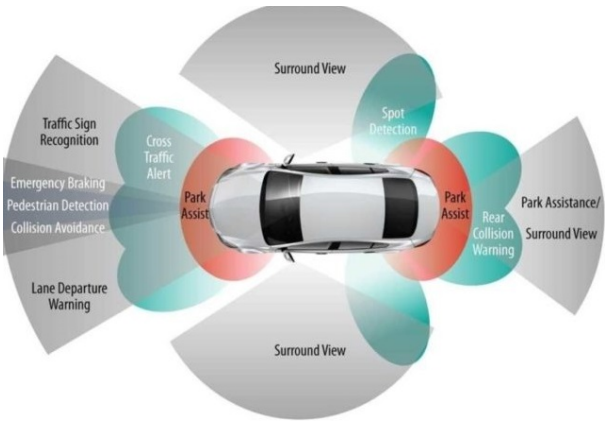
ADAS Made Trivial

Elaborato Progetto

Sistemi Operativi 2019/2020



Bernardo Manfriani 6344894

bernardo.manfriani@stud.unifi.it

Andrea Santi 6014033

andrea.santi2@stud.unifi.it

3 febbraio 2020

Sommario

Progetto realizzato da Bernardo Manfriani e Andrea Santi per il corso di Sistemi Operativi. Questo è un elaborato descrittivo riguardante simulatori di gestione di sistemi ADAS, estremamente stilizzati, attraverso l'utilizzo di architetture a 64 bit, processori intel e distribuzione **Linux Mint 19.2 Tina**.

Indice

1 Compilazione ed esecuzione **2**

2 Progettazione ed implementazione **3**

3 Esempi e dettagli esecutivi **7**

Capitolo 1

Compilazione ed esecuzione

Il progetto è contenuto in una cartella contenente 5 sotto-cartelle:

- *data* (contenenti i dati: .data e .binary)
- *exe* (executable: i file compilati)
- *lib* (library: le librerie)
- *log* (logger: i file di log)
- *src* (source code: i file sorgenti).

Per avviare il progetto dobbiamo preventivamente entrare nella cartella *exe* dove si troverà un makefile grazie al quale vengono unite le varie compilazioni dei file sorgenti in un unica chiamata:

```
$ cd ADAS-Man-San/exe
$ make allFiles
```

La compilazione, di tutti i sorgenti e delle librerie, viene così eseguita.

Per l'esecuzione vengono eseguiti i passi indicati dalla HumanMachineInterface, la quale accetta inizialmente un avvio o ***NORMALE*** o ***ARTIFICIALE*** (come da specifica) con i seguenti comandi:

```
$ ./hmi NORMALE
```

o

```
$ ./hmi ARTIFICIALE
```

e durante l'avvio si può solo digitare il comando di ***INIZIO*** per avviare la guida o, una volta avviata la guida, ***PARCHEGGIO*** per parcheggiare.

In caso di pericolo (sequenze maligne o segnale di **PERICOLO**), il sistema si resetta reiterando da zero la procedura di avvio.

Capitolo 2

Progettazione ed implementazione

Componenti e funzionalità implementate

Di seguito sono riportati i componenti, i sensori e gli attuatori implementaticome processi e dunque un file sorgente (contenuto in *src*) associato ad ogni componente:

- Human Machine Interface (hmi)
- Central ECU (ecu)
- Sensori
  - front windshield camera (fwc)
  - park assist (pa)
  - forward facing radar (ffr) (*facoltativo*)
  - blind spot (bs) (*facoltativo*)
  - surround view cameras (svc)(*facoltativo*)
- Attuatori
  - steer-by-wire (sbw)
  - throttle control (tc)
  - brake-by-wire (bbw)

Facoltativi

Di seguito riportiamo le funzionalità facoltative implementate che ci sembravano più caratteristiche:

- Il processo **parkAssit** è creato dalla ecu solamente una volta entrati in modalità parcheggio. Questo per evitare di creare troppi processi inutilizzati dall'avvio del sistema.
- Il processo **blindSpot** è in attesa di un seganle direttamente da **steerByWire** il quale per comodità lo abbiamo pensato come processo padre di **blindSpot**. Quest'ultimo è creato da **steerByWire** così da semplificare la comunicazione tramite *signal* tra i due processi.
- I dati ricevuti da **forwardFacingRadar** sono continuamente analizzati e raccolti: se i valori raccolti contengono almeno uno tra i valori 0xA00F,0xB072,0x2FA8,0x8359, 0xCE23, allora la Central ECU invia un segnale di pericolo a brake-by-wire.
- L'attivazione della procedura di parcheggio termina tutti i processi superflui ovvero i componenti che non vengono utlizzati durante la sessione di parcheggio. Questo comporta una terminazione istantanea di questi componenti in esecuzione tramite l'utilizzo di una *signal* SIGTERM.

- Durante una sterzata, la Central ECU legge i valori di Blind spot rear radars. Se per la durata della sterzata la Central ECU non riceve nessuno dei valori i) 0x414E, ii) 0x4452, iii) 0x4C41, iv) 0x424F, v) 0x5241, vi) 0x544F, vii) 0x8359, viii) 0xA01F, ix) 0x5249, x) 0x5100, allora la sterzata termina correttamente, altrimenti si invia un segnale di pericolo a brake-by-wire. Il segnale viene inoltre inviato anche ad *hmi* poiché vogliamo che venga reiterata la procedura di partenza e dunque reinserito il comando *INIZIO*.

Architettura software

Ad alto livello l'applicativo è strutturato tramite interazioni Client-Server. La ECU è suddivisa in ECU-Client e ECU-Server:

- ECU-Sever comunica con i sensori i quali fanno da Client per essa, ovvero prendono informazioni da file (frontCamera.data e generatori di random) e li spediscono tramite *socket* a Ecu-Server, il quale tramite *pipe* li invia a Ecu-Client.
- ECU-client è composto da metodi **manager**, uno per ogni sensore, i quali catturano i dati che ECU-Server ha smistato. Una volta gestite le informazioni le manipola e le invia agli attuatori che fanno da server per ECU-Client. **ParkAssist** e **surroundViewCameras** si differenziano dagli altri sensori poiché il loro server processa direttamente i dati senza spedirli al manager opportuno.

HMI

Il programma al suo avvio esegue ***HMI***.

L'HMI è divisa sostanzialmente in tre processi:

- un processo che fa da interlocutore fra l'utente e l'applicativo e può essere avviato in modalità ***NORMALE*** o ***ARTIFICIALE***,
- un processo che avvia ***ECU*** la quale resta in attesa di una *signal* da parte dell'***HMI*** che viene inviata una volta scritto il comando ***INIZIO***,
- un altro processo che fa da interfaccia di output e stampa su un secondo terminale il contenuto di ECU.log. Tale funzionalità è ottenuta sfruttando il comando di linux *tail -F ECU.log*.

ECU

Una volta arriavato il segnale, la ***ECU*** viene sbloccata e si occupa di creare sensori ed attuatori, inizializzando le varie connessioni fra essi tramite i metodi ***init()*** e ***start()***. Il metodo ***start()*** si occupa di generare un processo per ogni sensore e per ogni attuatore attraverso i metodi ***creaSensori()*** e ***creaAttuatori()*** i quali creano solo i processi generabili all'avvio del sistema grazie ad un Template Method creaComponente() (fwc, ffr, tc, bbw, sbw).

Sensori

I sensori sono programmi gestiti come Client della ECU-Server e scambiano messaggi con quest'ultima tramite socket. Tutti i sensori vengono creati dalla ECU non appena questa riceve il comando di "INIZIO", ad eccezione di blindSpot (bs) che viene creato da steerByWire mentre parkAssist (pa) e surroundViewCamera (svc) vengono creati sempre dalla ECU ma solo una volta avviata la procedura di parcheggio.

Nella logica dell'applicativo il primo sensore ad agire è

FrontWindShieldCamera il quale legge dati da frontCamera.data ed ogni 10 secondi li invia alla ECU tramite *socket* la quale li gestisce (tramite pipe li invia ad Ecu-Client) e li gira agli attuatori. I dati inviati sono salvati in camera.log.

ForwardFacingRadar quando viene creato dalla ECU si connette alla socket, e ogni 2 secondi prova a leggere 24 byte da /dev/random o /randomARTIFICIALE.binary. Se riesce a leggere correttamente 24 byte, li trasmette alla ECU e li scrive nel file di log (radar.log), altrimenti non invia niente.

ParkAssit viene creato dalla ECU una volta attivata la procedura di parcheggio e, per la durata di 30 secondi, legge iterativamente una volta al secondo 4 byte da /dev/urandom o /urandomARTIFICIALE.binary e li invia alla Central ECU. I dati inviati sono salvati in assist.log.

BlindSpot una volta creato da SteerByWire è in costante attesa di una *signal* da esso, inviata ogni volta che viene eseguita una sterzata, e si interrompe quando finisce la procedura di sterzata. Ricevuta la signal legge 8 byte ogni mezzo secondo da /dev/urandom o /urandomARTIFICIALE.binary e li invia su socket alla ECU.

SurroundViewCameras come per ParkAssist, viene creato dalla ECU una volta attivata la procedura di parcheggio. Legge 16 byte ogni secondo da /dev/urandom o /urandomARTIFICIALE.binary e li invia su socket alla ECU.

Attuatori

I tre attuatori sono programmi che si comportano come server per la ECU-client, e utilizzano le socket per scambiarsi messaggi con la ECU. Sono tutti creati all'avvio del sistema.

ThrottleControl una volta avviato si divide in due processi. Il processo padre crea la socket di servizio con la ***ECU*** rimanendo in ascolto su di essa, prende i dati dalla socket e li invia su pipe al processo figlio. Il figlio si occupa di aprire e, ogni secondo, scrivere sul file throttle.log *NO ACTION* se non c'è variazione di velocità o *AUMENTO 5* se riceve il comando di accelerazione.

BrakeByWire anch'esso si divide in due processi padre e figlio. Il padre scrive i dati presi dalla socket con la ***ECU*** e li immette su pipe. Il figlio legge da pipe e scrive, ogni secondo, su brake.log *NO ACTION* se non c'è variazione di velocità, *DECREMENTO 5* se c'è un comando di decelerazione.

SteerByWire si divide in tre processi:

- un processo si occupa di creare il sensore BlindSpot,
- un processo che legge da socket e invia su pipe i dati,
- un altro legge da pipe e processa i dati:
  - se legge *DESTRA/SINISTRA* esegue la sterzata che dura 4 secondi e scrive, ogni secondo, su steer.log *STO GIRANDO A DESTRA/SINISTRA*. In caso contrario non compie nessuna azione e scrive sul file di log, ogni secondo, *NO ACTION*.
  - Inoltre quando avviene la sterzata vengo inviati i segnali SIGCONT e SIGSTOP a BlindSpot rispettivamente per attivarlo e disattivarlo durante la sterzata.

Scelte implementative

- Creiamo il sensore **BlindSpot** direttamente in **SteerByWire** così da poter aver il **pid** di BlindSpot ed inviare le *signal* adeguate ad attivare ed interrompere il sensore all'occorrenza tutte le volte che si presenta una curva.
- Per la creazione di ECU-server dei sensori abbiamo deciso di implementare un *Template Method*, **creaServer()**, chiamato nel metodo **creaServers()**, il quale fattorizza l'implementazione comune in un unico metodo richiamato nella creazione di ogni server.

- Sono state create due librerie: *fileManager.h e socketManager.h.*
  - In *fileManager.h* è stato fatto un override di openFile(), così da poter fare il controllo sull'apertura del file e fare il controllo sul filepointer che non fosse nullo, evitando di doverlo inserire in ogni apertura di file.
  - In *socketManager.h* abbiamo fattorizzato il *connectClient(char *socketName)* dato che il client per connettersi alla socket usava lo stesso metodo per tutti i componenti. Per comodità, inoltre sono stati inseriti dentro la libreria anche i metodi di scrittura e lettura.
- Una volta terminata la lettura da frontCamera.data avviamo la procedura di *PARCHEGGIO* così da far terminare definitivamente il programma. Ci sembrava una cosa intelligente poiché, quando il file frontCamera.data termina, logicamente non ci saranno più indicatori di velocità o di direzione e perciò la macchina è predisposta alla procedura di parcheggio.

Capitolo 3

Esempi e dettagli esecutivi

Esempi Rappresentativi

Le esecuzioni si dividono fondamentalmente in due rami: esecuzioni *NORMALI* o *ARTIFICIALI*, e ogni test è stato eseguito sia in modalità *NORMALE* che *ARTIFICIALE*. Nel caso di esecuzioni artificiali, abbiamo testato i casi critici inserendo combinazioni *malevole* che compromettono l'esecuzione del sistema. Le interruzioni avvengono correttamente ed il programma viene bloccato e gestite le interruzioni a seconda del tipo di combinazione trovata. Verrano volta volta specificati i dati di **input** (frontCamera.data) e quelli di **output** (HMI output) stampati a schermo.

Esempio 0 (banale)

input : *20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20*

HMI output : *INCREMENTO 20*

L'input banale fa accelerare inizialmente, dopodiché la velocità rimane costante nel tempo difatti non viene più effettuato nessun incremento.

Esempio 1

input : *15 50 10 DESTRA SINISTRA PERICOLO 20 20 20*

HMI output : *INCREMENTO 15 INCREMENTO 35 FRENO 40 DESTRA SINISTRA INCREMENTO 20*

Una volta letto il segnale di *PERICOLO* il veicolo viene arrestato correttamente e non viene stampato più niente da terminale. Per riavviare l'esecuzione è necessario scrivere nuovamente *INIZIO* e l'esecuzione ripartirà dalla riga successiva a l'ultima letta prima della terminazione.

Esempio 2 :

input : *20 20 30 30 30 80 80 40 40 DESTRA PERICOLO 20 20 40 40 70*

output : *INCREMENTO 20 INCREMENTO 10 INCREMENTO 50 FRENO 40 DESTRA INCREMENTO 20 INCREMENTO 20 INCREMENTO 30*

Una volta letto il segnale di *PERICOLO*, come prima, il veicolo viene arrestato correttamente e non viene stampato più niente da terminale. Per riavviare l'esecuzione è necessario scrivere nuovamente *INIZIO* e l'esecuzione ripartirà dalla riga successiva a l'ultima letta prima della terminazione.

Figura 3.1: Esempio di corretto funzionamento: il programma si stoppa alla ricezione del segnale di perciolo e riparte chiedendo l'inserimento del comando *INIZIO*.

Casi limite e situazioni critiche

Esempio 3

input : *20 20 90 105 20 SINISTRA DESTRA PERICOLO*

output : *INCREMENTO 20 INCREMENTO 70 INCREMENTO 15 FRENO 85 SINISTRA DESTRA DESTRA INCREMENTO 20 INCREMENTO 70 INCREMENTO 15 FREN0 85 SINISTRA DESTRA DESTRA*

Questo esempio è un caso particolare. Infatti la variazione di velocità e maggiore di 50 con un ulteriore aumento seguente. Il programma funziona comunque correttamente, l'output è corretto e i valori dei file di log corrispondono con quelli attesi.

Esempio 4 Per completezza di seguito è riportato uno *screen* della situazione in cui viene trovata una combinazione esadecimale maligna con la conseguente segnalazione e terminazione della simulazione. L'immagine rappresenta la terminazione a causa della rilevazione di una sequenza maligna da parte di *ForwardFacingRadar* con **input**: 15 10 DESTRA.

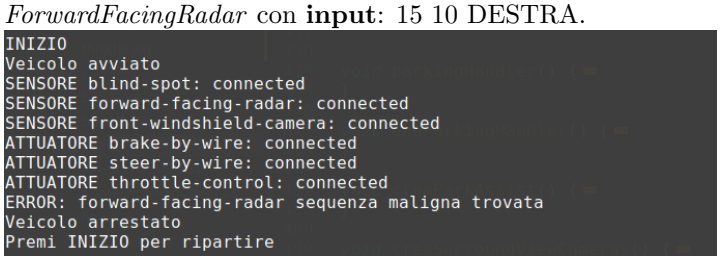


Figura 3.2: FFR ERROR

Esempio 5 L'ultimo esempio rappresenta l'esecuzione in fase di parcheggio. La macchina rallenta fino a raggiungere velocità nulla, poi avvia un conto alla rovescia per la fasa di parcheggio.

input:15-DESTRA-30-30-30-30

```
INIZIO
Veicolo avviato
SENSORE blind-spot: connected
ATTUATORE brake-by-wire: connected
SENSORE front-windshield-camera: connected
ATTUATORE throttle-control: connected
ATTUATORE steer-by-wire: connected
SENSORE forward-facing-radar: connected
PARCHEGGIO
Sto fermando il veicolo...
speed: 30
speed: 25
speed: 20
speed: 15
speed: 10
speed: 5

parcheggio in corso...
SENSORE surround-view-cameras: connected
SENSORE park-assist: connected
remaining time: 30sec
remaining time: 25sec
remaining time: 20sec
remaining time: 15sec
remaining time: 10sec
remaining time: 5sec
Terminato
```

output:

```
HMI OUTPUT:
INCREMENTO 15
DESTRA
INCREMENTO 15
PARCHEGGIO 30
```