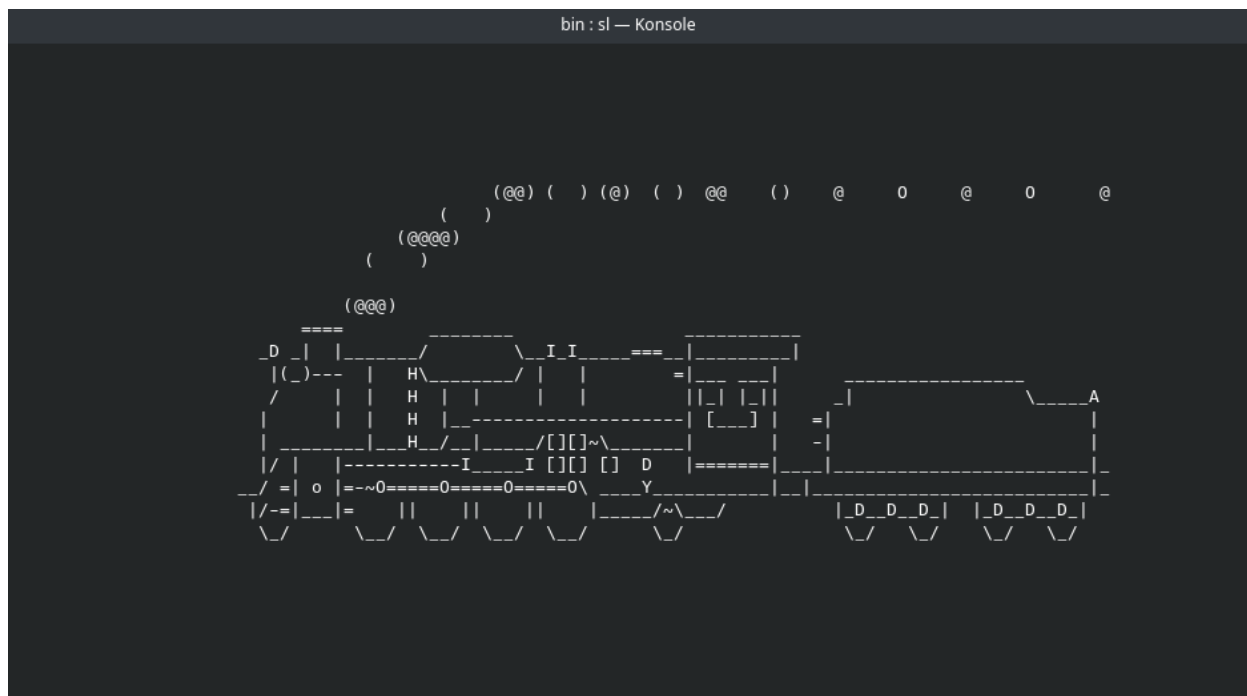


# RELAZIONE PROGETTO DI SISTEMI OPERATIVI

*Sistema ferroviario ETCS1/2 in C*



**Riccardo Degli Esposti**

[riccardo.degli1@stud.unifi.it](mailto:riccardo.degli1@stud.unifi.it) - Mat. 7047904

25/06/2022

A.S.: 2021-2022

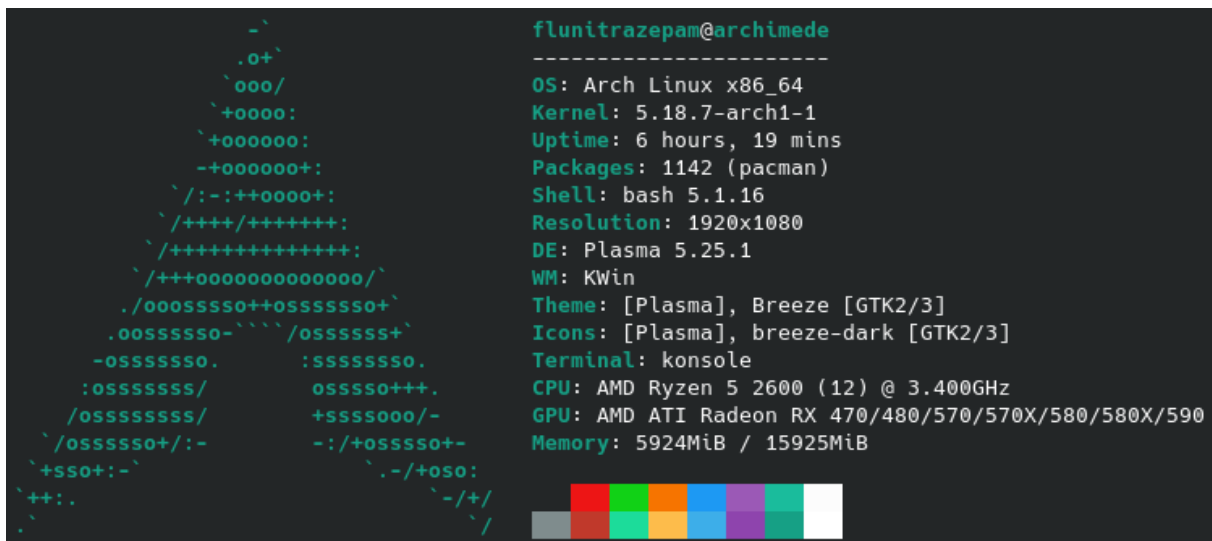
## Compilazione e Esecuzione

Per compilare il progetto è sufficiente eseguire il comando `make` nella directory in cui è stato estratto.

Per eseguire il programma è necessario spostarsi nella cartella `bin` ed eseguire `padre_treni` passando gli argomenti come da specifica.

## Sistema Obiettivo

```
flunitrazepam@archimede
-----
OS: Arch Linux x86_64
Kernel: 5.18.7-arch1-1
Uptime: 6 hours, 19 mins
Packages: 1142 (pacman)
Shell: bash 5.1.16
Resolution: 1920x1080
DE: Plasma 5.25.1
WM: KWin
Theme: [Plasma], Breeze [GTK2/3]
Icons: [Plasma], breeze-dark [GTK2/3]
Terminal: konsole
CPU: AMD Ryzen 5 2600 (12) @ 3.400GHz
GPU: AMD ATI Radeon RX 470/480/570/570X/580/580X/590
Memory: 5924MiB / 15925MiB
```



## Elementi Facoltativi

Elemento Facoltativo	Realizzato	Descrizione e metodo o file principale
Soluzioni per gestire letture/scritture concorrenti	SI	<code>processo_treno.c:occupaSegmento()</code> utilizzando <code>fcntl()</code> e lock sui file
In caso di informazione discordante tra RBC e boe, il TRENO rimane fermo	SI	<code>processo_treno.c:missione()</code> <code>rbc_client.c</code> <code>rbc.c:servizio()</code>
Terminazione di <code>padre_treni</code> e <code>processo_treno</code> basata sul segnale <code>SIGUSR1</code>	SI	<code>padre_treni.c:aspettaTreni()</code> <code>processo_treno.c:postMissione()</code>
Terminazione di RBC basata sul segnale <code>SIGUSR2</code>	SI	<code>padre_treni.c:getRBCpid()</code> <code>padre_treni.c:ETCS2()</code> <code>rbc.c:SIGUSR2_handler()</code>

## Progettazione e Implementazione

L'applicativo è suddiviso in 4 eseguibili, ognuno dei quali è composto da alcuni sottomoduli comuni distribuiti sui vari file .c del progetto. Di seguito si descrivono brevemente i vari moduli, le parti principali delle loro interfacce e gli eseguibili, nonché le interazioni tra i processi, presentando un diagramma semplificato di un'esecuzione tipo del programma.

### log

Fornisce un'interfaccia per la creazione dei file di log di ciascun processo e per la stampa su di essi. Fornisce 5 livelli di verbosità:

- **LOG\_FATAL:** per quando si incontra un errore irreversibile ed il programma deve essere terminato immediatamente. È il livello più basso.
- **LOG\_ERROR:** per quando si incontra un errore temporaneo che può essere risolto senza terminare il programma.
- **LOG\_WARN:** per quando si verificano eventi o condizioni inaspettate; il programma può continuare, ma potrebbe comportarsi in maniera inaspettata.
- **LOG\_INFO:** per quando il programma sta funzionando correttamente e vuole riportare alcune informazioni sul suo stato. È il livello di default.
- **LOG\_DEBUG:** per quando si vogliono registrare informazioni utili al fine di verificare il funzionamento del programma e la correttezza del suo stato.

*log\_setLogLevel:* imposta il livello di verbosità del log per il processo corrente e per i suoi figli. Vengono stampati soltanto messaggi con livello minore o uguale a quello impostato.

*log\_init:* inizializza il sistema di logging. Crea il file e recupera il livello impostato.

*log\_printf:* stampa, utilizzando una stringa di formattazione, sul file di log. Sono fornite anche alcune macro per ogni livello di logging che aggiungono ulteriori informazioni utili a fine di debug.

Per una miglior leggibilità dei file di log (soprattutto se il livello è settato a **LOG\_DEBUG**) si consiglia l'utilizzo di Visual Studio Code (o derivati) e l'estensione [Log File Highlighter di Emil Åström](#), i cui file di configurazione, realizzati appositamente, sono inclusi nel progetto.

### mappa

Mette a disposizione varie funzioni per convertire o reperire alcune informazioni sulla mappa, sugli itinerari e sulle tappe nei vari formati utilizzati.

## socket

Mette a disposizione alcune funzioni per semplificare la creazione e l'utilizzo delle socket. Può essere utilizzato direttamente dagli eseguibili, ma viene utilizzato anche da altri moduli.

## registro\_client

Permette di interfacciarsi e comunicare con il processo registro in modo semplice.

*rc\_init*: inizializza il client, si connette a registro e si autentica. I treni possono accedere soltanto alle informazioni che li riguardano; la stessa cosa vale per gli altri tipi di client.

*rc\_getItinerario*: recupera l'itinerario e l'ID del treno. Utilizzabile soltanto dai treni.

*rc\_getMappa*: recupera il numero di treni e l'itinerario di ciascuno di essi. Utilizzabile soltanto da client SUPER.

*rc\_getNumeroTreni*: recupera soltanto il numero di treni presenti nello scenario. Utilizzabile soltanto da client SUPER.

## rbc\_client

Permette di interfacciarsi e comunicare con il processo RBC in modo semplice.

*rbc\_init*: inizializza il client, si connette a RBC e comunica il suo ID.

*rbc\_richiediMA*: Richiede l'autorizzazione ad occupare il segmento successivo. RBC può negarla in caso ci sia un altro treno già presente oppure se il treno sta cercando di accedere ad un segmento che non fa parte del suo itinerario.

*rbc\_comunicaEsitoMovimento*: comunica a RBC se il treno si è mosso oppure no. Un treno può decidere di non muoversi in caso il segmento sia già occupato. Realisticamente l'esito sarà sempre positivo.

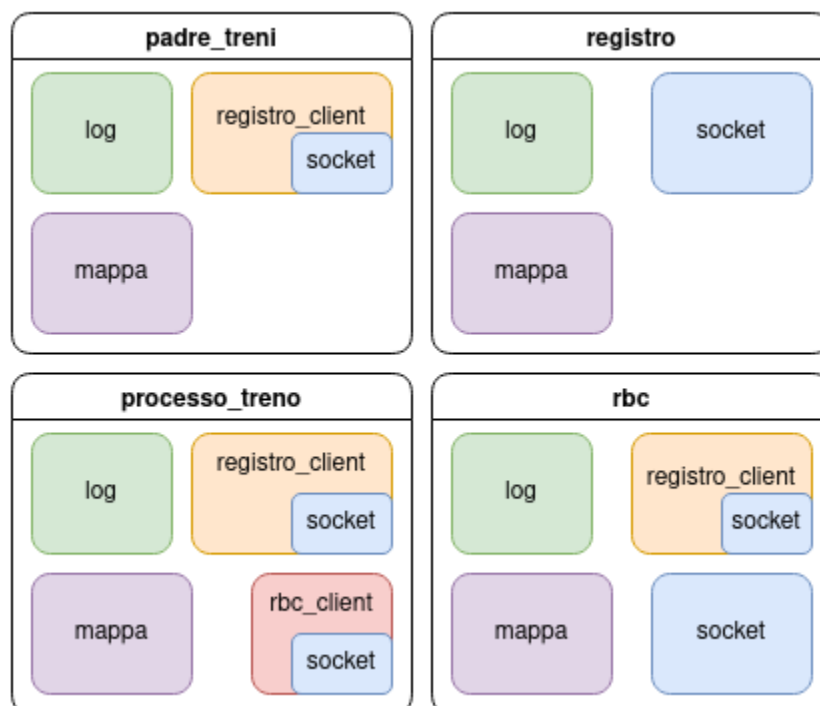
## Eseguibili

Il programma inizia con l'esecuzione di **padre\_treni**. Questo processo è responsabile di creare tutti gli altri processi; inoltre crea le directory e i file necessari al funzionamento di questi.

**Registro** è responsabile di mantenere le informazioni relative alle due mappe e di fornirle agli altri processi. Implementa un server concorrente per servire i vari client.

**RBC** controlla gli accessi dei treni ai vari segmenti, fornendo o negando l'autorizzazione a ciascuno di essi. In più mantiene aggiornato lo stato dell'intera rete ferroviaria, quindi è a conoscenza della posizione di ciascun treno e dello stato di ciascun segmento. Per motivi di consistenza di dati, implementa un server iterativo, così da non dover sincronizzare i vari sottoprocessi sulle strutture dati interne.

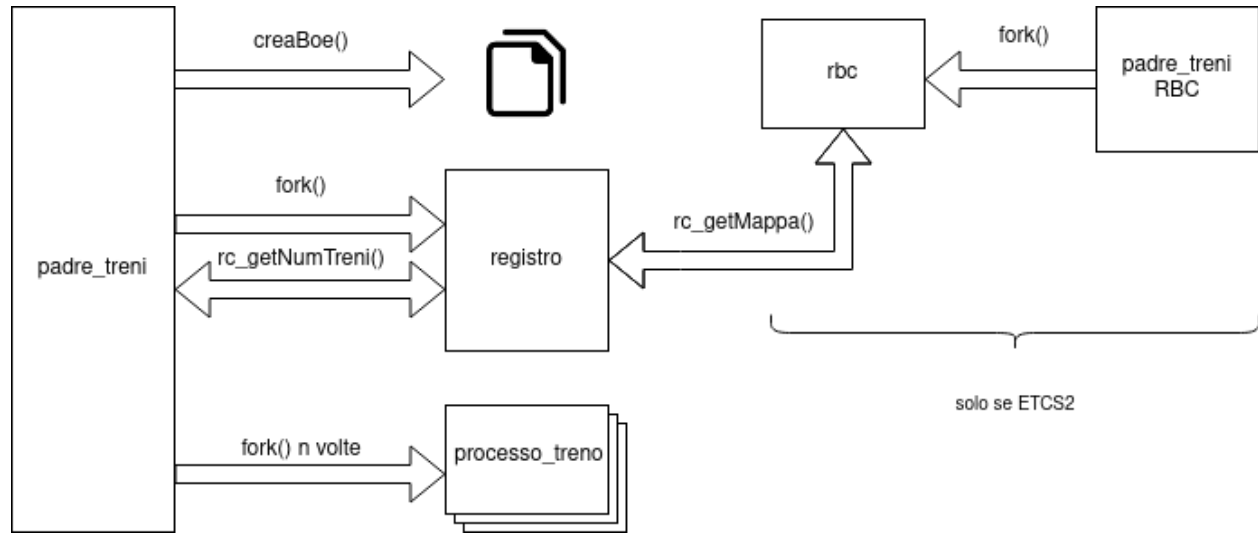
I vari **processo\_treno** sono i processi che eseguiranno la missione ferroviaria. Il loro compito è quello di percorrere l'intero itinerario, organizzandosi in modo da non essere mai più di uno su ciascun segmento (escluse le stazioni). Una volta giunti alla stazione di arrivo, lo comunicano a **padre\_treni**, il quale farà pulizia ordinando ai vari processi di terminare.



*Diagramma dei moduli utilizzati dai processi.*

## Esecuzione Tipo

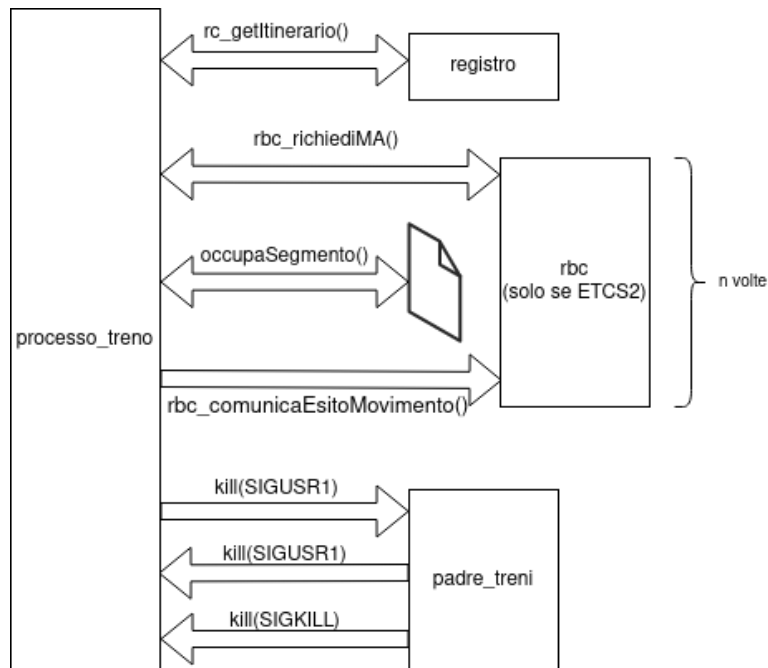
### Avvio



Una volta avviato `padre_treni`, questo, dopo aver controllato la correttezza degli argomenti, fa pulizia eliminando i file di log delle esecuzioni precedenti, poi crea i file che rappresentano i vari segmenti di binario. Successivamente crea il processo `registro` (passando la mappa come parametro) e recupera il numero di treni presenti. Infine crea tanti `processo_treno` quanti indicati da `registro`, fornendogli la modalità. Se invece la modalità è ETCS2 e viene fornito il parametro apposito, allora `padre_treni` crea soltanto RBC, il quale, dopo esser stato avviato comunicherà con `registro` per ricevere la mappa (e quindi tutti gli itinerari).

## Missione

Per prima cosa, ogni processo\_treno richiede l'itinerario a registro; durante questo passaggio, viene anche assegnato l'ID al treno. Adesso il treno percorre iterativamente le varie tappe del suo itinerario. Nel caso di ETCS2 chiede ad RBC l'autorizzazione a muoversi. Se l'autorizzazione è accordata oppure se la modalità è ETCS1 allora il treno prova ad occupare il segmento, controllando che sul file della boa ci sia scritto 0. In tal caso procede ad occuparlo (scrivendoci 1) e a liberare il

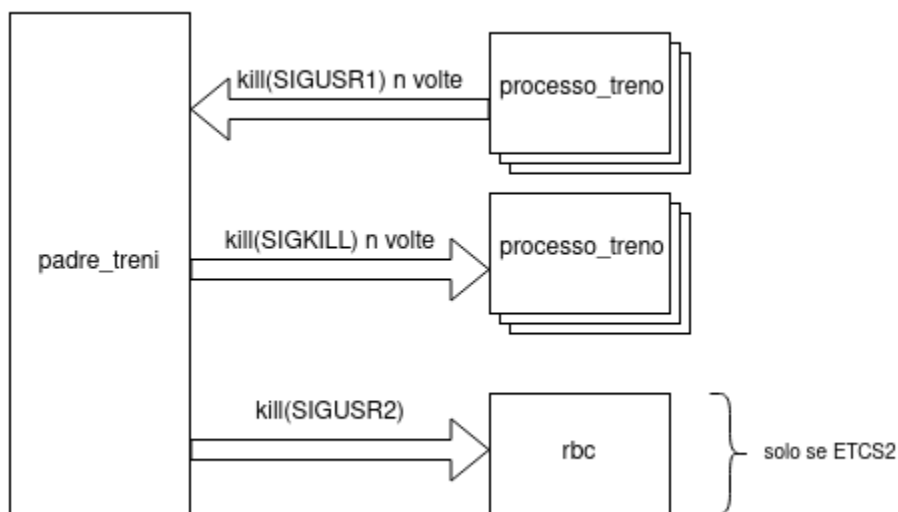


segmento su cui si trova al momento. Eventualmente comunica a RBC se si è spostato. Una volta terminata la missione lo comunica a padre\_treni inviandogli un segnale SIGUSR1. Siccome su linux la consegna dei segnali non è garantita (ad esempio se è già presente un altro segnale in coda, un altro segnale dello stesso tipo non viene consegnato) padre\_treni invia un segnale SIGUSR1 di acknowledgement al treno. Ricevuto questo segnale, il treno si mette in attesa di essere ucciso, altrimenti riprova.

## Terminazione

Una volta che tutti i treni hanno finito, padre\_treni li uccide tutti inviando un SIGKILL. In ETCS2 uccide anche RBC, inviandogli un SIGUSR2. Si preoccupa anche di eliminare eventuali socket non rimosse.

Registro invece muore da solo, dopo 20 secondi dall'ultima connessione ricevuta.



## Esecuzione e verifica di funzionamento

### ETCS1 MAPPA2

[2022-07-01 20:16:53]	[Attuale: --]	[Successiva: S2]
[2022-07-01 20:16:55]	[Attuale: S2]	[Successiva: MA5]
[2022-07-01 20:16:57]	[Attuale: MA5]	[Successiva: MA6]
[2022-07-01 20:16:59]	[Attuale: MA6]	[Successiva: MA7]
[2022-07-01 20:17:01]	[Attuale: MA7]	[Successiva: MA3]
[2022-07-01 20:17:03]	[Attuale: MA7]	[Successiva: MA3]
[2022-07-01 20:17:05]	[Attuale: MA3]	[Successiva: MA8]
[2022-07-01 20:17:07]	[Attuale: MA8]	[Successiva: S6]
[2022-07-01 20:17:09]	[Attuale: S6]	[Successiva: --]

*T1.log*

[2022-07-01 20:16:53]	[Attuale: --]	[Successiva: S6]
[2022-07-01 20:16:55]	[Attuale: S6]	[Successiva: MA8]
[2022-07-01 20:16:57]	[Attuale: MA8]	[Successiva: MA3]
[2022-07-01 20:16:59]	[Attuale: MA8]	[Successiva: MA3]
[2022-07-01 20:17:01]	[Attuale: MA3]	[Successiva: MA2]
[2022-07-01 20:17:03]	[Attuale: MA2]	[Successiva: MA1]
[2022-07-01 20:17:05]	[Attuale: MA1]	[Successiva: S1]
[2022-07-01 20:17:07]	[Attuale: S1]	[Successiva: --]

*T4.log*

Come si può vedere dalle immagini soprastanti, in modalità ETCS1 i treni attendono che il segmento su cui si devono spostare sia libero, altrimenti attendono il ciclo successivo. Nel caso in oggetto il segmento MA3 è stato occupato prima da T4; successivamente, quando T1 ha provato anch'esso ad occuparlo, ha rilevato che il segmento non era libero, perciò non si è spostato, ma ha tentato di nuovo al turno successivo.



## ETCS2 MAPPA2

```
[2022-06-23 14:44:40] [Treno: T5] [Attuale: --] [Richiesto: S5] [Autorizzato: SI]
[2022-06-23 14:44:42] [Treno: T5] [Attuale: S5] [Richiesto: MA4] [Autorizzato: SI]
[2022-06-23 14:44:44] [Treno: T5] [Attuale: MA4] [Richiesto: MA3] [Autorizzato: NO]
[2022-06-23 14:44:46] [Treno: T5] [Attuale: MA4] [Richiesto: MA3] [Autorizzato: SI]
[2022-06-23 14:44:48] [Treno: T5] [Attuale: MA3] [Richiesto: MA2] [Autorizzato: SI]
[2022-06-23 14:44:50] [Treno: T5] [Attuale: MA2] [Richiesto: MA1] [Autorizzato: SI]
[2022-06-23 14:44:52] [Treno: T5] [Attuale: MA1] [Richiesto: S1] [Autorizzato: SI]
```

*rbc.log filtrato per visualizzare solamente T5*

```
[2022-06-23 14:44:40] [Attuale: --] [Successiva: S5]
[2022-06-23 14:44:42] [Attuale: S5] [Successiva: MA4]
[2022-06-23 14:44:44] [Attuale: MA4] [Successiva: MA3]
[2022-06-23 14:44:46] [Attuale: MA4] [Successiva: MA3]
[2022-06-23 14:44:48] [Attuale: MA3] [Successiva: MA2]
[2022-06-23 14:44:50] [Attuale: MA2] [Successiva: MA1]
[2022-06-23 14:44:52] [Attuale: MA1] [Successiva: S1]
[2022-06-23 14:44:54] [Attuale: S1] [Successiva: --]
```

*T5.log*

In modalità ETCS2 si può verificare che il treno e RBC comunicano correttamente. Nell'esempio si nota che al 3° turno RBC non autorizza il treno a muoversi (un altro treno aveva già richiesto e ottenuto l'autorizzazione per quel segmento e vi si era spostato), perciò T5 ritenta 2 secondi dopo, questa volta ottenendo l'autorizzazione.