Trabalho Prático 2 de Redes de Computadores

[Rafael Sant'Ana]

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

rafael-sant-ana@ufmg.br, 2023087834

1 Introdução

O desenvolvimento do sistema distribuído "Patrulha Antidesmatamento"para o monitoramento da Amazônia Legal apresentou desafios bem diferentes do primeiro trabalho prático. Enquanto o TP1 focava na confiabilidade inerente do TCP e na lógica de jogo, este TP2 exigiu que eu entendesse as limitações do protocolo UDP (User Datagram Protocol), o gerenciamento de concorrência com *Threads* POSIX e a aplicação de algoritmos de grafos (Dijkstra) para roteamento geoespacial. Este documento descreve as decisões de projeto, as dificuldades de sincronização e as estratégias adotadas para garantir a robustez de um sistema sem conexão permanente.

2 Desafios Encontrados na Implementação

O UDP ser "não confiável"e a necessidade de paralelismo no cliente trouxeram obstáculos técnicos significativos:

2.1 1. Comunicação UDP e Endianness

Diferente do TCP, o UDP não garante entrega nem ordem, e não mantém estado de conexão.

- **Gerenciamento de Estado sem Conexão:** Como não há `connect()` permanente no servidor, tive que estruturar a comunicação baseada inteiramente em troca de mensagens atômicas (`sendto`/`recvfrom`). O servidor precisava ser capaz de receber um pacote de qualquer cliente a qualquer momento e identificar o contexto apenas pelo cabeçalho.
- **Endianness (Byte Order):** Um desafio crucial foi garantir a interoperabilidade. Aprendi na prática que a conversão de todos os inteiros para *Network Byte Order* (Big Endian) usando `htonl()` e `htons()` é obrigatória. Tive problemas ao testar com servidores de terceiros que liam a memória diretamente (*raw casting*), o que resultava em interpretação errada dos valores em arquiteturas Little Endian. A solução foi padronizar rigorosamente a serialização e desserialização em ambos os lados.

2.2 2. Concorrência e Sincronização (Threads)

O cliente exigiu a coordenação de 4 threads simultâneas, criando cenários de condição de corrida:

- **Produtor-Consumidor de Estado:** A *Thread de Monitoramento* altera o estado das cidades aleatoriamente, enquanto a *Thread de Telemetria* lê esse estado para enviar ao servidor. O

uso de `pthread_mutex_t` foi essencial para evitar leituras inconsistentes (leitura suja) durante a atualização dos sensores.

- **Coordenação com Timeout:** Implementar a lógica de espera por ACK com timeout foi complexo. Utilizei `pthread_cond_timedwait()` para que a thread de envio pudesse dormir até receber um sinal da thread de recepção ou até o tempo expirar. Isso evitou o uso de *busy waiting* (loops vazios) que desperdiçariam CPU.

2.3 3. Algoritmo de Dijkstra e Lógica de Grafo

- **Seleção de Drones:** A lógica não era só "o nó mais próximo", mas "a capital mais próxima que possui equipe disponível". Por isso, tive de adaptar a busca no vetor de distâncias do Dijkstra para filtrar apenas os nós que são do tipo `CAPITAL` (tipo 1) e verificar uma flag de ocupação no servidor antes de despachar.

3 Estratégias Adotadas

Para cumprir os requisitos de robustez e funcionalidade, adotei as seguintes abordagens:

- **Definição de Protocolo Binário (Packed Structs):** Criei um arquivo `common.h` contendo estruturas com o atributo `__attribute__((packed))`. Isso removeu o *padding* (alinhamento de memória) que o compilador insere automaticamente, garantindo que o tamanho da estrutura `payload_telemetria_t` fosse exatamente o mesmo no cliente e no servidor, independente da máquina.
- **Mecanismo de Reenvio (Stop-and-Wait ARQ):** Na thread de telemetria, implementei um loop de reenvio. Se o ACK não chegar em 5 segundos, o cliente reenvia o pacote, até um limite de 3 tentativas. Isso adiciona uma camada de confiabilidade sobre o UDP, essencial para sistemas de monitoramento crítico.
- **Arquitetura Orientada a Eventos no Servidor:** O servidor opera em um loop infinito de recepção. Ao receber um pacote, ele faz um *switch-case* no tipo da mensagem (HEADER). Isso permite que o servidor trate Telemetria, ACKs e Conclusões de forma assíncrona e intercalada, sem bloquear o processamento de outros drones.
- **Independência de Versão IP:** Assim como no TP1, utilizei `getaddrinfo` com `AF_UNSPEC` e `struct sockaddr_storage`. Isso permite que o código funcione transparentemente tanto com IPv4 (`./server v4`) quanto com IPv6 (`./server v6`), abstraindo a complexidade das estruturas de endereço subjacentes.

4 Possíveis Melhorias

Apesar de funcional, o sistema permite evoluções futuras:

- **Servidor Multi-thread:** Atualmente o servidor processa uma requisição por vez. Para escalar para milhares de sensores na Amazônia real, o servidor deveria despachar o processamento de pacotes pesados (como o cálculo de Dijkstra) para um *pool* de threads trabalhadoras.
- **Janela Deslizante:** O protocolo atual usa *Stop-and-Wait* (envia um, espera confirmação). Implementar uma janela deslizante aumentaria significativamente o *throughput* da telemetria, permitindo enviar múltiplos dados de cidades sem esperar cada ACK individualmente.

- **Segurança (DTLS):** Como os dados trafegam em texto plano (ou binário sem criptografia), um atacante poderia injetar falsos alertas de desmatamento. A implementação de DTLS (Datagram Transport Layer Security) seria necessária para um ambiente de produção.

5 Ambiente de desenvolvimento

Para desenvolver e testar o sistema, foi utilizado:

- Ubuntu 22.04.3 LTS (WSL2 e PC nativo)
- Linguagem C (Padrão C99/GNU99)
- Compilador GCC versão 11.4.0
- Ferramentas de depuração: GDB e Valgrind (para verificação de vazamento de memória no carregamento do grafo).

6 Instruções de Compilação e Execução

O projeto utiliza um `Makefile` configurado para gerar os binários na raiz do projeto, conforme solicitado.

- **Compilação:** No diretório raiz do projeto, execute:

  ```
  make
  ```
  (1)

  Isso gerará os executáveis `server` e `client`.

- **Executando o Servidor:** O servidor deve ser iniciado primeiro, especificando a versão do IP:

  ```
  ./server v4   (ou ./server v6)
  ```

- **Executando o Cliente:** Em outro terminal, inicie o cliente. Opcionalmente, pode-se passar o IP do servidor (padrão é localhost):

  ```
  ./client v4   (ou ./client v6 ::1)
  ```

7 Conclusão

O TP2 permitiu consolidar os conceitos de comunicação sem conexão e programação concorrente. A implementação do algoritmo de Dijkstra sobre a topologia da Amazônia Legal demonstrou a aplicação prática de Grafos em redes de computadores. A maior lição aprendida foi a importância da definição rigorosa do protocolo (tamanho de estruturas e ordem de bytes) para garantir que sistemas heterogêneos possam "conversar"sem erros de interpretação, um pilar fundamental da Engenharia de Redes.