INSTITUTO DO EMPREGO
E FORMAÇÃO PROFISSIONAL

RELATÓRIO

Projecto3 - TFTPy

Síntese

Este projecto é uma aplicação de transferência de ficheiros com base no Trivial File Transfer Protocol (TFTP), para isso será usado o Python 3 e sockets para a comunicação em rede entre servidor e cliente.

Tiago Domingos

TEGRSI07 – UFCD5119

Projecto 3 – TFTPy: Cliente TFTP

O TFTP define 5 tipos de pacotes ou mensagens:

RRQ: Read Request - Opcode 1

WRQ: Write Request - Opcode 2

DAT: Data transfer - Opcode 3

ACK: Acknolwedge - Opcode 4

ERR: Error - Opcode 5

Estrutura de RRQ/WRQ entre o cliente e o servidor:

```
#     Cliente    #     Servidor   #
```

1. Requisição de Leitura

```
|  ---------------------> |
|       ACK 0             |
| <----------------------  |
```

2. Dados do Bloco 1

```
| ----------------------> |
|       ACK 1             |
| <----------------------  |
```

3. Dados do Bloco 2

```
| ----------------------> |
|       ACK 2             |
| <----------------------  |
```

O cliente envia uma solicitação de leitura (RRQ - Read Request) para o servidor, especificando o nome do ficheiro que deseja transferir.

O servidor responde com um pacote de confirmação de ACK 0 (Acknowledgment 0), indicando que está pronto para começar a enviar os dados.

O cliente responde com um pacote de ACK 1, confirmando a recepção do primeiro bloco de dados.

O servidor envia os dados do Bloco 1 para o cliente.

O cliente responde com um pacote de ACK 2, confirmando a recepção do segundo bloco de dados.

Esse processo continua até que todos os blocos de dados tenham sido enviados e confirmados pelo cliente.

Este é um exemplo simplificado do fluxo de comunicação entre um cliente e um servidor TFTP. É importante observar que o TFTP não fornece autenticação ou criptografia por padrão, portanto, não é recomendado para transferências de arquivos sensíveis. No entanto, devido à

sua simplicidade, ele é amplamente utilizado em cenários de rede onde a eficiência é mais importante do que a segurança.

O estado actual do meu ficheiro tftp.py é:

```
"""
This module handles all TFTP related "stuff": data structures, packet
definitions, methods and protocol operations.

Data de entrega 14/07/2023

(C) João Galamba && Tiago Domingos, 2023
"""

import ipaddress
import re
import struct
import string
from socket import (
    socket,
    herror,
    gaierror,
    gethostbyaddr,
    gethostbyname_ex,
    AF_INET, SOCK_DGRAM,
)

##############################################################################
##
##      PROTOCOL CONSTANTS AND TYPES
##
##############################################################################

MAX_DATA_LEN = 512              # bytes
MAX_BLOCK_NUMBER = 2** 16 - 1   # 0..65535
INACTIVITY_TIMEOUT = 25.0       # segs
DEFAULT_MODE = 'octet'
DEFAULT_BUFFER_SIZE = 8192      # bytes

# TFTP message opcodes
# RRQ, WRQ, DAT, ACK, ERR = range(1, 6)
```

```
RRQ = 1   # Read Request
WRQ = 2   # Write Request
DAT = 3   # Data transfer
ACK = 4   # Acknowledge DAT
ERR = 5   # Error packet; what the server responds if a read/write
        # can't be processed, read and write errors during file
        # transmission also cause this message to be sent, and
        # transmission is then terminated. The error number gives a
        # numeric error code, followed by an ASCII error message that
        # might contain additional, operating system specific
        # information.

ERR_NOT_DEFINED = 0
ERR_FILE_NOT_FOUND = 1
ERR_ACCESS_VIOLATION = 2
ERR_DISKFULL_OR_ALLOCEXCEEDED = 3
ERR_ILLEGAL_TFTP_OP = 4
ERR_UNKNOWN_TRANSFER_ID = 5
ERR_FILE_ALREADY_EXISTS = 6
ERR_NO_SUCH_USER = 7

ERROR_MESSAGES = {
   ERR_NOT_DEFINED: 'Not defined, see error message (if any)',
   ERR_FILE_NOT_FOUND: 'File not found',
   ERR_ACCESS_VIOLATION: 'Access violation',
   ERR_DISKFULL_OR_ALLOCEXCEEDED: 'Disk full or allocation exceeded',
   ERR_ILLEGAL_TFTP_OP: 'Illegal TFTP Operation',
   ERR_UNKNOWN_TRANSFER_ID: 'Unknown transfer ID',
   ERR_FILE_ALREADY_EXISTS: 'File already exists',
   ERR_NO_SUCH_USER: 'No such user'
}

INET4Address = tuple[str, int]      # TCP/UDP address => IPv4 and port

##################################################################
##
##      SEND AND RECEIVE FILES
##
##################################################################
```

```
def get_file(server_addr: INET4Address, filename: str, dest_file: str = None):
    """
    Get the remote file given by `filename` through a TFTP RRQ
    connection to remote server at `server_addr`.
    """

    # 1. Criar um socket para o servidor em server_addr
    # 2. Abrir ficheiro para escrita binária
    # 3. Enviar pacote RRQ para o servidor
    # 4. Esperar por pacote enviado pelo servidor [1]
    #       4.1 Recebemos pacote.
    #       4.2 Se pacote for DAT:
    #           a) Obter block_number e data (ie, o bloco de dados)
    #           b) Se block_number não for next_block_number ou
    #              next_block_number - 1 [2] => ERRO de protocolo
    #           c) Se block_number == next_block_number [3], gravamos
    #              bloco de dados no ficheiro e incrementamos contador
    #           d) Enviar ACK reconhecendo o último pacote recebido
    #           e)
    #       4.3 Se pacote for ERR: assinalar o erro lançando a excepção apropriada
    #       4.4 Se for outro tipo de pacote: assinalar ERRO de protocolo
    #
    # [1] Terminar quando dimensão do bloco de dados do pacote
    #   DAT for < 512 bytes
    # [2] next_block_number indica o próximo block_number, contador
    #       inicializado a 1 antes do passo 4.
    # [3] Recebemos novo DAT
#:
    with socket(AF_INET, SOCK_DGRAM) as sock:
        sock.settimeout(INACTIVITY_TIMEOUT)
        rrq = pack_rrq(filename)
        sock.sendto(rrq, server_addr)
        next_block_num = 1
        save_file = dest_file if dest_file is not None else filename
        with open(save_file, 'wb') as file:
            while True:
                packet = sock.recvfrom(DEFAULT_BUFFER_SIZE)
                opcode = unpack_opcode(packet)
```

```
            if opcode == DAT:
                block_number, data = unpack_dat(packet)
                if block_number not in (next_block_num, next_block_num - 1):
                    raise ProtocolError(f'Unexpected block number: {block_number}')

                if block_number == next_block_num:
                    file.write(data)
                    next_block_num += 1

                ack = pack_ack(block_number)
                sock.sendto(ack, server_addr)

                if len(data) < MAX_DATA_LEN:
                    break;
            #:
            elif opcode == ERR:
                error_code, error_msg = unpack_err(packet)
                raise Err(error_code, error_msg)
            #:
            else:
                raise ProtocolError(f'Invalid opcode {opcode}')
            #:
        #:
    #:
#:

def put_file(server_addr: INET4Address, filename: str, dest_file: str = None):
    with socket(AF_INET, SOCK_DGRAM) as sock:
        sock.settimeout(INACTIVITY_TIMEOUT)
        wrq = pack_wrq(filename)
        sock.sendto(wrq, server_addr)
        block_num = 0
        save_file = dest_file if dest_file is not None else filename
        with open(save_file, 'rb') as file:
            while True:
                packet = sock.recvfrom(DEFAULT_BUFFER_SIZE)
                opcode = unpack_opcode(packet)
```

```
            if opcode == ACK:
                received_block_number = unpack_ack(packet)
                if received_block_number not in (block_num):
                    raise ProtocolError(f'Unexpected block number: {received_block_number}')

                data = file.read(MAX_DATA_LEN)
                packet = pack_dat(block_num, data)
                sock.sendto(packet, server_addr)

                if len(data) < MAX_DATA_LEN:
                    break

                block_num += 1

            elif opcode == ERR:
                error_code, error_msg = unpack_err(packet)
                raise ERR(error_code, error_msg)

            else:
                raise ProtocolError(f'Invalid opcode {opcode}')
#:

##################################################################
##
##      PACKET PACKING AND UNPACKING
##
##################################################################

def pack_rrq(filename: str, mode: str = DEFAULT_MODE) -> bytes:
    return pack_rrq_wrq(RRQ, filename, mode)
#:

def unpack_rrq(packet: bytes) -> tuple[str, str]:
    return unpack_rrq_wrq(packet)
#:

def pack_wrq(filename: str, mode: str = DEFAULT_MODE) -> bytes:
    return pack_rrq_wrq(WRQ, filename, mode)
```

```
#:

def unpack_wrq(packet: bytes) -> tuple[str, str]:
    return unpack_rrq_wrq(packet)
#:

def pack_rrq_wrq(opcode: int, filename: str, mode: str) -> bytes:
    encoded_filename = filename.encode() + b'\x00'
    encoded_mode = mode.encode() + b'\x00'
    rrq_fmt = f'!H{len(encoded_filename)}s{len(encoded_mode)}s'
    return struct.pack(rrq_fmt, opcode, encoded_filename, encoded_mode)
#:

def unpack_rrq_wrq(packet: bytes) -> tuple[str, str]:
    delim = packet.index(b'\x00', 2)
    filename = packet[2:delim].decode()
    mode = packet[delim + 1:-1].decode()
    return (filename, mode)
#:

def pack_dat(block_number: int, data: bytes) -> bytes:
    if not 0 <= block_number <= MAX_BLOCK_NUMBER:
        raise ValueError(f'Invalid block number: {block_number}')
    if len(data) > MAX_DATA_LEN:
        raise ValueError(f'Invalid data length: {len(data)}')

    fmt = f'HH{len(data)}s'
    return struct.pack(fmt, DAT, block_number, data)
#:

def unpack_dat(packet: bytes) -> tuple[int, bytes]:
    opcode, block_number = struct.unpack('!HH', packet[:4])
    if opcode != DAT:
        raise ValueError(f'Invalid opcode: {opcode}')
    return block_number, packet[4:]
#:

def pack_ack(block_number: int) -> bytes:
    if not 0 <= block_number <= MAX_BLOCK_NUMBER:
```

```
        raise ValueError(f'Invalid block number: {block_number}')
    return struct.pack('!HH', ACK, block_number)
#:

def unpack_ack(packet: bytes) -> int:
    if len(packet) > 4:
        raise ValueError(f'Invalid packet length: {len(packet)}')
    return struct.unpack('!HH', packet[2:4])[0]
#:

def pack_err(error_code: int, error_msg: str) -> bytes:
    if error_code not in ERROR_MESSAGES:
        raise ValueError(f'Unknown error code {error_code}')

    encoded_error_msg = error_msg.encode() + b'\x00'
    fmt =f'!HH{len(encoded_error_msg)}s'
    return struct.pack(fmt, ERR, error_code, encoded_error_msg)
#:

def unpack_err(packet: bytes) -> tuple[int, str]:
    fmt = f'!HH{len(packet)-4}s'
    opcode, error_num, error_msg = struct.unpack(fmt, packet)
    if opcode != ERR:
        raise ValueError(f'Invalid opcode: {opcode}')
    return error_num, error_msg[:-1].decode()
#:

def unpack_opcode(packet: bytes) -> int:
    opcode = struct.unpack('!H', packet[0:2])
    if opcode in (RRQ, WRQ, DAT, ACK, ERR):
        raise ValueError(f"Invalid opcode {opcode}")
    return opcode[0]
#:

# TODO: pack_dat, unpack_dat, pack_ack, unpack_ack, pack_err, unpack_err

##########################################################################
##
##      ERRORS AND EXCEPTIONS
```

```
##
#####################################################################

class NetworkError(Exception):
    """
    Any network error, like "host not found", timeouts, etc.
    """



class ProtocolError(NetworkError):
    """
    A protocol error like unexpected or invalid opcode, wrong block
    number, or any other invalid protocol parameter.
    """


class Err(Exception):
    """
    An error sent by the server. It may be caused because a read/write
    can't be processed. Read and write errors during file transmission
    also cause this message to be sent, and transmission is then
    terminated. The error number gives a numeric error code, followed
    by an ASCII error message that might contain additional, operating
    system specific information.
    """


    def __init__(self, error_code: int, error_msg: str):
        super().__init__(f'TFTP Error {error_code}')
        self.error_code = error_code
        self.error_msg = error_msg
    #:
#:

#############################################################################
####
##
##      COMMON UTILITIES
##      Mostly related to network tasks
##
#############################################################################
####
```

```
def _make_is_valid_hostname():
    allowed = re.compile(r"(?!-)[A-Z\d-]{1,63}(?<!-)$", re.IGNORECASE)
    def _is_valid_hostname(hostname):
        """
        From: http://stackoverflow.com/questions/2532053/validate-a-hostname-string
        See also: https://en.wikipedia.org/wiki/Hostname (and the RFC
        referenced there)
        """
        if len(hostname) > 255:
            return False
        if hostname[-1] == ".":
            # strip exactly one dot from the right, if present
            hostname = hostname[:-1]
        return all(allowed.match(x) for x in hostname.split("."))
    return _is_valid_hostname
#:
is_valid_hostname = _make_is_valid_hostname()


def get_host_info(server_addr: str) -> tuple[str, str]:
    """
    Returns the server ip and hostname for server_addr. This param may
    either be an IP address, in which case this function tries to query
    its hostname, or vice-versa.
    This functions raises a ValueError exception if the host name in
    server_addr is ill-formed, and raises NetworkError if we can't get
    an IP address for that host name.
    TODO: refactor code...
    """
    try:
        ipaddress.ip_address(server_addr)
    except ValueError:
        # server_addr not a valid ip address, then it might be a
        # valid hostname
        # pylint: disable=raise-missing-from
        if not is_valid_hostname(server_addr):
            raise ValueError(f"Invalid hostname: {server_addr}.")
        server_name = server_addr
```

```
        try:
            # gethostbyname_ex returns the following tuple:
            # (hostname, aliaslist, ipaddrlist)
            server_ip = gethostbyname_ex(server_name)[2][0]
        except gaierror:
            raise NetworkError(f"Unknown server: {server_name}.")
    else:
        # server_addr is a valid ip address, get the hostname
        # if possible
        server_ip = server_addr
        try:
            # returns a tuple like gethostbyname_ex
            server_name = gethostbyaddr(server_ip)[0]
        except herror:
            server_name = ''
    return server_ip, server_name
#:

def is_ascii_printable(txt: str) -> bool:
    return set(txt).issubset(string.printable)
    # ALTERNATIVA: return not set(txt) - set(string.printable)
#:
```

Do client.py é:

```
"""
TFTPy - This module implements an interactive and command line TFTP
client.

This client accepts the following options:
    $ python3 client.py (get|put) [-p serv_port] server source_file [dest_file]
    $ python3 client.py [-p serv_port] server

    Data de entrega 14/07/2023

(C) João Galamba && Tiago Domingos, 2023
"""
# import docopt
import sys
from tftp import *
```

```
def main():
    if len(sys.argv) < 2:
        print("Comando inválido. Use 'get', 'put', ou 'tftf'.")
        sys.exit(1)

    comando = sys.argv[1]

    if comando == 'get':
        if len(sys.argv) < 4:
            print("Usage: python3 client.py get [-p serv_port] server source_file [dest_file]")
            sys.exit(1)

        server = sys.argv[2]
        source_file = sys.argv[3]
        dest_file = sys.argv[4] if len(sys.argv) >= 5 else None
        serv_port = int(sys.argv[3][3:]) if len(sys.argv) >= 5 and sys.argv[3].startswith('-p') else
69
        server_addr = (server, serv_port)

        try:
            get_file(server_addr, source_file, dest_file)
        except NetworkError as e:
            print(f"Error reaching the server '{server}' ({e}).")
        except ProtocolError as e:
            print(f"Protocol error: {e}")

    elif comando == 'put':
        if len(sys.argv) < 4:
            print("Usage: python3 client.py put [-p serv_port] server source_file [dest_file]")
            sys.exit(1)

        server = sys.argv[2]
        source_file = sys.argv[3]
        dest_file = sys.argv[4] if len(sys.argv) >= 5 else None
        serv_port = int(sys.argv[3][3:]) if len(sys.argv) >= 5 and sys.argv[3].startswith('-p') else
69
        server_addr = (server, serv_port)

        try:
```

```
            put_file(server_addr, source_file, dest_file)
        except NetworkError as e:
            print(f"Error reaching the server '{server}' ({e}).")
        except ProtocolError as e:
            print(f"Protocol error: {e}")

    elif comando == 'tftp':
        while True:
            print("TFTPy - Cliente de TFTP")
            opcao = input('TFTP > ')
            if opcao in ('get'):
                print("Receber ficheiro - get ficheiro_remoto [ficheiro_local]")
                # get ficheiro_remoto [ficheiro_local]
                server_addr = input("Server > ")
                source_file = input("Ficheiro fonte > ")
                dest_file = input("Opcional destino do ficheiro > ")
                try:
                    get_file(server_addr, source_file, dest_file)
                except NetworkError as e:
                    print(f"Error reaching the server '{server}' ({e}).")
                except ProtocolError as e:
                    print(f"Protocol error: {e}")

            elif opcao in ('put'):
                print("Enviar ficheiro - put ficheiro_local [ficheiro_remoto]")
                # put ficheiro_local [ficheiro_remoto]
                server_addr = input("Server > ")
                source_file = input("Ficheiro fonte > ")
                dest_file = input("Opcional destino do ficheiro > ")
                try:
                    put_file(server_addr, source_file, dest_file)
                except NetworkError as e:
                    print(f"Error reaching the server '{server}' ({e}).")
                except ProtocolError as e:
                    print(f"Protocol error: {e}")
            elif opcao in ('help', '?'):
                ajuda()
            elif opcao in 'quit':
                print("Exiting TFTP client.")
```

```
                print("Goodbye!")
                sys.exit(0);
            else:
                print(f"Opção {opcao} inválida")

        #elif opcao.upper() in ('HELP', 'AJUDA', '?'):
        #    ajuda()

        else:
            print(f"Opção {comando} inválida")
        #    print(ajuda())
#:

def ajuda():
    print("Commands:")
    print("  get remote_file [local_file] - get a file from server and save it as local_file")
    print("  get remote_file [local_file] - send a file to server and store it as remote_file")
    print("  dir                          - obtain a listing of remote files (not implementable)")
    print("  quit                         - exit TFTP client")

if __name__ == '__main__':
    #arguments = docopt(__doc__, version='TFTPy Python - 1')
    #print(arguments)
    main()

#:
```

Do servidor é este e está muito incompleto:

```
"""
```

Aqui é o módulo do servidor e faz a gestão do mesmo.

Data de entrega 14/07/2023

(C) Tiago Domingos, 2023

```
"""

from socketserver import BaseRequestHandler, ThreadingUDPServer
from socket import socket, AF_INET, SOCK_DGRAM
import time
```

```
class TimeHandler(BaseRequestHandler):
    def handle(self):
        # Obter mensagem e um socket cliente
        msg = self.request[0]
        print('Got request', msg, 'from', self.client_address)
        resp = time.ctime()

        # Criar outro socket com um porto efémero para a resposta
        with socket(AF_INET, SOCK_DGRAM) as sock:
            sock.sendto(resp.encode(), self.client_address)

if __name__ == '__main__':
    # Threading server permite servir múltiplos pedidos ao mesmo tempo
    ThreadingUDPServer.allow_reuse_address = True
    serv = ThreadingUDPServer(('localhost'), 20022), TimeHandler)
    print('Starting server...', serv)
    serv.serve_forever()
```

Conclusão

Infelizmente ainda não consegui fazer funcionar o tftpy mas penso que está no bom caminho. Pelo menos o cliente interactivo está implementado excepto o teste para ver se se pode transferir ficheiros entre servidor e cliente. Acrescentei também as mensagens de erro que faltavam. Vou fazer o pull de todo o código fonte para o meu github e incluir este relatório.

Apesar de alguma dificuldade com a password do git para fazer upload do projecto, depois de seguir o que o professor disse consegui com uma key gerada anteriormente durante uma das aulas.

Gestão de Código Fonte em Git para o Github

A gestão de código fonte é feita através de git e guardado num repositório no github como se vê em seguida:

https://github.com/TiagoDomingos/Tegrsi07TFTP

Através deste URL podemos aceder ao projecto sendo que pode ser feito download para outro PC.

Docstring do cliente.py:

"""

TFTPy - This module implements an interactive and command line TFTP client.

This client accepts the following options:

$ python3 client.py (get|put) [-p serv_port] server source_file [dest_file]

$ python3 client.py [-p serv_port] server

Data de entrega 14/07/2023

(C) João Galamba && Tiago Domingos, 2023

"""

E também do tftp.py:

"""

This module handles all TFTP related "stuff": data structures, packet definitions, methods and protocol operations.

Data de entrega 14/07/2023

(C) João Galamba && Tiago Domingos, 2023

"""

Anexo I:

O protocolo UDP (User Datagram Protocol) é um protocolo de transporte na camada de transporte do modelo TCP/IP. Ao contrário do TCP (Transmission Control Protocol), o UDP é um protocolo de transporte orientado a conexão não confiável. Aqui está uma breve descrição do UDP em comparação com o TCP:

Conexão e orientação a datagramas: O UDP não estabelece uma conexão antes da transmissão de dados, ao contrário do TCP que estabelece uma conexão ponto a ponto. O UDP é um protocolo de entrega de datagramas independentes, onde cada datagrama é tratado separadamente e não há garantias de que eles serão entregues corretamente ou na ordem correta.

Confiabilidade: O UDP não fornece mecanismos de controle de congestionamento, retransmissão de pacotes perdidos, reordenação de pacotes ou controle de fluxo. Isso significa que não há garantia de entrega confiável dos dados, e os pacotes podem ser perdidos, duplicados ou entregues fora de ordem.

Overhead: O UDP possui um menor overhead em comparação com o TCP. Isso ocorre porque o UDP não possui os mecanismos complexos de controle de fluxo, retransmissão e reordenação do TCP. Portanto, o UDP é mais eficiente em termos de recursos de rede e processamento.

Velocidade e latência: Como o UDP não possui os mecanismos de controle de fluxo e retransmissão do TCP, ele é geralmente mais rápido em termos de velocidade e tem menor latência. No entanto, isso também significa que o UDP é menos confiável em ambientes de rede com perda de pacotes ou congestionamento.

Aplicações adequadas: O UDP é frequentemente utilizado em situações onde a entrega rápida dos dados é mais importante do que a confiabilidade, como transmissões de áudio e vídeo em tempo real, streaming de mídia, jogos online e serviços de DNS (Domain Name System).

É importante observar que a escolha entre TCP e UDP depende do tipo de aplicação e dos requisitos específicos de entrega de dados. O TCP é mais adequado para cenários que exigem uma entrega confiável e ordenada de dados, enquanto o UDP é preferível para situações em que a velocidade e a latência são mais importantes do que a confiabilidade.