Autonomous Driving Recruitment Task

Francisco Pedrosa – 2181248

Sensor Simulation System

To solve the task, I decided to create 2 programs: The sensor simulating software (client) and the sensor reading software (server). The code can be found in the repository that I have created for this propose (github.com/francsp1/lart_sensor_simulator). The programs were written in the C programming language and in a Linux environment. The project can be compiled using the given Makefile.

To compile and run the program you should follow these instructions:

```
sudo apt install build-essential

sudo apt install gengetopt

git clone https://github.com/francsp1/lart_sensor_simulator.git

cd lart_sensor_simulator

sudo cp ./src/lib/libqueue.so /usr/local/lib

sudo ldconfig

make
```

The libqueue.so is an implementation of a thread safe queue necessary to run the server.

1. Project structure

This project has the following directory structure:

- **bin/**: This directory contains the two compiled binary executables (server and client).
- **inc/**: This directory contains all the header files (.h) of the project. This directory has a similar subdirectory structure as src/ and obj/.
- **obj/**: This directory contains all object files (.o) of the project. This directory has a similar subdirectory structure as src/ and inc/.
- **src/**: This directory contains all the source code files (.c) for the project.
    - **src/cli/**: Contains the sensor simulation software (client) source code files (.c)
        - **src/cli/args/**: Contains the .ggo file and the code generated by gengetopt to handle command line arguments of the client.
            - args.ggo**:** This file is used by gengetopt to generate the code for the client command line arguments.
            - args.c: This is the code generated by gengetopt for the client command line arguments.

        - main.c: This file contains the main function for the client application and the function ran by the client threads.
        - client_socket.c: This file implements the functions to create and manage the client socket, pack, serialize and send sensor data to the server.
        - client_threads.c: This file contains the functions needed to handle client threads.
    - **src/lib/**: Contains all the shared objects files (.so) necessary to run and compile the code. The .so files must be placed where the operating system can find them (For example in "/usr/local/lib").
        - libqueue.so: This is a shared object file that contains a thread safe queue implementation.
    - **src/srv/**: Contains the sensor reading software (server) source code files (.c)
        - **src/cli/args/**: Contains the .ggo file and the code generated by gengetopt to handle command line arguments of the server.
            - args.ggo**:** This file is used by gengetopt to generate the code for the server command line arguments.
            - args.c: This is the code generated by gengetopt for the server command line arguments.
        - main.c: This file contains the main function for the server application, the function ran by the server threads and the signal handler function.
        - server_socket.c: This file implements the functions to create the server socket, receive and deserialize sensor data.
        - server_threads.c: This file contains the functions needed to handle server threads.
        - server_queues.c: This file implements the functions to create and destroy the server queues.
    - common.c: This file contains common functions, structures and macros used by both the server and the client.

2. Sensor reading software (Server)

To run the server, you need to specify the port argument to tell in which port you want the server to listen in. The server will bind to every IP address. Example:

- ./bin/server --port 8080

The sensor reading software starts by:

- Validate the port argument given by the user (most be an integer number between 1024 and 65535).
- Initialize the signal handlers for SIGINT and SIGTERM.
- Create the server socket and bind it to every IP address of the system where it is running, with the given port.

- Open the logs file and create a mutex to safely write to it. The server logs will be placed in the current directory within a file named “server_logs.txt”.
- Create as many queues as the number of sensors and place them in the queues array. Each sensor was its own queue. When a message from a sensor arrives, it will be placed in the corresponding queue so it can be processed in the future. The index of the queue array corresponds to the sensor id (Example: the message sent by sensor with the id 0 will the placed in queues[0]).
- Create as many threads as the number of sensors. Each sensor was its own thread. The sensor id, the thread id (tid) and the index of the corresponding queue in the array of queues are the same. (Example: the thread with a tid of 0 will remove messages sent by the sensor with the id 0 from the queue in queues[0]).

After this initial process the server is ready to receive messages from the sensor simulation software (client). Until the server receives an SIGTERM or SIGINT the server will:

- Receive data from the client, reading it from the socket and place it into a buffer.
- Deserialize the received data by converting it from network endianness to host endianness.
- Place the data in the corresponding queue.

While the main thread is receiving the data and placing it into the corresponding queue, the rest of threads are removing the received data from the corresponding queue and producing a log in the “./server_logs.txt” file with the timestamp and the received float value.

If the server receives a SIGINT or SIGTERM the main thread will stop receiving messages and wait for the rest of the threads to empty the queues. After this process it will print to the standard output (stdout) some statistics like the number of messages received from each sensor, the sensor average received value and the total messages received. After all this steps the server terminates.

3. Sensor simulating software (Client)

To run the sensor simulation software (client) you must specify these arguments:

- --ip (-i): This is the server ip address.
- --port (-p): Use this argument to specify the server port.
- --packets_per_sensor (-s): This argument is used to tell the client how many messages you want to send from each sensor. If this argument was not given it will assume a default value of 100 messages.

Example:

- ./bin/client --ip 127.0.0.1 --port 8080 --packets_per_sensor 500

The sensor simulating software will:

- Validate the port argument given by the user (most be an integer number between 1024 and 65535).
- Create a socket to send the sensor data.
- Open the logs file and create a mutex to safely write to it. The client logs will be placed in the current directory within a file named “client_logs.txt”. Each log contains a timestamp and the sensor float value.
- Create as many threads as the number of sensors. Each thread will send random float values from a specific sensor at 10hz (1 packet every 100 milliseconds). The threads are also responsible to produce a log and write it to the “./client_logs.txt” file.
- Wait for the threads to send the number of messages specified in the --packets_per_sensor argument.
- When the threads finish sending all messages, the client will terminate.

4. Communication between the server and the client

For the communication between the server and the client I have decided to create a small protocol. This protocol is composed of one enumeration and 2 structures defined in the “common.h” file.

The protocol supports the following types of messages:

- PROTO_SENSOR_DATA: This message type is used to exchange sensor data between the server and the clients.

The type of the message is defined by the “proto_type_e” enumeration.

Each message consists of a header and data. The header and data are defined by the “proto_hdr_t” and “proto_sensor_data_t” structures, respectively.

The header of each message is created using the “proto_hdr_t” stucture and contains the following fields:

- type: This field specifies the type of the message. It is an enumeration of type “proto_type_e”.
- sensor_id: This field specifies the ID of the sensor that the data is associated with. This is of the type “uint32_t”.
- len: This field specifies the length of the data in the message. Its type is “uint16_t”.

The “proto_sensor_data_t” structure defines the messages that the client will send to the server with the sensor data. It was the following parameters:

- hdr: This field is the header of the message. It is of type “proto_hdr_t”.
- data: This field holds the float value that the sensor sends to the server. It is of the type “uint32_t”. If you need to use this value, access it using the “get_float_value” function.

When a client has sensor data to send to the server, it creates a “proto_sensor_data_t” message. The client sets the “type” field of the message header to “PROTO_SENSOR_DATA”, the “sensor_id” field to the ID of the sensor, and the “len” field to the length of the data. The client then sets the data field of the message to the sensor data. After packing the data, the client serializes it, converting it from host endianness to network endianness. The serialization process is handled by the “serialize_sensor_data” function. This function takes a “proto_sensor_data_t” message as input and converts all its fields from host byte order to network byte order using the “htonl” and “htons” functions. This ensures that the message can be correctly interpreted by the server, regardless of the endianness of the server's system."

The client sends the proto_sensor_data_t message to the server. The server reads the message, extracts the sensor data, and processes it as necessary.