# Relatório do Trabalho 1 - Protocolo de Comunicação P2P

## 1. Introdução

Este trabalho tem como objetivo implementar um protocolo de comunicação P2P (peer-to-peer) confiável sobre UDP, permitindo descoberta automática de dispositivos, troca de mensagens e transferência de arquivos com integridade garantida. O projeto foi desenvolvido em Python, utilizando sockets e múltiplas threads para garantir desempenho e robustez, mesmo em cenários de rede adversos.

## 2. Arquitetura do Sistema

### 2.1 Componentes Principais

O sistema é composto por dois módulos principais:

* **dispositivo.py**: Responsável por toda a lógica do protocolo, comunicação UDP, controle de estado, envio/recebimento de mensagens e arquivos, e mecanismos de confiabilidade.
* **main.py**: Implementa a interface de linha de comando, permitindo ao usuário interagir com o sistema, listar dispositivos, enviar mensagens e arquivos.

Além disso, há arquivos auxiliares para testes e logs.

### 2.2 Classes e Estruturas

#### 2.2.1 Classe Dispositivo

A classe Dispositivo centraliza toda a lógica de rede e protocolo. Ela gerencia o socket UDP, threads de envio/recebimento, controle de dispositivos ativos, retransmissão, controle de duplicatas e integridade de arquivos.

**Principais atributos:** - nome: Nome do dispositivo (identificador único na rede). - porta: Porta UDP utilizada para comunicação. - socket: Socket UDP configurado para broadcast e unicast. - dispositivos\_ativos: Dicionário com informações dos dispositivos descobertos (nome, IP, porta, timestamp). - mensagens\_recebidas: Cache de IDs de mensagens já processadas (evita duplicatas). - arquivos\_recebidos: Estrutura para controle de recebimento de arquivos (nome, tamanho, blocos, hash, progresso). - acks\_recebidos: Controle de confirmações de recebimento (ACKs) para cada mensagem enviada.

**Principais métodos:** - \_\_init\_\_: Inicializa o dispositivo, socket, threads e estruturas de controle. - \_enviar\_heartbeat: Envia periodicamente mensagens de broadcast para descoberta de dispositivos. - \_receber\_mensagens: Thread que escuta e processa todas as mensagens UDP recebidas. - \_limpar\_inativos: Remove dispositivos inativos da lista. - enviar\_mensagem: Envia mensagens de texto confiáveis para outro dispositivo, com retransmissão e confirmação. - enviar\_arquivo: Gerencia a transferência de arquivos em blocos, com controle de ACKs, retransmissão e verificação de integridade. - \_processar\_\*: Métodos privados para processar cada tipo de mensagem do protocolo (TALK, FILE, CHUNK, END, ACK, NACK). - \_log: Registra eventos e mensagens em arquivo de log, podendo exibir no terminal.

**Exemplo de inicialização:**

meu\_dispositivo = Dispositivo(nome="Joao", porta=5000)

#### 2.2.2 Classe Interface

A classe Interface provê a interação com o usuário via terminal. Permite listar dispositivos, enviar mensagens e arquivos, e exibe o menu principal.

**Principais métodos:** - mostrar\_menu: Exibe as opções disponíveis ao usuário. - listar\_dispositivos: Mostra todos os dispositivos ativos na rede. - enviar\_mensagem: Solicita dados ao usuário e chama o método correspondente do dispositivo. - enviar\_arquivo: Solicita o nome do arquivo e o destino, e inicia a transferência. - executar: Loop principal da interface.

## 3. Protocolo de Comunicação

### 3.1 Tipos de Mensagens e Formatos

O protocolo define os seguintes tipos de mensagens, cada uma com formato e semântica específicos:

1. **HEARTBEAT** (broadcast)
   * Formato: HEARTBEAT <nome>
   * Exemplo: HEARTBEAT Joao
   * Função: Descoberta automática de dispositivos na rede. Enviada a cada 5 segundos para todas as portas do intervalo.
2. **TALK** (unicast)
   * Formato: TALK <id> <mensagem>
   * Exemplo: TALK 12345 Olá, tudo bem?
   * Função: Envio confiável de mensagens de texto. Cada mensagem tem um ID único para controle de duplicatas.
3. **FILE** (unicast)
   * Formato: FILE <id> <nome> <tamanho>
   * Exemplo: FILE 67890 foto.png 204800
   * Função: Inicia a transferência de arquivo, informando nome e tamanho. Aguarda ACK antes de enviar os blocos.
4. **CHUNK** (unicast)
   * Formato: CHUNK <id> <seq> <dados\_base64>
   * Exemplo: CHUNK 67890 0 SGVsbG8gV29ybGQ=
   * Função: Transfere um bloco do arquivo, codificado em base64. Cada bloco tem número de sequência e requer ACK individual.
5. **END** (unicast)
   * Formato: END <id> <hash>
   * Exemplo: END 67890 5d41402abc4b2a76b9719d911017c592
   * Função: Finaliza a transferência, enviando o hash SHA-256 do arquivo para verificação de integridade.
6. **ACK** (unicast)
   * Formato: ACK <id> [seq|END]
   * Exemplo: ACK 67890 0 ou ACK 67890 END
   * Função: Confirma o recebimento de mensagens, blocos ou finalização.
7. **NACK** (unicast)
   * Formato: NACK <id> <motivo>
   * Exemplo: NACK 67890 HASH\_MISMATCH
   * Função: Indica falha na transferência, como erro de integridade ou timeout.

### 3.2 Mecanismos de Confiabilidade

O protocolo implementa diversos mecanismos para garantir a entrega correta das mensagens e arquivos:

* **Confirmação de Recebimento (ACK):** Todas as mensagens importantes requerem confirmação. Se não houver ACK em tempo hábil, a mensagem é retransmitida até 3 vezes.
* **Controle de Duplicatas:** IDs únicos para cada mensagem e bloco. Mensagens duplicadas são descartadas.
* **Transferência em Blocos:** Arquivos são enviados em blocos de 1KB, cada um com número de sequência e ACK individual.
* **Verificação de Integridade:** O hash SHA-256 do arquivo é enviado ao final. O receptor compara com o hash calculado localmente e envia ACK ou NACK.
* **Timeouts e Retransmissão:** Se não houver resposta, o bloco/mensagem é retransmitido. Após 3 tentativas sem sucesso, a transferência é abortada.
* **Tratamento de Falhas:** NACKs são enviados em caso de erro de integridade, timeout ou formato inválido.

## 4. Análise das Capturas

As capturas de pacotes foram realizadas com o Wireshark, e falhas de rede foram simuladas com o Clumsy. A seguir, detalhamos o comportamento do protocolo em cada cenário.

### 4.1 Captura de HEARTBEAT

**Arquivo:** capturaNova2HeartbeatT1.pcapng - Mostra o envio periódico de mensagens HEARTBEAT via broadcast. - Permite que dispositivos recém-iniciados descubram rapidamente os demais. - O Wireshark exibe pacotes UDP para todas as portas do intervalo configurado.

### 4.2 Captura de TALK

**Arquivo:** capturaNova2TalkT1.pcapng - Mostra o envio de uma mensagem TALK de um dispositivo para outro. - O receptor responde com ACK, confirmando o recebimento. - IDs únicos garantem que mensagens duplicadas não sejam processadas. - O Wireshark mostra o fluxo TALK → ACK.

### 4.3 Captura de Transferência de Arquivo

**Arquivo:** capturaNova2SendfileT1.pcapng - Mostra a sequência completa: FILE → ACK → CHUNKs → ACKs → END → ACK. - Cada bloco CHUNK é confirmado individualmente. - O hash do arquivo é verificado ao final, garantindo integridade. - O Wireshark mostra todos os pacotes trocados, incluindo retransmissões se houver perda.

### 4.4 Testes de Falhas

#### 4.4.1 Perda de Pacotes (Drop)

**Arquivo:** capturaNova2DropT1.pcapng - Simulação de 10% de perda de pacotes. - O protocolo detecta ausência de ACK e retransmite automaticamente. - A transferência é completada com sucesso após algumas retransmissões.

#### 4.4.2 Duplicação (Duplicate)

**Arquivo:** capturaNova2DuplicateT1.pcapng - Simulação de 10% de duplicação de pacotes. - O receptor descarta duplicatas usando IDs e números de sequência. - Não há impacto na integridade da transferência.

#### 4.4.3 Atraso (Lag)

**Arquivo:** capturaNova2LagT1.pcapng - Simulação de atraso de 200ms. - O protocolo aguarda o tempo adequado antes de retransmitir. - A transferência pode ser um pouco mais lenta, mas é completada com sucesso.

#### 4.4.4 Reordenação (Out of Order)

**Arquivo:** capturaNova2OutOfOrderT1.pcapng - Pacotes CHUNK chegam fora de ordem. - O receptor reordena os blocos antes de salvar o arquivo. - O hash final garante que o arquivo está correto.

#### 4.4.5 Corrupção (Tamper)

**Arquivo:** capturaNova2TamperT1.pcapng - Simulação de corrupção de dados em alguns pacotes. - O hash SHA-256 detecta qualquer alteração. - O receptor envia NACK e solicita retransmissão dos blocos afetados.

#### 4.4.6 Teste Combinado

**Arquivo:** capturaNova2TesteCombinadoT1.pcapng - Combinação de todas as falhas anteriores (perda, duplicação, atraso, reordenação, corrupção). - O protocolo lida com todas as adversidades, completando a transferência com sucesso após múltiplas retransmissões e verificações.

## 5. Conclusão

O protocolo desenvolvido demonstrou robustez e confiabilidade, sendo capaz de garantir a entrega correta de mensagens e arquivos mesmo em condições adversas de rede. Os mecanismos de confirmação, retransmissão, controle de duplicatas e verificação de integridade funcionaram conforme esperado.

### 5.1 Pontos Fortes

* Descoberta automática de dispositivos na rede local.
* Transferência confiável de arquivos, mesmo com falhas de rede.
* Logs detalhados para depuração e análise.
* Estrutura modular e fácil de expandir.

### 5.2 Limitações

* Dependência de broadcast UDP para descoberta (restrito a redes locais).
* Necessidade de configuração de firewall para permitir comunicação.
* Não há criptografia nativa (dados trafegam em texto claro/base64).
* Interface apenas em linha de comando.

### 5.3 Melhorias Futuras

* Suporte a redes maiores (NAT traversal, relay, etc).
* Compressão e criptografia de dados.
* Interface gráfica para facilitar o uso.
* Ajuste dinâmico de timeouts e retransmissões.
* Suporte a múltiplas transferências simultâneas.

## 6. Referências

* Python Documentation: https://docs.python.org/3/
* Wireshark Documentation: https://www.wireshark.org/docs/
* Clumsy Documentation: https://jagt.github.io/clumsy/
* RFC 768 - User Datagram Protocol (UDP)
* RFC 793 - Transmission Control Protocol (TCP) (para inspiração nos mecanismos de confiabilidade)

## 7. Anexos

* Códigos-fonte do programa (dispositivo.py, main.py).
* Arquivos de captura do Wireshark (.pcapng).
* Prints de tela e logs relevantes.
* Arquivo de teste: grande\_teste.txt.
* Roteiro de testes e instruções de uso.