EXTENDING TFTP

The Trivial File Transfer Protocol (TFTP) is a simple protocol for transferring data. On small devices, it is frequently one of the first high-level protocols to be implemented, because of its usefulness (transferring data) and its simplicity. The basic TFTP protocol is defined in RFC 1350, later RFCs define extensions to the basic protocol.

In this article, I propose a few more extensions. For interoperability, the extended protocol is compatible with standard (or "basic") TFTP hosts. In the design of TFTP, there is actually little difference between a client and a server, and I have usually built combined TFTP clients + servers.

Handling large files

The basic TFTP protocol works with packets with 512 bytes of data —with an exception for the final block of the transfer. Each packet contains a 16-bit block counter and this counter starts at 1 (one) for the first block. As long as the data to transfer is less than 65535 blocks (the highest value for a 16-bit counter), all is well. This sums up to (nearly) 32 MiB.

For bigger files, there are three options: larger TFTP data blocks, allowing the block counter to roll over and concatenation of separate transfers to a single file.

- **Larger data blocks**

  RFC 2348 defines an extension for larger block sizes. Even though the RFC allows blocks of up to 65464 bytes, in practice the limit is set to 1468 bytes: the size of an Ethernet MTU minus the headers of TFTP (4 bytes), UDP (8 bytes) and IP (20 bytes). Larger packets run a high risk of being fragmented at the IP level, and this is generally considered a *bad thing*. With a block size of 1468 bytes, the maximum transfer size is over 91 MiB. Increasing the block size improves the transfer performance at the same time as allowing larger files to be transferred.

  Next to the standard "blksize" option, there is also a non-standard "blksize2" option. The latter has the restriction that the block size must be a power of two (512, 1024, 2048, ...). The server should not use a block size below 512 bytes unless a client has explicitly requested it.

- **Block counter roll-over**

  RFC 1350 does not mention block counter roll-over, but several TFTP hosts implement the roll-over be able to accept transfers of unlimited size. There is no consensus, however, whether the counter should wrap around to zero or to one.

Many implementations wrap to zero, because this is the simplest to implement. The "ARINC 615A Data Load Protocol" mandates a roll-over to 1.

Although the block counter roll-over does not need a change in (or extension of) the TFTP protocol per se, it *does* REQUIRE that both host agree on the method of rolling over. While many TFTP hosts use block counter roll-over implicitly, it should be an option in the TFTP option negotiation. I propose the option name "rollover" for this purpose. The client sets this option if it supports block counter roll-over, and it indicates whether it wants to wrap to zero or wrap to one. The server acknowledges this option if it supports roll-over and the method.

The block counter roll-over scheme conflicts with the multicast modes. In multicast transfers, a client may step into the middle of a transfer that is already streaming out of the server. In that situation, a client cannot know how many roll-overs have occurred before it stepped into the stream.

- **Collect separate TFTP transfers into a single file**

  Instead of requesting full files, a client could ask the server to transfer part of a file. The client sets the offset from the beginning of the file through an option. The server acknowledges the option if it supports it.

The "toffset" option would normally be combined with the "tsize" option to set both the start offset and the amount of data to transfer. The "toffset" option is my proposal; "tsize" is a standard option defined in RFC 2349.

Performance

The TFTP protocol uses a lock-step algorithm. After the transmitter sends a block of data, it waits for an acknowledgement of reception before sending the next block. The transfer rate is therefore limited to the round-trip time (RTT). If the round-trip time between two hosts is 20 ms, for example, then the transmitter can send up to 50 blocks per second. With a default block size of 512 bytes, the transfer rate is bound to 25 KiB/s.

One way to improve this is to use larger blocks. On links with a large round-trip time, transfer speed may improve dramatically by using block sizes of 1024 or 1468 bytes. (As was mentioned earlier, 1468 bytes is a common limit, chosen so that the total size of the data payload and protocol headers does not exceed the 1500-bytes Ethernet MTU.)

Although the TFTP protocol is standardized as "lock-step", it will actually work with a transfer window. The WvTftp server pioneered this design. With a transfer window, the file *transmitter* sends new packets before having received the acknowledgements for previous packets.

With a transfer window, multiple packets may be "in flight" on the network: these are packets that have been sent out, but have not yet been acknowledged. The WvTftp implementation refers to the transfer window as "negative latency", by the way. I prefer the term "transfer window", because the technique is quite similar to how the TCP protocol handles data transfer.

A transfer window does not REQUIRE changes in the receiver of the file: each received data block is still acknowledged individually. A receiver should check that the received packet is the one that it expects, because with multiple packets in flight the packets may arrive out of order. However, TFTP receivers should already check the packet number for reasons of avoiding the Sorcerer's Apprentice Syndrome.

A transfer window on TFTP should just work out of the box. When you choose the size of the window too large, it may damage performance instead of help it, because the receiver may not be able to handle the packets at the speed that the transmitter tries to pump them over the network. If the receiver "back-log queue" is smaller than the transmitter window, packets may get dropped, and the transfer will then stall on the time-outs.

To avoid the above scenario, I propose the option "window" that the client can set.

In option negotiation, the transmitter and sender can then select the smallest window that either side handles optimally. The parameter of this option is the number of data packets, not bytes. So if you negotiate a window of 4 and the block size is 1024, there may be 4 packets with 1024 bytes each in flight at any time.

I have referred to the TCP protocol essentially using the same technique for a (sliding) transfer window. The TCP protocol uses a combination of negotiation and detection of the network congestion to determine an optimal window, using slow-start and back-off algorithms. I feel that option negotiation lies more in the nature of the TFTP protocol than these adaptive algorithms.

Authentication

TFTP has no session control and you cannot "log in" onto a TFTP server. Traditionally, files on a TFTP server can be accessed without password. Several Linksys routers (e.g. WAG54G) use TFTP for updating firmware. To make sure that not anyone can do this, Linksys made a minor adjustment to the protocol in which the initial transfer request holds a password (in addition to the filename and the file mode).

Simply adding a password string to the TFTP read or write request has two problems. Since the password is sent in clear-text it can easily be sniffed.

A worse problem is that this choice breaks option negotiation (RFC 2347). I feel that the password should be added to the request as an option, so that it does not break option negotiation. There is no easy solution to avoid sniffing the password, since the option negotiation must be initiated by the client and it is just a two-way handshake. A full challenge-handshake protocol cannot be built with TFTP option negotiation.

I propose the option "password" with a zero-terminated string as the parameter. In case a user name and a password must be sent, these can be combined in the string with a colon (or another "special" character) separating the user name and the password. For example, in the parameter "thiadmer:secret", "thiadmer" would be the user name and "secret" the password.

Handling time-outs

According to RFC 1350, both the client and the server should check for time-outs for the packets that they transmit. The packets can be data packets or acknowledgements. For the protocol, it is sufficient that only one side uses a time-out. A common simplification is only data blocks time out. Acknowledgements are simply transmitted once, and never re-transmitted.

In this scheme, if a data packet gets lost, it will be re-transmitted.

If an acknowledgement packet gets lost, the transmitter will not see an acknowledgement for the data packet and it will re-transmit the data packet. The receiver receives a duplicate packet, which it acknowledges again. In conformance with RFC 1350, a transmitter must be prepared to receive a duplicate ACK.

This is purely an implementation issue. In addition, the two hosts do not need to agree on how to handle time-outs: one host that re-transmits only data packets can interoperate with another host that re-transmits both data and acknowledgement packets.

Multicasting

In multicast mode, a single data stream is received by multiple hosts. Standard connections ("unicast") require a separate connection for each data stream going out of the server, but a multicast transfer needs only a single connection. Multicasting therefore reduces the load at the server and increases the network efficiency.

There exist two proposals for multicast TFTP: RFC 2090 and mTFTP defined in the Intel PXE specification. RFC 2090 is a fairly complex protocol if implemented in full at both the client and the server.

PXE's mTFTP is much simpler, but it has important limitations:

- No option negotiation is possible.
- The client cannot ask for a specific file, it will always receive a particular file from the server (the PXE specification suggests to use a different multicast IP address for every file).
- When the "master client" disappears, the other clients are not signalled, meaning that another client will only restart the transfer (and become the new master client) after a time-out —which may be quite long.

The master client is the client that sends the acknowledgements: in a multicast situation, several hosts receive the same data packets from a server, but only one of these may (and must) answer.

The complexity of RFC 2090 lies in the ways that it implements partial transfers. According to the RFC, each client should maintain a list of packets that it has received and (if it is selected by the server as the "master client") ask the server for the packets that it still lacks. A mTFTP client/server will always restart the transfer with the first block, and therefore a client only has to remember the first block number that it received. An RFC 2090 client can be simplified accordingly (without needing to change the protocol): instead of a map of all received packets, let it just remember a single span of consecutive packets.

Any received series packet that expands this range is accepted. Any series of received packets that lies outside the span is ignored.

For example, assume that a client drops into an existing stream and it sees the packets 683 to 1254 pass by. It can determine whether that last packet (1254) marks the end of the file, by checking whether that packet is a *full* data block. A next sequence of packets from 1 to 200 would be ignored, but if the client receives packets 500 to 700, it will accept 500 to 682 and update its span to 500 .. 1254. When the client becomes the master client before seeing a complete file, it asks the server for at most two sequences: from packet 1 to the start of its span and from the end of its span to the end of the file.

Neither type of multicast TFTP works well with block counter roll-over. Since mTFTP does not support option negotiation at all, data transfers are limited to 32 MiB (minus 512 bytes). A multicast TFTP host implementing RFC 2090 could negotiate larger block sizes and it could be extended to handle the proposed (non-standard) "toffset" option.

Peer-to-peer transfer

For peer-to-peer file transfers, a complication is that one or both peers may be behind a NAT router or a firewall (NAT stands for network address translation). Both NAT routers and firewalls often block unsolicited incoming network traffic.

If you run a server behind a NAT router or firewall, you must typically configure the router and/or firewall MANUALLY.

Several proposals exist for automatic configuration of NAT routers and firewalls (UPnP, SOCK5, MIDCOM), but none is widely used. However, most NAT routers and firewalls *do* support a technique called "UDP NAT traversal" or "UDP hole punching". As the name implies, this technique works with the UDP protocol; it is standardized in RFC 3489. There is some experimental success in making TCP hole punching work, but the TCP variety is much more dependent on a particular implementation of the TCP/IP protocols in a router. NAT traversal works far more reliably with UDP. NAT traversal REQUIRES a "rendez-vous" server, typically called a "STUN" server.

Despite what the name "UPD hole punching" implies, the technique does not open ports at the firewall behind your back. The hole punching protocol tells the NAT router and the firewall that incoming traffic on a particular port is solicited.

When you browse the web (and your PC is behind a NAT router and/or a firewall), the CONNECTION to the remote server tells the NAT router and firewall that any data coming from the server is in reply to the request that your browser made. That is: the browser initiated the transfer, so the router/firewall considers data arriving on that particular connection as solicited and valid.

So, your browser made a "hole" in the NAT router/firewall, so to speak, but only for that single connection. UDP hole punching is an extension of this for the case that the client (e.g. your browser) and the server are both behind a NAT router/firewall.

TFTP uses UDP as the transport protocol, rather than TCP. As such, peer-to-peer file transfer may be more easily and more reliably implemented with TFTP than with FTP, HTTP or other protocols based on TCP. Another advantage is that the implementation of TFTP clients and TFTP servers are similar, and it is therefore easy to build combined client/server hosts.

There is one issue with the TFTP protocol that hinders NAT traversal: according to RFC 1350, the client should choose a pseudo-random port number and send its request to the well-known TFTP server port number 69. For its reply, RFC 1350 states, the server should choose a new pseudo-random port number. That is, the server receives a request at port 69, but it sends its reply from, say, port 4362. NAT routers and firewalls may block this: a reply from port 69 would pass (because it was solicited) but a reply from a different port might not pass —this depends on the rules that a NAT router and/or firewall use; RFC 3489, describing STUN, describes four classes of NAT routers.

The reason that RFC 1350 describes the protocol as such, is for ease of implementation. A TFTP client should be prepared to receive a reply from a TFTP server at a different port then where the request was sent (*only* or read-requests and write-requests, by the way). However, there is no fundamental reason that the server *must* choose a different port. It is just simpler to implement a TFTP server where all data transports run over unique ports, because the demultiplexing of the network packets then happens in the operating system. When the TFTP server returns all replies over the well-known port 69, it must demultiplex incoming packets itself, using the source IP and port addresses. In fact, this is precisely how Weird Solutions made their "TFTP Turbo" product firewall-friendly and NAT-enabled. Other TFTP servers use a similar design (e.g. "Open TFTP Server", "Managed TFTP server").

Unreliable data streaming

Sometimes a quick delivery of packets is more important than the recovery of a lost packet. Streaming of audio and video are an example of this. For these purposes the UDP protocol is more appropriate than TCP, because TCP has reliable resending of lost packets built into the protocol. TFTP uses UDP as the transport medium, but it adds "reliability" itself.

To make TFTP more suitable for multimedia data streaming, I propose that the transmitter *never resends* blocks, that the transmitter uses a transmit window and that the receiver drops blocks that arrive out of order.

If the transmitter knows at what data rate it must send the blocks, it can ignore any ACKs that it receives. It will simply push any next block out of its queue at regular intervals. If the transmitter does *not* know how fast the receiver will process the data being streamed, the transmitter should fill a transfer window and use the received acknowledgements to determine how many blocks have been received and how much space that produces in the window. On receiving ACKs, the transmitter then sends new blocks to fill the window to maximum capacity. The client and the server should, of course, negotiate a window size.

Data streaming will work in combination with multicast mode. If the transmitter determines the pace of sending blocks, both mTFTP (PXE) and RFC 2090 are suitable. If the receiver must INDICATE the server what pace to use, via block acknowledgements, the RFC 2090 procedure has the advantage that the server can switch to another host as the master client as soon as the active master client disconnects from the group.

Summary of the extensions

The table below list the options that are standardized in various RFCs, in use is diverse TFTP implementations, or proposed in this article.

Option	Parameter	Notes
blksize	8 .. 65464	Block size, excluding protocol headers. The default block size is 512. Defined in RFC 2348.
blksize2	8 .. 32768	Block size restricted to powers of 2, excluding protocol headers. *Non-standard*, but common.
multicast	addr, port, master	Multicast, defined in RFC 2090.
password	text	Password or a combined string of the user name and the password. *Non-standard*.
rollover	0 or 1	Block counter roll-over (roll back to zero or to one). *Non-standard*.
timeout	1 .. 255	Time-out in seconds. Defined in RFC 2349.
toffset	numeric	Transfer offset in bytes, for partial transfers. *Non-standard*.
tsize	numeric	Transfer size in bytes (size of the file being transferred). Defined in RFC 2349.
window	1 .. 255	Window size, in blocks of "blksize" (or "blksize2") bytes. *Non-standard*.

The mythical TFTP flaws

In closing, a few words on the suitability of the protocol. TFTP is *by intent* a light-weight and low-complexity protocol. It may not be suitable for purposes that REQUIRE optimal performance or services beyond basic file transfer. However, the critique of the protocol is often exaggerated. As an example, below is an excerpt of RFC 3617:

Use of TFTP has been historically limited to those devices where a more full protocol stack is impractical due to either memory or CPU constraints. While this still may be the case with a toaster, it is unlikely to be the case for even the simplest piece of network support hardware, such as simple routers or switches. There are a myriad of reasons to use some protocol other than TFTP, only a few of which are listed below.

TFTP has no mechanism for access control within the protocol, and there is no protection from a man in the middle attack. Implementations are left to their own devices in this area. Because TFTP has no way to determine file sizes in advance, implementations should be prepared to properly check the bounds of transfers so that neither memory nor disk limitations are exceeded.

TFTP is not well suited to large files for the following reasons. TFTP has no inherent integrity check. There is no way to determine what one side sent is what the other received. There is no way to restart TFTP transfers from anywhere other than the beginning. TFTP is a lock step protocol. Only one packet may be in flight at any one time. There is no slow start or smart backoff mechanism in TFTP, but very simple timeouts.

TFTP is not well suited to file transfers across administrative domains. For one thing, TFTP utilizes UDP, and many NATs will not either support or allow TFTP transfers. More likely firewalls will prohibit transfers.

There are no caching semantics within TFTP. There is no safe way to cache information using the TFTP protocol. This diatribe lacks nuance and it is inaccurate in part. Below, I will attempt to give a more balanced view of the TFTP protocol, in relation to the prevalent file transfer protocols (HTTP, FTP).

- Small, memory-constrained devices exist and are still actively developed & researched. To give just one example: wireless sensor networks: these are severely memory/CPU-constrained devices that still must take part in a LAN. Routers and switches are actually complex, mains-powered, network-centric devices. For battery-powered devices low-power operation may be a quite important design

criterion. So while "there are [sic] a myriad of reasons to use some protocol other than TFTP", there are also reasons to choose a light-weight protocol like TFTP.

- TFTP has indeed no defined way for authentication and it does not standardize access control. For the specialized task that TFTP is often used for, authentication and access control is not important. Note that by far most transfers over HTTP and FTP are anonymous, unauthenticated. If authentication and access control are important, they can be added via options. That said, there is currently no standard for such options, other than a rudimentary implementation in Linksys routers.
- TFTP is prone to "man in the middle attacks", like FTP, HTTP, SMTP, POP3 and many, many others. To avoid "man in the middle attacks", the typical solution is to run a standard protocol over a secure tunnel —this is how HTTPS and SFTP work. TFTP can be tunnelled equally well. Implementations are therefore *not* truly "left to their own devices in this area".
- TFTP *does* have a way to determine file sizes in advance, see RFC 2349.
- TFTP *does* have an "inherent" integrity check of received data. In fact, it uses *the same* integrity check as FTP and HTTP (and many other protocols), namely: the Internet checksum (RFC 1071).
- In the standard unicast mode (RFC 1350) TFTP does not support partial transfer, but multicast mode according to RFC 2090 may be used to support partial

transfers: TFTP clients could request a multicast transfer just for the purpose of a partial transfer.

- Most TFTP implementations indeed use simple time-outs (the time-out value itself can be negotiated, see RFC 2349).

  However, if packets are lost due to errors in reception (wireless networks, cable interference) rather than network/host congestion, the slow-start and back-off algorithms in the TCP protocol decrease throughput for the wrong reason, leading to poor performance. In these situations, TFTP's simple time-outs may well *give better performance* than TCP-based protocols. This is a well-known TCP limitation, see for example RFC 908, and tests performed at the University of Amsterdam and Oak Ridge National Lab.

- NAT router and firewall (mis-)configuration is barely a valid criticism of a protocol.

  Similarly to TFTP, "active mode" FTP has an inherent problem with NAT routing, because it opens the "data channel" at a random port. NAT routers and firewalls have been adapted to specifically filter FTP command packets. If adjusting routers and firewalls is a proper fix to make FTP work, why would it be a "protocol issue" for TFTP?

  Moreover, TFTP servers can be (and have been) implemented with a static port without changing the protocol —nullifying this argument.

- Numerous key protocols use UDP, including the Domain Name System (DNS), the Simple Network Management Protocol (SNMP), the Dynamic Host Configuration Protocol (DHCP) and the Routing Information Protocol (RIP). Many peer-to-peer applications (e.g. multi-player games), digital streaming media and Voice-over-IP systems (VoIP, H.323 protocol) use UDP as the transport protocol. Therefore, NAT routers and firewalls blocking UDP completely are much, much rarer than folklore has it.
- I do not understand why caching semantics are a protocol issue; they appear to me an implementation issue. I do not understand why caching would be safer with other protocols than with TFTP. I think this is a bogus argument.

There you have it. Some arguments are irrelevant; some are false. TFTP compares well with HTTP and FTP, the prevalent file transfer protocols. TFTP has disadvantages and limitations: no authentication, slow on high-latency links and potential problems with large files (over 32 MiB) are the most prominent. It has advantages over HTTP and FTP too: a simple implementation, a low resource usage, good performance in the presence of dropped packets (wireless networks in noisy environments), support for multicast transfers (where multiple clients receive the same data from a server concurrently) and support for peer-to-peer transfers over most NAT routers (without needing to configure the router).

Source: http://www.compuphase.com/tftp.htm