CS2911 – Trivial File Transfer Protocol (TFTP)

1 Introduction

The Trivial File Transfer Protocol (TFTP) was developed to be a simple protocol for serving files to clients. It is like the File Transfer Protocol (FTP) but has fewer and simpler protocol messages. You might see TFTP used today for computers that boot from the network, also known as PXE (Pre-Execution Environment) boot. While not a required test for this lab, with a working TFTP server, you should be able to use it to boot another computer. The goal of this lab is to create a server in Python that will allow clients to download files via the Trivial File Transfer Protocol (TFTP).

2 References

The full details of TFTP as well as TFTP message formats sent between client and server are in RFC-1350:

https://tools.ietf.org/html/rfc1350

3 The Exercise

For this lab you will be writing the Python code for a TFTP server. The server will handle requests sent from a TFTP client. The first step is to install the TFTP client on your windows computer. To do this follow the following steps:

1. Click on the “Start” menu or press the Windows key and search for “Control Panel” (You can also open search using Windows Key + S)
2. Click on “Programs” -> “Programs and Features” and then click on “Turn Windows features on or off”
3. Scroll down and check the box labeled “TFTP Client” then click “OK”
4. The installer will run and install the TFTP client software

To use the TFTP client, open the Windows command prompt (you can do this by pressing the Windows Key + R, then typing ‘cmd’, and pressing enter). From the Windows command prompt you can run the TFTP client using the “tftp” command. If you run the command without any parameters, you will see the help text for the command:

```
C:\>tftp

Transfers files to and from a remote computer running the TFTP service.
TFTP [-i] host [GET | PUT] source [destination]

  -i              Specifies binary image transfer mode (also called
                  octet). In binary image mode the file is moved
                  literally, byte by byte. Use this mode when
                  transferring binary files.
  host            Specifies the local or remote host.
  GET             Transfers the file destination on the remote host to
                  the file source on the local host.
  PUT             Transfers the file source on the local host to
                  the file destination on the remote host.
  source          Specifies the file to transfer.
  destination     Specifies where to transfer the file.
```

The command says to retrieve a file you have to specify the host name for the TFTP server, use GET, and finally the name of the file you want to retrieve. For example, to get a file named ‘myfile.txt’ from the server mytftp.msoe.edu you would type:

```
C:\>tftp mytftp.msoe.edu GET myfile.txt
```

The TFTP client also supports 'PUT' which is used to send a file to be stored on the server. The TFTP server that you will write for this lab only has to support 'GET' to retrieve a file.

TFTP uses User Datagram Protocol (UDP) as the transport layer so you will need to use a DGRAM socket in your Python code. A TFTP server must send and receive data on port 69.

The TFTP RFC defines 5 messages used by TFTP.

1. Read Request
2. Write Request
3. Data Message
4. Acknowledgement Message
5. Error Message

The first message that the server receives is a request (either a read or a write). Your server will only have to support read requests. The RFC defines a request as follows:

```
 2 bytes     string    1 byte     string   1 byte
 ------------------------------------------------
| Opcode |  Filename  |   0  |    Mode    |   0  |
 ------------------------------------------------
```

The operation code (Opcode) for a request is a 2-byte integer represented in big endian. The opcode for a read request is 1. The Filename tells the server which file to read (GET). It is represented as an ASCII string and ends with a zero byte (null byte). The Mode represents how the file is to be transferred. It is represented as an ASCII string and ends with a zero byte (null byte). Your server will not need to use this value. Instead all files will be sent in binary. You will need to parse it from the message, however.

Once the request is parsed, the TFTP server sends a file to a client by sending blocks of data. The TFTP server breaks a file up into pieces called blocks. Each block is 512 bytes of data except for the last block which is the rest of the file after the last 512-byte block. For example, if you have a file that is 1423 bytes long, you would have 3 data blocks: two would be 512 bytes long and the last would be 399 bytes long (512 + 512 + 399 = 1423 bytes).

Data blocks are sent to the client one at a time using a data message which is formatted as follows:

```
 2 bytes     2 bytes      n bytes
 ----------------------------------
| Opcode |   Block #  |   Data     |
 ----------------------------------
```

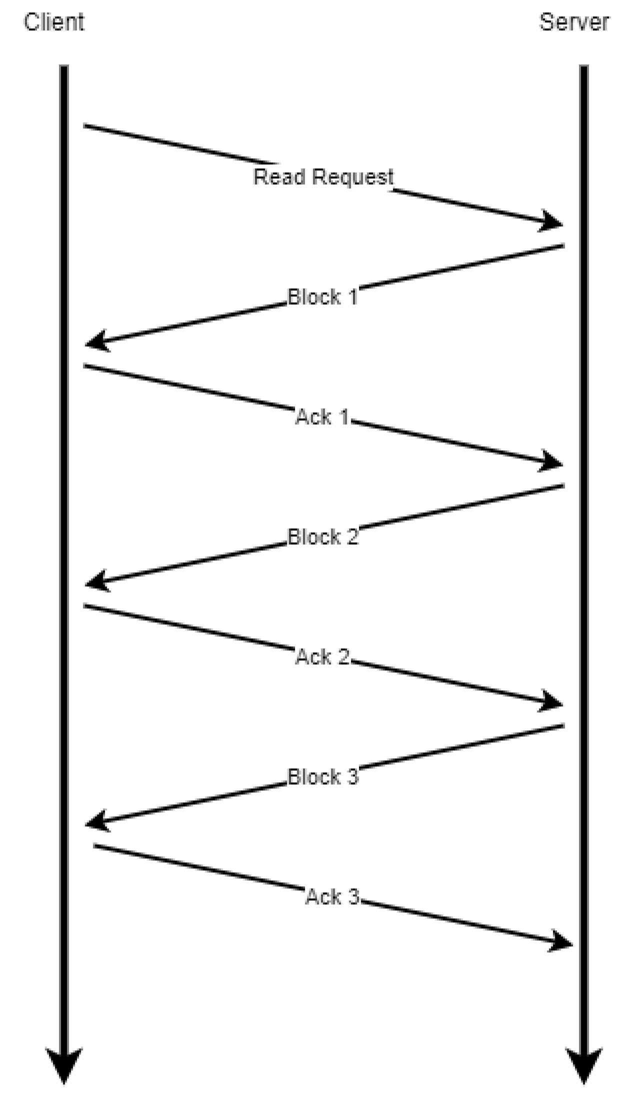
The operation code (Opcode) for a data message is a 2-byte integer represented in big endian. The opcode for a data message is 3. All data blocks sent to the client will use the opcode 3. The block number is a 2-byte integer represented in big endian. The block number tells the client which block of data is currently being sent (either 1, 2, or 3 in the example above). Data blocks start at 1 and count until the last block.

Once the server sends the first data block to the client, it must wait for an acknowledgement. An acknowledgement is sent from the client using the following message format:

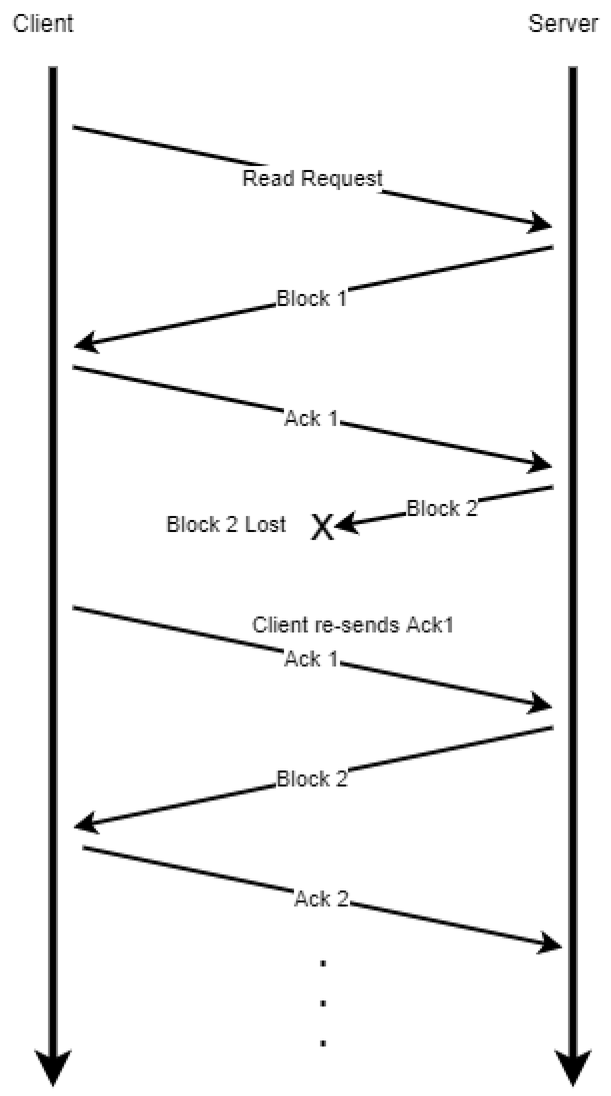
```
 2 bytes     2 bytes
 ---------------------
| Opcode |   Block #  |
 ---------------------
```

The operation code (Opcode) for an acknowledgement message is a 2-byte integer represented in big endian. The opcode for a acknowledgement message is 4. The block number is a 2-byte integer represented in big endian. In an acknowledgement message, the block number indicates which block is being acknowledged. For example, if the server sends block 1 to the client, once the client receives the block successfully, it sends an acknowledgement message (with opcode 4) with block number 1.

Let’s look at the three-block example a little differently:



In the picture on the left, the client sends a read request for “myfile.txt” (1423 total bytes) which consists of three blocks (block 1, block 2, and block 3). The server responds to the read request with the first block in a data message (opcode 3, block 1, and 512 bytes of data) and waits for an acknowledgement (Ack) from the client. The client receives block 1 and sends an ack message (opcode 4 and block 1) to the server. The server continues by sending block 2, waits for ack 2, and finally sends block 3 (opcode 3, block 3, and 399 bytes of data). The client knows that block 3 is the last data message because the size is less that 512 bytes. The client sends an ack for block 3 and the protocol ends.



WARNING WARNING – TFTP uses UDP as the transport layer
Since TFTP uses UDP as the transport layer, which doesn’t use a connection like TCP does, there are no guarantees that the data messages will reach the client successfully. If a client doesn’t receive a data block, it resends an acknowledgement for the last block it successfully received. For example, say the message for data block 2 was lost. The client would send ANOTHER ack for block 1 even though your server already got an ack for block 1. The server needs to handle this by sending block 2 again. Your TFTP server will need to be able to handle this case.

Let’s look at this in another way: In the image on the right, block 2 is lost. When the client sent an acknowledgement for block 1, it started a stopwatch to wait for the next block (block 2). However, that block was lost. So, after waiting long enough, the client resends an acknowledgement for 1. The server receiving ANOTHER acknowledgement for 1 sends block 2 again. Your server will need to respond by sending data blocks to the client based on the acknowledgements received. The next data block to send is 1 higher than the last acknowledgement, except for block 1 which is sent immediately after receiving the read request.

What if the file the client requests doesn't exist? You will need to send an error message if the client requests a file that you can't find. The TFTP RFC defines an error message format communicating errors to the client.

Here is the format for an error message:

```
 2 bytes     2 bytes      string    1 byte
 -----------------------------------------
| Opcode |  ErrorCode |   ErrMsg   |   0  |
 -----------------------------------------
```

The operation code (Opcode) for an error message message is a 2-byte integer represented in big endian. The opcode for an error message always has the value 5. The error code is a 2-byte integer represented in big endian. The value of the error code indicates specifically what the error is. The TFTP RFC defines a set of error codes and what they mean:

```
Value     Meaning
0         Not defined, see error message (if any).
1         File not found.
2         Access violation.
3         Disk full or allocation exceeded.
4         Illegal TFTP operation.
5         Unknown transfer ID.
6         File already exists.
7         No such user.
```

The last part of the error message is the error message represented as an ASCII encoded string that ends with a zero (null) byte. So, if a client sends a read request message for a file that doesn't exist, your server should send an error message (op code 5) with error code 1 with an error message of b'File not found'. Don't forget to add the null byte at the end of the message.

Can the client send error messages to the server? Yes, there could be several situations where the client sends an error to the server. If your script receives an error message from the client, make sure you print out the error code, print the error message, and quit.

4 Development Procedure

1. Work through a design as you did for labs 3, 5, and 6
2. Download the tftpserver.py skeleton file to get you started. Getting the right data for a block can be a bit tricky so the skeleton contains some helpful functions to get you started.
3. Edit the header of the file to include your team members' names
4. Complete the script:
    a. Receive a request from the client
    b. Parse the request and verify that it is a read request
    c. Send the data blocks to the client one by one waiting for an ack after each data message
    d. Don't forget to handle error messages as well as acks for blocks that need to be retransmitted.
    e. Close the socket after successfully receiving an ack for the last block
    f. Cleanly exit the program

NOTE: for this lab you will not be hosting your TFTP server on the network so use 'localhost' for the TFTP client. This will prevent you from having to use the S-343 network.

5 Deliverables

Please refer to your instructor's checklist and/or verbal instructions on submitting your lab

6 FAQ

Q: How do I receive TFTP responses?
A: Refer to the slides on how to set up a Python UDP socket: socket.recvfrom

Q: How do I receive data without using next_byte()?
A: Since UDP is not a stream protocol like TCP, you can receive entire packets at once. A TFTP message will never be more than one UDP packet in size (in fact it will probably be much less than that). To receive data without using next_byte() you can use socket.recvfrom(MAX_UDP_PACKET_SIZE).

Q: When I request a text file from my server everything works great, but when I request an image (e.g. JPG) the image doesn't look correct. How do I fix this?
A: The problem is with the TFTP client. By default, the TFTP client on windows uses ASCII mode, which works great for text files which are encoded in ASCII. Images are, however, represented in binary. So, to successfully transfer an image or any other binary file, you need to tell the TFTP client to use binary mode. If you consult the help for TFTP (see above), you find that to tell the TFTP client to use binary mode, you use the -I parameter. For example, to request a file "image.jpg" you would need to use:

```
C:\>tftp -i localhost GET image.jpg
```

7 The challenge

Your challenge, should you choose to accept it is to expand on your TFTP server. Here are some challenges:

- Your TFTP server only must handle a single read (GET) request. Can you make it handle another request after the first one finishes?
- What about write requests? Can you make your TFTP server handle GET and PUT? Consult the RFC to learn how a write request should be processed and implement that into your TFTP server.
- What if you get another read request while your server is in the middle of sending data blocks for a different client's read request? Can you make your TFTP server handle multiple requests at the same time?
    - This one is tricky, you will need to be able to keep track of which blocks you've sent to which client.
    - There isn't a separate data socket like when using TCP so when your server receives data (via socket.recv) it will need to record the address of who sent the data in order to know what to send back.
    - Ask your instructor for more details if you get stuck. This one is a challenge!