottiglia.

Questo processo di valutazione fornirà informazioni preziose non solo sulle performance del sistema, ma anche sulla sua resilienza e affidabilità in contesti di maggiore complessità. Questi dati, a loro volta, costituiranno una base solida per l'ottimizzazione continua e la progettazione di soluzioni in grado di affrontare con successo scenari sempre più realistici e impegnativi.

7.3 Simulatore

Date le dimensioni contenute della rete nel prototipo, si è riusciti a sviluppare direttamente su schede fisiche, nonostante alcune difficoltà incontrate lungo il percorso. Tuttavia, proiettandoci verso una futura implementazione su larga scala, potrebbe rivelarsi estremamente vantaggioso concepire un simulatore in grado di emulare il comportamento di un insieme di schede.

Attraverso questa strategia, si potrebbe notevolmente accelerare il processo di test e di debug del codice, riducendo i tempi e aumentando l'efficienza complessiva. Inoltre, si aprirebbero interessanti opportunità per valutare la fattibilità e l'impatto di nuove funzionalità, senza l'onere di impegnare risorse considerevoli in termini di hardware e infrastrutture.

Oltre a ciò, questa metodologia presenterebbe un ulteriore vantaggio: la possibilità di simulare agevolmente le diverse distanze tra i nodi all'interno della rete. Questa simulazione potrebbe essere ottenuta attraverso l'implementazione di ritardi di comunicazione a livello software, consentendo un controllo più flessibile e una migliore comprensione dell'interazione tra i vari componenti.

Quindi l'adozione di un simulatore rappresenta un passo logico per prepararsi alle sfide dell'implementazione su vasta scala, fornendo un ambiente controllato e altamente efficiente per lo sviluppo, il test e la sperimentazione delle funzionalità, mentre si mitigano i rischi e si ottimizzano le risorse.

7.4 Distribuire la conoscenza

Nell'attuale stato del prototipo, le informazioni riguardanti la struttura e la composizione della rete vengono inizialmente detenute esclusivamente dall'host. Sarà poi compito di quest'ultimo condividere queste informazioni con i vari nodi che si collegheranno. Inoltre, nel caso in cui vi sia una richiesta di accesso ai log, i nodi stessi dovranno stabilire una connessione diretta con l'host per trasmettere tali dati. Tuttavia, come parte dello sviluppo futuro, potrebbe rivelarsi altamente benefico implementare dispositivi all'interno della rete specificamente dedicati a svolgere il compito di intermediari tra i nodi e l'host.

Questi intermediari, noti anche come broker, avranno un ruolo multifunzionale all'interno del sistema. Prima di tutto, saranno incaricati di ricevere costantemente aggiornamenti sullo stato operativo dei vari nodi e sui log di esecuzione e successivamente, in caso di richieste provenienti dagli host, risponderanno prontamente fornendo tali informazioni. Un secondo ruolo chiave dei broker sarà quello di agire come fornitori di configurazioni personalizzate per ciascun nodo.

L'implementazione di questa infrastruttura di brokeraggio può offrire vari vantaggi. Innanzitutto, decentralizza il processo di gestione delle informazioni di rete, alleggerendo il carico di lavoro sui singoli nodi. Inoltre, semplifica la gestione delle richieste e delle risposte tra i nodi e l'host centrale, consentendo una distribuzione più efficiente delle informazioni in tutta la rete.

In definitiva, questo approccio potrebbe portare a un'ottimizzazione del flusso di informazioni e dei processi di comunicazione all'interno della rete, migliorando complessivamente l'efficienza e la scalabilità del sistema.

7.5 Itinerari dinamici

Al momento, la struttura della rete è considerata un'informazione statica, cioè immutabile nel corso del funzionamento del sistema. Tuttavia, un percorso di sviluppo potenziale potrebbe prevedere l'aggiunta di una funzionalità dinamica che consenta di apportare modifiche alla composizione della rete durante l'esecuzione del

sistema. Questo significherebbe introdurre la capacità di aggiungere e rimuovere nodi dalla configurazione della rete e di modificare gli itinerari esistenti in tempo reale.

L'implementazione di questa caratteristica avrebbe implicazioni significative per la flessibilità e la gestione del sistema. Ad esempio, potrebbe risultare particolarmente vantaggiosa quando si ha la necessità di eseguire lavori di manutenzione o di riorganizzare un percorso ferroviario. In questi scenari, l'abilità di modificare la struttura della rete in modo dinamico potrebbe portare a un'ottimizzazione sostanziale del flusso operativo e dell'efficienza complessiva.

Per realizzare questa funzionalità, potrebbe essere progettato un nuovo task operante in parallelo con il normale funzionamento del sistema. Questo task avrebbe la responsabilità di monitorare eventuali richieste di modifica provenienti dagli utenti o da fonti interne, come nodi adiacenti o l'host centrale. Quando una modifica viene richiesta, il task può interagire con i nodi coinvolti e gli elementi della rete per implementare le modifiche richieste in modo sincronizzato.

La chiave per il successo di questa implementazione risiederà nell'efficacia dell'interazione tra il task di gestione dinamica della rete e gli altri elementi del sistema. Sarà fondamentale garantire la coerenza dei dati, evitare conflitti e assicurarsi che le modifiche vengano eseguite in modo sicuro.

7.6 Priority nelle prenotazioni

Una direzione intrigante per lo sviluppo futuro potrebbe essere l'implementazione di un sofisticato meccanismo di prenotazione basato sulle priorità dei treni. Attualmente, se più treni tentano di riservare lo stesso itinerario, prevale la semplice logica "chi arriva prima, prenota prima". Tuttavia, sarebbe altrettanto interessante esaminare la gestione delle richieste di prenotazione che si verificano in prossimità temporale, provenienti da treni diversi, e adottare un sistema di assegnazione basato su priorità.

Questo nuovo meccanismo introducerebbe un elemento di decision-making più sofisticato, in grado di valutare diverse variabili per determinare quale treno debba ricevere l'assegnazione di un particolare itinerario. Tra le diverse variabili che potrebbero essere considerate per stabilire l'importanza di un treno vi sono:

- Numero di passeggeri trasportati: un treno che trasporta un numero maggiore di passeggeri potrebbe ricevere una priorità più elevata, poiché soddisfa le esigenze di un maggior numero di viaggiatori
- Ritardi accumulati: un treno che ha accumulato un ritardo maggiore rispetto agli orari programmati potrebbe ricevere una priorità superiore per recuperare il tempo perso
- Storia delle prenotazioni rifiutate: un treno che ha subito molte richieste di prenotazione rifiutate in passato potrebbe ricevere un vantaggio per assicurare che possa completare il suo itinerario
- Tipologia di servizio: treni con servizi speciali o di alta priorità, come quelli di soccorso o trasporto merci urgenti, potrebbero ricevere una priorità elevata
- Disponibilità di percorsi alternativi: se un treno ha opzioni di percorsi alternativi, la scelta potrebbe dipendere dalla sua priorità globale

Questi parametri potrebbero essere ponderati e valutati all'interno di un algoritmo di assegnazione delle priorità. L'obiettivo principale sarebbe quello di massimizzare l'efficienza dell'utilizzo delle risorse disponibili, assicurando al contempo che i treni più rilevanti dal punto di vista operativo ricevano l'attenzione necessaria.

7.7 Migrazione a Rust

Il progetto è stato inizialmente sviluppato utilizzando il linguaggio C. Tuttavia, uno dei punti focali nel percorso futuro potrebbe essere la transizione verso il linguaggio Rust, grazie all'implementazione diretta di VxWorks del supporto a questo linguaggio di programmazione. Rust è un linguaggio di basso livello che coniuga le prestazioni tipiche del C/C++ con un sistema di gestione della memoria estremamente sicuro. Questo è reso possibile dai meccanismi integrati che assicurano type-safety e memory-safety, contribuendo a prevenire i classici errori legati alla gestione della memoria e a migliorare la robustezza delle applicazioni.

Oltre alle sue caratteristiche di sicurezza avanzata, Rust offre il vantaggio di essere stato progettato con l'obiettivo di integrarsi facilmente con il software già sviluppato in C/C++. Questa flessibilità si dimostra particolarmente preziosa nei contesti in cui è necessario coesistere con sistemi preesistenti o componenti legacy. Inoltre, Rust è altamente adattabile agli ambienti embedded, dove il controllo a basso livello è fondamentale. La documentazione completa di Rust è disponibile sulla pagina ufficiale del linguaggio al seguente [link](#). Questa risorsa costituisce un punto di partenza cruciale per coloro che desiderano approfondire Rust, esplorarne le caratteristiche peculiari e comprendere come integrarlo efficacemente nei loro progetti.

References

- [1] Fantechi Alessandro. “Distributing the challenge of model checking interlocking control tables”. In: *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies - 5th International Symposium, pp. 276–289, Heraklion, Crete, 15-18 October 2012*. Springer, Berlin, DEU, 2012. DOI: [10.1007/978-3-642-34032-1_26](https://doi.org/10.1007/978-3-642-34032-1_26).
- [2] Giacomo Bucci, Laura Carnevali, Lorenzo Ridi, and Enrico Vicario. “Oris: A tool for modeling, verification and evaluation of real-time systems”. In: *STTT* 12 (Sept. 2010), pp. 391–403. DOI: [10.1007/s10009-010-0156-8](https://doi.org/10.1007/s10009-010-0156-8).
- [3] Alessandro Fantechi, Anne Haxthausen, and Michel Nielsen. “Model Checking Geographically Distributed Interlocking Systems Using UMC”. In: Jan. 2017, pp. 278–286. DOI: [10.1109/PDP.2017.66](https://doi.org/10.1109/PDP.2017.66).