Relatório - Trabalho Prático 2
Redes de Computadores
Patrulha antidesmatamento

Arthur Araujo Rabelo
Universidade Federal de Minas Gerais

2025/2

1 Introdução

O presente trabalho tem como objetivo a implementação de um simulador de uma patrulha antidesmatamento na Amazônia Legal. O sistema foi implementado completamente em C, utilizando *sockets* UDP e *threads POSIX*. Para a implementação do projeto, foram criadas algumas estruturas de dados e módulos, descritos abaixo juntamente com as decisões de projeto envolvidas.

2 Estruturas de Dados e Módulos

- **Communication (.h e .c)**: Declara e define todas as *structs*, variáveis, funções e *libs* que serão usadas para a comunicação entre o cliente e o servidor. Entre as principais funções estão *send_message* e *receive_message*, que utilizam *recv_from* e *send_to*. Todas as mensagens são recebidas e enviadas por meio de um *buffer* de tipo *char*, sobre o qual é feito um *cast* para as estruturas de comunicação. Essas funções também chamam outras auxiliares que tratam os problemas de *endianness*, convertendo as *structs header* e *payload* para os tipos da rede e do *host*. A sua implementação foi motivada principalmente para organização e facilidade. Dessa forma, a responsabilidade de receber, enviar e tratar (*casting*, *prints* e conversão de *endianness*) os diferentes tipos de mensagens fica mais nessas funções do que no cliente e no servidor.

- **Graph (.h e .c)**: Define as estruturas que serão os vértices e as arestas do grafo. A implementação foi feita com uma lista de adjacência (estrutura de dados usada para representar um grafo, consistindo em um *array* onde cada posição corresponde a um vértice e em cada posição contém os vértices que são diretamente conectados a esse vértice, ou seja, os seus vizinhos). A sua principal função é o *dijkstra*, usado para achar as menores distâncias entre um vértice de destino e todos os outros. Essa estrutura foi escolhida pela simplicidade e por minha própria familiaridade com ela.

- **Utils (.h e .c)**: Implementa duas funções importantes para o projeto: *handle_telemetry* e *findNearestAvailableDroneCrew*. As duas são usadas pelo servidor. A primeira percorre os dados da telemetria recebida e insere os alertas na fila de alertas (explicada

logo abaixo). A segunda recebe o grafo da floresta, chama o *dijkstra* para ele e acha a capital mais próxima da cidade em alerta.

- **AlertQueue (.h e .c)**: Fila FIFO (*First-In-First-Out*) que registra os alertas para que o servidor possa atendê-los. A sua implementação foi feita através de uma lógica bem parecida com listas encadeadas, mas com um ponteiro para o primeiro elemento e para o último.

- **AnswerQueue (.h e .c)**: Fila FIFO (*First-In-First-Out*) que registra respostas do servidor para que controle das *threads*. Essas respostas são armazenadas na estrutura *answer_t*, que guarda as estruturas de uma mensagem recebida, além de uma *flag* que indica se a mensagem foi completamente recebida ou não, uma vez que a comunicação é feita por UDP. As *threads* possuem filas desse tipo para registrar os *acks* e as equipes de drone que são enviadas pelo servidor. Cada *thread* terá sua(s) própria(s) filas, que só guardarão um tipo de resposta. Como será explicado com mais detalhes adiante, existe uma *thread* receptora que ficará responsável por encaminhar as mensagens do servidor para a *thread* correta, inserindo-as em uma de suas filas de acordo com o tipo de mensagem.

- **ThreadsUtils (.h e .c)**: Declara, define e implementa as funções, os *mutexes*, os *inputs*, as variáveis compartilhadas, condicionais e barreiras utilizadas pelas *threads*. Cada *thread* possui a sua função:

  - *Dispatcher*: *foward_message*
  - 1: *modify_telemetry_data*
  - 2: *send_telemetry*
  - 3: *confirm_crew_received*
  - 4: *simulate_drones*

  Cada uma dessas funções utiliza os *mutexes* e *conds* para evitar condições de corrida entre as estruturas compartilhadas pelas *threads*.

- **Servidor (server.c)**: Recebe e envia mensagens ao cliente. De início, espera mensagens do cliente. Quando as recebe, executa operações de acordo com o tipo da mensagem. Algumas decisões de implementação:

  - Para despachar drones, o servidor não tira os alertas da fila. Ele apenas faz isso (*pop operation*), quando chega um *MSG_CONCLUSAO*.
  - O atendimento dos alertas não leva em conta o seu atributo *timestamp*. Ele apenas atende o primeiro alerta que está na fila (evidentemente, por ser uma fila FIFO).

- **Cliente (client.c)**: Recebe e envia mensagens ao servidor. Inicializa as *threads* e as estruturas, variáveis e parâmetros usados por elas. Usa uma barreira (*barrier*) para que todas as *threads* comecem a sua execução ao mesmo tempo.

3 Threads

- Thread Receptora: responsável por encaminhar as mensagens recebidas do servidor para as outras *threads*. Sua implementação foi uma decisão pensando no problema de concorrência pelo *socket* entre as *threads*. Uma poderia acabar recebendo as mensagens da outra, causando confusões no programa. Sendo assim, apenas esta *thread* recebe as mensagens e as multiplexa, adicionando nas filas da *thread* responsável as mensagens. O acesso a essas filas é controlado por *mutexes*. Junto a eles, também estão as variáveis condicionais, que são "acordadas"quando uma resposta do servidor é adicionada a fila. As *threads* ficam esperando as mensagens chegarem na fila (verificando se esta está vazia) por meio de *pthread_cond_wait's*.

- Thread 1: altera o vetor *telemetry_data* durante toda a execução do programa, concorrendo o acesso a ele com a *thread* 2.

- Thread 2: copia os dados alterados pela *thread* 1 para os dados de telemetria que serão enviados ao servidor. Possui um controle de tentativas.

- Thread 3: recebe as ordens de drones e se comunica com *thread* 4 através de variáveis globais: *mission_available*, que diz se há uma missão disponível;*mission_completed*, que indica quando uma missão é completa; *mission_city_id*, que recebe a cidade recebendo a missão; *mission_crew_id*, que recebe a equipe atuando na missão e *th4_busy*, que indica se a *thread* 4 está ocupada.

- Thread 4: simula a execução das missões, o que é feito com um simples *sleep()*.

4 Melhorias e Decisões de Projeto

Durante a implementação do programa, algumas melhorias que poderiam ser feitas, preenchendo algumas lacunas das decisões de projeto e das estruturas definidas no enunciado, foram identificadas:

- Quando o servidor recebe uma mensagem do tipo *MSG_CONCLUSAO*, é feito um *pop()* na fila de alertas. Isso parece funcionar bem, não causando confusões no programa, mas semanticamente é incorreto. O alerta que deveria ser retirado é o da cidade atendida, indicada no *payload* de conclusão.

- As mensagens de tipo *ACK*, especialmente para o recebimento de equipes e de conclusão de uma missão, não especificam qual cidade recebeu o drone ou teve sua missão concluída. Isso poderia ser implementado para maior organização.

- A estrutura que guarda os alertas possui um campo *timestamp*, que não foi utilizado. Pensando na lógica FIFO da fila, aqueles alertas que chegam primeiro (mais antigos) são atendidos primeiro. Portanto, esse campo, explicitado no enunciado, acabou por ser irrelevante.

- O controle de tentativas só foi explicitado no enunciado quando se tratava do envio da telemetria. Pensando em melhorias, isso poderia ser feito em todo envio de mensagens, tanto no cliente quando no servidor.

- A interrupção por CTRL+C do cliente não é muito responsiva, por causa das *threads* e suas variáveis condicionais, *mutexes* e *sleep's*. É provável que ao interromper, você deva esperar um pouco até que a finalização seja concluída e que mensagens aleatórias sejam imprimidas no *console*. Isso não ocorre no servidor.

5 Conclusão e Resultados

Os resultados do programa foram bons, funcionando correta e eficientemente. Não foram identificados vazamentos de memória, o que indica que esta foi alocada e desalocada corretamente. Abaixo estão testes realizados com *valgrind*:

```
[Encerrando servidor...]
Liberando recursos...
Servidor encerrado com sucesso.
==6720==
==6720== HEAP SUMMARY:
==6720==     in use at exit: 0 bytes in 0 blocks
==6720==   total heap usage: 20 allocs, 20 frees, 6,308 bytes allocated
==6720==
==6720== All heap blocks were freed -- no leaks are possible
==6720==
==6720== For lists of detected and suppressed errors, rerun with: -s
==6720== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

```
Cliente encerrado com sucesso.
==167611==
==167611== HEAP SUMMARY:
==167611==     in use at exit: 0 bytes in 0 blocks
==167611==   total heap usage: 33 allocs, 33 frees, 12,332 bytes allocated
==167611==
==167611== All heap blocks were freed -- no leaks are possible
==167611==
==167611== For lists of detected and suppressed errors, rerun with: -s
==167611== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Por fim, este trabalho permitiu aplicar, de forma prática, os conceitos de comunicação UDP, controle de concorrência e roteamento vistos em sala, como confirmações de mensagem, tempo para retransmitir, *mutexes* e *threads* e encontrar o caminho mais curto.