Instituto Superior de Engenharia de Coimbra

Licenciatura em Engenharia Informática

Relatório

Plataforma de gestão de serviço de táxi

Sistemas Operativos 2025/2026

David Quaresma 2021133301

Gonçalo Almeida 2022136657

Índice

1. Introdução

Este trabalho prático, desenvolvido no âmbito da unidade curricular de Sistemas Operativos, tem como objetivo a implementação de um sistema de simulação para a gestão de uma plataforma de táxis.
Este relatório detalhará a arquitetura e os métodos de comunicação adotados.

2. Estrutura do projeto

2.1. Arquitetura

A estrutura do projeto é fundamentalmente definida por uma arquitetura cliente servidor, sendo o servidor o controlador e o cliente o cliente.
Cada cliente que queira usar o sistema, possui um terminal diferente, que permite interagir com o controlador.
O servidor, neste contexto o controlador, é o que centraliza a gestão de todo o sistema, sendo responsável pela receção e agendamento dos pedidos dos clientes, pela alocação dos veículos e pela monitorização contínua do estado da frota. Dada a sua função, a terminação do controlador exige o encerramento de todos os componentes associados do sistema.

2.1.1. Estruturas

Data.h

O header data.h tem o objetivo de definir todas as estruturas, constantes e tipos de dados necessários à gestão do sistema e à comunicação entre os processos. Este ficheiro é incluído pelos três programas executáveis (controlador, cliente, veiculo).

```c
typedef struct {
    int id;
    VehicleActiveStatus active;
    VehicleAvailability available;
    int progress_percent;  // 0-100
    int service_id;  // -1 se não atribuído
    pid_t process_pid;
    double total_km;
} VehicleInfo;
```

Figura 1 - Estrutura VehicleInfo

```c
typedef struct {
    int id;
    char client_name[50];
    int client_pid;
    int scheduled_time;  // em segundos
    char origem[100];
    char destino[100];
    int vehicle_id;  // -1 se não atribuído
    ServiceStatus status;
    double distance_km;
} ServiceInfo;
```

Figura 2 - Estrutura ServiceInfo

```c
// --- Estrutura Mensagem (Cliente -> Controlador) ---
typedef struct {
    pid_t client_pid;
    char client_name[50];
    RequestType type;
    char data[BUFFER_SIZE];
} ClientMessage;
```

Figura 3 -Estrutura ClientMessage

controller.c

Este ficheiro contém a função main que inicializa o sistema, cria o pipe de servidor e lança todas as threads de gestão como por exemplo, client_listener_thread, scheduler_thread, vehicle_telemetry_thread. Também implementa a lógica para manipular as estruturas de dados globais, com o mutex, e as funções para os comandos administrativos.

client.c

Contém a lógica da interface do utilizador. A função main, analisa os comandos e envia os pedidos. É responsável por criar o named pipe e lançar a thread server_response_listener para ouvir as respostas e notificações do controlador.

vehicle.c

Contém a lógica do veículo. Recebe os detalhes do serviço, na função main executa o loop de simulação de viagem e, através da função send_telemetry, reporta o progresso ao controlador. Também tem a função contact_client para interagir diretamente com o processo cliente via namedpipe.

2.1.2. Threads

Neste projeto, foram desenvolvidas quatro threads para além da main thread no controlador.

Thread client_listener_thread

Esta thread é responsável por ficar à espera de atividade no pipe, que significa que chegaram novos pedidos por parte dos clientes. Trata todos os passos para o processamento de pedidos, desde o registo inicial até ao agendamento, cancelamento e consulta de serviços, enviando as respetivas respostas aos clientes.

```
// --- Thread de Leitura ---
void* client_listener_thread(void* arg) {
    int fd = open(PIPE_SERVER, O_RDWR);
    if (fd == -1) return NULL;

    ClientMessage msg;
    while (keep_running) {
        if (read(fd, &msg, sizeof(ClientMessage)) > 0) {
            // Processamento seguro com Mutex
            pthread_mutex_lock(&data_mutex);
```

Figura 4 - Thread de leitura

Thread time_simulator_thread

Esta thread é responsável pela gestão do tempo simulado do sistema. Funciona num ciclo contínuo, com pausas de 1 segundo de tempo real, para incrementar a variável global simulated_time.

Thread scheduler_thread

Esta thread é responsável pela gestão dos serviços agendados. Verifica de tempo em tempo se existem serviços cuja hora agendada foi atingida pelo tempo simulado. Caso encontre um serviço válido e um veículo disponível na frota, faz o lançamento do processo “veículo” e atualiza o estado do serviço para "em progresso".

```
void* scheduler_thread(void* arg) {
    while (keep_running) {
        sleep(1);

        pthread_mutex_lock(&data_mutex);

        // Verificar serviços agendados que devem ser iniciados
        for (int i = 0; i < num_services; i++) {
            if (services[i].status == STATUS_SCHEDULED &&
                services[i].scheduled_time <= simulated_time &&
                services[i].vehicle_id == -1) {

                // Encontrar veículo disponível
                int vehicle_idx = find_available_vehicle();
                if (vehicle_idx != -1) {
                    services[i].vehicle_id = vehicles[vehicle_idx].id;
                    services[i].status = STATUS_IN_PROGRESS;
                    vehicles[vehicle_idx].available = VEHICLE_OCCUPIED;
                    vehicles[vehicle_idx].service_id = services[i].id;

                    // Atualizar cliente para em viagem
                    for (int c = 0; c < num_clients; c++) {
                        if (clients[c].pid == services[i].client_pid) {
                            clients[c].status = CLIENT_ON_TRIP;
                            break;
                        }
                    }
```

Figura 5 - Thread Schedule

Thread vehicle_telemetry_thread

Esta thread é responsável por monitorizar a atividade de toda a frota. Fica à escuta nos pipes de telemetria de todos os veículos ativos. Recebe e processa atualizações do estado do veiculo, como o início de viagem, o progresso percentual e a conclusão do serviço, atualizando as estruturas do controlador.

Quando é indicado ao controlador para terminar, todas as threads terminam o seu ciclo.

3. MakeFile

O makefile define as variáveis do compilador e as flags essenciais, garantindo a consistência do build para os três componentes do sistema. O ficheiro tem cinco regras, onde o target all desencadeia a criação dos executáveis controlador, cliente e veiculo. Adicionalmente, a regra clean assegura a manutenção do ambiente.

```
# --- Variáveis ---
CC = gcc
CFLAGS = -Wall -pthread -g
OBJ_COMMON = common/data.h

# --- Targets ---
all: controlador cliente veiculo

controlador: controller.c $(OBJ_COMMON)
    $(CC) $(CFLAGS) controller.c -o controlador

cliente: client.c $(OBJ_COMMON)
    $(CC) $(CFLAGS) client.c -o cliente

veiculo: vehicle.c $(OBJ_COMMON)
    $(CC) $(CFLAGS) vehicle.c -o veiculo

clean:
    rm -f controlador cliente veiculo
    rm -f /tmp/taxi_*
```

Figura 6 - Makefile

4. Conclusão

Os testes realizados não detetaram anomalias na comunicação crítica entre o servidor e os clientes, demonstrando a eficácia dos mecanismos implementados. Verificou-se que o sistema assegura um encerramento ordenado, garantindo que todas as threads terminam a sua execução corretamente e que os recursos do sistema são integralmente removidos, não deixando restos de execuções anteriores.