Roteiro de Estudos: Roteamento de Rotas em C

Módulo 1: Estrutura da Tabela de Rotas

1. **Objetivo do Módulo:** Modelar a estrutura de dados central do roteamento: a **Tabela de Rotas** (Routing Table) em C.

Exercício Prático (Progressivo):

- **Foco:** Definição de `struct` e manipulação de endereços IP de 32 bits.

Tarefa: Crie uma `struct` chamada `Rota` que deve conter:

- `destino` (o endereço da rede de destino, ex: $192.168.1.0$).
- `mascara` (a máscara de sub-rede do destino, ex: $255.255.255.0$).
- `gateway` ou `proximo_salto` (o endereço IP do próximo roteador/máquina, se houver).
- `interface` (um identificador ou string para a interface de saída, ex: "eth0").

Crie uma função `adicionar_rota` que insira várias instâncias dessa `struct` em um array (ou lista, para ser mais avançado) que representará a Tabela de Rotas.

Provocação de Raciocínio (Crucial):

- Por que é crucial armazenar o **destino** e a **máscara** de forma separada? O que o valor $ ext{destino} ext{ AND } ext{máscara}$ realmente representa nessa tabela?
- Qual é o caso especial que você deve incluir na tabela que representa a **Rota Padrão** ($0.0.0.0/0$)? Como os valores $ ext{destino}$ e $ ext{máscara}$ são preenchidos para essa rota?

3. **Ponto de Consolidação:** Se você consegue armazenar rotas específicas (ex: $172.16.1.0/24$) e a rota padrão na mesma estrutura de dados, e entende por que $0.0.0.0$ com máscara $0.0.0.0$ define o *default*, você está pronto para a lógica.

Módulo 2: O Algoritmo de Lookup (Best Match)

1. **Objetivo do Módulo:** Implementar o algoritmo que um roteador usa para escolher a **melhor rota** para um pacote de destino.

Exercício Prático (Progressivo):

- **Foco:** A lógica de **Maior Correspondência de Prefixo** (Longest Prefix Match - LPM).

Tarefa: Crie a função central do roteamento: `const struct Rota* buscar_rota(unsigned int ip_destino, const struct Rota tabela[])`.

1. A função deve iterar sobre cada rota na tabela.
2. Para cada rota, ela deve aplicar o **AND bitwise** do `ip_destino` com a `máscara` da rota, e comparar o resultado com o `destino` da rota.
3. Se houver uma correspondência, armazene essa rota.
4. A melhor rota (o *match*) é aquela que tem a **maior máscara** (mais bits de rede).

Provocação de Raciocínio (Crucial):

- Se o destino do pacote for $192.168.1.10$ e a tabela tiver rotas para $192.168.1.0/24$ e $192.168.0.0/16$, por que a rota $/24$ deve ser a escolhida, mesmo que a rota $/16$ também corresponda?
- O que acontece se duas rotas tiverem o mesmo prefixo de máscara (ex: duas rotas $/24$) e ambas corresponderem ao IP de destino? Como o roteador (ou sua implementação) deve lidar com essa ambiguidade?

3. **Ponto de Consolidação:** Se você consegue buscar o IP de destino $10.1.1.5$ em uma tabela que contém rotas $/8$, $/16$ e $/24$ (e $/32$), e sempre seleciona a rota com o prefixo mais longo que *inclui* o destino, você dominou o LPM.

Módulo 3: Fluxo de Pacotes e Decisão de Entrega

1. **Objetivo do Módulo:** Simular o fluxo do pacote e a decisão final do roteador: entregar **direto** (na rede local) ou enviar para o **próximo salto** (Gateway).

Exercício Prático (Progressivo):

- **Foco:** Lógica de interface e manipulação do TTL (Time-to-Live).

Tarefa: Crie uma função `void processar_pacote(unsigned int ip_origem, unsigned int ip_destino, int ttl, const struct Rota tabela[], const struct Interface interfaces[])` que simula um roteador:

1. Verifique se o `ip_destino` é o IP de uma das suas próprias interfaces ("packet not for me" - ou, neste caso, "packet *is* for me").
2. Execute o `buscar_rota` (Módulo 2) para encontrar a melhor rota.
3. Se a rota for uma rede local (sem `gateway`), o pacote é entregue diretamente na interface.
4. Se a rota tem um `gateway`, o pacote é encapsulado e enviado ao `gateway` via a `interface`.
5. Se o TTL (simulado como um inteiro) for zero, "descarte" o pacote. Se não, decremente-o.

Provocação de Raciocínio (Crucial):

- Ao receber um pacote, por que o roteador deve **sempre** decrementar o TTL *antes* de decidir para onde enviá-lo? Qual problema na rede esse campo previne?
- Em um roteador, quando o `buscar_rota` retorna uma rota que aponta para um `gateway` (próximo salto), o roteador precisa executar outro `buscar_rota` para descobrir como alcançar esse próprio `gateway`? Ou o `gateway` sempre deve estar em uma de suas redes diretamente conectadas?

3. **Ponto de Consolidação:** Se você consegue simular o Roteador A recebendo um pacote para $10.1.1.100$, consultando sua tabela, decrementando o TTL e decidindo que o próximo passo é enviar o pacote para o Roteador B ($192.168.10.1$), você dominou a lógica do encaminhamento (*forwarding*).

Módulo 4: Interpretação de Logs de Roteamento (NetPractice)

1. **Objetivo do Módulo:** Correlacionar a lógica implementada em C (lookup e decisão) com os logs de depuração do NetPractice (e sistemas Unix).

Exercício Prático (Progressivo):

- **Foco:** Mapeamento de condições de erro.

Tarefa: Crie um *módulo de logging* dentro do seu programa C. Em vez de apenas executar a lógica, faça a função `processar_pacote` (Módulo 3) imprimir mensagens que simulam os logs do NetPractice/Kernel Linux:

- Quando o pacote é descartado por TTL zero: Imprimir "Log: Packet discarded. TTL expired."
- Quando o `buscar_rota` não encontra *nenhuma* correspondência (e não há rota padrão): Imprimir "Log: Destination network unreachable."
- Quando o pacote é para uma rede remota e encaminhado: Imprimir "Log: Forwarding packet to next hop via [interface]."
- **Análise de Log:** Pegue um log de erro do NetPractice (como "reverse way is not possible") e determine **exatamente** qual rota faltaria na **máquina de destino** para que a resposta não pudesse voltar.

Provocação de Raciocínio (Crucial):

- Por que o log "forward way" ser bem-sucedido e o "reverse way" falhar significa quase sempre que o problema **não** está na sua configuração atual, mas sim na configuração de **retorno** de outro dispositivo?
- O que o sistema faz se o `buscar_rota` falhar e não houver uma **Rota Padrão**? Como o kernel comunica essa falha ao remetente original (dica: *ICMP*).

3. **Ponto de Consolidação:** Se, ao ver um log de falha de comunicação, você consegue apontar qual `struct Rota` está faltando ou está incorreta (destino/máscara/gateway) em **qual** dispositivo (remetente ou receptor), você está pronto para o projeto.

Módulo 5: Abstração de Rede de Baixo Nível

1. **Objetivo do Módulo:** Conectar a lógica de roteamento (Módulos 1-4) com a realidade do sistema operacional, entendendo como o kernel vê e armazena os endereços IP.

Exercício Prático (Progressivo):

- **Foco:** Funções de conversão de ordem de bytes (`ntohl`, `htonl`) e as *estruturas* de endereço de rede.

- **Tarefa:** Crie um pequeno programa que converta um endereço IP de **notação pontuada** (string, ex: `"192.168.1.1"`) para o formato de **inteiro de 32 bits** (usando `inet_addr` ou `inet_pton`). Em seguida, use a função de sua escolha (`ntohl` ou `htonl`) no resultado e imprima o IP nos dois formatos (o *host order* e o *network order*).

Provocação de Raciocínio (Crucial):

- Se você estiver executando seu programa em um processador Intel (Little Endian) e receber um pacote de rede (Network Byte Order, que é Big Endian), o que acontece se você tentar usar o inteiro de 32 bits **diretamente** para sua lógica de **AND bitwise** sem usar as funções de conversão? Qual número você calculará incorretamente?

- Qual é o propósito da `struct sockaddr_in` e como ela se relaciona com o `unsigned int` que você usou até agora para representar o endereço IP?

3. **Ponto de Consolidação:** Se você consegue converter corretamente $1.2.3.4$ (decimal pontuado) para seu valor binário/inteiro de 32 bits, e entende por que ele pode estar invertido na memória do seu PC (Endianness), você garantiu a base de baixo nível necessária para a manipulação correta dos IPs.