

# **Rede Local em Anel**

Arthur Santos, Gabriel Moura e Lucas Ramon

98709-04 - Fundamentos de Redes de Computadores - Turma 030

Relatório

## 1. Introdução

Este documento detalha a concepção, arquitetura e validação de um software desenvolvido para simular o funcionamento de uma rede local em anel com passagem de token. O objetivo principal do projeto foi implementar uma aplicação robusta capaz de gerenciar a comunicação entre múltiplos nós, utilizando o protocolo UDP como meio de transporte.

A solução implementada abrange funcionalidades essenciais de redes de computadores, como:

- Um protocolo de acesso ao meio controlado por token.
- Uma estrutura de pacotes para dados e controle.
- Mecanismos para controle de erro com CRC32.
- Um sistema de confirmação e retransmissão de mensagens (ACK/NAK).

## 2. Concepção e Estratégia de Desenvolvimento

A elaboração deste projeto foi guiada por princípios fundamentais de engenharia de software, visando criar uma solução que não fosse apenas funcional, mas também modular, robusta, e de fácil compreensão e manutenção. A estratégia de desenvolvimento pode ser decomposta nos seguintes pilares:

### 2.1. Modularidade e Separação de Responsabilidades (SoC)

Desde o início, a principal decisão arquitetônica foi evitar um único arquivo monolítico. Em vez disso, o sistema foi decomposto em componentes lógicos, cada um com uma responsabilidade única e bem definida. Esta abordagem oferece diversas vantagens, como legibilidade, facilidade de depuração e manutenibilidade. Se, por exemplo, o formato dos pacotes precisasse ser alterado no futuro, apenas a classe `Packet.py` necessitaria de modificação, sem impactar a lógica central do nó.

A divisão foi a seguinte:

- **RingNode.py:** O "cérebro" da aplicação, responsável exclusivamente pela lógica de estado do protocolo Token Ring (reter token, passar token, processar pacotes, etc.).
- **Packet.py:** Uma "fábrica" de pacotes, que abstrai toda a complexidade da formatação e da interpretação das strings de dados enviadas via UDP. O `RingNode` apenas manipula dicionários Python, delegando a serialização/desserialização para esta classe.
- **MessageQueue.py:** Isola o gerenciamento da fila de mensagens pendentes. Isso simplifica o `RingNode`, que apenas precisa interagir com métodos claros como `enqueue`, `peek` e `dequeue`.
- **CRC32.py e ErrorInserter.py:** O controle de erro e a simulação de erro foram encapsulados em seus próprios módulos. Isso não apenas organiza o código, mas também permite que esses mecanismos sejam testados ou até substituídos de forma independente.

### 3. Análise dos Testes de Validação

Aqui apresentaremos a análise dos testes de validação realizados sobre a implementação do software. O objetivo foi verificar a conformidade do sistema com todos os requisitos especificados no documento disponibilizado pela professora. Os testes, executados em um ambiente de três nós (Alice, Bob e Charlie), demonstraram que, após um ciclo de depuração, a aplicação atende totalmente aos requisitos funcionais, incluindo a circulação de token, comunicação unicast e broadcast, controle de erros com CRC32, e tratamento de falhas (ACK/NAK/máquina inexistente).

#### **Ambiente e Metodologia de Teste**

Os testes foram conduzidos em uma topologia de anel com três nós, configurados da seguinte forma:

**Nó 1 (Alice):** Porta 6001, vizinho 127.0.0.1:6002 (Bob), gerador inicial do token.

**Nó 2 (Bob):** Porta 6002, vizinho 127.0.0.1:6000 (Charlie).

**Nó 3 (Charlie):** Porta 6000, vizinho 127.0.0.1:6001 (Alice).

A metodologia consistiu em executar uma série de casos de teste, cada um projetado para validar uma funcionalidade específica do enunciado. A análise foi realizada através da inspeção cruzada dos logs gerados por cada nó.

#### 3.1. Teste de Circulação de Token Ocioso

**Objetivo:** Validar a geração e circulação contínua do token na rede, conforme especificado.

**Execução:** Os nós foram iniciados e mantidos em estado ocioso.

**Resultados Observados:** Às 17:52:54, o nó Alice gerou o token inicial e o enviou para Bob. Os logs subsequentes mostram o token circulando corretamente pela rede (Alice -> Bob -> Charlie -> Alice) com um período de aproximadamente 4 segundos, sem que o monitor de perda de token fosse acionado.

**Veredito:** Conformidade Atingida.

```
[17:52:54] 🔧 [Alice] Gerando token inicial...
[17:52:54] 📦 [Alice] Enviou TOKEN para ('127.0.0.1', 6002)
[17:52:54] 🚛 Token agora em trânsito para ('127.0.0.1', 6002)
```

```
[17:52:54] [Bob] TOKEN chegou de ('127.0.0.1', 6001) – Agora em Bob
[17:52:56] [Bob] Enviou TOKEN para ('127.0.0.1', 6000)
[17:52:56] Token agora em trânsito para ('127.0.0.1', 6000)
```

```
[17:52:56] [Charlie] TOKEN chegou de ('127.0.0.1', 6002) – Agora em Charlie
[17:52:58] [Charlie] Enviou TOKEN para ('127.0.0.1', 6001)
[17:52:58] Token agora em trânsito para ('127.0.0.1', 6001)
```

### 3.2. Teste de Comunicação Unicast com Sucesso (ACK)

**Objetivo:** Verificar o fluxo completo de uma transmissão bem-sucedida: envio, recepção, cálculo de CRC, retorno com ACK e liberação do token.

**Execução:** O nó Alice enviou uma mensagem para o nó Bob.

#### Resultados Observados:

1. [17:53:17] Alice envia o pacote para Bob.

```
[17:53:16] [Alice] Token retornou em 4.02s (esperado mínimo: 4.5s)
[17:53:17] [Alice] Enviando para Bob (tentativa 1) via ('127.0.0.1', 6002)
[17:53:17] Pacote agora em trânsito para ('127.0.0.1', 6002)
```

2. [17:53:17] Bob recebe o pacote, confirma que o CRC está correto e envia o pacote de volta com o status alterado para ACK.

```
[17:53:14] Token agora em trânsito para ('127.0.0.1', 6000)
[17:53:17] [Bob] Pacote de dados recebido de ('127.0.0.1', 6001) (origem: Alice, destino: Bob, status: maquinanaoexiste)
[17:53:17] [Bob] CRC OK. Mensagem de Alice: "Teste de ACK bem-sucedido". Enviando ACK.
[17:53:18] [Bob] TOKEN chegou de ('127.0.0.1', 6001) – Agora em Bob
```

3. [17:53:18] Alice recebe o pacote de retorno com ACK, remove a mensagem da fila e, em seguida, libera o token para o próximo nó. Isso confirma a regra de que o token é retido até o fim da transação.

```
[17:53:18] Token agora em trânsito para ('127.0.0.1', 6002)
[17:53:18] [Alice] Pacote de dados recebido de ('127.0.0.1', 6000) (origem: Alice, destino: Bob, status: ACK)
[17:53:18] [Alice] Mensagem para Bob entregada com ACK. Removendo da fila.
[17:53:22] [Alice] TOKEN chegou de ('127.0.0.1', 6000) – Agora em Alice
```

### 3.3. Teste de Detecção de Erro e Retransmissão (NAK)

**Objetivo:** Validar a detecção de erro via CRC, o envio da resposta NAK, e a retransmissão única do pacote pela origem.

**Execução:** O nó Bob enviou uma mensagem para Charlie, e o módulo de inserção de falhas corrompeu o pacote.

#### Resultados Observados:

1. [17:53:36] O nó Charlie recebe um pacote corrompido de Bob, logo CRC falhou e envia um NAK de volta.

```
[17:53:36] [Charlie] Pacote de dados recebido de ('127.0.0.1', 6002) (origem: Bob, destino: Charlie, status: maquinanaoexiste)
[17:53:36] [Charlie] CRC falhou (origem=Bob). Enviando NAK.
```

2. [17:53:36] Bob recebe o NAK, registra a falha e prepara-se para a retransmissão.

```
[17:53:36] 🔴 Pacote agora em transito para ('127.0.0.1', 6000)
[17:53:36] [Bob] Pacote de dados recebido de ('127.0.0.1', 6001) (origem: Bob, destino: Charlie, status: NAK)
[17:53:36] [Bob] Falha (NAK) ao entregar para Charlie. Tentativa 1. Será retransmitida.
[17:53:36] 📧 [Bob] Enviando para Charlie (tentativa 2) via ('127.0.0.1', 6000)
[17:53:36] 📦 Pacote agora em transito para ('127.0.0.1', 6000)
```

3. [17:53:36] Na sua próxima posse do token, Bob envia o pacote novamente (a retransmissão única), que desta vez é recebido com sucesso por Charlie e confirmado com ACK.

```
[17:53:36] [Charlie] Pacote de dados recebido de ('127.0.0.1', 6002) (origem: Bob, destino: Charlie, status: maquinanaoexiste)
[17:53:36] [Charlie] CRC OK. Mensagem de Bob: "Teste de NAK e retransmissao". Enviando ACK.
[17:53:38] 🟢 [Charlie] TOKEN chegou de ('127.0.0.1', 6002) - Agora em Charlie
[17:53:36] 📦 Pacote agora em transito para ('127.0.0.1', 6000)
[17:53:36] [Bob] Pacote de dados recebido de ('127.0.0.1', 6001) (origem: Bob, destino: Charlie, status: ACK)
[17:53:36] [Bob] Mensagem para Charlie entregada com ACK. Removendo da fila.
```

### 3.4. Teste de Comunicação Broadcast (TODOS)

**Objetivo:** Validar o envio de uma mensagem para todas as máquinas da rede usando o apelido TODOS, a correta recepção por todos os nós e o tratamento adequado do pacote ao retornar para a origem.

**Execução:** O nó Charlie enviou uma mensagem de broadcast para o destino TODOS. Por coincidência, durante este teste, o módulo de inserção de falhas foi ativado, corrompendo a mensagem antes de ser enviada.

**Resultados Observados:** A análise cronológica dos logs mostra o comportamento robusto e correto do sistema em múltiplas frentes:

- **Envio (Charlie):** [17:54:02] Charlie envia o pacote de broadcast. O ErrorInserter altera a mensagem de "todos" para "toeos".

```
[17:54:02] 🔴 Token agora em transito para ('127.0.0.1', 6000).charlie.ACK.0.TESTE DE NAK E RETRANSMISSAO -> 5522710750
TODOS Mensagem de broadcast para todos
[MessageQueue] Mensagem enfileirada para TODOS
[CRC32] Calculado para dados: '2000;Charlie:TODOS:maquinanaoexiste:0:Mensagem de broadcast para todos' -> 1067159061
[ErrorInserter] Caractere na posição 29 alterado de 'd' para 'e'
6 [17:54:02] 🔴 [Charlie] Token retornou em 4.01s (esperado minímo: 4.5s)
7 [17:54:02] 📧 [Charlie] Enviando para TODOS (tentativa 1) via ('127.0.0.1', 6001)
8 [17:54:02] 📦 Pacote agora em transito para ('127.0.0.1', 6001)
```

- **Recepção e Detecção de Erro (Alice):** [17:54:02] Alice recebe o pacote, identifica que é um broadcast de Charlie e, crucialmente, seu módulo CRC detecta a corrupção, logando: [Alice] Broadcast de Charlie: "Mensagem de broadcast para toeos" (CRC FALHOU). O que eh um comportamento esperado.

```
[17:54:00] 🔴 Token agora em transito para ('127.0.0.1', 6002)
[17:54:02] [Alice] Pacote de dados recebido de ('127.0.0.1', 6000) (origem: Charlie, destino: TODOS, status: maquinanaoexiste)
[17:54:02] [Alice] Broadcast de Charlie: "Mensagem de broadcast para toeos" (CRC FALHOU)
[17:54:02] 🟢 [Alice] TOKEN chegou de ('127.0.0.1', 6002) - Agora em Alice
```

- **Recepção e Detecção de Erro (Bob):** [17:54:02] De forma similar, Bob recebe o pacote, também detecta o erro de CRC e loga: [Bob] Broadcast de Charlie: "Mensagem de broadcast para toeos" (CRC FALHOU).

```
[17:54:02] 🔴 Token agora em transito para ('127.0.0.1', 6000)
[17:54:02] [Bob] Pacote de dados recebido de ('127.0.0.1', 6001) (origem: Charlie, destino: TODOS, status: maquinanaoexiste)
[17:54:02] [Bob] Broadcast de Charlie: "Mensagem de broadcast para toeos" (CRC FALHOU)
[17:54:02] 🟢 [Bob] TOKEN chegou de ('127.0.0.1', 6001) - Agora em Bob
```

- **Retorno e Tratamento (Charlie):** [17:54:02] O pacote de broadcast, após passar por todos os nós, retorna ao seu remetente. O nó Charlie, com o código corrigido,

agora identifica corretamente a situação e loga o sucesso da operação: [Charlie] Broadcast para TODOS completou o anel e foi removido da fila.

```
[17:54:02] 🟤 Pacote agora em transito para ('127.0.0.1', 6001)
[17:54:02] [Charlie] Pacote de dados recebido de ('127.0.0.1', 6002) (origem: Charlie, destino: TODOS, status: maquinanaoexiste)
[17:54:02] [Charlie] Broadcast para TODOS completou o anel e foi removido da fila.
[17:54:02] [Bob] Enviando TOKEN para ('127.0.0.1', 6001)
```

### 3.5. Teste de Destino Inexistente

**Objetivo:** Validar o tratamento de pacotes enviados a um destino que não existe na rede.

**Execução:** O nó Alice enviou uma mensagem para "David", um apelido não existente.

```
David Teste de maquina inexistente
[MessageQueue] Mensagem enfileirada para David
```

**Resultados Observados:** O pacote circulou por todo o anel e, às 17:54:17, retornou ao nó Alice com o status maquinanaoexiste inalterado. Alice logou a falha e removeu a mensagem da fila, como especificado.

### 3.6. Teste de Fila de Mensagens

**Objetivo:** Verificar se a fila de mensagens armazena múltiplos pacotes e os envia na ordem correta, um por vez.

**Execução:** O nó Bob enfileirou duas mensagens rapidamente, uma para Alice e outra para Charlie.

```
Alice mensagem 1 para Alice
[MessageQueue] Mensagem enfileirada para Alice
Charlie mensagem 2 para Charlie
[MessageQueue] Mensagem enfileirada para Charlie
```

**Resultados Observados:** Os logs mostram que Bob, ao obter o token, enviou a primeira mensagem (para Alice), esperou a transação ser concluída, e somente na sua próxima posse do token ele iniciou a transmissão da segunda mensagem (para Charlie). Isso demonstra o funcionamento FIFO da fila e a regra de uma transmissão por vez.

```
[17:54:39] 🟤 Token agora em transito para ('127.0.0.1', 6000)
[17:54:39] [Bob] Pacote de dados recebido de ('127.0.0.1', 6001) (origem: Bob, destino: Alice, status: NAK)
[17:54:39] [Bob] Falha (NAK) ao entregar para Alice. Tentativa 1. Sera retransmitida.
[17:54:43] 🟢 [Bob] TOKEN chegou de ('127.0.0.1', 6001) - Agora em Bob
[17:54:43] 🟠 [Bob] Token retornou em 4.02s (esperado minimo: 4.5s)
[17:54:43] 🟧 [Bob] Enviando para Alice (tentativa 2) via ('127.0.0.1', 6000)
[17:54:43] 🟤 Pacote agora em transito para ('127.0.0.1', 6000)
[17:54:43] [Bob] Pacote de dados recebido de ('127.0.0.1', 6001) (origem: Bob, destino: Alice, status: ACK)
[17:54:43] [Bob] Mensagem para Alice entregada com ACK. Removendo da fila.
[17:54:43] 🟧 [Bob] Enviando para Charlie (tentativa 1) via ('127.0.0.1', 6000)
[17:54:43] 🟧 [Bob] Enviando para Charlie (tentativa 1) via ('127.0.0.1', 6000)
[17:54:43] 🟤 Pacote agora em transito para ('127.0.0.1', 6000)
[17:54:43] [Bob] Pacote de dados recebido de ('127.0.0.1', 6001) (origem: Bob, destino: Charlie, status: ACK)
[17:54:43] [Bob] Mensagem para Charlie entregada com ACK. Removendo da fila.
[17:54:45] 🟧 [Bob] Enviou TOKEN para ('127.0.0.1', 6000)
```

#### 4. Análise e conclusão

As imagens das telas da simulação mostram claramente como funciona a rede em anel e os protocolos implementados:

- O token passando entre os nós Alice, Bob e Charlie.
- Envio e recebimento de mensagens específicas (unicast) e para todos os nós (broadcast).
- Uso do mecanismo CRC32 para detectar erros nas mensagens.
- Confirmação positiva (ACK) e retransmissão (NAK) quando ocorrem falhas.
- Gerenciamento das mensagens aguardando envio pelos nós.

Esses testes visuais confirmam que o protocolo Token Ring está funcionando corretamente, incluindo o gerenciamento de pacotes, o controle de envio e recebimento de mensagens e a capacidade de lidar com erros, exatamente como planejado no desenvolvimento do projeto.

A classe RingNode.py é o núcleo do projeto. Ela gerencia toda a lógica relacionada ao protocolo Token Ring para cada nó individualmente, como capturar o token, verificar mensagens pendentes, enviar dados, processar mensagens recebidas e passar o token adiante. O RingNode foca exclusivamente nas regras do protocolo e delega tarefas auxiliares para outros módulos específicos.

Para facilitar a troca de informações, foi criada a classe Packet.py. Essa classe converte os dados internos (como dicionários Python) em formatos adequados para transmissão pela rede e vice-versa. Essa abordagem simplificou o gerenciamento dos pacotes e facilita alterações futuras.

O gerenciamento das mensagens pendentes ficou a sob responsabilidade da classe MessageQueue.py. Com uma interface simples (adicionar, visualizar próxima mensagem e remover após envio), garante que as mensagens sejam enviadas na ordem correta (FIFO - primeira que entra é a primeira a sair) e conforme as regras do Token Ring.

Para garantir a integridade dos dados, foi implementado o CRC32. As classes CRC32.py e ErrorInserter.py cuidam disso: o CRC32 verifica a integridade das mensagens enviadas e recebidas, enquanto o ErrorInserter simula erros para testar a capacidade do sistema de detectar e corrigir problemas.

Esses módulos juntos permitem uma simulação eficaz da rede em anel:

- Um nó adiciona uma mensagem à fila.
- Ao receber o token, verifica a fila e envia a mensagem formatada pelo Packet.py.
- O pacote circula até chegar ao destino.

- O destino verifica a integridade (CRC32), responde com ACK (sucesso) ou NAK (falha).
- O remetente remove a mensagem após ACK ou retransmite após NAK, conforme regras definidas.
- O ErrorInserter pode ser ativado para testar cenários de falhas.

Essa estrutura mantém a rede ativa, confiável e pronta para novas transmissões a qualquer momento.