RELATÓRIO SOBRE O TRABALHO DE CAMADA DE TRANSPORTE

TRABALHO DA M2

Estudantes: André Aliardi, Eduardo Amaro Maciel, Jhonatan Mariani, Jhonatan Fernandes e João Vitor Dubiela
Disciplina: Redes de Computadores 1
Professor: Felipe Viel
Data: 20/10/2021

ÍNDICE

Link do projeto no GitHub: https://github.com/jhonatanmariani/redes1-av2

Link da apresentação da implementação do código: https://youtu.be/lLn9wwMvrkU

subir docx

subir pdf

add o prof no github

1. Descrição do trabalho proposto

Neste trabalho deve-se implementar em 2 partes as versões propostas do protocolo Reliable Data Transfer (RTP). Sendo, a primeira parte sobre a versão 2.1 e 2.2. Já na segunda parte deverá ser implementada a versão 3.0 do protocolo.

Para a primeira parte da implementação do algoritmo, ele deve englobar os seguintes requisitos:

- Seqnum – sequência do pacote enviado
- Acknum – número do Ack gerado
- Checksum – valor da soma de verificação calculado
- Carga útil

Para a segunda parte deve-se utilizar, ou seja, reaproveitar a parte 1 já implementada anteriormente, adicionando o temporizador para o controle do tempo para envio de resposta do pacote.

Para a simulação da implementação realizada, deve-se realizar simulações que contenham uma transmissão normal do pacote, uma transmissão do pacote com atraso, uma transmissão do pacote com perda e uma transmissão do pacote com corrompimento dos dados.

2. Descrição do protocolo implementado

2.1. O que é o RDT?

O RDT (Reliable Data Transfer) que em português significa transferência confiável de dados. Ele foi criado advindo do protocolo UDT que é um protocolo da camada de transporte, muito utilizado em aplicações que exigem um transporte rápido e contínuo de dados entre os equipamentos, exemplos de tipo de aplicações que utilizam são streaming de áudio e vídeo.

Com isso, não há verificação do recebimento e a integridade dos dados enviados, sendo os dados transmitidos apenas uma vez. Quando os pacotes chegam corrompidos, eles são descartados, sem que o emissor fique sabendo do problema.

Com a utilização do RTD, nenhum dos dados transferidos são corrompidos ou perdidos, e todos são entregues na ordem em que foram enviados. Portanto, um protocolo de transferência confiável de dados possui essa implementação de abstração de serviço.

2.2. Versões existentes do RDT

2.2.1. Versão 1.0

Na versão mais simples do RDT, tem-se a implementação onde o canal de transferência é completamente confiável. Com isso, temos uma transmissão unidirecional, sem a confirmação e entrega dos dados ao remetente.

2.2.2. Versão 2.0

Na versão 2.0 do RDT, temos a verificação de bits do pacote verificando se estes estão ou não corrompidos. Esses erros podem ocorrer nos componentes físicos da rede enquanto está sendo transmitido o pacote. Embora, ainda nessa versão do protocolo é feita uma suposição de que todos os pacotes transmitidos sejam recebidos pelo destinatário, na ordem que foram enviados pelo remetente.

Portanto, para verificar se há presença de erros são exigidas 2 ferramentas:

- **Detecção de erros:**

O destinatário deve detectar se há erros de bits nos pacotes recebidos. Para isso, é utilizado técnicas onde o remetente envia bits extras, além dos bits do dado original para que o destinatário possa detectar se há erros.

- **Realimentação do destinatário:**

Envio de feedback ao remetente, assim, o remetente saberá se um pacote chegou ao seu destino. Para isso, são utilizados NAKs e ACKs são exemplos dessas respostas.

O remetente nessa versão (2.0) possui dois estados:

1. **Espera a chamada de cima:**

Nesse estado o protocolo está esperando que os dados sejam passados pela camada superior. Quando houver um evento de envio de dados, o remetente criará um pacote contendo os dados que serão enviados juntamente com a soma de verificação do pacote.

2. **Espera o ACK ou NAK:**

Nesse estado o protocolo fica esperando um pacote ACK ou NAK do destinatário. Caso chegue um pacote ACK, o remetente saberá que o pacote mais recente chegou ao destinatário. Se um NAK for recebido, o protocolo retransmitirá o último pacote e esperará por uma nova resposta do destinatário. Enquanto o remetente está esperando por uma resposta, ele não pode receber mais dados da camada superior.

Já o destinatário nessa versão 2.0 do RDT tem apenas um estado, sendo muito parecido com o RDT na versão 1.0. Quando ele recebe um pacote, ele responde ao remetente um pacote com um ACK ou NAK dependendo do estado do pacote recebido. Mesmo funcionando nessa versão 2.0 essa resposta, ainda possui defeitos, sendo um deles a possibilidade de haver uma corrupção dos pacotes NAK e ACK. Para isso, uma solução viável é a numeração pelo remetente dos pacotes enviados, sendo essa numeração sequencial.

2.2.3. Versão 2.1

Na versão 2.1 do RDT há a implementação contra falha de recebimento dos pacotes, ainda que com a falha citada anteriormente. Por isso, quando um pacote fora de ordem é recebido pelo destinatário é enviado ao remetente um ACK, já quando um pacote recebido esteja corrompido um NAK é enviado. Um remetente que recebe dois ACKs para o mesmo pacote, interpreta que o destinatário não recebeu corretamente o pacote seguido.

2.2.4. Versão 2.2

Na versão 2.2 do RDT há uma melhoria da versão 2.1, que é a implementação da numeração em sequência do pacote reconhecido pelo destinatário por uma mensagem ACK e o remetente verifica esse número que está sendo reconhecido também por uma mensagem ACK.

2.2.5. Versão 3.0

Na versão 3.0 do RDT, possui um temporizador de contagem regressiva para que o remetente fique esperando um ACK sem saber se ele foi perdido no meio do caminho, se está atrasado ou se o pacote nem chegou ao destino.

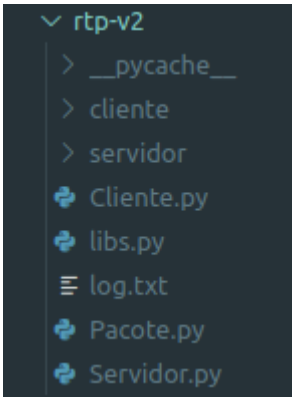
Então o remetente terá que adotar um tempo intermediário de recebimento do pacote ACK dentro desse período, caso contrário fará o reenvio do pacote para o destinatário. Mas como o

destinatário controla os pacotes recebidos quando há um atraso no envio? Através da numeração deles, consegue-se ter esse controle.

O remetente após o tempo definido e com o não recebimento do ACK, não se sabe se o pacote teve atraso na entrega, se o mesmo foi perdido, se o ACK teve atraso na entrega ao remetente ou ainda se o ACK no meio do caminho (destinatário-remetente) foi perdido. Nesse caso é feito o reenvio do pacote para o destinatário.

3. Códigos importantes da implementação

3.1. Versões 2.1 e 2.2



A imagem acima representa a estrutura do projeto. A pasta cliente representa o diretório de uma máquina que simula uma requisição cliente, enquanto a pasta servidor representa o diretório de arquivos de um servidor simulado.

O arquivo libs.py possui a importação de todas as bibliotecas sendo utilizadas por outras classes.

```
import os
import socket
import pickle

from socket import timeout
from threading import Thread
from random import randint
from termcolor import colored
from datetime import datetime
from hashlib import sha1
```

O arquivo Cliente.py juntamente com a pasta Cliente representa o cliente requisitando um arquivo ao servidor. A classe possui o método **request** que cria uma conexão **socket** utilizando o endereço do servidor, que nesse caso é localhost/127.0.0.1.

```
class Cliente:
    def __init__(self, ip='localhost', porta=9999):
        self.ip = ip
        self.porta = porta
        self.endereco_servidor = (self.ip, self.porta)
        print(self.endereco_servidor)
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Nesse mesmo método foi implementada uma máquina de estados através do laço de repetição **while** onde é chamado continuamente enquanto o número máximo de tentativas não for alcançado. É

nesse escopo que é identificado os possíveis problemas de conexão como por exemplo desconexão ou arquivo não encontrado.

```
# Envia uma requisição com arquivo e espera a resposta
timeout_tentativas = TIMEOUT_TENTATIVAS_CLIENTE
while timeout_tentativas:
    # Envia a requisição
    pacote = Pacote(arquivo=arquivo)
    self.socket.send(pacote.__dumb__())
```

A simulação de pacote corrompido está implementada no método **fn_arquivo_recebido**. Foi implementada uma regra aleatória para a simulação de ACKs positivos e NACKs.

Todas as mensagens são exibidas no terminal que está simulando o cliente.

O arquivo Servidor.py juntamente com a pasta Servidor representa o servidor no qual possui o arquivo que será transmitido ao cliente.

```
class Servidor:
    def __init__(self, ip='localhost', porta=9999):
        self.ip = ip
        self.porta = porta
        self.endereco = (self.ip, self.porta)
        self.threads = []
        self.thread_contador = 0
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.socket.bind(self.endereco)

        print('Servidor: ', self.ip, self.endereco)
```

O método **aguarda_requisicao** implementa um laço de verificação da conexão com o cliente.

O método **listener** implementa a conexão **socket** com o endereço especificado na classe. Também possui um laço de repetição no qual só é interrompido quando o servidor é intencionalmente finalizado.

O método **disponibiliza_pacote_cliente** faz a verificação do arquivo dentro do diretório especificado no servidor e juntamente com o método **envia_pacote** realiza o envio dos chunks de acordo com a variável **TAMANHO_PACKET.** É esperado um ACK positivo do cliente para dar sequência no envio, caso contrário haverá uma nova tentativa de envio até que o número de tentativas seja alcançado. A verificação de arquivo corrompido ocorre no método **disponibiliza_pacote_cliente.**

```
# Envia e verifica arquivo corrompido
sender_checksum = pacote.__get__('checksum')
reciever_checksum = pacote.checksum(sha1(dados_coletados).hexdigest())

bin_sender_checksum = bin(pacote.checksum(sha1(dados_coletados).hexdigest()))
bin_reciever_checksum = bin(pacote.__get__('checksum'))

integer_sum = int(bin_sender_checksum, 2) + int(bin_reciever_checksum, 2)
binary_sum = bin(integer_sum)
only_bin = binary_sum.replace('0b', "")
arquivo_corrompido = bin_sender_checksum == bin_reciever_checksum
print(arquivo_corrompido)
```

Todas as mensagens são exibidas no terminal que está simulando o servidor.

O arquivo **Pacote.py** implementa a classe que parametriza as propriedades necessárias que um packet deve possuir (status, arquivo, ack, seq_num, checksum, dados_coletados). A classe possui o método **checksum** que é utilizado pelo **Servidor.py** para verificar se o pacote recebido é equivalente ao pacote inicialmente transmitido.

```
class Pacote:
    def __init__(self, pickled=None, seq_num=0, dados_coletados=b'', ack='', arquivo='', status=''):
        if pickled is not None:
            self.pacote = pickle.loads(pickled)
        else:
            self.pacote = {
                "status": status,
                "arquivo": arquivo,
                "ack": ack,
                "seq_num": seq_num,
                "checksum": self.checksum(sha1(dados_coletados).hexdigest()) if dados_coletados else '',
                "dados_coletados": dados_coletados
            }
```

A classe verifica o ACK, status do arquivo e status de requisição que são exibidas nos terminais que estão simulando o servidor e o cliente.

3.2. Versão 3.0

Na implementação da versão dois do protocolo, foi adicionado o uso do temporizador para o controle do tempo para o envio de resposta do pacote.

```
# Envia pacote para o cliente e espera um ACK positivo
# Retorna 1 para sucesso ou 0
def comeca_transmissao(self, packets, cliente, endereco):
    base = 0
    self.enviar_janela(packets, base, cliente, endereco)

    contagem_tempo_limite_cliente = TIMEOUT_TENTATIVAS_CLIENTE
    while contagem_tempo_limite_cliente:
        if base >= len(packets):
            break

        fim = base + TAMANHO_JANELA
        fim = fim if fim < len(packets) else len(packets)

        cliente.settimeout(TIME_OUT)
        try:  # Aguarda por resposta ou timeout
            res = cliente.recv(TAMANHO_PACKET)
            if not res:
                break
            pkt = Pacote(res=res)
            pkt.__print__(endereco_origem=endereco)
            seq_num = int(pkt.__get__('seq_num'))
            if base <= seq_num <= base + TAMANHO_JANELA:
                if pkt.__get__('ack') == '+':
                    cliente.settimeout(None)
```

Uma vez que o remetente tenha enviado todos os quadros na janela (window), ele detectará que todos os quadros desde o primeiro quadro perdido estão pendentes, e retornará ao número de sequência do último ACK que recebeu do processo receptor e preencherá seu janela começando com aquele quadro e continue o processo novamente.

4. Resultados obtidos com a implementação

Foi possível implementar uma conexão real entre cliente e servidor via socket através de um endereço local previamente configurado e parametrizado. Todo o projeto contempla a transmissão normal do pacote, transmissão do pacote com atraso, transmissão do pacote com perda (timeout) e transmissão do pacote com corrompimento dos dados (e nova tentativa de envio).

O arquivo **text.txt** do servidor é enviado para o cliente criando ou atualizando um arquivo já existente.

5. Resultados obtidos com as simulações

As simulações desse projeto limitaram-se ao corrompimento de dados e perda de dados através de uma regra de probabilidade de 0 a 100 parametrizada nos arquivos. Essa probabilidade é calculada para cada tentativa de transmissão para que a simulação seja a mais realista possível.

As outras funcionalidades do protocolo especificadas estão funcionando de forma fiel, através da conexão socket entre cliente e servidor. Todas as transmissões de dados acontecem localmente em IPs e portas configuradas previamente.

6. Análise e discussão sobre os resultados

Com base na implementação realizada das versões, concluímos que a versão do protocolo 3.0 é funcionalmente correta, mas é improvável que alguém fique feliz com seu desempenho, particularmente nas redes de alta velocidade de hoje. No cerne do problema de desempenho da versão 3.0 está o fato de que ele é um protocolo de “parar e esperar”.