Laboratório de Redes de Computadores - Trabalho 1

Rafael Almeida de Bem

25 de abril de 2022

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

Este trabalho implementa um protocolo de comunicação customizado que não faz uso do protocolo TCP/IP. As comunicações se dão via *raw sockets* na camada 2 do modelo OSI [4]. Nesse caso, utilizamos frames Ethernet II [8]. Os *headers* (cabeçalhos) Ethernet são populados manualmente pelo sistema no seguinte formato:

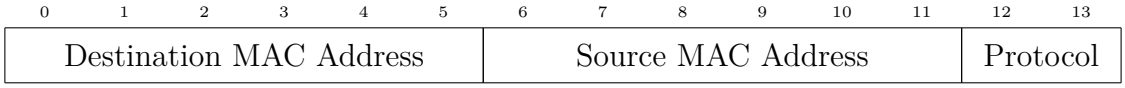

Figura 1: Header Ethernet II

2 Definição do Protocolo

O protocolo customizado, chamado de `T1Protocol`, se baseia em: um tipo de mensagem, um campo de endereço, e um campo de dados. O tipo de mensagem pode ser:

START <NAME> Mensagem enviada para o endereço de Broadcast, indicando que um novo nodo entrou na rede. O "nome" (*name*) do nodo é seu endereço MAC.

HEARTBEAT <NAME> Mensagem de *alive* enviada via Broadcast para todos os nodos a cada 5 segundos, indicando que aquele dispositivo ainda está ativo na rede. O campo *NAME* indica o endereço MAC do nodo que está enviando a mensagem. Essa mensagem também é utilizada ao receber uma mensagem do tipo *START*. Nesse caso, é enviado um *HEARTBEAT* diretamente ao nodo que enviou a mensagem de *START*, e não ao endereço de Broadcast.

TALK <NAME> <DATA> Mensagem enviada quando um dispositivo deseja se comunicar com outros dispositivos via Broadcast. O campo *NAME* contém o nome do dispositivo que está enviando a mensagem. O campo *DATA* contém dados quaisquer.

O protocolo foi descrito na linguagem Protobuf 3 [5] para versioná-lo. Utilizamos o programa *protoc* [9] juntamente do programa *betterproto* [1] para gerar as classes necessárias.

Todas as máquinas da rede possuem uma lista de roteamento (lista de máquinas conhecidas). Mensagens vindas de máquinas fora dessa lista são ignoradas. O sistema dispõe de uma maneira de remover máquinas automaticamente pelo tempo do último *HEARTBEAT* de cada nodo. Caso um nodo não envie um *alive*/*HEARTBEAT* por mais de 15 segundos, ele é removido da lista dessa máquina e dado como offline. Para que ele seja adicionado novamente à lista do nodo atual, ele deve enviar uma mensagem de *START*.

Sempre que uma máquina entra na rede e manda uma mensagem *START*, os demais nodos adicionam essa nova máquina à suas listas de *hosts* conhecidos/tabelas de roteamento e respondem-na diretamente com um *HEARTBEAT*. A máquina que enviou o *START* adiciona em sua lista de máquinas conhecidas todas as máquinas que responderam seu *START* com um *HEARTBEAT*. A mensagem *HEARTBEAT* enviada diretamente ao remetente que enviou o *START* será chamada de *ACK_ALIVE* daqui em diante.

Não foi definido um limite máximo no tamanho dos dados na mensagem *TALK*, então assumiu-se que os dados, juntamente dos headers e outras informações necessárias, não ultrapassarão o tamanho máximo de um *frame* Ethernet, 1518 B.

As mensagens de *HEARTBEAT* (para Broadcast) e *START* não funcionariam corretamente sem o campo *name*, que contém o endereço MAC da máquina em si. Como a máquina de origem envia esses *frames* para o endereço de Broadcast, os outros nodos recebem o *frame* como se

estivesse sido enviada do endereço de Broadcast. Acessando o campo *name* podemos salvar o endereço MAC do nodo original e ignoramos o endereço de origem nesses casos

3 Implementação Prática

O trabalho foi escrito em Python 3.10 [6], atual (25 de abril de 2022) versão oficial da linguagem. Também utilizamos o sistema de descrição de protocolos Protobuf [5]/gRPC [3], da Google.

O relatório utiliza as palavras pacote, *packet* e *frame* para representar sempre um *frame* Ethernet II. Um "pacote" (*packet*) só existe da camada 3 do modelo OSI [4] para cima.

3.1 Protocolo

Como mencionado acima, o protocolo foi definido na linguagem Protobuf 3 [5]. O *enum* `MessageType` define o tipo de mensagem que será enviado, podendo ser um de:

- `START`
- `HEARTBEAT`
- `TALK`

O campo `type` do tipo `MessageType` indica o tipo de mensagem daquele pacote. O campo `string name` contém o endereço MAC do remetente (a máquina que está enviando a mensagem). O campo `string data` contém os dados quaisquer que serão enviados na mensagem *TALK*.

```
syntax = "proto3";

package t1_protocol;

message T1Protocol {
    enum MessageType {
        START = 0;
        HEARTBEAT = 1;
        TALK = 2;
    }
    MessageType type = 1;
    string name = 2;
    string data = 3;
}
```

Figura 2: Descrição do protocolo

Os frames enviados pelo sistema têm seu formato descrito pela Figura 3.

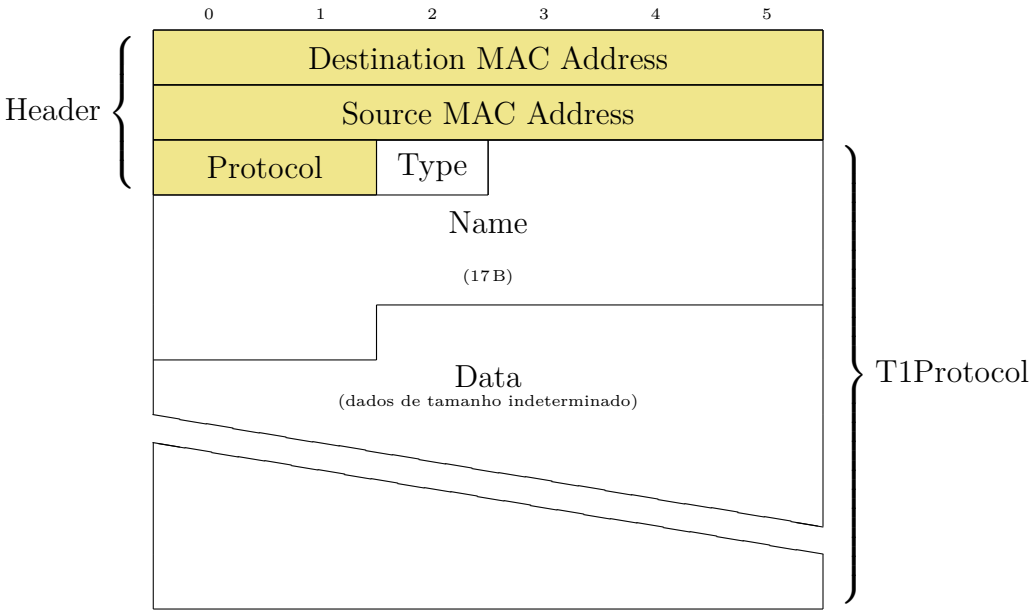

Figura 3: Formato dos frames

O tamanho máximo do campo *Data* é de 1486 B.

$$1518\,\text{B (max)} - 14\,\text{B (header)} - 1\,\text{B (type)} - 17\,\text{B (name)} = 1486\,\text{B}$$

3.2 Implementação

O sistema consiste de uma interface simples via CLI e duas *threads*, uma para recebimento de mensagens, e outra para envio de mensagens. A thread de recebimento é uma instância da classe `ReceiverDaemon`, que herda da classe `RawSocketDaemon`. A thread de envio é uma instância da classe `SenderDaemon` que também herda de `RawSocketDaemon`.

A classe `RawSocketDaemon` define algumas propriedades comuns entre recebimento e envio como interface, endereço MAC e o objeto do socket em si. A função `create_and_bind_socket(iface: str)` cria e retorna o *raw socket* com o protocolo `ETH_P_ALL` (3) para recebermos todos os pacotes da camada 2 e faz *bind* à interface desejada.

```python
def create_and_bind_socket(iface: str) -> socket.SocketType:
    s = socket.socket(socket.AF_PACKET, socket.SOCK_RAW,
                      socket.htons(ETH_P_ALL))
    s.bind((iface, 0))
    return s
```

Figura 4: Criação do raw socket

Essa e outras funções de uso geral estão definidas no arquivo `src/socket_utils.py`, como as funções de `pack()` e `unpack()` dos headers Ethernet.

3.2.1 Os Daemons

O `ReceiverDaemon` é a classe que recebe dados do socket e que possui a tabela de roteamento (`self.known_hosts`) e cuida da lista de nodos "vivos" (`self.alive_table`). A thread possui um *timer* que roda de 15 em 15 segundos que faz a verificação da lista de nodos "vivos".

Ao receber uma mensagem, a classe verifica se a mensagem é destinada ao seu MAC/Broadcast e, se sim, verifica se o remetente está em sua lista de nodos conhecidos. Caso não esteja e a mensagem seja do tipo *START* ou *HEARTBEAT* com destino ao seu MAC (e não ao Broadcast), a mensagem é processada (Figura 5).

```
    match data.type:
        case T1ProtocolMessageType.START:
            # Salva nodo na lista de nodos conhecidos
            self.add_to_routing_table(data.name)
            # Responde com HEARTBEAT
            self.ack_alive(data.name)
        case T1ProtocolMessageType.HEARTBEAT:
            # Verifica se é uma resposta ao nosso START
            if src != sock_util.MAC_BROADCAST:
                self.add_to_routing_table(data.name)
            # Atualiza o tempo do último alive desse host
            self.update_alive_table(data.name)
        case T1ProtocolMessageType.TALK:
            # Imprime a mensagem recebida
            print(
                f'TALK[From {src} ({data.name}) to {dst}]: {data.data}')
            # Salva o último contato (para REDIAL)
            self.last_contact = data.name
        case _:
            print(f'Unknown type. Type: {packet_type}')
```

Figura 5: Processamento de mensagens

O `SenderDaemon` é a classe que faz o envio de dados e é a responsável por enviar o um *HEARTBEAT* e 5 em 5 segundos. A classe possui uma fila do tipo `Queue` para adição/remoção de comandos para que o sistema seja thread-safe. O método `SenderDaemon.put()` é quem faz a codificação dos dados da mensagem e adiciona na fila de processamento. O `put()` invoca o método `encode_message()` para instanciar a classe `T1Protocol` com os dados da mensagem.

```
    def encode_message(self, type: T1ProtocolMessageType, data: str = '') →
     ↪  T1Protocol:
        return T1Protocol(type=type, name=self.name, data=data)

    def put(self, type: T1ProtocolMessageType, dst: str, data: str = '') →
     ↪  None:
        dest = [int(d, 16) for d in dst.split(':')]
        msg = self.encode_message(type, data)
        header = Header(pack_eth_header(self.mac_address, dest,
         ↪  ETH_CUSTOM_PROTOCOL))
        self.q.put((header, msg))
```

Figura 6: Inclusão na lista de mensagens

No *loop* dessa thread, ao receber uma mensagem, o header e mensagem são extraídos. A mensagem é serializada para *bytes* pelo método `SerializeToString` do gRPC [3]. Então, é feito o envio do header somado com os dados serializados.

```
    def run(self) -> None:
        while True:
            message = self.q.get()
            header, proto_data = message
            proto_data = proto_data.SerializeToString()
            self.socket.send(header + proto_data)
```

Figura 7: Envio de mensagens

4 Uso

O programa possui um menu rudimentar que explica mais a fundo as opções, que são: `MENU`, `START`, `HEARTBEAT`, `TALK`, `TALKTO`, `REDIAL` e `TABLE`. As funções `TALKTO` e `REDIAL` são para aumentar a qualidade de vida do usuário, onde `TALKTO` envia para um ou mais destinos uma mensagem, e `REDIAL` envia uma mensagem ao último nodo que nos enviou um *TALK*.

Antes de rodar o programa é preciso instalar os pacotes necessários, de preferência em um ambiente virtual.

```
$ python3.10 -m venv venv && source venv/bin/activate
$ pip install -r requirements.txt
```

Estando com o ambiente virtual habilitado, o programa pode ser invocado:

```
$ python main.py
```

O programa também pode receber como argumento uma interface de rede para fazer o *bind*:

```
$ python main.py $INTERFACE
```

5 Testes

Utilizamos a arquitetura de rede demonstrada na Figura 8 no CORE Emulator [2]. Possuímos quatro nodos (*n2*, *n3*, *n4* e *n6*) conectados por uma switch. Os endereços MAC dos nodos estão definidos na Tabela 1.

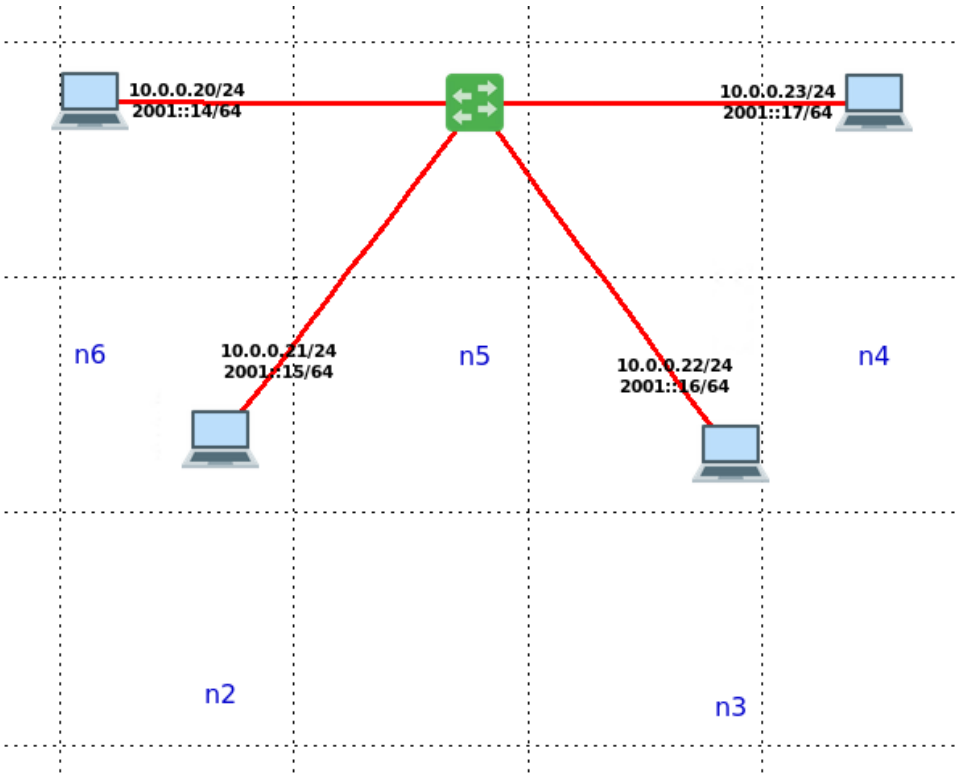

Figura 8: Arquitetura da rede no CORE

O nodo *n5* é a switch de rede que faz a interconexão entre todos os nodos.

Nodo	**Mac**
n2	00:00:00:AA:00:01
n3	00:00:00:AA:00:02
n4	00:00:00:AA:00:03
n6	00:00:00:AA:00:00

Tabela 1: Mapeamento de nodos para seus MACs

START Enviamos o *START* apenas quando o usuário solicita. O *START* é enviado usando a opção **1** ou **START** do menu. Demonstrado na Figura 9.

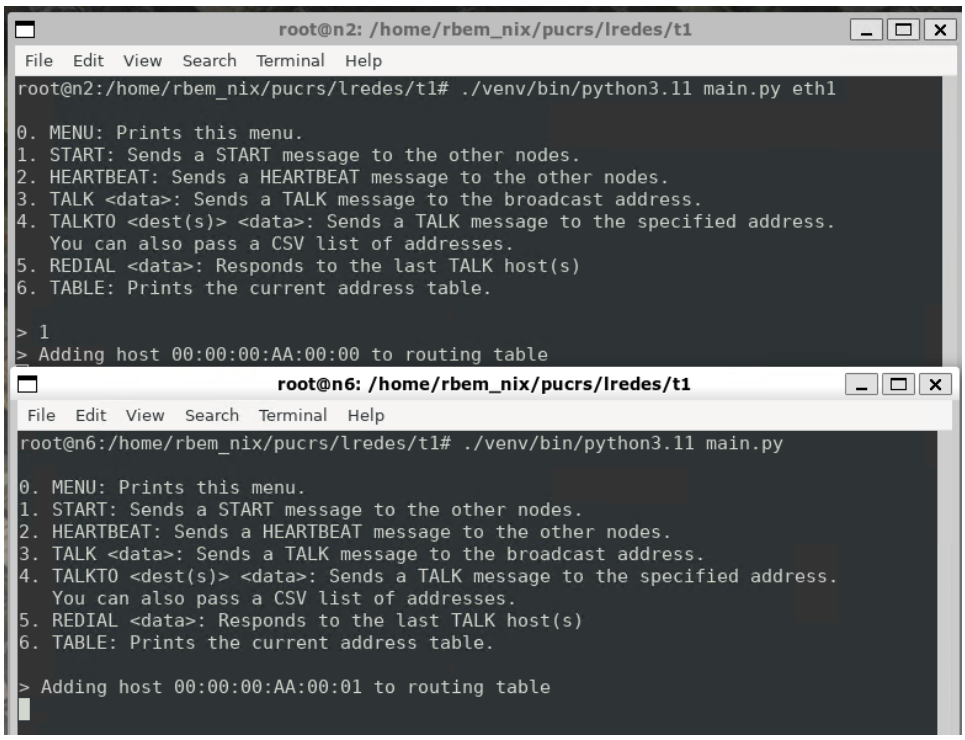

Figura 9: START enviado por *n2* e recebido por *n6*

HEARTBEAT de Resposta Enviamos um *HEARTBEAT* com destino ao nodo que nos enviou a mensagem de *START*. Esse processo é feito de maneira autônoma e está demonstrado na Tabela 2.

Origem	Destino	Mensagem
00:00:00:AA:00:02	Broadcast	*START*
00:00:00:AA:00:00	00:00:00:AA:00:02	*HEARTBEAT*

Tabela 2: *START* enviado por *n3*, recebido por *n6* e respondido com *HEARTBEAT*. Dados coletados com Wireshark [7].

HEARTBEATS Os nodos enviam um *HEARTBEAT* para o endereço de Broadcast de 5 em 5 segundos.

Timeout Passados 15 segundos sem receber um *HEARTBEAT*, o nodo é removido da tabela de roteamento local. Essa funcionalidade está evidenciada na Figura 10.

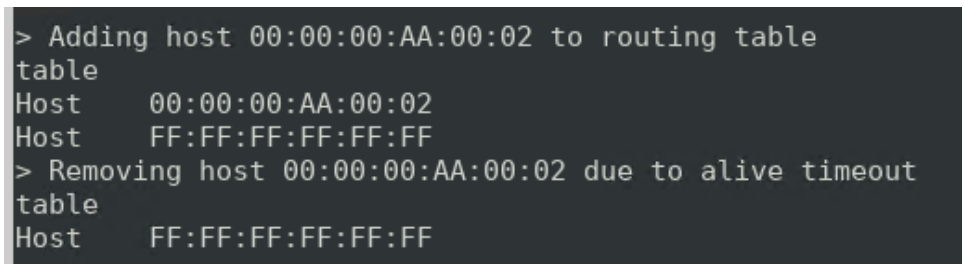

Figura 10: Timeout de *n3* em *n6* por falta de *HEARTBEAT*

TALK Mensagem enviada a todos os nodos via Broadcast com um payload de dados quaisquer. Demonstrado na Figura 11.

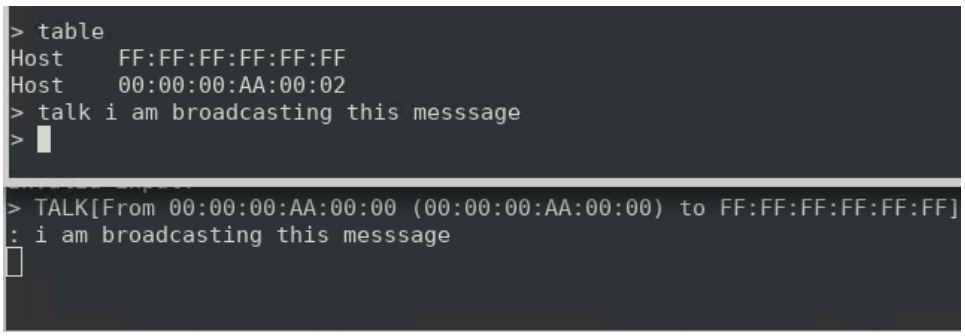

Figura 11: *TALK* de *n6* para Broadcast, recebido por *n3*

TALKTO Envia um *TALK* para um ou mais nodos diretamente, sem ser via Broadcast. A mensagem é definida por um ou mais endereços MAC separados por vírgula, seguidos de um espaço e os dados a serem enviados.

REDIAL Envia um *TALK* para o último nodo que nos contatou por último. Não faz nada quando não foi recebido um *TALK* de outro nodo – último contato está vazio.

6 Conclusão

Em suma, a utilização de raw sockets nos permite a utilização de protocolos diferentes, indo além dos clássicos TCP/IP e UDP. Os *frames* do protocolo `T1Protocol` (definidos na Figura 3) são enviados diretamente pela segunda camada do modelo OSI [4]. A ferramenta Wireshark [7] nos permitiu analisar os dados dos *frames* enviados e recebidos e garantir que o sistema funcione corretamente.

Referências

[1] Better Protobuf / gRPC Support for Python. https://github.com/danielgtaylor/python-betterproto. [Online; acessado em 24-Abr-2022]. 2

[2] Common Open Research Emulator. https://github.com/coreemu/core. [Online; acessado em 24-Abr-2022]. 6

[3] gRPC. https://grpc.io/. [Online; acessado em 24-Abr-2022]. 3, 6

[4] ISO/IEC 7498-1:1994, OSI Model. https://www.iso.org/standard/20269.html. [Online; acessado em 24-Abr-2022]. 2, 3, 9

[5] Protocol Buffers - Google's data interchange format. https://github.com/protocolbuffers/protobuf. [Online; acessado em 24-Abr-2022]. 2, 3

[6] Python 3.10.0. https://www.python.org/downloads/release/python-3100/. [Online; acessado em 24-Abr-2022]. 3

[7] Wireshark. https://www.wireshark.org/. [Online; acessado em 24-Abr-2022]. 1, 8, 9

[8] IEEE SA - IEEE Standard for Ethernet. https://standards.ieee.org/ieee/802.3/7071/, 2018. [Online; acessado em 24-Abr-2022]. 2

[9] Protocol Buffer Compiler. https://grpc.io/docs/protoc-installation/, 2021. [Online; acessado em 24-Abr-2022]. 2