2024-2학기 네트워크 프로그래밍 기말 과제 보고서

TFTP 프로토콜을 이용한 파일 전송 클라이언트 구현

2024. 12. 19.

성명: 박주민

Github URL: https://github.com/zo0mini

I. TFTP 프로토콜 개요

TFTP(Trivial File Transfer Protocol)는 UDP(User Datagram Protocol)를 기반하는 단순한 파일 전송 프로토콜입니다. 이 프로토콜은 TCP/IP 프로토콜 스택의 응용 계층에서 동작하며, 최소한의 기능만을 제공하여 효율적인 파일 전송을 가능하게 합니다.

TFTP의 주요 특징은 비연결형 통신 방식입니다. 이로 인해 데이터 전송 시 발생하는 오버헤드가 적어 전송 속도가 빠르다는 장점이 있습니다. 그러나 이러한 구조적 특성으로 인해 데이터 전송의 신뢰성과 보안성이 상대적으로 낮다는 한계를 가지고 있기도 합니다.

TFTP는 주로 네트워크 부팅 환경이나 임베디드 시스템에서 활용됩니다. 특히 PXE 부팅 시스템에서 운영체제 이미지를 전송하거나, 네트워크 장비의 펌웨어 업데이트 과정에서 자주 사용됩니다. 그러나 보안 기능이 부재하고 신뢰성이 부족하여, 중요한 데이터를 전송하는 환경에서는 적합하지 않습니다.

II. 프로그램 구현 및 분석

1. 프로그램 개요

TFTP(Trivial File Transfer Protocol)는 UDP(User Datagram Protocol) 기반의 파일 전송 프로토콜입니다. 본 프로그램은 TFTP 클라이언트를 구현하여 서버와의 파일 송수신 기능을 제공합니다. TFTP는 TCP와 비교하여 낮은 오버헤드와 빠른 전송 속도를 특징으로 하며, 주로 네트워크 부팅(PXE)이나 임베디드 시스템과 같이 리소스가 제한적인 환경에서 활용됩니다.

2. 구현 기능

본 프로그램은 다음과 같은 주요 기능을 구현하였습니다.

- 파일 다운로드(get): 서버로부터 지정된 파일을 다운로드
- 파일 업로드(put): 로컬 시스템의 파일을 서버로 업로드
- 데이터 송수신: RRQ(Read Request) 및 WRQ(Write Request) 처리
- 응답 처리: 데이터 블록 수신에 대한 ACK 처리
- 오류 제어: 타임아웃 및 재전송 메커니즘 구현

3. 프로그램 사용법

프로그램의 기본 사용법은 다음과 같습니다.

파일 다운로드(get):

```
bash
python3 tftp_client.py <서버 IP> get <파일명> [-p <포트번호>]
```

파일 업로드(put):

```
bash
python3 tftp_client.py <서버 IP> put <파일명> [-p <포트번호>]
```

4. 주요 구현 내용

4.1 소켓 통신 구현

```
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.settimeout(TIMEOUT)  # 타임아웃 설정
```

UDP 통신을 위한 소켓을 생성하고 타임아웃을 설정하여 안정적인 통신 보장.

4.2 파일 요청 처리

```
def send_request(opcode, filename, mode, server_address):
    # TFTP 요청 메시지를 패킹하여 서버로 전송
    # opcode: 요청 유형 (RRQ/WRQ), filename: 파일 이름,
              mode: 전송 모드, server_address: 서버 주소
    format = f'>h{len(filename)}sB{len(mode)}sB'
    request_message = pack(format, opcode, bytes(filename, 'utf-8'),
                           0, bytes(mode, 'utf-8'), 0)
    sock.sendto(request_message, server_address)
```

파일 전송 요청 시 TFTP 프로토콜 명세에 따라 메시지를 구성하여 전송.

4.3 데이터 수신 및 응답

```
# 서버로부터 최대 516byte 크기의 데이터를 수신
data, server_new_socket = sock.recvfrom(516)

# 수신한 데이터의 Opcode가 'DATA'인지 확인
```

```
if opcode == OPCODE['DATA']:
    # 수신한 데이터에서 블록 번호를 추출 (데이터의 2번째부터 4번째 바이트)
    block_number = int.from_bytes(data[2:4], 'big')

    # 예상된 블록 번호와 일치하는지 확인
    if block_number == expected_block_number:
        # 일치하면 해당 블록 번호에 대해 ACK 패킷을 서버로 전송
        send_ack(block_number, server_new_socket)
```

수신된 데이터의 무결성을 확인하고 적절한 응답을 전송.

4.4 파일 다운로드(get) 작업

```
if operation == "get": # 읽기 요청(RRQ) 전송
    send_request(OPCODE['RRQ'], filename, mode, server_address)
    file = open(filename, 'wb')  # 파일 열기 (쓰기 모드)
    expected_block_number = 1  # 첫 번째 블록 번호 예상
    while True:
        try:
            data, server_new_socket = sock.recvfrom(516)  # 데이터 수신
            opcode = int.from_bytes(data[:2], 'big')  # Opcode 추출
            if opcode == OPCODE['DATA']:
                # 블록 번호 추출
                block_number = int.from_bytes(data[2:4], 'big')
                # 예상된 블록 번호와 일치하는지 확인
                if block_number == expected_block_number:
                    send_ack(block_number, server_new_socket)
                    file_block = data[4:]  # 데이터 블록 추출
                    file.write(file_block)  # 파일에 데이터 쓰기
                    expected_block_number += 1  # 다음 블록 번호 예상
                if len(file_block) < BLOCK_SIZE:  # 마지막 블록인지 확인
                    print("File transfer completed.")
                    break  # 파일 전송 완료
            elif opcode == OPCODE['ERROR']:  # 에러 발생 시
                error_code = int.from_bytes(data[2:4], byteorder='big')
                print(f"Error: {ERROR_CODE.get(error_code, 'Unknown error')}")
                file.close()
                os.remove(filename)  # 부분적으로 받은 파일 삭제
                break  # 에러 종료
        except socket.timeout:  # 타임아웃 시 재전송 요청
```

```
            print("Timeout occurred. Retrying...")
            send_request(OPCODE['RRQ'], filename, mode, server_address)
    file.close()
```

서버에 읽기 요청 후 데이터를 받아 파일에 저장하고, 전송 완료 시 종료.

4.5 파일 업로드(put) 작업

```
elif operation == "put":
    # 쓰기 요청(WRQ) 전송
    send_request(OPCODE['WRQ'], filename, mode, server_address)
    file = open(filename, 'rb')  # 파일 열기 (읽기 모드)
    block_number = 0  # 블록 번호 초기화
    while True:
        file_block = file.read(BLOCK_SIZE)  # 파일에서 데이터 읽기
        block_number += 1
        # 데이터 패킷 생성
        data_packet = pack('>hh', OPCODE['DATA'], block_number) + file_block
        sock.sendto(data_packet, server_address)  # 데이터 전송
        try:
            ack, _ = sock.recvfrom(4)  # ACK 수신
            ack_opcode = int.from_bytes(ack[:2], 'big')
            ack_block_number = int.from_bytes(ack[2:4], 'big')
            if ack_opcode == OPCODE['ACK'] and ack_block_number ==
block_number:  # ACK 확인
                if len(file_block) < BLOCK_SIZE:  # 마지막 블록인지 확인
                    print("File upload completed.")
                    break  # 파일 업로드 완료
        except socket.timeout:  # 타임아웃 시 블록 재전송
            print("Timeout occurred. Resending block...")
            sock.sendto(data_packet, server_address)
    file.close()
```

서버에 쓰기 요청 후 파일을 전송하고, ACK를 받아 전송 완료.

III. 프로젝트 평가

1. 느낀 점

이번 프로젝트에서 TFTP 클라이언트를 직접 구현하면서, UDP 프로토콜의 특징을 몸소 체험할 수 있었습니다. 수업 시간에 배웠던 UDP의 비연결성이나 신뢰성 없는 전송 방식이 실제로 어떤 의미인지, 그리고 이를 보완하기 위해 어떤 처리가 필요한지 확실히 알게 되었습니다.

특히 파일 전송 중에 발생하는 패킷 손실이나 타임아웃 상황을 처리하면서, 네트워크 프로그래밍에서 예외 처리가 얼마나 중요한지 깨달았습니다. TCP를 사용했다면 자동으로 처리됐을 많은 부분들을 직접 구현하면서, 전송 계층 프로토콜의 역할과 중요성도 더 깊이 이해하게 되었습니다.

처음에는 단순해 보였던 TFTP 프로토콜이 실제 구현 과정에서는 꽤 복잡한 고려사항들이 필요하다는 점이 인상 깊었습니다. 데이터 블록의 순서를 관리하고, ACK 응답을 처리하고, 타임아웃이 발생했을 때 적절히 재전송하는 과정을 구현하면서 네트워크 프로그래밍의 실제를 경험할 수 있었습니다.

2. 개선할 점

- 파일 전송 중 네트워크 끊김 현상에 대한 더 나은 복구 방법 구현
- 블록 크기를 네트워크 상황에 맞게 조절하는 기능 추가
- 사용자가 전송 진행 상황을 더 쉽게 확인할 수 있는 인터페이스 개발
- 여러 번의 재전송 실패 시 효과적인 오류 처리 방안 마련

이번 과제를 통해 실제 네트워크 환경에서의 프로그래밍이 교과서와는 또 다른 차원의 문제들을 포함하고 있다는 것을 배웠습니다. 이러한 경험은 앞으로 더 복잡한 네트워크 프로그래밍을 하는 데 있어 큰 도움이 될 것 같습니다.