CSE123 · Spring 2013

Term Project

Implementing and testing *tftp*

Checkpoint: May 10, 2013
Due: May 29, 2013

Project Description

For this project you will program a client/server network application in C on a UNIX platform. You will use the UDP/IP family of Internet protocols via the BSD sockets application programming interface. Your task is to design, implement and test a version of the UNIX tftp program.

Your TFTP will implement a subset of the Trivial File Transfer Protocol, Revision 2, as described in RFC 1350, modified as explained below. TFTP is used over local area networks with low error rates, low delays, and high speeds. It employs a simple stop-and-wait scheme over UDP.

In a TFTP file transfer there are two parties involved: a client and a server. The client may read a file from the server, or write a file to the server. You must write and run two programs, a server called `tftpserver.c` and a client called `tftpclient.c`, that support file transfer in both directions.

The server runs in the background, listening on a group-specific UDP port (see group/account section below) for client requests. If a received client request can be satisfied, a file transfer session begins that ends either after successful completion, or after a fatal error occurs. While the clients exit after each session terminates, the server must remain in the background to listen for more client requests, if it has not been terminated due to a fatal error.

You will start your server in one terminal by typing `tftpserver` and your clients in another terminal by typing either `tftpclient -r filename` or `tftpclient -w filename`, to read or write files respectively. A read request from the client will download the file from the server, while a write request from the client will upload the file to the server.

During normal protocol operation, your programs send and receive messages according to protocol rules. The core logic of the program consists of a main receive/send loop plus some initialization and termination code. In the loop, you wait for a message from the peer. If it is the next expected message, you send your next message or terminate the session successfully. You need to know the message number (block #) that you are next expecting to see, so that you can distinguish between new and duplicate messages, whether DATA or ACK.

There will be a midpoint in the sixth week on May 10 to check your progress and the final due date of the project is May 29.

Project Breakdown

Step 0: Getting started

To help you get familiar with socket programming, we provided a simple sample program.

Two .c files have been uploaded for a sample client-server model running over UDP. The programs have been added with comments whenever something new has been introduced.
Try to run these programs on your laptops, or on the ieng6 machine. The port number is defined in the program as follows.

```
#define SERV_UDP_PORT   12345
```

When you are running the program on the ieng6 machine, make sure that you replace 12345 with your group specific port number.

Compile the programs using the following commands:

```
$gcc test_echo_client.c -o test_echo_client
$gcc server_test.c -o server_test
```

The -o directives select the output file name. Run the server in the background "./server_test &" and the client in the foreground "./test_echo_client" on the same machine.

Whatever you type in the client, is sent to the server, the server reads it and returns an acknowledgment - the same message to the client for display. The client may send another message to the server on receipt of the acknowledgment, and this loop continues till the client process is killed.

The programs use UDP and show how you can use socket calls to set up connection endpoints, specify addresses and ports for them, and send and receive data to and from another process. Please study the sample program carefully, especially the usage of system calls sendto(), recvfrom(), bind(), and the socket struct.

You should use the man pages at a Linux machine to get a full description of the calls mentioned here and references to other related calls. The man pages also give you programming information such as parameters, return values, header files that you need to include and any libraries that must be linked to your program via the -l compiler directive. This should help you solve problems during compilation and linking. You may also refer to the following web-page -
http://www.faqs.org/faqs/unix-faq/socket/

Step 1: transmit a simple small file (less than 512 bytes)

After step 0, now you should understand how sockets are bind to port and how server and client establish connections. At this point, please read the RFC 1350 specification thoroughly to understand the protocol. At the end of step 1, you should be able to transmit a simple small file that is less than 512 bytes, which only needs one packet to send. You would have 2 directories named CLIENT and SERVER, with the client and server programs running in their respective directories. File transferred from server to client (READ) operation means a file in the SERVER folder should be transferred to the CLIENT folder. (And vice versa for WRITE operation)

How do I manipulate text and raw data packet fields?

All text fields are terminated by a null byte, which is also the string terminator in the C language. As a result, you can read or write them directly by employing C string functions, using a pointer to the beginning of the appropriate field. The filename and error message fields are in fixed offsets from the beginning of the packet, so you can use the procedure described above. For the mode field, you need also to calculate the size of the filename field to add to the offset (include the null byte), which is an easy task using a C function. The raw data in data messages on the other hand are binary and may contain null bytes, so C string functions are inappropriate for them. Instead, you should use the bcopy() function to read or write them, using a pointer to their fixed offset in the packet. When sending data, the size of this field is either 512 bytes or 0-511 bytes for the last packet, as indicated by the file read call. When receiving data, the size of the packet is indicated by the recvfrom() call (remember to subtract the size of the other packet fields to get the data size). The function bzero() is also useful for initializing packets.

How do I manipulate numeric packet fields?

All numeric fields are unsigned short integers (2 bytes) in network byte order, which means that multibyte numbers are stored with the high byte first. Adopting this convention ensures compatibility between systems with different byte ordering. To convert a number from the host byte order, whatever this order is, to network byte order, you usehtonl() for long integers and htons() for short integers. ntohl() and ntohs() perform the reverse conversion. By always performing host to network conversion before storing a number and network to host conversion after reading it, your program will work regardless of the platform used. The numeric fields (packet type, block number and error code) are in fixed offsets from the beginning of the packet. To read from or store to them, get a pointer to the beginning of the packet, add (if required) the correct number of bytes to it, and convert it to the correct pointer type before dereferencing it. IP and UDP numeric fields are also in network byte order and unsigned.

Please note: Since your programs will execute on a multi-user environment, you must ensure that your server does not conflict with others by using your unique, group specific, port for your programs. Your server can also conflict with older copies of itself that uses the same group specific port. To avoid this, shutdown (kill) your server when you do not need it anymore, so that newer versions of your server can reuse the group specific port.

Since TFTP does not support authentication, anyone can send requests to your server to read or write files, by using your group specific port on purpose or accidentally. To avoid having your private files stolen or overwritten, have the server operate in a directory where it has read and write privileges, and only store test files there. You should check that read and write requests use simple filenames and not full pathnames, so that no files outside this directory are accidentally accessed.

Step 2: Transmitting large files.

In this step, you need to improve your program in step 1 such that it can transmit larger files that need multiple packets to transmit. Your program should handle files that include white space, special characters, carriage return, blank lines, and should be able to transmit files very large (eg: 10MB)

You may print protocol events for client and server on the screen, e.g.:

```
Received [Read Request]
Sending block# 1 of data
Received Ack #1
Sending block #2 of data ...etc.
```

This will help you and us understand the flow of the program. At a later stage, when your program supports multiple clients, the protocol events printed on screen may have appended the client source port # to distinguish events for multiple clients.

Since the protocol allows bi-directional transfer (read and write), both the server and the client can be either senders or receivers of data for any given session. The protocol also specifies that all messages are acknowledged, so programs must both send and receive messages during the same session. This symmetry means that most of the code can be shared between the client and the server. *Therefore, you should put all necessary shared code in a file tftp.c and the function prototypes in tftp.h. These two files will then be shared between tftpserver.c and tftpclient.c.*

Step 3: Handling packet loss and enabling timeout

The protocol does not specify any error control, so the contents of the transferred files are not guaranteed to be correct. However, all data messages must be received, so lost packets must be retransmitted until they are acknowledged.

Packet Loss

To detect lost messages and crashed or unavailable peers, you must use timeouts. After sending a message, set a timer of duration T seconds (of your choice) and wait for a reply. If the timer expires before a message is received, a timeout has occurred. By retransmitting the last message sent, you can recover from lost messages in either direction. If the peer has crashed or is unavailable however, you must terminate the connection after 10 consecutive timeouts. For this reason, you need to keep a timeout counter that is increased on every timeout, and reset to zero when a message is received. It does not matter whether the received message is new or duplicate, what matters is that connectivity is being maintained.

The client may terminate due to timeouts while sending its initial request, meaning the server is unavailable, while both client and server may terminate due to timeouts during a session, meaning the peer has crashed.

Timeouts

For timeout implementation you can use the UNIX supported timers, by setting and catching SIGALRM signals. To set a timer to expire after t seconds of real (as opposed to process) time, you call `alarm()` as `x = alarm(t);`, where `x` and `t` are unsigned integers. Value `t` holds the interval in seconds until the alarm goes off and value `x` gets the previous remaining interval, if an alarm was already set but not expired. This method only supports one timer, but this is sufficient for your protocol.

After you send a message, before waiting to receive a reply, you set the alarm. If the alarm goes off before a message is received, we have a timeout so the program receives a SIGALRM signal. If the message is received instead, we reset the timer so that it will not expire (later) and confuse our program, by calling `alarm()` with `0` as the interval.

The SIGALRM signal sent by UNIX will interrupt your program from whatever it is doing and look at whether a handler function has been associated with this signal. If not, the program is terminated. To associate a handler with a specific signal you can use the `signal()` call. The handler should take an integer parameter and return nothing, for example `void handler(int);`. You can associate this handler with the SIGALRM signal using the call `signal(SIGALRM, handler);`. The return value of `signal()` is a pointer to the previous handler associated with this signal.

If UNIX finds a handler associated with SIGALRM, it interrupts the program, executes the handler, and continues from where it left off when the signal occurred. The parameter passed to the handler is the identifier of the signal (this is useful if you want to handle multiple signals with the same function).

When the SIGALRM handler is called, there is no guarantee that the timeout occurred during the wait to receive the message, since the alarm may have expired right between message reception and execution of the next instruction in your program. You should thus avoid taking any actions inside the handler since it may be a false alarm, apart maybe from re-associating the handler with the signal. You can reliably detect whether a timeout occurred or not by looking at the return value of the receive message system call (for example, `recvfrom()` for UDP sockets)

Checkpoint (May 10, 2013)

At this checkpoint, you should finish step 3. You are required to turn in a half to one page project progress report, and attach screen shots for a successful file read/write session. More details of this mid point checkpoint will be available soon.

Step 4: Specify port number at runtime from command line

Implement a runtime option to **choose the group specific port from the command line,** for example, `tftpserver -p 7000`. If the option is passed to the server, it specifies the port to listen to, while if it is passed to the client, it specifies the port to which the initial request must be sent to. This makes your programs interoperable with standards-compliant implementations that use "well-known" UDP port 69. If the option is not present, your programs must use your assigned port # as the default.

Step 5: Handling error cases

Program should support processing of TFTP ERROR message, sent out for abnormal scenarios. Your program should handle the following subset of error situations:

1) File does not exists
2) File already exists (overwrite warning)
3) No permission to access the requested file

In the event of an error case, the client that started the request should receive an error message with the corresponding error information from the server or the client itself, depending on the type of error.

Step 6: Enable your server to handle multiple clients simultaneously

You can achieve this by splitting your server in two pieces. The main server is always waiting in the background for requests to the group specific port, executing auxiliary servers to handle the actual file transfers. The auxiliary servers employ temporary ports provided by the operating system so that they do not conflict with each other or the main server. The clients initially contact the main server at the group specific port and then switch to the auxiliary server and temporary port for the remainder of the session. The client changes its destination port after receiving the first message from the (auxiliary) server.

To create a new process (auxiliary server) you must use the `fork()` system call which creates an identical copy of your server. Depending on the return value of `fork()`, you can see whether the process is the parent (you get the process identifier of the child) or the child (you get zero). The parent may return to its main loop, waiting for a new request, while the child takes care of the client request. It is easier to have the auxiliary server be the second part of the main server program, since the parent and child processes share the same state right after the `fork()` call. One exception is the `alarm()` timer, so wait until you are in the child before setting any timeouts. The parent must not wait for the child to exit using the wait() system call, as this is a blocking call that would freeze the main server and thus prohibit simultaneous sessions.

Requirements

1. All project requirements are included either in the RFC 1350 or in this document. In case of a conflict, this document takes precedence.
2. You will implement only the octet file transfer mode, not the netascii or mail modes mentioned in RFC 1350. As a result, no data conversion is required at the receiving end.
3. Note that timeout and retransmission handling (and code) is symmetric between client and server regardless of file transfer direction. If the expected data or acknowledgment message does not arrive before a timeout occurs, the program retransmits its last message. Choose a reasonable timeout value T seconds to detect inactivity from the other side. If after 10 consecutive attempts to send a DATA or ACK message the program still gets a timeout before a reply, the session is assumed to have failed (connection broken down) and your program should terminate and report this on screen output.
4. The 512-byte limit refers to the actual TFTP data part, not TFTP and lower layer protocol headers and trailers. The server uses your group unique port, while the clients use UDP ports provided by the operating system.
5. Despite only supporting the octet mode, your TFTP RRQ/WRQ message headers must contain the mode field for compatibility with other TFTP implementations. Your client must fill in these fields and the server must inspect them for correctness. The filename and mode fields in RRQ/WRQ messages must be in plain ASCII, terminated by the 0 byte that serves as a field separator. Your server must check that the filename field contains a simple filename, not a pathname, to avoid security intrusions into some other directory.
6. No data conversions will be performed by your programs. New data messages are acknowledged by ACK messages and vice versa, in a manner symmetric between client and server regardless of data transfer direction.
7. On the other hand, duplicate messages are handled in an asymmetric manner. When you receive a duplicate data message, you must retransmit the last ACK message, but when you

receive a duplicate ACK message do not retransmit the last DATA message, to avoid the sorcerer's apprentice bug.

8. Do not implement dallying. After the data receiver (whether it is the client or the server) sends an ACK message as a reply to the final DATA message, it can terminate the session. The data sender terminates after getting an ACK message for its last data message.
9. All message formats are the same as in the standard. Protocol fields that contain text are in plain ASCII, while numeric fields are in network byte order.

Group

This is a group project, with each group consisting of two students. You must pick your group number by selecting a blank group in the group section on Ted. Use this number to identify your group and determine the default port number to use in your programs: `600 + group #`. For example, if your group number is 2, then you should use 60002 as the default port number in your program.

Finally, you will be doing a demo of your program, and turn in all your code (tftp.c, tftp.h, tftpclient.c, tftpserver.c) and screenshots. More details of the deliverables will be made available around the checkpoint.

Additional Information and Turn-in Requirements

Details of when, how and what to turn-in, including perhaps details on what to test and demonstrate, will be provided as the project progresses. Project discussion will take place during tutorials only and not at the lectures.