TFTP: Trivial File Transfer Protocol

15.1 Introduction

TFTP is the Trivial File Transfer Protocol. It is intended to be used when bootstrapping diskless systems (normally workstations or X terminals). Unlike the File Transfer Protocol (FTP), which we describe in Chapter 27 and which uses TCP, TFTP was designed to use UDP, to make it simple and small. Implementations of TFTP (and its required UDP, IP, and a device driver) can fit in read-only memory.

This chapter provides an overview of TFTP because we'll encounter it in the next chapter with the Bootstrap Protocol. We also encountered TFTP in Figure 5.1 when we bootstrapped the host `sun` from the network. It issued a TFTP request after obtaining its IP address using RARP.

RFC 1350 [Sollins 1992] is the official specification of version 2 of TFTP. Chapter 12 of [Stevens 1990] provides a complete source code implementation of a TFTP client and server, and describes some of the programming techniques used with TFTP.

15.2 Protocol

Each exchange between a client and server starts with the client asking the server to either read a file for the client or write a file for the client. In the normal case of bootstrapping a diskless system, the first request is a read request (RRQ). Figure 15.1 shows the format of the five TFTP messages. (Opcodes 1 and 2 share the same format.)

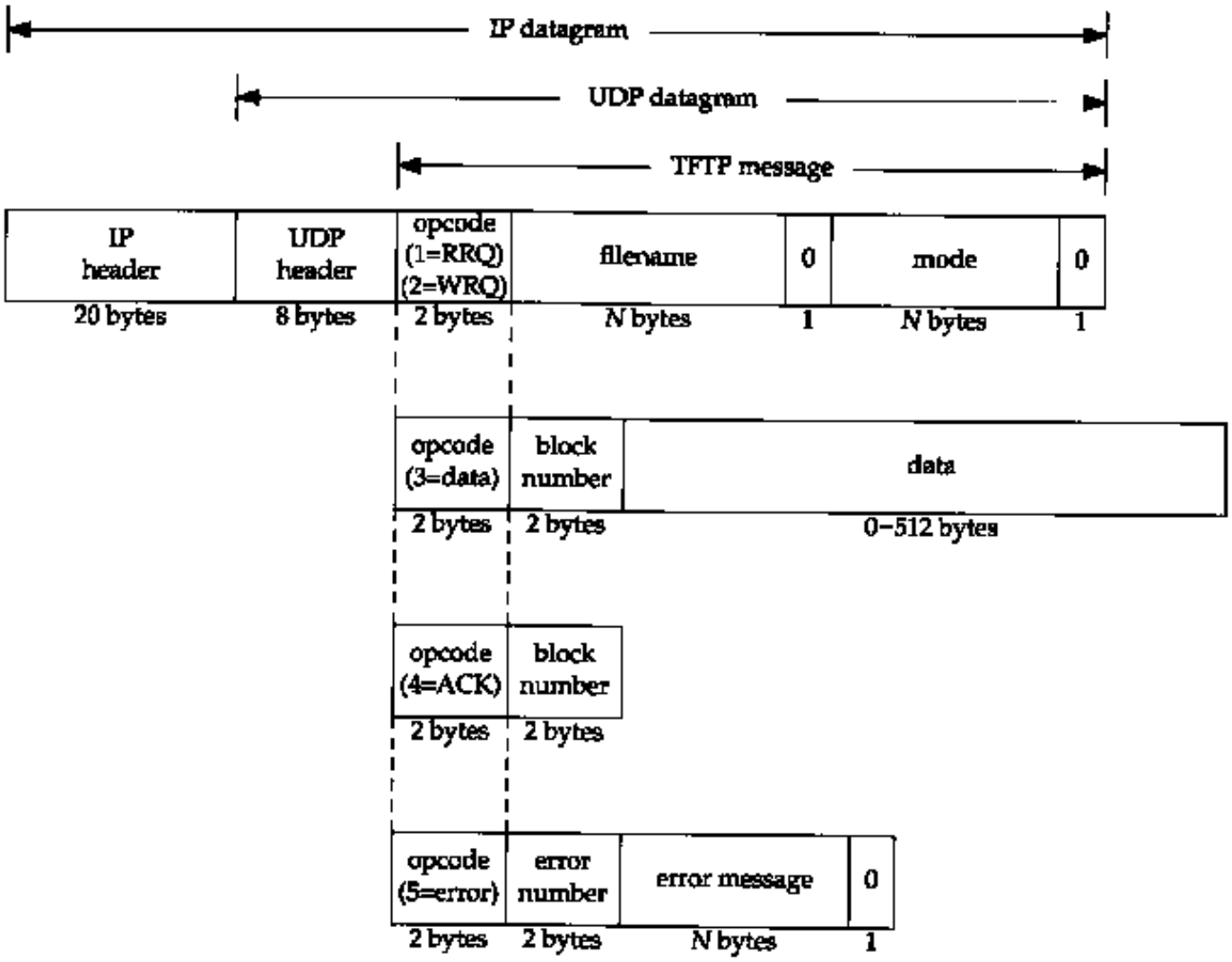

Figure 15.1 Format of the five TFTP messages.

The first 2 bytes of the TFTP message are an *opcode*. For a read request (RRQ) and write request (WRQ) the *filename* specifies the file on the server that the client wants to read from. or write to. We specifically show that this filename is terminated by a byte of 0 in Figure 15.1. The *mode* is one of the ASCII strings `netascii` or `octet` (in any combination of uppercase or lowercase), again terminated by a byte of 0. `netascii` means the data are lines of ASCII text with each line terminated by the 2-character sequence of a carriage return followed by a linefeed (called CR/LF). Both ends must convert between this format and whatever the local host uses as a line delimiter. An `octet` transfer treats the data as 8-bit bytes with no interpretation.

Each data packet contains a *block number* that is later used in an acknowledgment packet. As an example, when reading a file the client sends a read request (RRQ) specifying the filename and mode. If the file can be read by the client, the server responds with a data packet with a block number of 1. The client sends an ACK of block number 1. The server responds with the next data packet, with a block number of 2. The client sends an ACK of block number 2. This continues until the file is transferred. Each data packet contains 512 bytes of data, except for the final packet, which contains 0-511 bytes of data. When the client receives a data packet with less than 512 bytes of data, it knows it has received the final packet.

In the case of a write request (WRQ), the client sends the WRQ specifying the filename and mode. If the file can be written by the client, the server responds with an ACK of block number 0. The client then sends the first 512 bytes of file with a block number of 1. The server responds with an ACK of block number 1.

This type of data transmission is called a *stop-and-wait* protocol. It is found only in simple protocols such as TFTP. We'll see in Section 20.3 that TCP provides a different form of acknowledgment, which can provide higher throughput. TFTP is designed for simplicity of implementation, not high throughput.

The final TFTP message type is the error message, with an *opcode* of 5. This is what the server responds with if a read request or write request 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.

Since TFTP uses the unreliable UDP, it is up to TFTP to handle lost and duplicated packets. Lost packets are detected with a timeout and retransmission implemented by the sender. (Be aware of a potential problem called the "sorcerer's apprentice syndrome" that can occur if both sides time out and retransmit. Section 12.2 of [Stevens 1990] shows how the problem can occur.) As with most UDP applications, there is no checksum in the TFTP messages, which assumes any corruption of the data will be caught by the UDP checksum (Section 11.3).

15.3 An Example

Let's examine TFTP by watching the protocol in action. We'll run the TFTP client on the host `bsdi` and fetch a text file from the host `svr4`:

```
bsdi % tftp svr4                        start the TFTP client
tftp> get test1.c                       fetch a file from the server
Received 962 bytes in 0.3 seconds
tftp> quit                              and terminate
bsdi % ls -l test1.c                    how many bytes in thefile we fetched?
-rw-r-r- 1 rstevens staff 914 Mar 20 11:41 test1.c
bsdi % wc -l test1.c                    and how many lines?
48 test1.c
```

The first point that catches our eye is that the file contains 914 bytes under Unix, but TFTP transfers 962 bytes. Using the `wc` program we see that there are 48 lines in the file, so the 48 Unix newline characters are expanded into

48 CR/LF pairs, since the TFTP default is a `netascii` transfer. Figure 15.2 shows the packet exchange that takes place.

```
1    0.0                    bsdi.ll06 > svr4.tftp: 19 RRQ "testl.c"
2    0.287080 (0.2871)      svr4.1077 > bsdi.ll06: udp 516
3    0.291178 (0.0041)      bsdi.1106 > svr4.1077: udp 4
4    0.299446 (0.0083)      svr4.1077 > bsdi.1106: udp 454
5    0.312320 (0.0129)      bsdi.ll06 > svr4.1077: udp 4
```

Figure 15.2 Packet exchange for TFTP of a file.

Line 1 shows the read request from the client to the server. Since the destination UDP port is the TFTP well-known port (69), `tcpdump` interprets the TFTP packet and prints RRQ and the name of the file. The length of the UDP data is printed as 19 bytes and is accounted for as follows: 2 bytes for the opcode, 7 bytes for the filename, 1 byte of 0, 8 bytes for `netascii`, and another byte of 0.

The next packet is from the server (line 2) and contains 516 bytes: 2 bytes for the opcode, 2 bytes for the block number, and 512 bytes of data. Line 3 is the acknowledgment for this data: 2 bytes for the opcode and 2 bytes for the block number.

The final data packet (line 4) contains 450 bytes of data. The 512 bytes of data in line 2 and this 450 bytes of data account for the 962 bytes of data output by the client.

Note that `tcpdump` doesn't output any additional TFTP protocol information for lines 2-5, whereas it interpreted the TFTP message in line 1. This is because the server's port number changes between lines 1 and 2. The TFTP protocol requires that the client send the first packet (the RRQ or WRQ) to the server's well-known UDP port (69). The server then allocates some other unused ephemeral port on the server's host (1077 in Figure 15.2), which is then used by the server for all further packet exchange between this client and server. The client's port number (1106 in this example) doesn't change, `tcpdump` has no idea that port 1077 on host `svr4` is really a TFTP server.

The reason the server's port number changes is so the server doesn't tie up the well-known port for the amount of time required to transfer the file (which could be many seconds or even minutes). Instead, the well-known port is left available for other TFTP clients to send their requests to, while the current transfer is under way.

Recall from Figure 10.6 that when the RIP server had more than 512 bytes to send to the client, both UDP datagrams came from the server's well-known port. In that example, even though the server had to write multiple datagrams to send all the data back, the server did one write, followed by the next, both from its well-known port. Here, with TFTP, the protocol is different since there is a longer term relationship between the client and server (which we said could be seconds or minutes). If one server process used the well-known port for the duration of the file transfer, it would either have to refuse any further requests that arrived from other clients, or that one server process would have to multiplex file transfers with multiple clients at the same time, on the same port (69). The simplest solution is to have the server obtain a new port after it receives the RRQ or WRQ. Naturally the client must detect this new port when it receives the first data packet (line 2 in Figure 15.2) and then send all further acknowledgments (lines 3 and 5) to that new port. In Section 16.3 we'll see TFTP used when an X terminal is bootstrapped.

15.4 Security

Notice in the TFTP packets (Figure 15.1) that there is no provision for a username or password. This is a feature (i.e., "security hole") of TFTP. Since TFTP was designed for use during the bootstrap process it could be impossible to provide a username and password.

This feature of TFTP was used by many crackers to obtain copies of a Unix password file and then try to guess passwords. To prevent this type of access, most TFTP servers nowadays provide an option whereby only files in a specific directory (often `/tftpboot` on Unix systems) can be accessed. This directory then contains only the

bootstrap files required by the diskless systems.

For additional security, the TFTP server on a Unix system normally sets its user ID and group ID to values that should not be assigned to any real user. This allows access only to files that have world-read or world-write permissions.

15.5 Summary

TFTP is a simple protocol designed to fit into read-only memory and be used only during the bootstrap process of diskless systems. It uses only a few message formats and a stop-and-wait protocol.

To allow multiple clients to bootstrap at the same time, a TFTP server needs to provide some form of concurrency. Because UDP does not provide a unique connection between a client and server (as does TCP), the TFTP server provides concurrency by creating a new UDP port for each client. This allows different client input datagrams to be demultiplexed by the server's UDP module, based on destination port numbers, instead of doing this in the server itself.

The TFTP protocol provides no security features. Most implementations count on the system administrator of the TFTP server to restrict any client's access to the files necessary for bootstrapping only.

Chapter 27 covers the File Transfer Protocol (FTP), which is designed for general purpose, high-throughput file transfer.

Exercises

15.1 Read the Host Requirements RFC to see what a TFTP server should do if it receives a request and the destination IP address of the request is a broadcast address.

15.2 What do you think happens when the TFTP block number wraps around from 65535 to 0? Does RFC 1350 say anything about this?

15.3 We said that the TFTP sender performs the timeout and retransmission to handle lost packets. How does this affect the use of TFTP when it's being used as part of the bootstrap process?

15.4 What is the limiting factor in the time required to transfer a file using TFTP?