Trivial File Transfer Protocol

153728

```
tftp client
enter '?' to print a list of commands
tftp> connect localhost 6009
tftp> get /Users/Guest/Public/image.jpg
received 3833 bytes in 0.1 seconds
tftp> put image.jpg /Users/Shared/returned.jpg
sent 3833 bytes in 0.1 seconds
tftp>
```

1 Core

In implementing the Trivial File Transfer Protocol, I attempted to abstract as much as possible so that a large amount of code could be shared between the client and server applications (in both the UDP version and the TCP version). So initially, I focused on translating the specifications of the TFTP to 'Java representation'. The `tftp.core` package contains enums describing error types (e.g. `ErrorType.FILE_NOT_FOUND`), transfer modes (e.g. `Mode.OCTET`) and classes describing TFTP packets (e.g. `WriteRequestPacket` and `DataPacket`). Each of these TFTP packet classes inherit from a base class `TFTPPacket`. Each TFTPPacket subclass has two constructors - one for creating a new TFTP-Packet from 'logical data' (e.g. `new ErrorPacket(ErrorType.FILE_NOT_FOUND, "file")`), and another for parsing the packet from an array of bytes. Byte array parsing is done using a `ByteBuffer` wrapping the buffer. The parent class `TFTPPacket` contains a static method `fromByteArray` which takes a byte buffer (and length) and returns a `TFTPPacket`. This was done by examining the first two bytes (the opcode) of the packet and then, based on this opcode, instantiating the relevant `TFTPPacket` subclass.

2 UDP

The `TFTPUDPServer` program takes two optional arguments: `-port port-number` to set the port to bind to and `-timeout time-in-ms` to set the timeout length - for example,

```
java tftp.udp.server.TFTPUDPServer -port 7000 -timeout 4000.
```

By default, the server binds to port 6009 and has a timeout of 3000ms.

The main server class `TFTPUDPServer` handles accepting write/read requests from clients, and spawning the relevant 'worker' thread to send or receive the client-specified file. This is done by binding a datagram socket to a given port (passed to the class constructor), port 6009 in my implementation. From there, a buffer to hold received packets is allocated, and then the server infinitely loops. In each pass through the loop, the socket waits to receive a datagram (blocking). When a datagram is received, the `TFTPPacket` is extracted from the buffer. Based on the type of request (`RRQ` or `WRQ`), the server submits a job to an asynchronous `ExecutorService` to respond to the request, and the loop repeats.

The two worker classes to respond to read and write requests are `ServerRRQHandler` and `ServerWRQHandler` respectively. Constructors for both these classes accept a client address and client port (to know where to send datagrams), and the original request packet (to obtain the file name and to ensure the transfer is taking place using octet mode).

Upon running, the `ServerWRQHandler` creates a new datagram socket and binds it to a random free port. A timeout is also specified based on the server configuration. Then the transfer mode of the RRQ packet is checked - if not `Mode.OCTET` then an error packet is sent to the client and the server worker terminates. If the mode is octet however, an input stream is opened to the file specified by the client (if the file does not exist, an error packet is sent to the client and the transfer terminates). Then an output stream is

opened to the file specified in the received WRQ packet. A new byte array and datagram packet is created to hold received datagrams. The initial acknowledgement packet (block number of 0) is converted to a datagram using the `UDPUtil` class. This datagram is sent to the client. Immediately after sending, the method `ServerSocket#receive(Datagram)` is called (blocking) to wait for a response from the client. If this is *not* received within the timeout length, a variable called `timeouts` is incremented and the last datagram is resent. If the server times out 5 times in a row, the server gives up. However all going well, when the response is received it is converted into a TFTPPacket. Using `instanceof`, the type of the packet is checked - if it is an error packet, the error message is printed and the transfer is cancelled. If it is a data packet with the expected block number (kept track of with a local variable called `ackNumber`), the data contained in the packet is written to the file output stream. `ackNumber` is incremented. Then `DataPacket#isFinalPacket()` is called - this method simply checks the length of the packet - if less than 512 bytes it is the final packet. If it is, the transfer is finished. If not, the server continues the loop by sending the current acknowledgement packet.

The `ServerRRQHandler` works similarly. Note that write requests and read requests do almost the same thing - both read from some input and write that input to some output. As before, the `ServerRRQHandler` opens a socket, binds it, sets up the timeout, and checks the transfer mode. Then the file is read from the input stream in 512-byte chunks. This 512-byte chunk is encapsulated in a data packet, initially with a block number of 1. The TFTP packet is then 'converted' to a datagram which is sent to the client. Using the same timeout method as described before, it waits for an acknowledgment of the correct packet from the receiver. Once received it increments the block number and continues the loop by again reading a chunk of 512 bytes from the file. The final packet is indicated by a less-than-512-byte data packet - once less than 512 bytes are read from the input file, it signals the last data packet. Once this packet is sent and the acknowledgement is received, the transfer is complete. A complication arises if the file is a multiple of 512 bytes long. As specified in the RFC, this is resolved by sending a 0-byte final data packet. So in the implementation if a read from the file returns -1 (signalling EOF), the length of the data packet to send is set to 0 in order to send an empty data packet.

After writing the server, the client was fairly straightforward because all code for transferring and receiving a file over TFTP had already been done. So most of the work developing the client involved coding the command line interface. My client is modelled after the OSX `tftp` program, a TFTP client bundled with OSX. As with the OSX program, entering '?' lists the available commands:

- `connect hostname [port]` sets the address and port of the TFTP server. If no port is specified port 6009 will be used. This is not connection in the TCP sense, but rather just sets the destination of outgoing datagrams.
- `get remote-path [local-path]` retrieves a file from the connected TFTP server. `remote-path` is the path to the file on the server. `local-path` is the path to the file on the local machine - if not specified the file is saved to the working directory.
- `put local-path [remote-path]` writes a file to the connected TFTP server. `local-path` is the path to the file on the local machine, and `remote-path` is where the file will be saved on the server - by default in the server's working directory.
- `timeout time-in-ms` sets the timeout length in milliseconds.
- `exit` quits the client.

As mentioned above, the code for client sending and receiving is similar to the code written for the server (but in reverse - i.e. for a RRQ the client reads and the server writes, for a WRQ the client writes and the server reads). As such, the shared code was abstracted into classes `FileReceiver` and `FileSender`: both the client and server use these. The only way that they differ is in the initial packet sent. The initial client packet sent is a RRQ/WRQ and the initial server packet sent is an ACK0/DATA1. So for a RRQ the client opens an output stream to the file, constructs the initial `ReadRequestPacket` and then proceeds as described in the server implementation above using `FileReceiver`. For a WRQ, it opens an input stream to the user-requested file (or prints an error if the file does not exist), constructs a `WriteRequestPacket`, and then uses `FileSender` to send the file to the server.

3 TCP

Since TCP provides a reliable transport service for the application layer, the explicit ACKs, timeout handling, etc. are not required in this case. Rather, the protocol can simply involve connecting the receiving a RRQ/WRQ, and then, upon 'acceptance' of the request, a direct sending of the file bytes to/from the server.

The read request interaction:

1. client opens connection to server
2. client sends RRQ to server
3. server sends TFTP ACK to indicate that the file transfer has been approved
4. server sends file bytes to client
5. after file is fully sent, server closes connection

The write request interaction:

1. client opens connection to server
2. client sends WRQ to server
3. server sends TFTP ACK to client (block number is irrelevant, this is just for the server to explicitly approve the transfer)
4. client sends file bytes to server
5. after file is fully sent, client closes connection

If the file does not exist, in step 3 the server will send back an error packet instead, containing an error message.

The server first opens a `ServerSocket` port on a particular port - by default 6009. The server then blocks until a client connects. When a client connects, a job is submitted to an `Executor` to be executed asynchronously - this job is in charge of responding to the client request. The first read from the socket input stream is interpreted as a request packet. The transfer mode of the request packet is ensured to be `octet`.

If the packet is a WRQ, the server sends an ACK to the client. Then the file is received by continually reading bytes from the input stream and writing them to file until the connection is closed by the client.

If the packet is a RRQ, the server then sends an ACK to the client to indicate that the read request has been accepted (or an error packet if the file does not exist). The server then immediately opens an input stream to the requested file, and sends the file bytes over TCP. Once the sending is complete, the server closes the connection.

The main difficulty in developing this protocol over TCP was that because TCP provides a continuous stream of bytes to the application layer, it is hard to tell where the server response packet ends and the file bytes begin. The response packet from the server (either an acknowledgement packet or an error packet) is of variable length depending on what it is and what error text it may contain - and so the client may read in file bytes while incorrectly interpreting them as TFTP packet bytes. There are other ways to get around this, but the simple solution I took was to pad out the packet to `Configuration.MAX_PACKET_LENGTH` bytes long. Then it is certain that the file bytes start at offset `Configuration.MAX_PACKET_LENGTH`.

The client uses the same structure as the UDP TFTP client, supporting the same commands. The only subtlety is with the `connect` command - as with the UDP client this does not mean connect in the TCP sense, rather it just sets the server's hostname and port. A separate TCP connection is initiated with every `get` and `put`.

A `get` operation causes a `ReadRequestPacket` with the appropriate filename to be sent to the server. Then the client simply receives the file bytes from the server and writes them to file. Once the transfer is complete the server will terminate the TCP connection.

A `put` operation causes a `WriteRequestPacket` to be sent to the server. The client then waits for an `AcknowledgementPacket` to be received from the server. Once received, the client sends the file bytes to the server and closes the connection.

This implementation supports the error packet `ErrorType.FILE_NOT_FOUND` by the client checking the initial response to the request - if it is of type `AcknowledgementPacket` the data is sent/received, but if it is of type `ErrorPacket` then the contained error message is printed to the output.