RELAZIONE PROGETTO “Fly-by-Wire: LabSO edition”

Autori:

Massimiliano Sirgiovanni 6357449 massimiliano.sirgiovanni@stud.unifi.it
Luca Donzelli 6355924 luca.donzelli1@stud.unifi.it

Data di consegna: 28/06/2020

Istruzioni per la compilazione ed esecuzione:

Per compilare il file è stato utilizzato un **makefile**, chiamato Makefile.
Di conseguenza, per la compilazione, vanno eseguite, sulla shell, le seguenti due istruzioni:

1. ***make clean***
2. ***make***

Il comando make, oltre a eseguire i comandi per la compilazione dei singoli processi, si occupa tramite il comando make install di creare le varie sottocartelle necessarie al progetto (src, bin, log, tmp) e di organizzare i file nei folder appositi.
Una volta eseguite le due istruzioni, sempre sulla shell, sarà necessario eseguire il file start, passandogli in input il file G18.txt. Questo può essere fatto con la seguente istruzione:

./start G18.txt

Ovviamente tale istruzione è corretta se ci si trova all’interno della cartella principale del progetto, dove è contenuto il makefile. Il file G18.txt è presente anche all’interno della *mainFolder*, ma si può anche inviare un percorso file differente, ad esempio:

./start /home/ceccarelli/labso/G18.txt

Sistema obiettivo:

Il computer, in cui è stato effettuato il debugging principale, ha installata la distribuzione di Linux Lubuntu 19.10. Le caratteristiche dello stesso computer sono:

- Processore → Intel I3 4170 3.6GHz
- Ram → 8 GB

Inoltre, dato il problema del distanziamento sociale e della chiusura delle università, è stato necessario utilizzare un altro computer per la programmazione. Tuttavia, si presuppone, per una bassa capacità di calcolo, non è stato possibile usare la virtual machine, poiché quest’ultima effettuava qualsiasi tipo di operazione con una lentezza snervante. Per questo è stata utilizzata una funzionalità di Windows, WSL (Windows Subsystem for Linux).
Purtroppo, in corso d’opera, ci siamo accorti che alcune funzioni di Linux non erano utilizzabili, più importanti fra tutte la pipe. Di conseguenza tale computer è stato utilizzato solo per programmare e per effettuare testing focalizzato su componenti in cui l’utilizzo di pipe era superfluo. Per il debugging principale è stato utilizzato il computer sopracitato.
La distribuzione di Linux installata è Ubuntu. Le caratteristiche del PC invece sono:

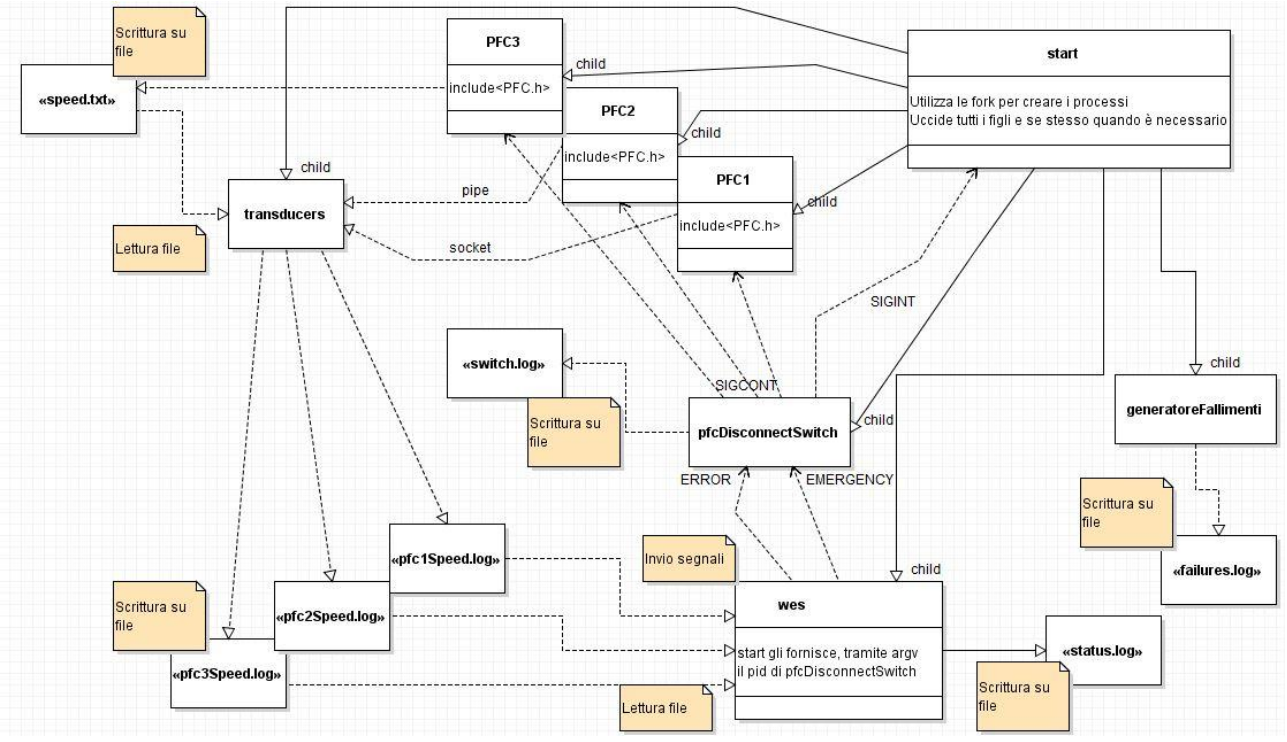
- Processore → Intel(R) Pentium(R) CPU N3710 @ 1.60GHz
- Ram → 4 GB

Elementi Facoltativi:

Elemento Facoltativo	Realizzato (SI/NO)	Metodo o file principale
Utilizzo Makefile per compilazione	SI	Makefile
Organizzazione in folder, e creazione dei folder al momento della compilazione	SI	Makefile
Riavvio di PFC1, PFC2, PFC3 alla ricezione di EMERGENZA	NO	
Utilizzo macro nel Generatore Fallimenti per indicare le probabilità.	SI	generatoreFallimenti.c
PFC Disconnect Switch sblocca il processo se bloccato, e lo riavvia altrimenti.	NO	
(continua da sopra) In entrambi i casi, il processo in questione deve riprendere a leggere dal punto giusto del file G18.txt.	NO	

Tuttavia, per quanto riguarda il penultimo elemento facoltativo, nonostante il **pfcDisconnectSwitch** non riesca a riavviare un pfc, nel caso in cui sia morto, *sblocca i pfc bloccati*, in caso di ricezione del segnale **ERRORE**.

Progettazione ed implementazione:



Come si può osservare dallo schema sopra presentato, il processo ***start*** è colui che si occuperà di eseguire gli altri processi e di ucciderli, quando necessario.
Dopo un breve controllo dell'input, *start* registra, per prima cosa, il segnale ***SIGINT***, assegnandogli, come **signal Handler**, **terminator**. Tutto ciò sarà utile per la terminazione del programma, sia nel caso in cui si dovesse terminare di leggere il file G18.txt, ma anche nel caso in cui dovesse verificarsi *"un'emergenza"*. Per terminare l'applicazione, **start ucciderà uno per uno i propri figli**, inviandogli un segnale **SIGKILL**, tramite la function kill. I pid dei processi figli vengono salvati in apposite variabili globali. Anche in caso di terminazione da tastiera, verrà richiamato il signal Handler terminator.
Passando alla creazione dei processi, **start richiama un metodo apposito per la creazione di ogni processo**. In tali metodi, i quali sono sette, **vengono creati i processi figli**, tramite **fork**. Una volta creato un figlio, quest'ultimo eseguirà un **execv**, eseguendo il codice dell'eseguibile passato a quest'ultima. Inoltre, se necessario verranno fornite in input tutte le informazioni necessarie.
Ad esempio, nel caso dei pfc viene passato loro il percorso del file G18.txt, oppure, nel caso del generatore di Fallimenti, vengono passati, a quest'ultimo, i pid dei tre pfc. Vengono, quindi, creati, in ordine: *PFC1, PFC2, PFC3, transducers, generatoreFallimenti, pfcDisconnectSwitch e wes*.

Dato che i **PFC** devono eseguire, in parte, lo stesso codice, abbiamo creato un file PFC.c, col quale realizzare un **Header File**, *"PFC.h"*, da includere in ogni pfc.
Nel file PFC.c, sono presenti otto metodi utili a tutti e tre i pfc. Importante, per un metodo nello specifico, è la macro **MAINSTRING**, **contenente** la stringa **"GPGLL"**. Nello specifico i metodi, ed il loro utilizzo, sono:

FILE *open_file(char *path, char *mode);	Utile per aprire un generico file. Utilizzato in tutti i pfc per aprire G18.txt ed in PFC3 per aprire il file per comunicare con transducers
int line_contains_coordinates(char *line);	Passatagli una riga del file G18.txt, restituisce 1 se i primi sei caratteri della riga sono uguali al valore della macro,"MAINSTRING". Altrimenti restituisce 0. Questo metodo è necessario per stabilire quali righe del file contengono le informazioni utili per i pfc.
struct Coordinates parse_line(char *line);	Estrapola le coordinate dalla riga del file G18.txt. Saranno fondamentali per il calcolo della distanza.
double calculate_distance(struct Coordinates c0, struct Coordinates c1);	Utilizzando le coordinate trovate dal metodo precedente, calcola la distanza compiuta "dall'aereo". La distanza è, ovviamente, necessaria per il calcolo della velocità.
double deg_to_rad(double deg);	Converte un valore espresso in gradi con l'equivalente in radianti (utilizzato nel metodo precedente)

void sendPid(pid_t pid, char * file);	Scrive, su un file, un pid passatogli in input. Utile per permettere ai pfc di inviare il proprio pid ad altri processi
int intOf(char c);	Converte una stringa in un intero. E' utilizzato per convertire le coordinate salvate sotto forma di stringhe in interi.

Inoltre, abbiamo implementato una struttura, sempre all'interno di *PFC.c*, utile per la gestione delle coordinate,rappresentate da latitudine e longitudine.
I **tre PFC** dovranno, quindi, **inviare il pid ed effettuare il calcolo della velocità** con i metodi presenti in *PFC.c*, importati tramite un'istruzione *#include<PFC.h>* .
La velocità è uguale alla distanza, dato che il tempo di percorrenza è sempre pari ad un secondo e che la formula della velocità è Spazio/Tempo. Per effettuare tali operazioni, è implementato in ogni PFC un **ciclo while che termina nel momento in cui sono state esaminate tutte le righe di *G18.txt***.
L'ultima funzionalità comune a tutti i PFC è la gestione del segnale **SIGUSR1**. E' richiesto, dal testo dell'esercizio, che tale segnale comporti un left shift di 2 bits della velocità calcolata una volta arrotondata all'intero più vicino. Questo viene effettuato tramite il metodo *shiftSpeed,* il quale realizza lo shift richiesto, ma soprattutto tramite la **variabile globale shift**. Infatti, quest'ultima è inizializzata a 0 di default, ma viene posta ad uno dal signal Handler che si occupa di gestire SIGURS1. Questo permette di riconoscere, all'interno del ciclo, è stato inviato tale segnale. Questo viene effettuato tramite le seguenti istruzione:

```
if(shift==1){
        distance = shiftSpeed(distance);
        shift = 0;
}
```

Una volta effettuato lo shift, alla variabile globale viene riassegnato il valore 0, in modo tale che all'iterazione successiva non venga effettuato lo shift.
Oltre a queste operazioni in comune, i tre processi comunicano con il transducers in tre modi diversi *(PFC1 tramite socket, PFC2 tramite pipe e PFC3 tramite scrittura su file)*. **La socket e la pipe, entrambe non bloccanti, vengono aperte prima del ciclo**, nel quale verranno esaminate tutte le righe di *G18.txt*, e **chiuse subito dopo**. Vengono aperte rispettivamente dai metodi *open_socket* ed *open_pipe*, che eseguono tutte le istruzioni necessarie per l'apertura in scrittura. Nello specifico, *open_pipe* tenta di aprire la pipe finché non sarà disponibile, quindi aspettando che venga aperta in lettura dal transducers. D'altra parte il **PFC3 apre un file temporaneo**, *"speed.txt"*, il quale viene **aperto e chiuso ad ogni iterazione del ciclo**.
Al termine del ciclo di ogni pfc, nel canale di comunicazione, da loro usato, **viene inserito un carattere speciale**, per consentire al transducers di capire quando i pfc hanno terminato di leggere il file G18.txt.

Una volta che i pfc hanno inviato le velocità calcolate, spetta al **transducers** riceverle. Il compito del transducers è **leggere le velocità inviategli e scriverle in tre file di log**, uno per ogni pfc. Di conseguenza le uniche operazioni che deve compiere il transducers sono quelle di apertura, lettura e chiusura della pipe, della socket e del file speed.txt. Inoltre, deve aprire, trascrivere le velocità e chiudere i file di log.

La maggior parte di **tali operazioni vengono effettuate all'interno di un ciclo while**, il quale termina quando il transducers ha letto tutti i dati in uno dei tre canali di comunicazione. La condizione del ciclo while è la seguente:

endFile == 0 && strcmp(str1, "-")!=0 && strcmp(str2, "-")!=0

Con endFile inizializzato a zero prima del ciclo. E' possibile cambiare il valore di endFile all'interno del ciclo tramite le seguenti istruzioni:

```
if(strcmp(str3, " ")==0){
        endFile = 1;
}
```

"str3" rappresenta l'ultima riga estratta dal file *speed.txt*
Quando il ciclo while termina, **l'applicazione ha portato a termine il suo compito** e può essere chiusa. Per fare ciò, **transducers invia al padre**, *start*, **un segnale SIGINT**, che, come descritto sopra, attiverà il signal Handler *terminate*.

Durante la comunicazione dei *PFC* con il *transducers,* possono avvenire delle complicazioni causate dal **generatore di fallimenti**. Infatti, il compito di tale processo è quello di **simulare, con una determinata probabilità, possibili guasti** che possano avvenire nei PFC. **Le probabilità** dei quattro eventi richiesti dal testo del progetto **sono state inizializzate in delle macro apposite**.
Al fine di poter indicare in maniera più elegante e funzionale i PFC all'interno del generatore di fallimenti, e per poter scrivere una documentazione adeguata nel file failures.log, è stata creata una **struttura apposita, "PFC"**, contenente come attributo il **nome ed il pid del PFC** rappresentato.
I pid dei tre pfc vengono mandati **in input** al generatore di fallimenti direttamente dal padre, start. Vengono, quindi, inseriti nell'attributo pid di tre strutture, appositamente create per identificare il PFC1, il PFC2 ed il PFC3. Ottenuto quindi il pid, ed assegnatogli di default il nome, si procede con le istruzioni per generare gli errori, tramite invio di segnali.
All'interno di un ciclo infinito, bisogna, primariamente, selezionare il pid a cui inviare i segnali. Quest'operazione viene effettuata tramite un metodo:

*"PFC * extractPFC(PFC * PFC1, PFC * PFC2, PFC * PFC3)"*

Il quale, tramite **rand()**, seleziona in maniera casuale uno dei tre pfc. Più nello specifico, la funzione rand() **estrae un numero casuale**, al quale verrà poi applicata l'operazione di **modulo tre (%3)**. Questo numero, che potrà assumere valore da 0, 1 o 2, viene salvato in una variabile e attraverso uno **switch** si seleziona il PFC da estrarre.
Una volta estratto, il puntatore al PFC viene passato alla funzione *sendSignals*. Questa funzione, per ognuno dei quattro errori previsti, utilizzerà il metodo rand(), visto nella funzione precedente, applicandogli la probabilità prevista.
Qualora il risultato di una di queste operazioni fosse pari ad uno, verrebbe inviato il segnale indicato al PFC estratto.
Ogni volta che viene inviato un segnale, tale operazione viene salvata nel file *failures.log*. Questo è realizzato tramite i metodi *improveDocumentation*, il quale rende più leggibile il contenuto del file, e *logFile*, il quale apre il file di log, vi scrive all'interno una stringa e lo chiude. Tutte queste istruzioni, come accennato precedentemente, sono inserite all'interno di

un ciclo infinito. Di conseguenza, il **generatore di fallimenti terminerà solo se ucciso da un altro processo**, in questo caso dal padre start.

A **verificare la corretta esecuzione** dei PFC, che potrebbe essere minata dal generatore di Fallimenti, è il **wes**. Il compito principale del wes è **verificare che i valori presenti nei tre file di log siano uguali**, in caso contrario **avvertire il pfcDisconnectSwitch** dell'anomalia. Partiamo con ordine, per prima cosa al wes servirà **il pid del pfcDisconnectSwitch**, per potervi comunicare. Come detto in precedenza, il pfcDisconnectSwitch viene creato, da start, prima del wes. Questo permette a **start** di **fornire**, come input, **il pid richiesto**. Il wes, quindi, **salva in una variabile il pid**, contenuto in *argv*, e prosegue con l'esecuzione. Vengono, ovviamente, inizializzati i buffer, per contenere le righe dei file, ed aperti i file di log richiesti *(speedPFC1.log, speedPFC2.log e speedPFC3.log)*. Come prossimo passo, **entrerà nel ciclo infinito**, dal quale uscirà solo una volta ucciso da start. In tale ciclo, legge le righe dei file di log, salvandole nei buffer precedentemente inizializzati, e le confronta. **Qualora i valori non dovessero essere identici**, stamperà a video l'errore trovato ed **invierà un segnale al pfcDisconnectSwitch**, tramite kill. **I segnali** che può inviare sono due, **ERROR ed EMERGENCY**, inizializzati prima del main tramite define. Dopo aver inviato un segnali, viene trascritto nel file *status.log*.
In più, qualora vi fosse una **EMERGENCY**, il wes **termina la sua esecuzione**. Questo per evitare che il wes continui a stampare a schermo o a scrivere nei file di log mentre start sta per eliminare tutti i processi.
Se i valori letti dai tre file dovessero risultare identici, il wes stamperà a video un messaggio di conferma e scriverà la stringa "OK" sul file status.log.

Quindi, per ogni errore trovato dal wes, viene immediatamente attivato il **pfcDisconnectSwitch**, pronto a **gestire le situazioni critiche**.
Dato che wes e pfcDisconnectSwitch comunicano tramite segnali, quest'ultimo, come il wes, dovrà inizializzare, tramite define, i due segnali, ERROR ed EMERGENCY.
Quindi il pfcDisconnectSwitch, di default, **non fa altro che registrare i due segnali ed aspetta l'arrivo di uno di essi, entrando in un ciclo infinito**. I segnali vengono **registrati tramite signal** e viene loro assegnato un signal Handler personalizzato, chiamato *"errorHandler"*. Oltre all'error Handler, vi sono altri due metodi, utilizzati in quest'ultimo, che sono *logFile* e *receivePid*. Il primo, presente in diversi altri processi, permette l'apertura del file di log, la scrittura di una stringa passatagli come argomento e la chiusura del file di log. Il secondo legge i file temporanei nei quali i tre pfc hanno trascritto i loro pid.
L'errorHandler può gestire entrambi i segnali, tramite due if. Più nello specifico, quando il pfcDisconnectSwitch riceve un segnale **ERROR** dal wes, per prima cosa inizializza una variabile intera, "block", e scrive sul file "switch.log" che ha ricevuto tale segnale. Successivamente, per ogni PFC, legge il pid nel file temporaneo creato dal PFC stesso. **Tramite un'istruzione kill verifica se il processo è ancora esistente, in tal caso invia un segnale di SIGCONT per farlo continuare**. Questo viene effettuato per tutti e tre i pfc, poiché se il processo non è bloccato la SIGCONT non influisce sull'esecuzione di quest'ultimo.
Qualora il processo dovesse essere deceduto, vengono effettuate delle operazioni sulla variabile block, che variano a seconda del pfc. Il PFC1 assegna il valore uno a block. Il PFC2 aumenta di due il valore di block, mentre PFC3 di quattro. In tal modo i valori possibili sono otto:

- block = 0 --→ Nessun PFC è morto
- block = 1 --→ PFC1 è morto
- block = 2 --→ PFC2 è morto
- block = 3 --→ PFC1 e PFC2 sono morti
- block = 4 --→ PFC3 è morto
- block = 5 --→ PFC1 e PFC3 sono morti
- block = 6 --→ PFC2 e PFC3 sono morti
- block = 7 --→ PFC1, PFC2 e PFC3 sono morti

Tramite uno **switch** si verifica il valore di block, e si loggano nel file switch.log i corrispondenti significati, presenti nell'elenco qui sopra.
Nel caso in cui **pfcDisconnectSwitch** dovesse ricevere un segnale di **EMERGENCY**, viene scritto sul file *"switch.log"* che è avvenuta una situazione di emergenza. Dopo aver fatto ciò, pfcDisconnectSwitch **invia al padre**, start, **un segnale SIGINT**, il quale attiverà la procedura di terminazione sopra descritta.
L'esecuzione dell'applicazione può, quindi, **concludersi in due modi.** O i PFC leggono tutte le righe di G18.txt ed il transducers termina di scrivere la velocità sui file di log, oppure si verifica un'emergenza. In entrambi i casi ad uccidere tutti i processi ci penserà start, padre di tutti gli altri processi.

ESECUZIONE

Proseguiamo con l'esecuzione di un caso standard. La prima cosa da fare è eseguire le due istruzioni chiave del *makefile*:

```
mizuiro@mizuiro-pc:~/Scaricati/mainFolder$ make
cp ./FILES/* ./
cc -o start start.c
cc -o PFC1 PFC1.c PFC.c -lm
cc -o PFC2 PFC2.c PFC.c -lm
cc -o PFC3 PFC3.c PFC.c -lm
cc -o transducers transducers.c
cc -o generatoreFallimenti generatoreFallimenti.c
cc -o pfcDisconnectSwitch pfcDisconnectSwitch.c
cc -o wes wes.c
mkdir -p src
mkdir -p bin
mkdir -p log
mkdir -p tmp
cp PFC.h PFC.c PFC1.c PFC2.c PFC3.c transducers.c start.c generatoreFallimenti.c p
fcDisconnectSwitch.c wes.c ./src
cp PFC1 PFC2 PFC3 transducers G18.txt generatoreFallimenti pfcDisconnectSwitch wes
 ./bin
rm -f PFC.h PFC.c PFC1.c PFC2.c PFC3.c transducers.c generatoreFallimenti.c pfcDis
connectSwitch.c wes.c start.c PFC1 PFC2 PFC3 transducers generatoreFallimenti pfcD
isconnectSwitch wes
```

Vengono così create le cartelle src, bin, log e tmp.

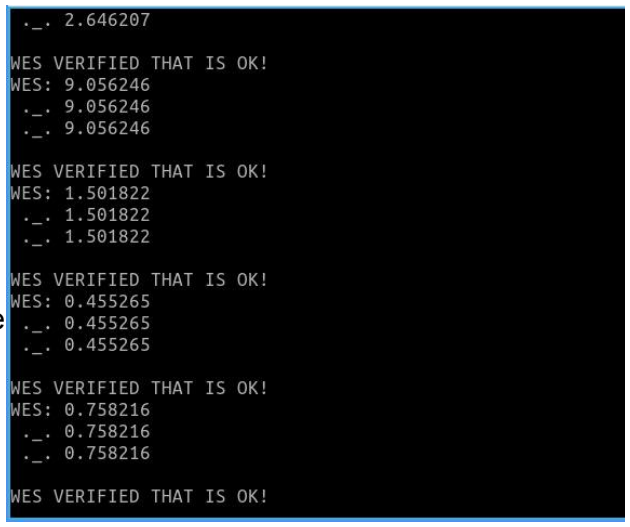
```
mizuiro@mizuiro-pc:~/Scaricati/mainFolder$ ls
bin  FILES  G18.txt  log  Makefile  src  start  tmp
```

Una volta effettuate queste operazioni, si passa all'esecuzione vera e propria, richiamando l'istruzione vista nella prima pagina della relazione:

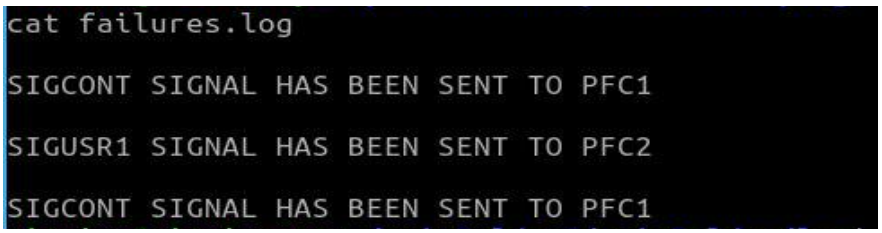
./start G18.txt

In questo modo comincia l'esecuzione. Nel nostro caso, per facilitare il testing e per poter aumentare le probabilità di possibili errori è stata usata una versione modificata del file G18.txt, rimuovendo tutte le righe per cui la velocità risultasse pari a zero. Inoltre, per

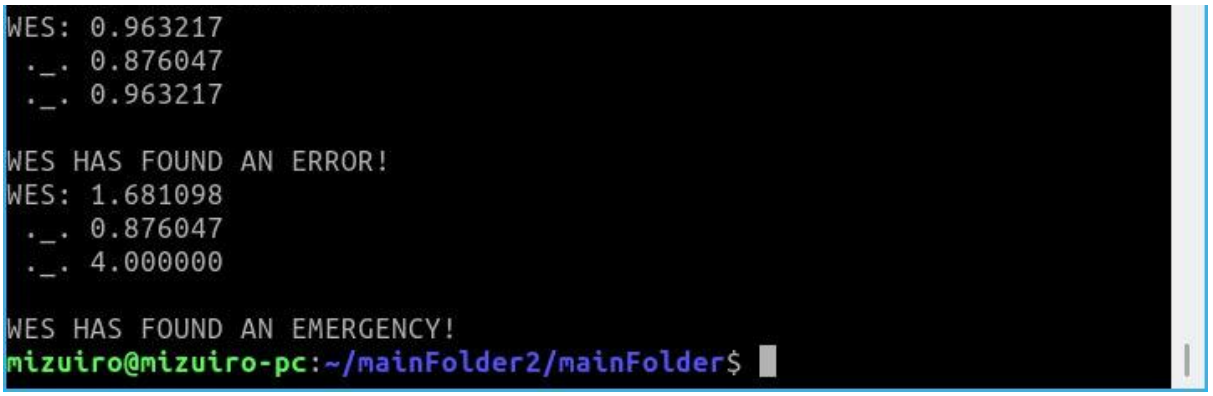
consentire una maggiore visibilità sono state inserite delle stampe, tramite l'istruzione **#ifdef DEBUG**. Nel codice inviato la macro DEBUG è inserita in un commento.
Per parte dell'esecuzione non risultano problemi di alcun tipo.
Infatti, si può verificare che i numeri stampati dal wes, rispettivamente prodotti dal PFC1, PFC2 e PFC3, sono identici. Di conseguenza il wes conferma che l'esecuzione sta procedendo come dovrebbe.
Però non è sempre così, poiché il generatore di fallimenti è sempre pronto ad inviare un segnale ai PFC.



Infatti, poco più avanti nell'esecuzione Si verifica un errore, che il wes prontamente segnala al pfcDisconnectSwitch e informa l'utente di quanto accaduto, tramite una stampa a video. Infatti, è possibile osservare come uno dei tre numeri raccolti dal wes sia differente dagli altri due. Questo è probabilmente stato causato da un segnale SIGUSR1 inviato al PFC2 dal generatore di fallimenti. In effetti andando a leggere il file failures.log è possibile notare che l'invio è effettivamente avvenuto.



Quando i numeri trovati dal wes risultano essere, tutti e tre, distinti, si verifica un'emergenza, come si può osservare in figura



Come previsto, in caso di EMERGENCY, l'applicazione termina.