

Ian Foster A09902766
Parry Wilcox A10015299
Project Report
11/26/2012

The Project

The purpose of this project was to build a tftp client and server in C to better understand network programming. First, most functions were planned out in advance to be as portable as possible. Then a server implementation was built while using the atftp client to test functionality and conformity to the tftp spec along the way. Once the server was completed the client was built while being tested against both our server implementation and the atftp server. Finally additional features such as error messages and timeouts were added to each and tested.

Project Documentation

tpft.h:

Main header file. Includes all other header files and defines globals including port number, supported mode, buffer size, maximum clients, and more.

packets.h/packets.c:

These files contain the packet structure definitions and functions to manipulate packets including serializing and deserializing packets and printing packet information to stdout.

filetransfer.h/filetransfer.c:

These files define sendFile() and recvFile() which handle all file transfers and work symmetrically for either the client or the server.

pong.h/pong.c:

All functions that send or receive packets are defined in here. Also includes timeout code.

tftpclient.c:

The client that either asks to read a file from the server or write a file to the server. Contains the main routine for the tftp client. All client specific code is in here.

tftpserver.c:

The server that will run continuously in the background waiting for a request sent from client(s).

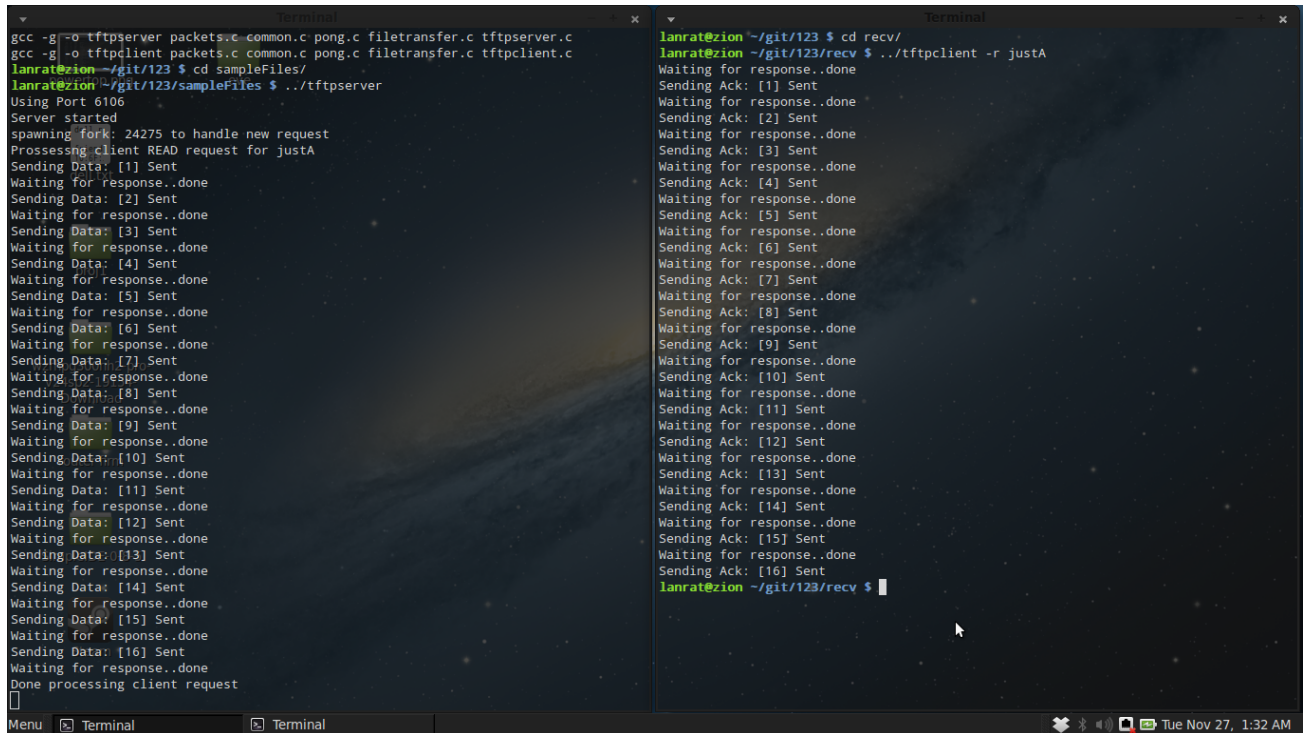
Contains the main routine for the tftp server. All server specific code is in here.

common.h/common.c:

Helper functions used to copy by character, translate network order or host order to the opposite order.

Screen Dumps:

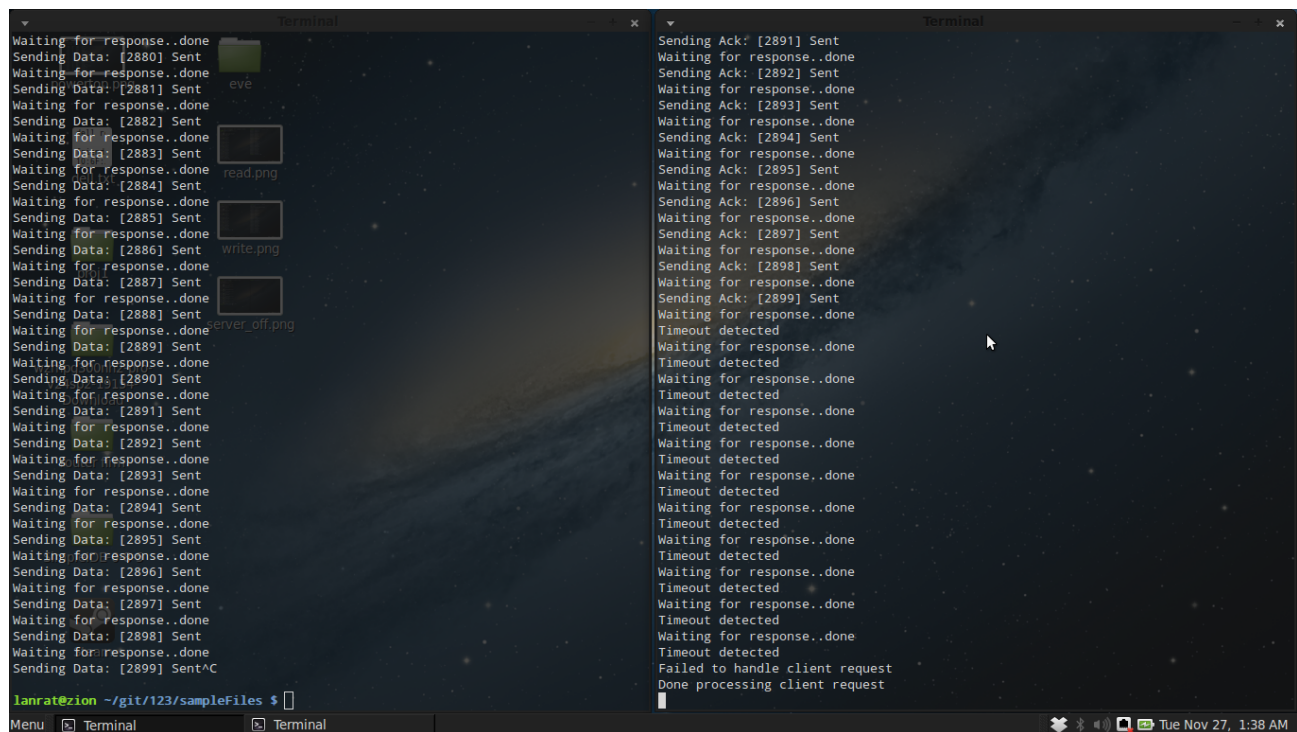
A successful read session.



```
gcc -g -o tftpserver packets.c common.c pong.c filetransfer.c tftpserver.c
gcc -g -o tftpcient packets.c common.c pong.c filetransfer.c tftpcient.c
lanrat@zion:~/git/123 $ cd sampleFiles/
lanrat@zion:~/git/123/sampleFiles $ ../tftpserver
Using Port 6106
Server started
spawning fork: 24275 to handle new request
Processing client READ request for justA
Sending Data: [1] Sent
Waiting for response..done
Sending Data: [2] Sent
Waiting for response..done
Sending Data: [3] Sent
Waiting for response..done
Sending Data: [4] Sent
Waiting for response..done
Sending Data: [5] Sent
Waiting for response..done
Sending Data: [6] Sent
Waiting for response..done
Sending Data: [7] Sent
Waiting for response..done
Sending Data: [8] Sent
Waiting for response..done
Sending Data: [9] Sent
Waiting for response..done
Sending Data: [10] Sent
Waiting for response..done
Sending Data: [11] Sent
Waiting for response..done
Sending Data: [12] Sent
Waiting for response..done
Sending Data: [13] Sent
Waiting for response..done
Sending Data: [14] Sent
Waiting for response..done
Sending Data: [15] Sent
Waiting for response..done
Sending Data: [16] Sent
Waiting for response..done
Done processing client request
[]

lanrat@zion:~/git/123 $ cd recv/
lanrat@zion:~/git/123/recv $ ../tftpcient -r justA
Waiting for response..done
Sending Ack: [1] Sent
Waiting for response..done
Sending Ack: [2] Sent
Waiting for response..done
Sending Ack: [3] Sent
Waiting for response..done
Sending Ack: [4] Sent
Waiting for response..done
Sending Ack: [5] Sent
Waiting for response..done
Sending Ack: [6] Sent
Waiting for response..done
Sending Ack: [7] Sent
Waiting for response..done
Sending Ack: [8] Sent
Waiting for response..done
Sending Ack: [9] Sent
Waiting for response..done
Sending Ack: [10] Sent
Waiting for response..done
Sending Ack: [11] Sent
Waiting for response..done
Sending Ack: [12] Sent
Waiting for response..done
Sending Ack: [13] Sent
Waiting for response..done
Sending Ack: [14] Sent
Waiting for response..done
Sending Ack: [15] Sent
Waiting for response..done
Sending Ack: [16] Sent
lanrat@zion:~/git/123/recv $
```

A successful write session.



Contributions

Ian was responsible for:

- Overall design of the program
- Shared code in common.c
- sendFile() function in filetransfer.c
- Designing packet structures in packets.h
- Implementing packet serialization and deserialization in packets.c
- Functions to send various packets in pong.c
- All code in tftpserver.c

Parry was responsible for:

- recvFile() in filetransfer.c
- waitForPacket() in pong.c including timeouts
- Most code in tftpclient.c
- Answering questions in this document

BUGS:

The only bug we are aware of is caused if you start the server and then immediately pause it (ctrl-z). Then start a client. after the client times out a few times resume the server (fg). The server will receive all the requests that the client sent and the OS queued, and spawn a form

to handle each one, however by this time the client is no longer listening for respond to them so the connection is not reestablished. This does not affect the negative scenarios that we are tested for which involve pausing the server or client mid-transfer and starting the server after the initial client requests have times out.

There are no known deviations from the spec and RFC. Our client and server were successfully tested with the atftp client and server to ensure compatibility.

Questions:

1. Is there a limitation in the maximum file size that the TFTP protocol can support to read/write? If yes, what? Is there a limitation in the maximum file size that your program implementation can support to read/write? If yes, what?

The original TFTP protocol has a limit of 32 MB, and then, in 1998, it was extended to 4 GB. There is the exception that if both the client and server support “block wraparound” then the file size can be unlimited.

Yes, the file size maximum of our program implementation is limited by the largest block number the program can hold because we do not support block wraparound, and the number of bytes that can be passed in each data packet.

Block Number is represented by 2 bytes, so max is $2^{16}-1 = 65535$

Max Data Size = 512 Bytes

$$\begin{aligned}\text{Max File Size} &= (\text{Max Block Number}) * (\text{Max Data Size}) \\ &= 65535 * 512 \text{ B} = 33553920 \text{ B}\end{aligned}$$

Although we need to keep in mind that the last data packet must be less than 512 bytes so that would give us 33553919 Bytes

2. What happens if the timeout value T (in seconds) is too low or too high? Can you use the average round trip time calculated by the ping command to select the timeout value? How can you set a good time-out value for your programs?

If the timeout value T is **too low**, the timeouts could happen prematurely more often if RTT is consistently longer, and therefore there would be a large number of unnecessary retransmissions from the sender to that receiver.

If the timeout value T is **too high**, there would be really slow reactions to a lost data packet. The entire transfer would require more time than it could have.

A good time-out value would be the double the RTT (RTT calculated by the ping command) because the RTT will differ depending on how long the receiving node takes to process the packet and send an ACK packet back.

3. Assume that you want to ensure that the data received is correct by employing checksums, but the UDP checksums are not used by your system. Would you use a checksum over the whole file or a checksum over each individual packet, and why? How should the operation of the protocol be modified to take advantage of the checksums?

We would use a checksum over each individual packet so that if the sum is incorrect, the data packet received should be resent.

The TFTP protocol could support checksums if a checksum field was added to the packet.

Extra Credit

All of the extra credit requirements were fulfilled and work without bugs.

Both the Client and server support the -p option allowing you to specify any port to use. The default port is 6106, our group number.

The server forks for each client request received allowing for it to handle multiple clients simultaneously.

If any error is received on either end and it is appropriate to send to the client/server the error is sent. Including any file handle errors using strerror.

Assessment

The most difficult part of this project was working with unfamiliar parts of C, specifically timeouts, forking and recvfrom buffers. Developing and testing our client and server on a local machine with root access while running a network packet analyzer such as tcpdump or wireshark greatly helped with debugging.

I definitely found this project useful and so far it has been the highlight of CSE 123 for me.