



École Centrale de Lille

CR TEA Informatique Industriel

Compte rendu du TEA de l'électif Système de transport autonome
Élèves : Yann Bruno ANDRADE MELLO
Gabriel GOSMANN

Villeneuve d'Ascq
Octobre 2023

“Trees are the most inspiring structures.”

—Kurt Gödel, Oppenheimer.

Table des matières

1	Introduction	3
2	Codage	4
2.1	railway.h	4
2.2	railway.c	5
2.3	server.h	9
2.4	server.c	9
2.5	client.c	16
2.6	main.c	17
2.7	train.c	19
2.8	makefile	24
3	DEMO	25
4	Conclusion	27

Chapitre 1

Introduction

Ce rapport décrit la conception et l'implémentation d'un simulateur distribué pour un système ferroviaire qui opère sur une ligne composée de **100 cantons**, numérotés de **1 à 100**. Ce système gère la circulation des trains, chacun caractérisé par un identifiant unique composé de 3 lettres et 3 chiffres, tels que TGV134 et TER203. L'objectif principal de ce système est de garantir la sécurité des opérations ferroviaires en s'assurant qu'un seul train peut occuper un canton à un moment donné.

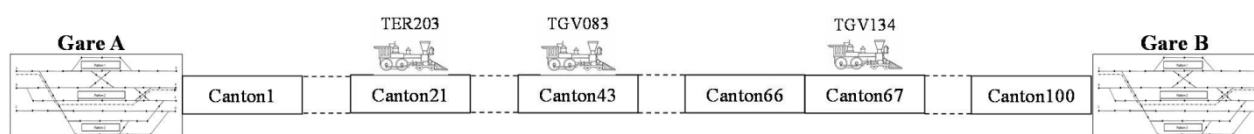


FIGURE 1.1 – Voie ferroviaire avec ses circulations

Le système se compose de deux éléments principaux : le **Radio Bloc Center(RBC)**, qui agit en tant que serveur central, et les mouvements de trains, qui sont représentés en tant que clients. Le RBC enregistre les mouvements de train lorsqu'ils entrent sur la voie, met à jour leurs positions, délivre les autorisations de mouvement et maintient les informations à jour. Chaque mouvement de train envoie régulièrement sa position actuelle à la RBC et demande de nouvelles autorisations de mouvement.

Pour réaliser ce système, nous avons opté pour une approche distribuée basée sur une architecture client/serveur utilisant la communication **TCP/IP**. De plus, une structure de messagerie industrielle a été développée pour permettre la communication entre les clients (trains) et le serveur (RBC).

Chapitre 2

Codage

2.1 railway.h

Essentiellement, cet en-tête fournit les structures de données et les fonctions nécessaires à la gestion des trains dans le système ferroviaire simulé. Les énumérations et les structures de données sont conçues pour représenter les trains et leurs propriétés, et les fonctions permettent de créer, de gérer et d'afficher des informations sur les trains. Ces structures et fonctions sont fondamentales pour le fonctionnement du simulateur de système ferroviaire.

Définitions des types :

```
#define NUM_STD_TYPES 3           // Number of train types

// standard train types
typedef enum std_types {
    TGV, TER, RER
} std_types_t ;
```

Ces lignes définissent un certain nombre de types de trains standards et une énumération qui représente ces types. Les types de train standard sont le TGV, le TER et le RER.

Structures de données :

```
typedef struct train_t {
    char type_id [5];           // string that holds type_id, ex. "TGV"
    char id [5];                // string that holds train id number, ex. "1"

    int location;               // current canton location [ 0 ~ 100 ]
    int eoa;                    // end_of_authority , max location the train can go
    int socket_fd;              // socket_fd

    pthread_t thread;           // thread_identifier
} train_t ;

// struct that defines the train sequence as a linked list
typedef struct train_set {
    train_t train;              // stores a train element in memory
    struct train_set * next;
```

```

    struct train_set * next;           // pointer to the next element
    struct train_set * prev;          // pointer to the prev element
} train_set_t;

// struct of messaging
typedef struct message_t {
    char message[256] ;
    train_t train ;
} message_t ;

```

- *train_t* est une structure qui représente un train. Elle stocke des informations telles que le type de train, l'identifiant du train, la position actuelle, le point d'autorité final (EOA), le descripteur de socket et l'identifiant de thread.
- *train_set_t* est une structure de liste chaînée qui stocke les trains. Chaque élément de la liste contient une structure *train_t*, un pointeur sur l'élément suivant et un pointeur sur l'élément précédent.

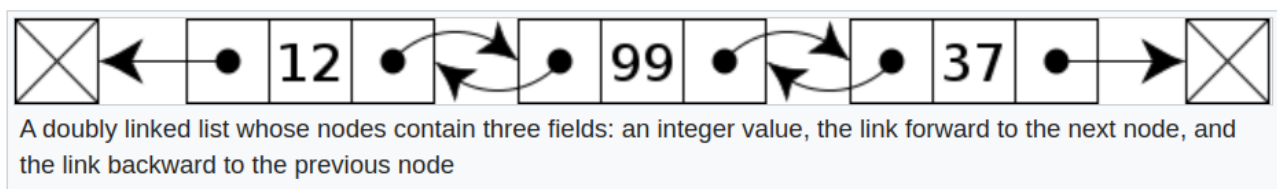


FIGURE 2.1 – Structure de liste chaînée de *train_set_t*

- *message_t* est une structure qui représente les messages relatifs aux trains.

2.2 railway.c

```

#include "../include/railway.h"

#include <stdio.h>
#include <stdlib.h>

#include "string.h"

// functions on [train_t]

// this function populates a raw train element with a type and an id
// can be thought as a constructor of the type train_t
int create_train(train_t * train, int type, char * train_id) {
    int status = 0;

    switch (type) {
    case TGV:
        strcpy(train->type_id, "TGV");
        break;
    case TER:
        strcpy(train->type_id, "TER");
        break;

```

```

    case RER:
        strcpy( train->type_id, "RER" );
        break;
    default:
        strcpy( train->type_id, "???" );
        status = -1;
}

train->location = 0;           // starts at default location
train->eoa = 0;                // with no authorization at all

strcpy( train->id, train_id ); // updates train_id

return status;
}

// prints the parameters of a train in the screen
void print_train_params( train_t * train ) {

    printf( "      _|_[%s%s]_|_[LOC,EOA]:_[%3d,_%3d_]_|_\\n",
        train->type_id, train->id, train->location, train->eoa );
}

// functions on [train_set_t]

// this functions creates a new train_set_t element from scratch
train_set_t * start_train_set( int type_id, char * train_id ) {

    // initializes first element with memory from the head
    // maybe use calloc
    train_set_t * first_train = (train_set_t *)calloc( sizeof(train_set_t), 1 )

    create_train( &(first_train->train), type_id, train_id );

    first_train->next = NULL ;
    first_train->prev = NULL ;

    return first_train ;
}

train_set_t * create_empty_train_set( train_set_t * last_train ) {

    train_set_t * train = (train_set_t *)calloc( sizeof(train_set_t), 1 ) ;
    train->next = NULL;
    train->prev = NULL;

    if( last_train != NULL ){
        last_train->next = train;
    }
}

```

```

        train->prev = last_train;
    }

    return train ;
}

// adds an element to the set of trains
train_set_t * add_train(train_set_t * set , int type_id , char * train_id) {

    if( set->next != NULL ){
        add_train(set->next , type_id , train_id );
    }
    else{
        // it is in the last element
        set->next = start_train_set( type_id , train_id ) ;
        set->next->prev = set;
    }

    return set->next;    // returns the element just added
}

// removes the first element of the set (FIFO)
train_set_t * remove_train(train_set_t ** set){

    train_set_t * old = set[0] ;
    train_set_t * ret ;

    if( set[0]->prev != NULL ){
        set[0]->prev->next = set[0]->next ;
    }

    if( set[0]->next != NULL ){
        set[0]->next->prev = set[0]->prev;
    }

    ret = set[0]->next;
    //set[0] = set[0]->next;
    //free( old ) ;

    return ret;    // returns the new first element
}

// Functions type_train
void print_all_trains( train_set_t * set )
{
    int num = 0;
    printf("      \t\tprinting all trains : \n");

    while( set->prev != NULL ){

```



```

        set = set->prev ;    // goes to first element
    }

    while( set->next != NULL ){

        printf( " [] \t [%d] | ", num );
        print_train_params( &(set->train) );

        set = set->next ;
        num++;
    }

    printf( "\n" );
}

// example and testing of the library
void testing( ){

    train_set_t * railway = start_train_set( TGV, "123" ) ;

    print_all_trains( railway );

    add_train( railway, TER, "666" );
    add_train( railway, RER, "122" );
    add_train( railway, TGV, "985" );

    print_all_trains( railway );

    remove_train( &railway );

    print_all_trains( railway );

    remove_train( &railway );

    print_all_trains( railway );

}

```

- *create_train()* : Cette fonction remplit une structure `train_t` avec des informations sur le type et l'identifiant du train. Elle gère également l'initialisation d'autres champs de la structure, tels que la localisation et le point d'autorité final (EOA).
- *print_train_params()* : Cette fonction permet d'imprimer à l'écran les informations relatives à un train, notamment son type, son identifiant, sa localisation et son EOA.
- *create_empty_train_set()* : Crée une nouvelle unité de train vide (liste chaînée) avec le premier élément. aux trains.
- *add_train()* : Ajoute un nouveau train à la liste chaînée, créant ainsi un nouvel élément.

- *remove_train()* : Retire le premier élément de la liste chaînée. La logique veut que le premier arrivé soit le premier parti(FIFO).
- *print_all_trains()* : Cette fonction imprime toutes les informations relatives aux trains dans la liste chaînée en parcourant la liste et en appelant la fonction *print_train_params()* pour chaque train.
- *testing()* : Cette fonction est un exemple de test de la bibliothèque. Elle crée un ensemble de trains, ajoute, supprime et imprime des informations sur les trains pour démontrer l'utilisation des fonctions.

2.3 server.h

Le fichier "server.h" est un fichier d'en-tête qui contient la déclaration des fonctions et des structures de données qui seront utilisées dans le code du serveur. Il fournit une interface pour l'implémentation des fonctions du serveur, permettant à d'autres codes, tels que "server.c", d'utiliser ces fonctions sans avoir besoin de connaître les détails de l'implémentation.

2.4 server.c

Le fichier "server.c" contient l'implémentation des fonctions déclarées dans le fichier d'en-tête "server.h.". Il représente la partie exécutable du serveur, où les opérations réelles du serveur sont définies.

- *treat_client()* : Cette fonction est exécutée dans un nouveau thread pour gérer la connexion du client. Elle obtient des informations sur le client, telles que l'adresse IP et le port, et met à jour les données du train avec les informations reçues du client.
- *init_tcp_server()* : Cette fonction démarre un serveur TCP. Elle crée un socket, le lie à un port spécifique et commence à écouter les connexions des clients. Elle renvoie le descripteur de fichier de la socket du serveur.
- *connect_to_server()* : Cette fonction est utilisée pour connecter le serveur à un client spécifié par l'adresse IP et le port. Elle crée une socket, définit l'adresse du serveur et établit la connexion. Elle renvoie le descripteur de fichier de la socket de connexion.
- *wait_for_connection()* : Cette fonction attend qu'un client se connecte au serveur. Lorsqu'une connexion est acceptée, elle renvoie le descripteur de fichier de la connexion.
- *connect_to_train* : Cette fonction est utilisée pour connecter un client de train au serveur. Elle crée un nouveau thread qui gère la connexion du client.

En général, "server.h" fournit une interface qui décrit les opérations disponibles pour le serveur, tandis que "server.c" met en œuvre les opérations réelles du serveur, y compris la logique de traitement des connexions des clients et de communication avec les trains. L'ensemble de ces codes permet de simuler un système ferroviaire avec des capacités de communication entre les trains et un serveur central.

server.h

```
#ifndef SERVER_H
#define SERVER_H

// from man socket
#include <sys/types.h>
#include <sys/socket.h>

#include "../include/railway.h"

int init_tcp_server( int port ) ;

int wait_for_connection( int socket_fd ) ;

int connect_to_train( train_set_t * train ) ;

int connect_to_server(char * ip , int port) ;

#endif
```

server.c

```
// TODO create a nice header

#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <arpa/inet.h> // from man htons
#include <unistd.h> // from man close

#include <pthread.h> // from man pthreads

#include <errno.h>

#include <assert.h>

#include "../include/server.h"

int calculate_eoa( train_set_t * train ){
    /*
    train_set_t * target = train;

    while( target->prev != NULL ){

        if( train->train.eoa < target->prev->train.location ){
            printf("define eoa by location");
            train->train.eoa = target->prev->train.location ;
        }
    }
    */
}
```

```

        return 0;
    }

    target = target->prev ;
}

train->train.eoa = 100 ;    // TODO make a define for this
*/

if( train->prev != NULL ){
    train->train.eoa = (train->prev)->train.location ;
}
else{
    train->train.eoa = 101 ;
}

return 0;
}

void * treat_client( train_set_t * train ){

    struct sockaddr_in peer_addr;          // this will hold info about the cli
    socklen_t addr_size = sizeof(struct sockaddr_in);
    int res = getpeername( train->train.socket_fd , (struct sockaddr *)&peer_

    assert( res == 0 );

    printf("      \t\tnew connection from [ip:%s] and [port:%d] at [PID:%d] \n"
        inet_ntoa( peer_addr.sin_addr ), htons(peer_addr.sin_port), getpid()

    // obtain data of the arriving train:
    train_t train_data ;
    int recv_r = recv( train->train.socket_fd , &train_data , sizeof(train_t),

    // update data
    strcpy( train->train.type_id, train_data.type_id );
    strcpy( train->train.id, train_data.id );

    // print current train params
    //print_train_params( &(train->train) );
    int running = 1;

    while( running == 1 ){
        //print_train_params( &(train->train) );
        print_all_trains( train );

        char buffer[256] = {0};
        message_t message = {0};
        int number = recv( train->train.socket_fd , &message , sizeof(message)

```

```

    if( number == -1 | number == 0 )
        //if( number == -1 )
            break;

    if( message.message[0] == 'u' ){        // update position
        train->train.location = message.train.location;

        calculate_eoa( train );

        //print_all_trains( train );
        //train->train.eoa = 66 ;

        message_t message = { {"EOA"}, train->train };
        int send_r = sendto( train->train.socket_fd, &message,
                             sizeof(message), 0, NULL, 0);

        assert( send_r != -1);
    }

    else if( message.message[0] == 'e' ){
        running = 0;
    }

    //printf("[%s] [%d]\n", buffer, number);
}

close( train->train.socket_fd );

remove_train( &train );

printf("    \tend of connection from [ip:%s] and [port:%6d] at [PID:%d]\n",
       inet_ntoa( peer_addr.sin_addr ), htons(peer_addr.sin_port), getpid() );

print_all_trains( train->next );

}

// inits the server to specified port
int init_tcp_server( int port ){

    // socket file descriptor
    // int socket(int domain, int type, int protocol);
    int socket_fd = socket( AF_INET, SOCK_STREAM, 0 ) ;

    if( socket_fd != -1 ){
        //printf("successfully created the socket! [%d] \n", socket_fd);
    }
}

```

```

else {
    printf("error creating the socket!\n");
    return socket_fd;
}

// bind (defining the addresses)
struct sockaddr_in my_addr;
my_addr.sin_family = AF_INET ;
my_addr.sin_port = htons(port);           // normally htons(1994)

//my_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
my_addr.sin_addr.s_addr = INADDR_ANY;    // listens from any address

// int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
int bind_r = bind( socket_fd, (struct sockaddr *)&my_addr, sizeof(my_addr) );

if(bind_r != -1){
    //printf("successfully binding! [%d] \n", bind_r);
}
else{
    printf("error binding!\n");
    return bind_r;
}

// listen
//int listen(int sockfd, int backlog);
int listen_r = listen( socket_fd, 10 );    // backlog int defines max queue size

if( listen_r == 0 ){
    //printf("successfully listen! [%d] \n", listen_r);
}
else{
    printf("error listening!\n");
    return listen_r;
}

// now it will always be listening
printf("    \tsocket created successfully\n");

return socket_fd;
}

int connect_to_server(char * ip, int port){

    // socket file descriptor
    // int socket(int domain, int type, int protocol);
    int socket_fd = socket( AF_INET, SOCK_STREAM, 0 );

    assert( socket_fd > 0 );
    printf("    \tsocket created successfully\n");
}

```

```

// connect (defining the address)
struct sockaddr_in server_addr;
server_addr.sin_family = AF_INET ;
server_addr.sin_port = htons(port);
server_addr.sin_addr.s_addr = inet_addr(ip);

int connect_r = connect( socket_fd, (struct sockaddr *)&server_addr,
    sizeof(struct sockaddr) );

assert( connect_r == 0 );

printf("    \tnew connection made at [ip:%s] and [port:%d] at [PID:%d]\n",
    inet_ntoa( server_addr.sin_addr ), htons(server_addr.sin_port), gettid());

return socket_fd;
}

int wait_for_connection( int socket_fd ){

    struct sockaddr_in peer_addr;
    socklen_t addr_size_peer = sizeof( struct sockaddr_in );

    int accept_fd = accept( socket_fd, (struct sockaddr *)&peer_addr, &addr_size_peer);

    //printf("successfully accepted! [%d] [%s] [%d]\n",
    //    accept_fd, inet_ntoa( peer_addr.sin_addr ), htons(peer_addr.sin_port));

    return accept_fd;
}

int connect_to_train( train_set_t * train ){

    // connect accept_fd to train_thread

    //int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *)
    int create_r = pthread_create( &(train->train.thread), NULL,
        (void *)(& treat_client ), (void *)(& train) );

    assert(create_r == 0);

    return create_r;
}

int old_main(){

    printf("Hello World! from server\n");

    int socket_fd = init_tcp_server(1994);

```

```

int running = 1;
int threads_count = 0;
pthread_t my_threads[100] ;

while( running ){

    // starts accepting connections
    // should create 1 thread for echo connection

    struct sockaddr_in peer_addr;
    socklen_t addr_size_peer = sizeof( struct sockaddr_in );
    int accept_fd = accept( socket_fd, (struct sockaddr *)&peer_addr, &addr_size_peer);

    if( accept_fd >= 0 ){
        printf("successfully accepted! [%d] [%s] [%d]\n",
            accept_fd, inet_ntoa( peer_addr.sin_addr ), htons(peer_addr.sin_port));

        // treat accept_fd to new thread
        //int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
        int create_r = pthread_create( &my_threads[ threads_count ], NULL,
            (void *)(* treat_client ), (void *)(&accept_fd) );

        //pthread_create( &thr[i], NULL, (void *)(* num_of_primes), (void *)(&accept_fd) );

        // pthread_join( thr[i], (void **) &status[i]);

        if(create_r == 0){
            printf("success creating thread\n");
        }
        else{
            printf("error creating thread\n");
        }

        threads_count++;
    }
    else{
        printf("error accepting!\n");
    }

    //int number = recv( accept_fd, buffer, 255, 0 ) ;

```



```

    }

    printf( "This is the end!\n" );

    return 0;
}

```

2.5 client.c

Le fichier "client.c" est un code qui représente un client dans un système de simulation ferroviaire. Ce client est chargé d'envoyer des messages au serveur du système ferroviaire, permettant ainsi la communication et l'interaction entre les trains et le serveur central.

Une boucle principale permet au client de saisir les messages à envoyer au serveur. Le client lit les messages à partir de l'entrée standard (stdin) à l'aide de la fonction "fgets", et les messages sont stockés dans un buffer.

Le code vérifie s'il y a quelque chose à envoyer (si la taille du message est supérieure à zéro) avant d'envoyer le message au serveur. Cela permet d'éviter l'envoi de messages vides ou invalides.

Envoi du message au serveur à l'aide de la fonction sendto. Le client tente d'envoyer le message au serveur et, en cas d'erreur, il imprime un message d'erreur.

```

// TODO create a nice header

#include <stdio.h>
#include <stdlib.h>

// from man socket
#include <sys/types.h>
#include <sys/socket.h>

// from man htons
#include <arpa/inet.h>

#include <unistd.h>      // from man close

#include <string.h>

int main(){

    //signal(SIGINT, handle_sigint);    // TODO this later

    char buffer[256] = {0};

    while(1){

        fgets( buffer , sizeof(buffer), stdin );
        //sprintf( buffer , "Hello World! \n" );

```

```

printf( "[%s] ", buffer );

// send only if there is something to send
if( strlen( buffer ) > 0 ){

    int send_to_r = sendto( socket_fd , buffer , strlen( buffer ) -1 , 0 ,

        if( send_to_r == -1 ){
            printf( "error!_\n" );
        }

    }

}

close( socket_fd );

return 0;
}

```

En résumé, "client.c" représente la partie client d'un système ferroviaire simulé. Il permet au client d'envoyer des messages au serveur, ce qui est essentiel pour la communication entre les trains et le serveur central.

2.6 main.c

Le fichier "main.c" représente la partie principale d'un système de simulation ferroviaire, simulant le centre de contrôle des trains (Radio Block Centre). Il est responsable du démarrage du serveur qui gère les communications avec les trains sur la voie ferrée

Nous commençons par créer une structure de données appelée "railway" qui stocke des informations sur les trains sur la voie ferrée. Elle est initialement créée sous la forme d'une liste vide.

Nous avons lu le numéro de port à partir des arguments de la ligne de commande pour la configuration du serveur. Le code ne traite pas des autres arguments de la ligne de commande, mais il pourrait être étendu pour inclure davantage d'options.

Nous initialisons le serveur à l'aide de la fonction `init_tcp_server` pour créer un socket serveur qui attend les connexions des clients sur le port spécifié.

Une boucle principale qui attend indéfiniment les connexions des clients. Lorsqu'un client se connecte, il vérifie que la connexion a réussi.

Nous assignons le socket du client au train actuel (`current_train`) et créons un thread pour gérer ce train, ce qui permet la communication entre le serveur et le client/train

À chaque connexion d'un client, une nouvelle structure de données de train est créée et liée à la structure existante, ce qui permet de contrôler plusieurs trains.

```
// TODO make good header
// TODO add date and time to main screen like a linux boot

// desc. this file simulates a train control station (Radio Block Center)

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#include "../include/railway.h"
#include "../include/server.h"

int main( int argc , char ** argv ){

    train_set_t * railway = create_empty_train_set( NULL ) ;

    int running = 1;

    // implement command line parsing to obtain :
    // TODO is there any other thing to obtain from command line interface?
    int port = 1994;

    if(argc == 2){
        port = atoi( argv[1] );
    }

    // start server integration
    int socket_fd = init_tcp_server( port );

    assert( socket_fd > 0 );    // has successfully created the socket

    //train_set_t * current_train = railway;

    // starts infinite loop for accepting connections
    while( running ){

        // waits until a client connects
        int accept_fd = wait_for_connection( socket_fd );

        assert( accept_fd > 0 ) ;    // client has logged in successfully

        //railway->train.socket_fd = accept_fd;
        railway->train.socket_fd = accept_fd;

        // make a thread to deal with this train
        int connect_r = connect_to_train( railway );
```

```

        assert( connect_r == 0);    // successfull thread connection

        railway = create_empty_train_set( railway );

    }

    printf( " This is the end! \n" );

    return 0;
}

```

En résumé, "main.c" est le composant principal qui démarre et gère le serveur du système de simulation ferroviaire. Il s'occupe de la création des connexions aux trains et de l'extension de la liste des trains gérés par le serveur.

2.7 train.c

Le fichier "train.c" représente une simulation du panneau de commande d'un conducteur de train sur le chemin de fer. Il permet à un utilisateur de contrôler un train simulé sur le système ferroviaire et de communiquer avec le serveur.

Nous avons créé un enum appelé `state_t` qui représente l'état actuel du programme. Il est utilisé pour guider l'utilisateur dans le menu d'interaction.

Nous avons défini une fonction `print_menu` qui affiche le menu du panneau de contrôle du conducteur en fonction de l'état actuel du programme.

Nous initialisons une structure de données `train_t` qui représente le train contrôlé par le conducteur. Initialement, le train est créé avec un type (TGV) et un identifiant (123), s'il n'est pas donné en paramètre par l'utilisateur.

Nous lisons le port et l'adresse IP du serveur dans les arguments de la ligne de commande. Ces arguments sont optionnels et le programme utilise les valeurs par défaut s'ils ne sont pas fournis.

Nous créons un socket client et nous nous connectons au serveur à l'aide de la fonction `connect_to_server`.

Nous vérifions les arguments de la ligne de commande pour déterminer l'état initial du programme (sélection du type de train ou du type de train et de l'ID).

Nous affichons le menu initial en appelant la fonction `print_menu`.

Les données relatives au train sont envoyées au serveur à l'aide de la fonction `sendto`.

Une boucle est lancée pour permettre à l'utilisateur d'interagir avec le train simulé via le menu.

```
// TODO make good header

// desc. this file simulates the train control board of the conductor

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

#include "../include/railway.h"
#include "../include/server.h"

typedef enum {
    select_type ,
    select_id ,
    base_case ,
} state_t ;

void print_menu( train_t * train , state_t state );

int main( int argc , char ** argv ){

    train_t train ;

    create_train( &train , TGV, "123" );

    int running = 1;
    state_t state = 0;

    // implement command line parsing to obtain :
    // TODO is there any other thing to obtain from command line interface?
    int port = 1994;
    if(argc == 2){
        port = atoi( argv[1] );
    }

    char * ip = "127.0.0.1";

    // start server integration
    //int socket_fd = init_tcp_server( port );
    int socket_fd = connect_to_server( ip , port );
    assert( socket_fd > 0 ); // has successfully created the socket

    if(argc > 2){
        /*
        if(argc == 3){ // has only type
            strcpy( train.type_id , argv[1] );
        }
        */
    }
}
```



```

        // send command to end
        message_t message = { {"e"}, train } ;
        int send_r = sendto( socket_fd , &message , sizeof(message) , 0 ,
        assert(send_r != -1);

        //printf("[ ] \t [%d] / ", 0 );
        //print_train_params( &train );
        //printf("\n");
    }

    //int send_r = sendto( socket_fd , buffer , strlen(buffer) -1 , 0 ,
    //assert(send_r != -1);

}
else if(buffer[0] == 'u' ){           // update location

    message_t message = { {"u"}, train } ;
    int send_r = sendto( socket_fd , &message , sizeof(message) , 0 , NUL
    assert(send_r != -1);

    int number = recv( socket_fd , &message , sizeof(message) , 0 ) ;
    //printf("[%s]", message2.message);

    train.eoa = message.train.eoa ;

    printf("[ ] \t [%d] | ", 0 );
    print_train_params( &train );
    printf("\n");

}
else{
    printf("unknown command, try again ... ");
}

//printf("[%s]", buffer);

}

printf("This is the end!\n");

return 0;
}

void print_menu( train_t * train , state_t state){

```



```

        printf( "\n" );

        printf( " possible_commands: \n" );
        printf( "$: m\t// move 1 unit forward \n" );
        printf( "$: u\t// requests new EOA \n\n" );

    }

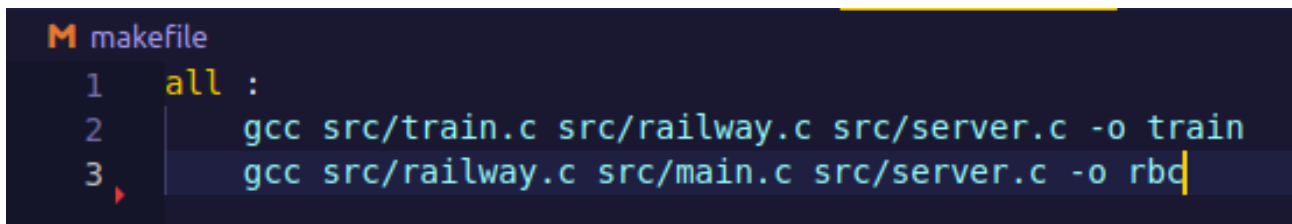
}

```

En bref, "train.c" simule l'interaction du conducteur avec un train sur la voie ferrée. Il permet à l'utilisateur de contrôler le train et de communiquer avec le serveur à l'aide de commandes spécifiques.

2.8 makefile

Le Makefile est un fichier utilisé pour automatiser la compilation d'un projet. Il contient des règles et des instructions qui indiquent au système de compilation (tel que make) comment compiler et lier les fichiers sources du projet en un programme exécutable.



```

M makefile
1  all :
2      gcc src/train.c src/railway.c src/server.c -o train
3      gcc src/railway.c src/main.c src/server.c -o rbc

```

FIGURE 2.2 – Fichier makefile

- *gcc src/train.c src/railway.c src/server.c -o train* : Cette ligne compile les fichiers sources "train.c", "railway.c" et "server.c" à l'aide du compilateur GCC. Les fichiers sources sont situés dans le dossier "src.". La commande -o train spécifie le nom du fichier de sortie du programme exécutable qui sera généré, dans ce cas "train".
- *gcc src/railway.c src/main.c src/server.c -o rbc* : Comme la ligne précédente, cette ligne compile les fichiers "railway.c", "main.c" et "server.c" dans le dossier "src" et génère l'exécutable appelé "rbc".

Globalement, ce Makefile est simple et ne contient que les instructions de base pour compiler les fichiers du projet. Il crée deux exécutables, "train" et "rbc", qui correspondent aux composants du système ferroviaire, le panneau de contrôle du conducteur et le centre de contrôle radio.

Chapitre 3

DEMO

```
gosmann@gosmann-G3-3579: ~/Documents/STA/systeme/trainspotting
File Edit View Search Terminal Help
gosmann@gosmann-G3-3579:~/Documents/STA/systeme/trainspotting$ ./rbc 2222
socket created successfully
new connection from [ip:127.0.0.1] and [port:51976] at [PID:22761]
printing all trains :
[0] | 🚂 | [TGV456] | [LOC, EOA]: [ 0, 0 ]

end of connection from [ip:127.0.0.1] and [port: 51976] at [PID:22761]

printing all trains :

new connection from [ip:127.0.0.1] and [port:51802] at [PID:22833]
printing all trains :
[0] | 🚂 | [TGV456] | [LOC, EOA]: [ 0, 0 ]

printing all trains :
[0] | 🚂 | [TGV456] | [LOC, EOA]: [ 0, 101 ]
```

FIGURE 3.1 – ./rbc

```
gosmann@gosmann-G3-3579: ~/Documents/STA/systeme/trainspotting
File Edit View Search Terminal Help
gosmann@gosmann-G3-3579:~/Documents/STA/systeme/trainspotting$ make
gcc src/train.c src/railway.c src/server.c -o train
gcc src/railway.c src/main.c src/server.c -o rbc
gosmann@gosmann-G3-3579:~/Documents/STA/systeme/trainspotting$ ./train 2222
socket created successfully
new connection made at [ip:127.0.0.1] and [port:2222] at [PID:22832]

🚂🚂🚂🚂🚂🚂🚂🚂🚂🚂🚂🚂🚂🚂🚂🚂
MACHINIST CONTROL PANEL

write your train type [ex. "TGV"] :
$ : TGV
write your train id [ex. "123"] :
$ : 456
print base case :
🚂 | [TGV456] | [LOC, EOA]: [ 0, 0 ]

possible commands:
$ : m // move 1 unit forward
$ : u // requests new EOA

$ : m
not allowed by "end of authority" (EOA)

$ : u
[0] | 🚂 | [TGV456] | [LOC, EOA]: [ 0, 101 ]

$ :
```

FIGURE 3.2 – ./train

```

gosmann@gosmann-G3-3579: ~/Documents/STA/systeme/trainspotting
File Edit View Search Terminal Help
gosmann@gosmann-G3-3579:~/Documents/STA/systeme/trainspotting$ ./rbc 2222
socket created successfully
new connection from [ip:127.0.0.1] and [port:51976] at [PID:22761]
printing all trains :
[0] | [TGV456] | [LOC, EOA]: [ 0, 0 ]
end of connection from [ip:127.0.0.1] and [port: 51976] at [PID:22761]
printing all trains :
new connection from [ip:127.0.0.1] and [port:51802] at [PID:22833]
printing all trains :
[0] | [TGV456] | [LOC, EOA]: [ 0, 0 ]
printing all trains :
[0] | [TGV456] | [LOC, EOA]: [ 0, 101 ]
new connection from [ip:127.0.0.1] and [port:44182] at [PID:23036]
printing all trains :
[0] | [TGV456] | [LOC, EOA]: [ 0, 101 ]
[1] | [RER999] | [LOC, EOA]: [ 0, 0 ]
new connection from [ip:127.0.0.1] and [port:46054] at [PID:23296]
printing all trains :
[0] | [TGV456] | [LOC, EOA]: [ 0, 101 ]
[1] | [RER999] | [LOC, EOA]: [ 0, 0 ]
[2] | [TER159] | [LOC, EOA]: [ 0, 0 ]
new connection from [ip:127.0.0.1] and [port:52662] at [PID:23348]
printing all trains :
[0] | [TGV456] | [LOC, EOA]: [ 0, 101 ]
[1] | [RER999] | [LOC, EOA]: [ 0, 0 ]
[2] | [TER159] | [LOC, EOA]: [ 0, 0 ]
[3] | [TGV666] | [LOC, EOA]: [ 0, 0 ]
end of connection from [ip:127.0.0.1] and [port: 52662] at [PID:23348]
printing all trains :
[0] | [TGV456] | [LOC, EOA]: [ 0, 101 ]
[1] | [RER999] | [LOC, EOA]: [ 0, 0 ]
[2] | [TER159] | [LOC, EOA]: [ 0, 0 ]
new connection from [ip:127.0.0.1] and [port:33050] at [PID:23444]
printing all trains :
[0] | [TGV456] | [LOC, EOA]: [ 0, 101 ]
[1] | [RER999] | [LOC, EOA]: [ 0, 0 ]
[2] | [TER159] | [LOC, EOA]: [ 0, 0 ]
[3] | [TGV123] | [LOC, EOA]: [ 0, 0 ]

```

FIGURE 3.3 – Connexions multiples à RBC

```

gosmann@gosmann-G3-3579: ~/Documents/STA/systeme/trainspotting
File Edit View Search Terminal Tabs Help
gosmann@gosmann-G3-3579:~/Documents/STA/systeme/trainspotting$ netstat | grep 2222
tcp        0      0 localhost:2222    localhost:46054   ESTABLISHED
tcp        0      0 localhost:52662   localhost:2222    TIME WAIT
tcp        0      0 localhost:46054   localhost:2222    ESTABLISHED
tcp        0      0 localhost:51802   localhost:2222    ESTABLISHED
tcp        0      0 localhost:2222    localhost:51802   ESTABLISHED
tcp        0      0 localhost:33050   localhost:2222    ESTABLISHED
tcp        0      0 localhost:2222    localhost:33050   ESTABLISHED
tcp        0      0 localhost:2222    localhost:44182   ESTABLISHED
tcp        0      0 localhost:44182   localhost:2222    ESTABLISHED

```

FIGURE 3.4 – Connexions actives à RBC

Chapitre 4

Conclusion

En résumé, le projet décrit un système ferroviaire simulé avec plusieurs composants. Le fichier "client.c" représente la partie client du système, permettant au client d'envoyer des messages au serveur, essentiels pour la communication entre les trains et le centre de contrôle central. Le fichier "main.c" est le composant principal du système, simulant le centre de contrôle des trains (Radio Block Centre), responsable du démarrage du serveur et de la gestion des connexions des trains.

Le fichier "train.c" simule le panneau de contrôle du conducteur, permettant à l'utilisateur de contrôler un train simulé et de communiquer avec le serveur. Le programme offre un menu interactif et prend en charge des commandes pour déplacer le train en avant et demander de nouvelles informations.

En résumé, le projet couvre l'ensemble de la simulation ferroviaire, de la partie cliente à la partie serveur, avec un panneau de contrôle du conducteur pour interagir avec le train. Chaque composant joue un rôle crucial dans la simulation et la communication au sein du système ferroviaire simulé.

En outre, dans notre travail, nous avons atteint avec succès l'objectif de créer un système de simulation ferroviaire, ce qui a non seulement consolidé notre formation en programmation C, mais nous a également permis d'explorer des concepts avancés tels que les sockets, les threads et la communication entre les processus. Tout au long du projet, nous avons beaucoup appris sur la mise en œuvre de systèmes de communication robustes et efficaces, qui sont cruciaux dans les scénarios du monde réel.

Le choix d'utiliser des listes chaînées pour représenter les trains et leurs connexions au serveur s'est avéré être une décision intelligente. Par rapport aux tableaux, les listes chaînées optimisent l'utilisation de la mémoire et rendent le système plus dynamique. Elles ont permis d'ajouter ou de supprimer des trains facilement, en s'adaptant aux changements dans le réseau ferroviaire simulé, un scénario qui est beaucoup plus compliqué à gérer avec des tableaux fixes.

Ce projet a non seulement renforcé nos bases en programmation C, mais nous a également donné l'occasion de créer une application pratique et fonctionnelle qui simule les interactions dans un environnement de contrôle ferroviaire. En outre, le projet a démontré l'importance des structures de données appropriées lors de la construction de systèmes efficaces et évolutifs.

En résumé, nous sommes satisfaits du résultat du projet et pensons qu'il a contribué de manière significative à notre apprentissage et à notre expérience en matière de programmation, de réseaux et de systèmes. Cette expérience nous a permis d'être mieux préparés à relever les défis futurs en matière de programmation et nous a donné une meilleure appréciation des complexités impliquées dans la simulation des systèmes du monde réel.

Bibliographie

- [RK88] D.M. RITCHIE et B.W. KERNIGHAN. *The C programming language*. Bell Laboratories, 1988.