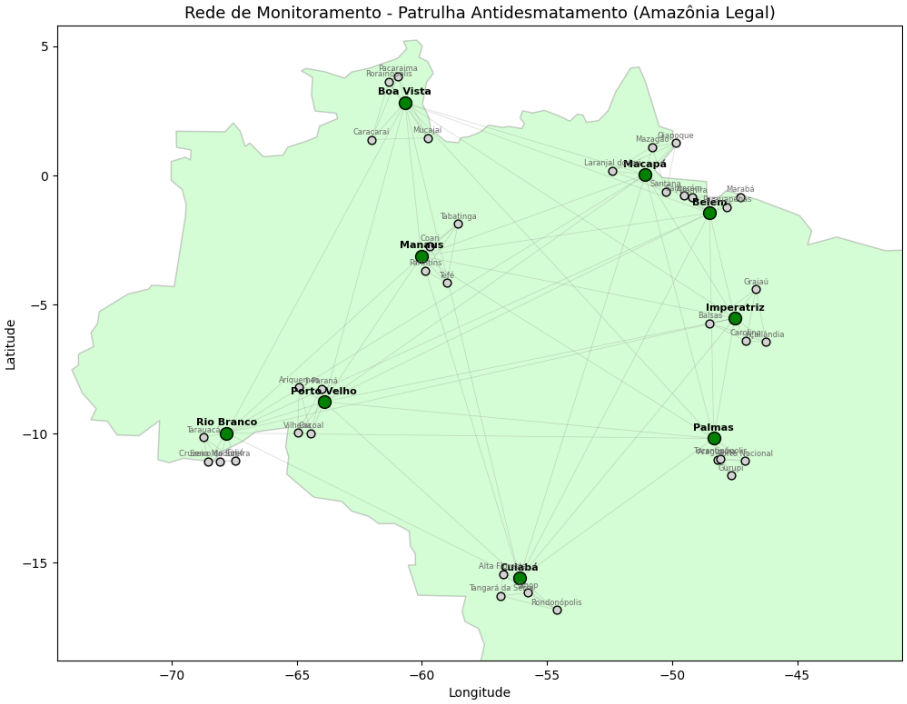
Trabalho Prático 2 — Patrulha Antidesmatamento (UDP + Threads em C)



Disciplina de Redes de Computadores - 2025/2
Universidade Federal de Minas Gerais

1. Introdução

Após a *Conferência do Clima das Nações Unidas (COP30)*, o governo brasileiro lançou um programa de alta prioridade: a **Patrulha Antidesmatamento da Amazônia Legal**. Você, engenheiro de software especializado em redes, foi convocado para implementar um sistema de comunicação distribuído entre sensores ambientais e drones de inspeção. O sistema deverá ser totalmente implementado em **C**, utilizando **sockets UDP** e **threads POSIX** e permitindo IPv4 e IPv6.

O objetivo é monitorar 45 regiões distribuídas na Amazônia Legal (5 por estado), detectando focos de desmatamento e despachando drones automaticamente. A simulação deve representar a troca de mensagens entre sensores, servidor e drones.

2. Objetivos do Trabalho

- Implementar um sistema distribuído baseado em UDP e utilizando threads.
- Modelar um grafo representando a Amazônia Legal
  - Vértices são cidades
  - Arestas são as rotas em linha reta entre duas cidades
  - O peso das arestas é a distância em Km entre as duas cidades
- Utilizar o algoritmo de Dijkstra para determinar o drone mais próximo.
- Trabalhar com estruturas de dados compartilhadas e sincronização via mutex.
- Demonstrar a integração entre rede, concorrência e estruturas de dados.

3. Descrição Geral do Sistema

O sistema é composto por dois binários principais:

1. **Servidor:** mantém o grafo, processa mensagens, detecta alertas e despacha drones.
2. **Cliente:** irá implementar 4 threads:
   - Thread 1 e 2: envia leituras simuladas dos sensores
   - Thread 3 e 4: equipes de drones que aguardam ordens para atuar nos alertas

3.1. Funcionamento básico

O cliente envia monitoramentos a cada 30 segundos com os dados de telemetria das cidades que ele monitora. O servidor processa estas mensagens. Caso alguma região esteja com índice de desmatamento acima do limite, o servidor computa a equipe de drones mais próxima à região em alerta. Em seguida, o servidor notifica o cliente com qual equipe de drones deverá se deslocar até o local de alerta. O cliente por sua vez, simula o trabalho de atender o chamado de alerta por 30 segundos.

A seguir detalharemos o funcionamento de cada um destes sistemas.

4. Protocolo de Comunicação UDP

A comunicação entre o cliente e o servidor segue um **protocolo binário**, desenvolvido para permitir diferentes tipos de mensagens dentro do mesmo socket UDP. Cada datagrama transmitido possui um cabeçalho comum, seguido de um payload específico para o tipo de mensagem.

4.1. Estrutura Geral das Mensagens

Todas as mensagens enviadas entre cliente e servidor compartilham o mesmo cabeçalho, definido pela estrutura a seguir:

```
typedef struct {
    // Tipo da mensagem
    // (1=telemetria, 2=ACK, 3=equipe de drones, 4=conclusão)
    uint16_t tipo;

    // tamanho do payload em bytes
    uint16_t tamanho;
} header_t;
```

O campo `tipo` informa ao destinatário como deve interpretar o conteúdo do payload. O campo `tamanho` facilita a leitura exata dos bytes subsequentes.

4.2. Tipos de Mensagem

O protocolo define quatro tipos principais de mensagem, representando todas as interações previstas entre cliente e servidor:

Tipo	Código	Descrição
`MSG_TELEMETRIA`	1	Cliente envia estado (OK/ALERTA) de todas as cidades
`MSG_ACK`	2	Servidor ou Cliente confirma recebimento de mensagem
`MSG_EQUIPE_DRONE`	3	Servidor designa qual equipe de drones atuará
`MSG_CONCLUSAO`	4	Cliente informa que a equipe de drones concluiu a missão

4.3. Estruturas de Payload

Cada tipo de mensagem possui um payload específico associado ao seu propósito:

1. Cliente → Servidor : Envio de telemetria (a cada 30s)

```
typedef struct {
    int total;                  // número de cidades monitoradas
    telemetria_t dados[50];     // lista de (id_cidade, status)
} payload_telemetria_t;

typedef struct {
    int id_cidade;   // identificador do vértice
    int status;      // 0 = OK, 1 = ALERTA
} telemetria_t;
```

2. Servidor → Cliente ou Cliente → Servidor: Confirmação de recebimento

```
typedef struct {
    int status; // 0=ACK_TELEMETRIA, 1=ACK_EQUIPE_DRONE, 2=ACK_CONCLUSAO
} payload_ack_t;
```

3. Servidor → Cliente : Ordem de envio de drone

```
typedef struct {
    int id_cidade;      // cidade onde o alerta foi detectado
    int id_equipe;      // equipe de drones designada
} payload_equipe_drone_t;
```

4. Cliente → Servidor : Conclusão de missão

```c
typedef struct {
    int id_cidade;      // cidade atendida
    int id_equipe;      // equipe que atuou
} payload_conclusao_t;
```

4.4. Fluxo de Comunicação

O ciclo completo de mensagens entre cliente e servidor é o seguinte:

1. **Cliente → Servidor:** envia uma mensagem `MSG_TELEMETRIA` contendo o estado (OK ou ALERTA) de todas as cidades monitoradas.
   - **Servidor → Cliente:** responde com uma mensagem `MSG_ACK` confirmando o recebimento da telemetria.
2. **Servidor → Cliente:** caso alguma cidade esteja em alerta, envia uma mensagem `MSG_EQUIPE_DRONE` informando qual equipe de drones deve atuar na região.
   - **Cliente → Servidor:** responde com uma mensagem `MSG_ACK` confirmando o recebimento da mensagem de equipes de drones.
3. **Cliente → Servidor:** após a execução da missão, envia uma mensagem `MSG_CONCLUSAO` notificando que o drone finalizou a atuação na cidade alertada.
   - **Servidor → Cliente:** responde com uma mensagem `MSG_ACK` confirmando o recebimento da conclusão da missão.

5. Implementação do Servidor

O servidor é o componente central. Ele deve:

- Carregar o grafo da Amazônia Legal e mantê-lo em memória.
  - você irá processar o arquivo $grafo_amazonia_legal.txt(formatdescritoàfrente)$
- Receber telemetria dos sensores enviados pelo cliente via UDP.
- Criar alertas, notificar o cliente para atuação dos drones e aguardar resolução.
- Escolher o drone mais próximo livre usando o algoritmo de Dijkstra.
- O servidor deve rodar na porta 8080 e aceitar comunicações UDP.

5.1. Leitura do Arquivo `grafo`$_amazonia_legal.txt$

O arquivo `grafo_amazonia_legal.txt` contém a representação do grafo utilizado pelo servidor para modelar as cidades da Amazônia Legal e as rotas disponíveis entre elas. Cada vértice corresponde a uma cidade e cada aresta representa uma rota de voo entre duas regiões. O grafo é não direcionado e todas as arestas tem pesos positivos.

O formato do arquivo é o seguinte:

- **Linha 1**: dois inteiros N e M, separados por espaço, indicando respectivamente o número total de vértices (cidades) e o número total de arestas (rotas).
- **Linhas 2 até** $N + 1$: cada linha descreve um vértice no formato:

  $$i \quad c \quad t$$

  onde:
  - i é o índice do vértice (inteiro);
  - c é o nome da cidade (string);
  - t é o tipo de cidade (0 para regional, 1 para capital).
- **Linhas seguintes (**M **linhas)**: cada linha descreve uma aresta no formato:

  $$u \quad v \quad p$$

  onde:
  - u e v são os índices dos vértices conectados pela rota;
  - p é o peso da aresta, representando a distância entre as cidades (em quilômetros).

Exemplo de arquivo:

```
45 122
0 RioBranco 1
1 CruzeiroDoSul 0
2 SenaMadureira 0
...
15 Manaus 1
16 Parintins 0
...
0 1 720
0 2 310
1 3 450
...
```

Durante a inicialização, o servidor deve abrir o arquivo, ler os valores e construir as estruturas de dados correspondentes em memória. Você é livre para implementar o grafo como achar melhor. Utilize suas estruturas de dados e algoritmos de preferência.

5.2. Recebimento da Telemetria

O servidor deve criar um `socket UDP` e mantê-lo em modo de escuta. Cada vez que um pacote do tipo `MSG_TELEMETRIA` for recebido, o servidor deve:

1. Ler o cabeçalho e confirmar o tipo da mensagem.
2. Interpretar o payload como um `payload_telemetria_t`.
3. Atualizar o status de cada cidade conforme os dados enviados.
4. Enviar imediatamente uma resposta `MSG_ACK` para o cliente confirmando o recebimento.

O servidor não precisa ser multi-thread.

5.3. Criação de alertas

Após receber e processar a telemetria, o servidor deve verificar se há cidades cujo campo `status == 1`. Para cada cidade em alerta, ele deve:

1. Registrar um novo evento de alerta em uma estrutura de dados à sua escolha contendo o identificador da cidade, o timestamp e se já existe equipe atuando.
    - A princípio, não existe equipe atuando.
    - Isto será confirmado ao receber um ACK do cliente quando servidor manda uma MSG_EQUIPE_DRONE.
2. Em seguida o servidor computa qual equipe de drones deve (processo descrito à seguir).
3. Enviar uma mensagem `MSG_EQUIPE_DRONE` ao cliente, informando qual equipe de drones foi designada para a missão.

5.4. Escolha de Equipe de Drones

Para escolher a equipe de drones mais próxima de uma cidade em alerta, o servidor deve:

1. Executar o algoritmo de Dijkstra no grafo, tendo como origem a cidade alertada.
2. As equipes de Drones estão SOMENTE nas capitais de cada estado. Ou seja, só existem 9 equipes de drones na Amazônia Legal.
3. Selecione a equipe livre com menor distância até a cidade alertada.

Uma vez selecionada a equipe, o servidor deve enviar uma mensagem `MSG_EQUIPE_DRONE` ao cliente, informando:

- o identificador da cidade em alerta;
- o número da equipe de drones designada. Considere que a equipe de Drone tem o mesmo ID da capital à qual ele pertence.

Enquanto a equipe estiver atuando, o servidor deve marcá-la como "ocupada". Somente quando o cliente enviar uma mensagem `MSG_CONCLUSAO` essa equipe volta ao estado "disponível".

6. Implementação do Cliente

O cliente faz as seguintes tarefas:

- Carregar a lista de cidades da Amazônia Legal e mantê-lo em memória.
    - você irá processar o arquivo $grafo_amazonia_legal.txt(nãoprecisalerasarestas)$
- Criar 4 threads
    - Thread 1: Simular o monitoramento das regiões

  - Thread 2: Enviar telemetria a cada 30 segundos
  - Thread 3: Receber mensagem de acionamento de equipe de drones
  - Thread 4: Simular 30 segundos de atuação da equipe selecionada.

- As threads 2 e 3 são as responsáveis pela comunicação com o servidor enquanto 1 e 4 fazem tarefas locais.
- O cliente precisa conectar no servidor na porta 8080

6.1. Simular monitoramento

A thread 1 do cliente é responsável por simular a coleta de dados das cidades. Deve-se gerar valores aleatórios para o campo `status` de cada cidade:

- 0 = normal
- 1 = alerta

Você deve garantir que a probabilidade de alertas seja de 3% por cidade. Ou seja, apenas 3% dos monitoramentos resultem em um alerta. Esse percentual foi escolhido de modo que não sejam gerados muitos alertas em cada iteração.

Esses dados são armazenados em memória compartilhada (ex.: vetor global protegido por mutex), para serem utilizados pela thread que envia a telemetria ao servidor que iremos detalhar à diante.

6.2. Enviar telemetria periodicamente

A thread 2 do cliente é responsável por enviar, a cada 30 segundos, uma mensagem `MSG_TELEMETRIA` contendo o estado atual de todas as cidades monitoradas.

1. Montar a estrutura `payload_telemetria_t` com o número total de cidades.
2. Copiar os valores de `id_cidade` e `status` para o vetor `dados`.
3. Enviar o pacote completo ao servidor.
4. Aguardar o recebimento do `MSG_ACK` antes de continuar o ciclo.

Caso o servidor não responda em um tempo limite (ex.: 5 segundos), o cliente pode reenviar o pacote anterior. Você pode tentar reenviar no máximo 3 vezes. Caso contrário, deixe o envio acontecer no próximo ciclo de 30 segundos.

6.3. Acionamento de equipes de drones

A threads 3 fica responsáveis por aguardar mensagens do tipo `MSG_EQUIPE_DRONE` enviadas pelo servidor. Cada vez que uma mensagem é recebida:

1. O cliente lê o identificador da cidade e o número da equipe designada.
2. Envia um `MSG_ACK` confirmando o recebimento da ordem.
3. Informa a thread de simulação

- via variável compartilhada ou fila para iniciar a missão. Você escolhe.

4. Quando a thread de simulação de drones te notificar/informar de volta que a missão foi concluída, notifique o servidor:
   - Construa uma mensagem `MSG_CONCLUSAO` ao servidor, contendo:
     - o identificador da cidade atendida;
     - o número da equipe que realizou a missão.
5. Aguarde uma mensagem `MSG_ACK` do servidor para confirmar o encerramento da operação e liberar a equipe para novas missões.

6.4. Simular ação dos drones

A thread 4 deve simular a execução da missão durante um intervalo aleatório de até 30 segundos. Entenda simular como um simples *sleep()*. Ao término da simulação, a thread notifica a thread 3 (que comunica com o servidor) que a missão foi concluída.

- Nesta etapa você irá utilizar sua estrutura de dados compartilhada entre as threads 3 e 4
- Como esta escolha será feita por você, é sua responsabilidade garantir que estas threads possam compartilhar estado.
- Se a thread que simula os drones estiver ocupada, você pode enviar um MSG_ACK para o servidor e ignorar aquele alerta (para simplificar o problema)

7. Saídas no terminal

Você não precisa seguir esses formatos estritamente. Nos exemplos a seguir, as palavras não estão acentuadas pois o LaTeX está com erro. Recomenda-se fazer as saídas no terminal devidamente acentuadas. A seguir vamos ver os exemplos de saída no terminal do servidor e do cliente:

Servidor

```
Servidor escutando na porta 8080...

[TELEMETRIA RECEBIDA]
Total de cidades monitoradas: 45
 ALERTA: Laranjal do Jari (ID=12)
 ALERTA: Belem (ID=25)
-> ACK enviado (tipo=0)

[DESPACHANDO DRONES]
Cidade em alerta: Laranjal do Jari (ID=12)
  -> Dijkstra: capital Macapa (ID=10) selecionada, distancia=108 km
-> Ordem enviada: Equipe Macapa (ID=10) -> Cidade Laranjal do Jari (ID=12)

[DESPACHANDO DRONES]
Cidade em alerta: Belem (ID=25)
  -> Dijkstra: capital Belem (ID=25) selecionada, distancia=0 km
-> Ordem enviada: Equipe Belem (ID=25) -> Cidade Belem (ID=25)

```

```
[ACK RECEBIDO]
Cliente confirmou recebimento de ordem de drone para Laranjal

[ACK RECEBIDO]
Cliente confirmou recebimento de ordem de drone para Belem

[MISSAO CONCLUDA]
Cidade atendida: Laranjal do Jari (ID=12)
Equipe: Macapa (ID=10)
-> Equipe Macapa liberada para novas missoes
-> ACK enviado (tipo=2)
```

Cliente

```
Conectado ao servidor 127.0.0.1:8080

Iniciando threads...

[Thread Monitoramento] Iniciada
[Thread Simulacao Drones] Iniciada
[Thread Telemetria] Iniciada
[Thread Recepcao Drones] Iniciada
. Todas as threads iniciadas com sucesso
Pressione Ctrl+C para encerrar...

[ENVIANDO TELEMETRIA]
Total de cidades: 45
 ALERTA: Laranjal do Jari (ID=12)
 ALERTA: Belem (ID=25)
-> Telemetria enviada (tentativa 1/3)
. ACK recebido do servidor

[ORDEM DE DRONE RECEBIDA]
Cidade: Laranjal do Jari (ID=12)
Equipe: Macapa (ID=10)
-> ACK enviado ao servidor
-> Missao registrada para execucao

[ORDEM DE DRONE RECEBIDA]
Cidade: Belem (ID=25)
Equipe: Belem (ID=25)
-> ACK enviado ao servidor
 Ja existe missao ativa, ordem ignorada

[MISSAO EM ANDAMENTO]
Equipe Macapa atuando em Laranjal do Jari
. Tempo estimado: 17 segundos
. Missao concluida!
-> Conclusao enviada ao servidor
```

8. Execução dos binários

Servidor

```
$> ./server v4
$> ./server v6
```

Cliente

```
$> ./client v4
$> ./client v6
```

9. Avaliação

O seu cliente será testado contra o meu servidor. E o seu servidor será testado com os meu cliente. Entenda que as saídas no terminal, assim como no TP 1, são recomendações. Na prática, os testes automatizados estão interessados no conteúdo das mensagens trocadas entre clientes e servidor. Ou seja, as estruturas de dados e seu conteúdo de acordo com cada tipo de mensagem.

Além disso, irei: ler, compilar e validar a execução do seu código visualmente. Portanto, o mais próximo que o seu código for das sugestões de saída no terminal, mais direta será esta etapa de avaliação.

Seja bem descritivo em sua documentação. Através dela, eu gostaria de entender o seu processo decisório para implementar o TP. Seja claro quanto à como e porquê você decidiu fazer alguma etapa do seu código.

9.1. Critérios de avaliação

- Comunicação UDP funcional (20%)
- Threads e sincronização (15%)
- Funcionamento global do sistema (15%)
- Algoritmo de seleção de drones (20%)
- Organização e clareza (10%)
- Robustez e documentação (20%)

10. Submissão

- Entrega em formato Tar gz: `TP2_MATRICULA.tar.gz`.
- A documentação somente será aceita se acompanhada de código fonte.
- Conteúdo do arquivo comprimido:
  - `server.c`
  - `client.c`
  - `grafo_amazonia_legal.txt`

  - `Makefile`
  - Você pode criar quantos outros arquivos de código você quiser além dessa estrutura básica
  - `documentacao.pdf`

- O seu Makefile deve gerar os binários (client, server) na raiz do projeto (diferente do TP 1 que utilizamos uma pasta bin).
- Após a data estabelecida para entrega no Moodle não haverá possíbilidade de entrega.

11. Detecção de Plágio

- O uso de código gerado por ferramentas de IA é proibido.
- Ferramentas automáticas de detecção serão utilizadas.
- Plágio resultará em nota zero.

12. Material complementar

- Playlist de programação com sockets do professor Ítalo Cunha
- Beej's Guide to Network Programming