Trivial File Transfer Protocol

Ben Andrews, Will Diedrick, Jimmy Hickey
CS 413: Advanced Networking and Telecommunications
https://github.com/JimmyJHickey/TFTP-Client

April 8, 2018

Contents

1 Testing **2**

2 Difficulties **2**
2.1 Unsigned Bytes 2
2.2 Carriage Returns in netascii Mode 3
2.3 Reading and Writing Bytes from a File 4
2.4 Dealing with Timeouts 5

1 Testing

To test our client, we connected to a local TFTP server. We set up the server using the `tftpd-hpa` package for Linux. We bound it to a local IP address (127.0.0.3) and were able to control the files in its directory. We sent packets to this server and monitored the traffic on Wireshark. We also tested over a switch.

We used the Linux `diff` command to determine whether the files were the same. Additionally, we sent the same file using the Linux TFTP utility and our client. We `diff`'d the sent files to check if they were the same. We repeated this process for receiving.

Things we tested:

- Text files
- Images
- Files of various sizes
- Octet and netascii sending modes
- Carriage return + new line spanning two packets
- Packets of 512 bytes in length
- Packets less than 512 bytes in length
- Packets of 0 bytes in length
- Temporary disconnects (testing the timeout functionality)

2 Difficulties

2.1 Unsigned Bytes

Oracle, in their infinite wisdom, has decided that Java is a complete programming language without support of unsigned primitive types. Thus, in order for us to access these

mythical, signless creatures required for many parts of TFTP (such as block numbers, error codes, and op codes) we wrote the following function:

```
private int unsignedByteToInt(byte b)
        return (int) b & 0xFF;
```

This casts our byte to an integer, and logically bitwise ANDs it with 0xFF to block out everything but the last eight bits. Thus, the result will be an integer with our byte value. To get a short (16 bit integer), we called this funtion with the MSB of the short, bitwise ORed it into our target int, and then bit shifted it up 8 times. Then we called the function again with the LSB of the short and ORed the result into the target int once more.

```
private int getOpcode(byte[] data)
        int code = 0;
        code |= unsignedByteToInt(data[Const.OPCODE_MSB_OFFSET]);
        code <<= 8;
        code |= unsignedByteToInt(data[Const.OPCODE_LSB_OFFSET]);
        return code;
```

2.2 Carriage Returns in netascii Mode

Netascii formats text with a carriage return (CR) and a line feed (LF) like the Windows operating systems. However, Unix based systems use only a LF character to denote a new line.

This causes issues when Unix systems (or any system that uses a different end of line sequence) want to transfer in netascii mode. When recieving a file on a Unix system all "CRLF"s need to be replaced with only a "LF". Fortunately Java provides a way to determine the correct line endings for your system. Calling `System.lineSeparator()` returns a string representation of the line seperators for your Operating System, usually "CRLF" or "LF". To make the text file correct for our system we replaced the known line ending "CRLF" with `System.lineSeparator()` using `String.replaceAll()`.

```
string.replaceAll("\r\n", System.lineSeparator());
```

This resolved our issue most of time. The exception is when the file coming from the server has a "CRLF" on the edge of a packet so the "CR" is in one packet and the "LF" is the next. In this scenario replacing "CRLF" with `System.lineSeparator()` does not work because we never have both characters at the same time. So, after we `replaceAll()` we look at the last character of the packet, and if it is a "CR" we assume it is part of a "CRLF" pair and remove it from the string before writing to file.

Sending files in netascii mode when your system does not use netascii line endings was another issue. We were still able to replace our system's line ending with the netascii line endings using `String.replaceAll()`. However, we did this after reading 512 bytes from the file to send. If a "LF" got replaced by a "CRLF", for example, we now have 513 bytes, and we can only send 512 of them. So, we saved any extra bytes to an overflow buffer, and the next time we came around we put any bytes in the overflow buffer at the front of the packet. This unfortunately guarantees more overflow, which is put into the overflow buffer along with any extra overflow created by replacing line endings. Eventually, the overflow buffer gets larger than 512 bytes and needs to be emptied before anymore data is read from the file.

2.3 Reading and Writing Bytes from a File

Reading and writing bytes to and from files was apparently not a use case devised by Sun when designing the Java standard libraries, so we had to go third party. We used the `org.apache.commons.io.FileUtils` and `org.apache.commons.io.IOUtils`. We load our data into a byte array and then, using the `FileUtils.writeByteArrayToFile` method, we write this directly to a file. Contrastingly, to get bytes out of a file, we read the file stream using `IOUtils.toByteArray`.

2.4 Dealing with Timeouts

We implemented a standard Java UDP socket, with a timeout of 5000ms. We decided upon this number by examining the behavior of the Linux TFTP server. It is possible to wait for too long when receiving a packet. We allow for a maximum of three timeouts before ending communication. In other words, our client will try to read the packet three times prior to “disconnecting". If the packet arrives, the timeout counter is reset. When we time out we resend the last sent packet, assuming that the server did not receive it. On a “disconnect" the user is kicked back to the UI prompt with an error message.