HTun: Providing IP Service Over an HTTP Proxy

Moshe Jacobson, Ola Nordström, Russell J. Clark
College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280
moshe@runslinux.net, ola@triblock.com, rjc@cc.gatech.edu

Abstract— **In many cases, application service is replacing IP service as the de facto form of network connectivity. One example of this is the popular use of an HTTP proxy service as a substitute for IP Internet connectivity. This work presents and analyzes *HTun*, a system for encapsulating IP traffic into HTTP requests. This allows users to create a tunneled IP-layer link between two hosts connected through nothing more than a web proxy. This paper will describe the design, implementation and performance analysis of the protocol.**

Keywords— **HTun, IP over HTTP, VPN, HTTP, tunnel, tunneling, proxy, IP, protocol, Linux**

I. Introduction

For many users, the Internet service is largely *web-centric* with little focus on other applications. This has led to the deployment of network environments that limit network access to HTTP via proxy servers. In such environments, new and emerging applications cannot be readily adopted by users. With HTun, we propose and demonstrate an IP over HTTP tunneling protocol that provides a full IP infrastructure in a strict HTTP environment.

There is some previous work in providing general tunneling services over the Internet. The VTUN [?] application supports tunneling virtual networks over TCP connections. The HTTP Tunnel [?] and Desproxy [?] projects provide specific TCP port tunneling over HTTP. However, there is no known general solution for supporting virtual IP connectivity through application proxies such as HTTP. An IP tunneling capability of this nature will allow users to create a full, IP-layer VPN between two hosts, complete with a virtual IP network interface on either side.

The goal of the current work is to design and implement a mechanism for providing full IP connectivity through a restrictive HTTP proxy service. The protocol, which we call *HTun*, addresses two main design challenges: (1) inefficiencies due to the large overhead associated with HTTP requests and (2) the limitation that inbound connections cannot be established through an HTTP proxy. While this work has focused on the issues of providing IP service over HTTP, the results are more generally applicable to the provision of a wide range of networking services over restrictive transport services such as the Small Message Service (SMS).

This paper is organized as follows. The next section presents an overview of the HTun solution. Section **??** describes the initial *Half-Duplex* design for HTun, followed in Section **??** by the implementation results of this approach. Section **??** describes the modified *Full-Duplex* design for HTun which overcomes several issues with the prior design. Section **??** describes the details of the HTun implementation project. Current open issues and future work are discussed in Section **??**.

II. Design Overview

The architecture for the HTun service is depicted in Figure **??**. The HTun user is on a client system in a restricted network, likely behind a firewall. This network is not routable to the Internet and likely uses a private address space. The client's only access to the Internet is through the HTTP proxy.

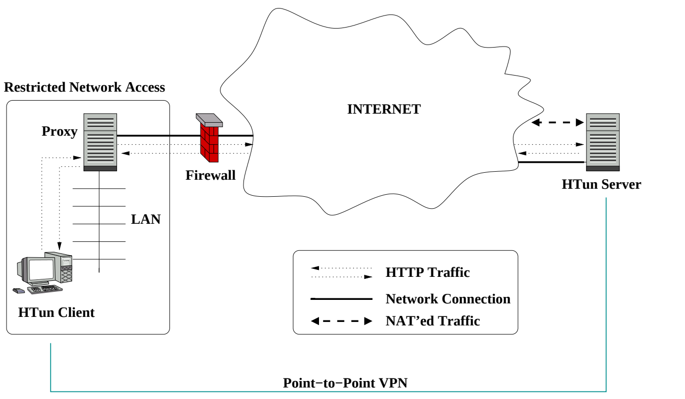

Fig. 1. HTun operating in a network

In order to provide IP level service, HTun utilizes a support server outside of the firewall. This *HTun Server* responds to standard HTTP requests, as any other web server, but it also performs the reverse encapsulation of the IP packets it receives from the HTun client. Routing and address translation are then performed by the operating system on which the HTun server is running. Effectively, all HTun clients will appear as though they are attached to the Internet through this HTun server.

A. HTTP Compliance

Of primary concern in the HTun design is the ability to provide the IP service through standard, unmodified HTTP proxies. The HTun protocol therefore uses standard HTTP requests to maximize the chances that an arbitrary proxy server will forward them properly.

The HTTP *POST* method allows for the inclusion of arbitrary amounts of data in both the request and response message bodies. This makes it the natural choice for providing the core transport method of the HTun protocol. This has the added advantage that proxy servers will not interfere with the intended IP packet flow by caching the responses [?]. The POST method also eliminates some overhead by sending and receiving data in the same request.

The basic idea is that a *send queue* of IP packets is maintained by both the *HTun client* (the restricted host behind the firewall and/or proxy) and the *server* (the open host on the Internet). When the client's send queue is nonempty, it makes a POST request to the server, transmitting its entire send queue in one POST request body. The POST request body may contain an arbitrary number of packets, whose aggregate length is specified in the `Content-Length` header of the request. The server then responds with any packets in its send queue by placing the raw packets into its response body, setting the Content-Length header in the same manner as with the request.

B. Polling

One obvious problem is that the server is unable to send its queued packets to the client without the client initiating a POST request. This is inherent in the client/server model of HTTP. The HTun design solves this problem by having the client periodically poll the server to check for pending data.

It is far more likely for the server to have queued response data immediately after data has been transferred (traffic is highly correlated) in either direction. To optimize overhead in light of this observation, the client is designed to poll more frequently at first and then reduce the polling rate as the idle time (time since the last data is transferred) increases. Any time a poll request returns data or the client has data to send, the poll rate is reset to its original high frequency.

C. Persistent Connections

It is desirable to avoid the overhead of requiring a separate HTTP request and TCP connection for each tunneled IP packet. HTun is therefore designed to take advantage of persistent connections with HTTP. HTTP/1.1 includes persistent connections as part of its base specification. HTTP/1.0 also supports persistent connections, but as an extension to the specification. Because of the obvious and immediate performance gains involved, however, it appears that almost all implementations of HTTP/1.0 include support for persistent connections.

D. Dynamic IP Configuration

The server should support multiple clients simultaneously. This means that it must bring up a separate virtual network interface for each one, as it must communicate with each client over its own VPN. Typically, one would simply use DHCP for dynamically assigning IP addresses to the clients. However, virtual interfaces such as those used with an HTun VPN are point-to-point interfaces; that is, they provide access to one and only one other host. They are also *virtual private* interfaces, meaning that there should be no way for outside hosts to route to them. Therefore, the assigned IP addresses are typically nonroutable private addresses, e.g. the 10.*.*.* or 192.168.*.* ranges.

Because the IP addresses are nonroutable, DHCP cannot be used, as a client may be multi-homed, and have interfaces on different private IP networks. If a DHCP server were to provide some nonroutable IP for the client to use, and it already had a route to a host by the same nonroutable IP, a conflict would arise, and the client would no longer be able to address the host on its own network by the same IP address.

Because of this problem, the HTun protocol must support some method of allowing the client and server to negotiate the set of IP addresses that are acceptable for use by both of them. The client tells the server what ranges of IP addresses are free for its own use, and the server compares this list to its own. The server would then assign a pair of IP addresses to be used with this interface, and inform the client of this decision so the client can bring up an interface with those IP addresses as well.

Once the client and server have both brought up the virtual interfaces with acceptable nonroutable IP addresses, they will be able to pass IP packets between each other successfully.

III. HTun Protocol 1 - (Half Duplex)

This section presents a detailed description of the HTun protocol as originally designed. This is referred to as the Half-Duplex version of the protocol. Several issues arose during performance testing, which led to the development of HTun's second (Full Duplex) protocol, described in Section **??**.

Since HTun requests and responses are transmitted as valid HTTP requests, we describe the HTun protocols in terms of HTTP. The following description assumes the terms Request, Method, Request-URI, HTTP-Version, Headers, Header Field, and Message Body as defined in RFC 2616 [?].

A. General message format

All HTun messages follow a similar format. The common elements are presented here to give the general flavor of the messages. These protocol grammars are not entirely formal, but are rather an approximation of the HTTP definitions given in RFC 2616.

A.1 Request message format

The general `request` message format looks like this:

```
request      := request-line[CRLF]
                Content-Length: [msg-body-len][CRLF]
                Proxy-Connection: [keep-alive][CRLF]
                [CRLF]
                [msg-body]

request-line := POST [server-addr]/[action] HTTP/1.0
server-addr  := http://[proxy-ip]:[proxy-port]
keep-alive   := Keep-Alive | Close
```

`server-addr` specifies to the proxy server on which host and port it can find the HTun server. This is the standard HTTP absolute URI format defined in RFC 2616, without a trailing pathname, and is specified in all client requests.

Specifically, the `proxy-ip` is the IP address of the HTun server specified in dotted decimal format. The `server-port` is a TCP port on which the HTun server is listening for connections. If a proxy server is not being used, this address:port specification is ignored by the server.

The `msg-body` is used for a different purpose in each message, but the `Content-Length:` header always has a value of `msg-body-len`, a decimal integer corresponding to the length, in octets, of this message body.

`CRLF` is used to indicate a carriage return (ASCII 015), followed by a linefeed (ASCII 012). This sequence should be present at the end of every line of headers, including the request-line.

A.2 Response message format

The general `response` message format looks like this:

```
response     := status-line[CRLF]
                Content-Length: [msg-body-len][CRLF]
                Proxy-Connection: [keep-alive][CRLF]
                [CRLF]
                [msg-body]


status-line  := HTTP/1.0 [status] [status-reason][CRLF]
```

`msg-body-len` is the message body length as specified above.

The `status` and `status-reason` are the HTTP status and short textual description of the status, as outlined in RFC 2616.

The `msg-body` is used for different things depending on the type of message being sent back by the server. If the `status` is an error value (in the 400s or 500s), the `msg-body` is a textual description of why the error response is being sent. If the `status` is 200, the `msg-body` is binary data representing raw IP packets. If the `status` is 204, the `msg-body` is empty (0 bytes long), and the `Content-Length` header is set to reflect this.

B. Establishing the communication channel

For the client to send data to the server, it must first establish the channel of communication. It sends a configuration request message to the server, which includes a unique client identification key, as well a range of IP addresses suitable for use by the client. When the server responds to this message, it lists the IP addresses the client should assign to its point-to-point interface. The client must then bring up the interface as specified.

B.1 Channel configuration request

The client initiates the channel with a config-request as follows:

```
config-request := POST [server-addr]/CP1 HTTP/1.0[CRLF]
                  Proxy-Connection: Keep-Alive[CRLF]
                  Content-Length: [config-req-body-len][CRLF]
                  [CRLF]
                  [client-key][CRLF]
                  [ip-range]
                  [ip-range]*

ip-range       := [ip-range-base]/[ip-range-maskbits][CRLF]
```

The `Proxy-Connection: Keep-Alive` causes the TCP connection to the server to be kept open. This is essential for the server to maintain a state with the client, as the client's subsequent requests will not contain any unique information about the client. The only way for the server to determine which client the request is coming from is by the socket on which it communicates.

The `client-key` is a identification string of 12 hexadecimal digits unique to this client. As there is no way for multiple clients to determine whether their key is unique among all clients, a useful value to use in this field is the Ethernet MAC address of one of the physical network interfaces in the machine on which the client daemon is running.

The `ip-range` is used to indicate to the server what IP addresses the server may consider when determining what IP address to dynamically assign to the client. The client should present to the server a set of addresses that will not collide with the local addresses already known or used by the client. Each ip-range specifies an `ip-range-base`, a contiguous block of IP addresses given by a base IP address, and an `ip-range-maskbits`, the number of '1' bits in the netmask.

For example, an ip-range of 10.0.0.0/8 is analogous to the more commonly-used notation 10.0.0.0/255.0.0.0, which indicates that all 10.*.*.* IP addresses are acceptable.

Multiple ip-range specifications may be sent to the server. The server should compare the entire list with its entire list to determine an acceptable address pair.

B.2 Successful configuration response message

When the server has successfully agreed on a pair of IP addresses, and allocated all necessary server resources, it must respond to the client with a message containing the IP addresses that the client should use. The config-response message is sent as follows:

```
config-response := HTTP/1.0 200 OK[CRLF]
                   Connection: Keep-Alive[CRLF]
                   Content-Length: [config-resp-body-len][CRLF]
                   [CRLF]
                   [ip-addr][CRLF]
                   [ip-addr][CRLF]
```

The first `ip-addr` is the IP address that the server has assigned the client, in dotted decimal format, and the second `ip-addr` is the IP address that the server has assigned itself, also in dotted decimal format.

C. Error response: Server Error or Busy

If an internal error occurs on the server, or the client limit has been reached while trying to process the client's configuration request message, the server must respond with a 500 error message as follows:

```
config-500-response := HTTP/1.0 500 [500-reason][CRLF]
                       Connection: Close[CRLF]
                       Content-Length: [500-desc-len][CRLF]
                       Content-Type: text/plain[CRLF]
                       [CRLF]
                       [500-desc]

500-reason          := Internal Server Error | Busy
```

The `500-reason` can specify that the server was too busy (i.e., there were already too many clients connected), or that there was some other sort of internal error that was not the client's fault. Regardless, a 500 response, as specified by the HTTP RFCs, is likely a transient error, and the client should try the operation again later. The HTun server may also send a `503 Access Denied` or `400 Bad Request` error response, as necessary, following the format given for `config-500-response`, above.

D. Sending and Receiving Data

This section describes the Protocol 1 Send Data request. This request actually involves sending *and* receiving data. The client sends a POST request, with a body containing an arbitrary number of raw IP packets, and the server responds with a message-body containing the raw IP packets it has queued to return to the client.

D.1 Client request

Once the client has established and configured an open channel with the server, and it has data to send, it can send the data to the server using a send request as follows:

```
send-request := POST [server-addr]/S HTTP/1.0[CRLF]
                Proxy-Connection: Keep-Alive[CRLF]
                Content-Length: [send-data-len][CRLF]
                [CRLF]
                [send-data]
```

The client places one or more raw IP packets into the body of the POST request, indicated in the above grammar by the `send-data` field. The server will read packets from the client stream until it has reached `send-data-len` bytes of raw IP data.

D.2 Server success response with data

If the server successfully processed the client request, and it has data to return to the client, its response is as follows:

```
resp-data := HTTP/1.0 200 OK[CRLF]
             Connection: Keep-Alive[CRLF]
             Content-Length: [resp-data-len][CRLF]
             [CRLF]
             [resp-data]
```

The message-body, shown here as `resp-data`, contains one or more raw IP packets to be sent to the client. The client reads packets until `resp-data-len` bytes of raw IP data has been read.

D.3 Server success response with no data

If the server successfully processed the client request, and it has no data to return to the client, its response is as follows:

```
resp-nodata := HTTP/1.0 204 No Data[CRLF]
               Connection: Keep-Alive[CRLF]
               [CRLF]
```

E. Polling for data

When the client has no data to send, but wishes to check if the server has data to send it, it sends the server a `poll-request`.

```
poll-request := POST [server-addr]/P HTTP/1.0[CRLF]
                Proxy-Connection: Keep-Alive[CRLF]
                Content-Length: 2[CRLF]
                [CRLF]
                :)
```

The server response message formats are the same as the valid responses for a *Send* request.

F. Closing the connection

When the client exits or is killed, it should send the server a `finish-request`, indicating to the server that it will not be making any more requests, and that the server can de-allocate any resources used to support that client.

```
finish-request := POST [server-addr]/F HTTP/1.0[CRLF]
                  Proxy-Connection: Keep-Alive[CRLF]
                  Content-Length: 2[CRLF]
                  [CRLF]
                  :(
```

The server should send a 204 response similar to the `resp-nodata` response defined above, but with a `Connection:` header of `Close`, so that the proxy server can close the connection:

```
resp-nodata := HTTP/1.0 204 No Data[CRLF]
               Connection: Close[CRLF]
               [CRLF]
```

IV. Protocol 1 Performance Analysis

In order to evaluate the HTun protocol we developed a prototype implementation for Linux 2.4.7. The details of the implementation are described in Section **??**. In this section we discuss the results of the half-duplex protocol and motivate the subsequent discussion.

We evaluated our initial protocol design in terms of latency and throughput. The network setup was similar to that shown in figure **??**. The HTun client was connected to a Proxy server via a 100Mbit/sec switched network. The Proxy, also running Linux, had a second interface which was connected to the Internet via a routable address in the Georgia Tech College of Computing.

We chose the widely available Squid Proxy Server [**?**] (version 2.4). The HTun server was 4 router hops away from the Proxy with an average ping response time of less than 10 milliseconds.

A. Throughput

Throughput was tested by transferring large (3-4MB) uncompressable files between the HTun client and a host on the IP network. The average data throughput observed using HTun in our test environment was 540 KBytes/sec. This test was performed using the *wget* application across HTun and a Squid proxy. Using the same system and network configuration but directly routed without the HTun and proxy interface, an average throughput of 860 KBytes/sec was achieved. Thus the overhead associated with HTun reduces throughput in our test environment by roughly 37%.

B. Latency

Latency was tested using a series of *ping* response times. Tests were performed both in isolation (no other traffic over the HTun VPN) and with a background file transfer. This background traffic offers additional load on the HTun channel but also serves to keep the data pipeline open across the proxy server. Latency was also tested empirically by observing the usability of a remote login terminal (via SSH) over the HTun VPN. Our experience is that the remote terminal session is quite usable as long as the variance of the response times is not too high – that is, the delay is predictably long for every packet sent.

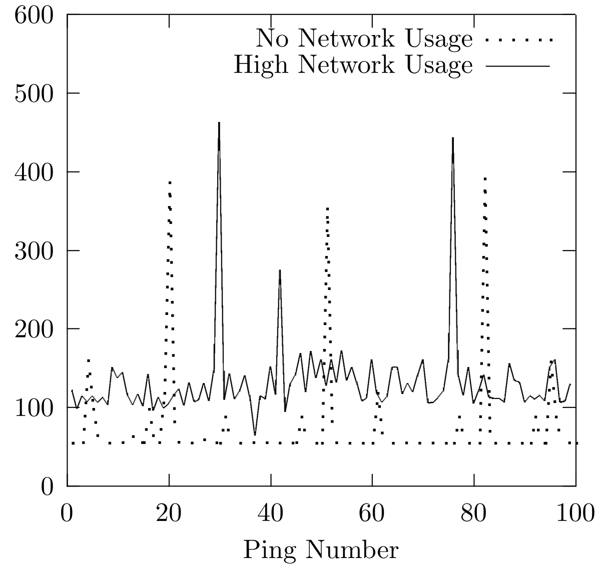

Fig. 2. Protocol 1 Outbound Ping times (msec)

The ping tests were performed in both directions across the HTun VPN. Figure **??** shows a comparison of the outbound client pings (from HTun client to Internet host) without background traffic and while transferring a file from the server to the client. The baseline ping round trip time increases and remains somewhat erratic during the file transfer (high network usage). The spikes during both tests represent ping response misses. In these cases, the ping response has not returned to the HTun server before the server responds to the current HTTP request. Thus the ping response is queued at the server until the next HTTP request (data or poll) arrives from the HTun client.

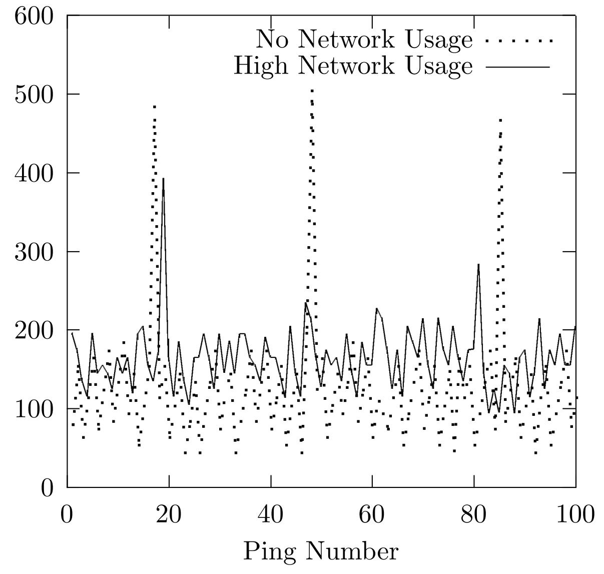

Fig. 3. Protocol 1 Inbound Ping times (msec)

Figure **??** shows a plot of Inbound (Internet host to HTun client) ping response times. Overall, the round trip time is higher and much more erratic than the outbound case. This is especially true for the tests with no background traffic. This behavior is attributed to the fact that the client drives the communication, polling the server for data and initiating server to client data transfers.

C. Theoretical Problems

While we were satisfied that HTun worked, there appeared to be several areas for improvement. In this section, we discuss the issues identified and their impact on performance.

C.1 Half Duplex Communication

Most common network applications utilize a two-way exchange of packets either explicitly in a command response protocol or implicitly through the use of TCP with its DATA/ACK exchange. The problem for HTun is that when data is sent from a client in the body of a POST request, the application-level reply is not immediately available. If the HTun server replies with only the currently queued data, the application response will likely not yet be available, and will therefore be delayed at the HTun server until the next POST request arrives from the client.

One solution we explored was to introduce a delay at the server before the POST response is sent. The hope is that during this interval, the packets containing the application response will arrive at the HTun server, making them available to the immediate POST response. We implemented this technique in the Protocol 1 version using a set of tunable parameters in the server's configuration.

C.2 HTTP Overhead and Fixed Sizes

The protocol overhead of the HTTP headers is fairly high and creates another hindrance to HTun performance. Each packet of data to and from the HTun client carries the additional overhead of three extra layers. First, there is the overhead of the lower layer TCP/IP, which adds approximately 40 bytes per packet of HTTP traffic. The HTTP traffic with HTun's minimal HTTP headers then adds approximately 100 bytes per POST request. However, most proxy servers add additional headers when making the proxied request. The final header size turns out to be around 200 bytes depending on the proxy server. In our testing with HTun we observed that most POST requests carry between 1 and 17 tunneled IP packets, with the large majority carrying between 1 and 3 packets.

Assuming an MTU of 1500 bytes, if the POST request carries 3 full packets, it is converted to 4 packets in the HTTP request, the 4500 bytes becomes $(4500 + 4 \times 40 + 200) = 4860$ bytes. If the tunneled packets are smaller or there are fewer of them available for a post request the effect of the HTTP protocol overhead becomes more pronounced.

While a certain amount of overhead is inevitable with any tunneling scheme, the problem is made worse by the fact that a new POST request and response must be initiated for every collection of packets that must be sent. This creates extra overhead both in terms of the time to make the request as well as the additional traffic from HTTP headers.

C.3 TCP Flow Window Limitations

Tunneled applications desiring high bandwidth experience another performance bottleneck due to the TCP flow control algorithms. Some applications, such as streaming video, are developed using datagram services which allow them to continuously send packets. One might imagine that HTun would queue up these packets and deliver a large number of them in a single POST request, thereby decreasing per-packet overhead. In reality, however, the TCP flow control mechanism causes the TCP stream to stop queueing unacknowledged packets after a certain point.

In our tests, we observed that the default Linux TCP implementation allows no more than 11 unacknowledged packets to be sent. After sending 11 data packets, TCP returns 6 ACK packets using piggybacked acknowledgments.

V. HTun Protocol 2 - (Full Duplex)

It was decided that two changes to the HTun protocol would eliminate many of the performance issues discovered in the original design. First, the protocol should be full duplex with two open channels to the server. Second, HTun should take advantage of the HTTP/1.1 feature of Chunked Transfer-Encoding.

A. Two Channels

If the HTun client maintains two open channels to the server, one is used for sending data, while the other is used for receiving data. This eliminates the half duplex communication problems and removes the need for the delayed response algorithms in the server.

The revised protocol allows for a client to negotiate a send and a receive channel with the server. By separating the two and enabling full duplex communication, we hoped to increase the throughput and minimize the erratic latency present in the half-duplex protocol.

In our design, the first channel continues to work over TCP port 80, the second will typically run over port 8080, since we discovered that the Squid default configuration of allowed ports includes 8080 as well. Multiple ports were used because we found that proxy servers generally did not allow two separate data streams to be opened between the client and one port on the server.

B. Chunked Transfer-Encoding

HTTP/1.1 allows for Chunked Transfer-Encoding which is useful when a sender does not know the size of the data it will transfer. The sender can specify the `Transfer-Encoding` as `Chunked`. This allows for an arbitrary amount of data to be sent in one *chunk*. The HTun client could then send data continuously in chunks over its send channel to the server. This would eliminate the need for HTTP headers at every transfer and would also do away with the need to wait for the other side to reply with a *200 OK* after each transfer. This would streamline the HTun protocol and reduce HTTP overhead.

Unfortunately, Chunked Transfer-Encoding is not supported by the current version of the Squid Web Proxy. The only Proxy we identified that currently supports Chunking is Jigsaw [**?**], a reference implementation HTTP/1.1 Proxy written by the W3C in Java. When testing the usability of Chunked Transfers we discovered that indeed they were supported in Jigsaw but Jigsaw does not forward a client's chunked data stream until it receives the entire stream. Jigsaw appears to assemble the data and transfer it in one contiguous HTTP request to the receiving HTTP server, rendering the feature useless to us. As a result, we had to continue the use of `POST` requests and not rely on Chunked Transfers for HTun.

C. Efficiency and Other Concerns

Protocol 2 contains roughly the same amount of HTTP overhead as protocol 1 since the same session semantics are used. However the required server wait which we designed into our implementation of Protocol 1 is not required since each channel operates independently of the other.

C.1 Behavior in Unexpected Situations

We decided that it would be desirable for the client to automatically reconnect should one of its data channels sporadically disconnect. Due to the nature of our channel negotiation process, a disconnected receive channel would simply mean a reconnect of that channel to restore the tunnel.

In the case that the client's send channel is disconnected, the problem is more serious, because the send channel is the primary channel for negotiation of the tunnel. In this case, the client must shut down all connections to the server and renegotiate from scratch.

Our server implementation allows the client to be disconnected for a tunable amount of time before its configuration data (such as its IP address and any pending packets) is removed. Therefore, if the client reconnects quickly, it should be able to resume its broken session seamlessly.

C.2 Client Crash

If the client crashes, the server should keep up its end of the VPN and wait for a config request matching the clients unique identifier. As mentioned above, the server shuts down VPNs after they have been inactive for an amount of time specified in the configuration.

C.3 Proxy Disconnect

If the proxy disconnects either channel, the client will simply reopen its send channel and re-negotiate the VPN IP addresses with the server before continuing.

C.4 Server Crash

If the server crashes, the proxy may disconnect the client as well. In this case, the client will also renegotiate. In the event that the proxy keeps its connection with the client and reopens its connection to the restarted server, the server will not know about the current connection it has with the client. It will respond with an error, forcing the client to reconnect.

VI. HTun Protocol 2 Specification - (Full Duplex)

The specifications for the second protocol use all the same semantics as the first. The send channel is used as the control channel to initially obtain the server-specified VPN IP addresses.

A. Send Channel Negotiation

The send channel configuration request is as follows:

```
config-request := POST [server-addr]/CP2 HTTP/1.0[CRLF]
                  Proxy-Connection: Keep-Alive[CRLF]
                  Content-Length: [config-req-body-len][CRLF]
                  [CRLF]
                  [client-key][CRLF]
                  [ip-range]
                  [ip-range]*

ip-range       := [ip-range-base]/[ip-range-maskbits][CRLF]
```

Note that the *Configure Protocol* indicator changed from CP1 to CP2 to reflect the change from Protocol 1 to Protocol 2. The server will respond with a `config-response` identical to that used in protocol 1.

B. Receive Channel Negotiation

Once the client's send channel has obtained a pair of IP addresses, the second (receive) channel is opened to the server with a config-request as follows:

```
config-request := POST [server-addr]/CR HTTP/1.0[CRLF]
                  Proxy-Connection: Keep-Alive[CRLF]
                  Content-Length: [config-req-body-len][CRLF]
                  [CRLF]
                  [client-key][CRLF]

ip-range       := [ip-range-base]/[ip-range-maskbits][CRLF]
```

The server uses the client-key to locate the requested VPN and then ties that socket to its receive channel.

C. Sending Data

The client sends data via the send channel, using the same send-request as with Protocol 1. Since HTTP is a request-oriented protocol, the server will send a *respond-nodata* back after each transfer.

D. Receiving Data

The client *receive* request is slightly different in Protocol 2. Because the client could potentially be waiting for data for a very long time, we need to ensure that there be an upper limit on the amount of time the server takes to respond to a *receive* request. Otherwise, the proxy may time out and close the connection due to inactivity.

With this protocol, the upper limit is specified by the client in the body of the request, as follows:

```
recv-request := POST [server-addr]/R HTTP/1.0[CRLF]
                Proxy-Connection: Keep-Alive[CRLF]
                Content-Length: [config-req-body-len][CRLF]
                [CRLF]
                [time][CRLF]
```

`time` is simply a positive integer representing the maximum response wait time in seconds. The server should respond a second or two *before* the timeout, and the client should optimally wait a second or two longer than it actually specified. This ensures that network lag does not fool the client into thinking that the server is unresponsive.

E. Closing the Connection

To close the connection, the client sends the `finish-request` as described in section ??.

VII. Protocol 2 Performance

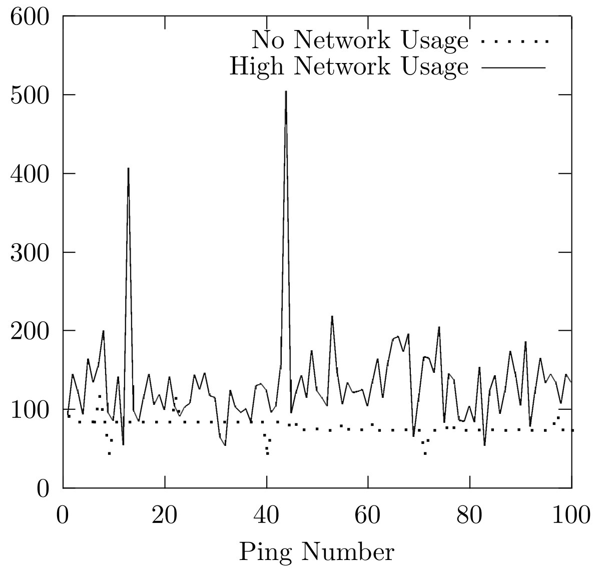

Fig. 4. Protocol 2 Outbound Ping Times (msec)

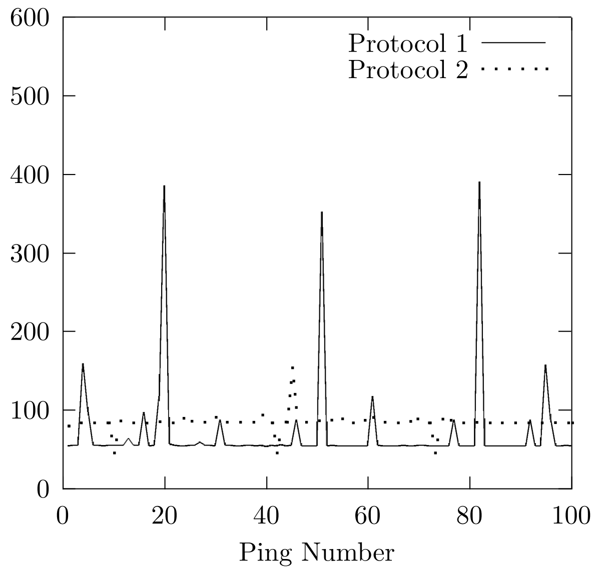

Fig. 5. Protocol 1 vs 2 Outbound Ping Times (msec)

Figure ?? shows the Outbound ping times for Protocol 2, contrasting the conditions under high and low network utilization. Figure ?? contrasts Outbound ping times between Protocols 1 and 2. It is clear that the latency with Protocol 2 is much more consistent with a lower variance than Protocol 1. This is especially true for the case where there is no background traffic over the HTun VPN. Our empirical ssh login test also confirmed this; the terminal was much more responsive and latency was more consistent.

The surprising result is that the average latency is higher under Protocol 2 (by up 15 ms) than under Protocol 1. This appears to be related to the overhead in maintaining two queues, though we are not completely satisfied with that explanation. One theory was that it was a load balancing problem in the Squid proxy, but our benchmarking without a proxy indicates that the anomaly is somewhere in the HTun implementation itself.

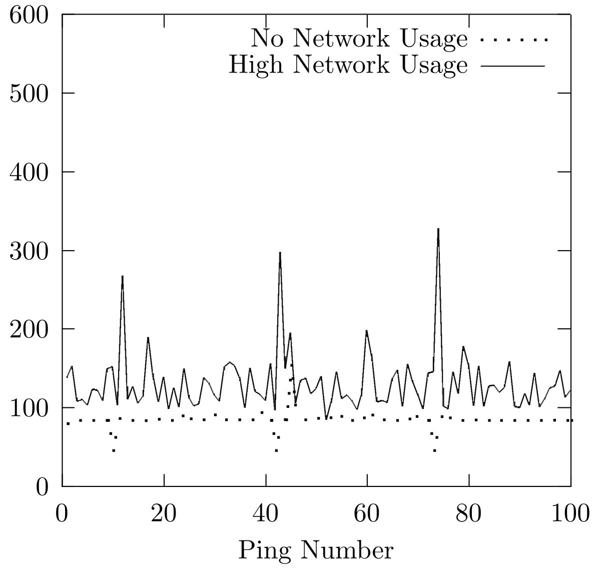

Fig. 6. Protocol 2 Inbound Ping Times (msec)

Figure ?? shows the Inbound ping times under high versus no network utilization. The average inbound and outbound ping times for Protocol 2 under no network load were 78 and 84 milliseconds, respectively. The average Inbound and outbound ping times during high network load were the same at 131 ms, but the variance for inbound ping times was slightly higher.

Figure ?? shows the dramatic improvement in inbound ping latencies for Protocol 2 in comparison to Protocol 1. Both figure ?? and figure ?? show the lack of ping response misses (latency spikes) in Protocol 2. This was one of the primary design goals for the full duplex approach, and it was succesfully achieved.

VIII. Implementation Details

This section describes some of the details of our implementation experience with HTun. The primary development platform was Linux 2.4.7. The final implementation of HTun (server and client) was written in C using POSIX threads. It was developed for Linux, though it should work

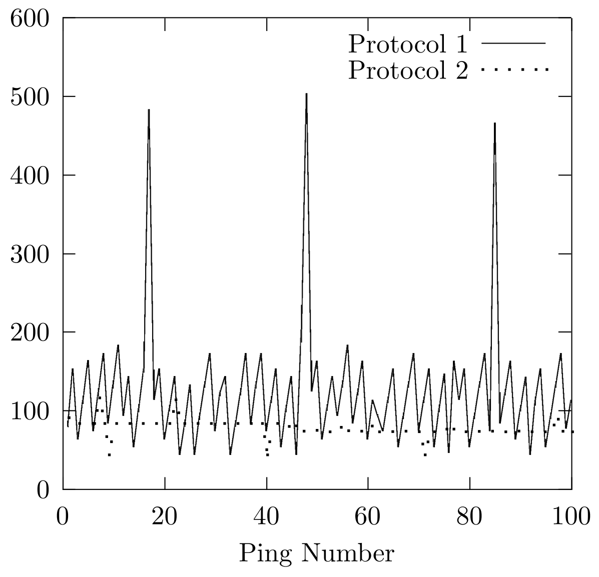

Fig. 7. Protocol 1 vs 2 Inbound Ping Times (msec)

on FreeBSD and Solaris with minor changes to the code that manipulates the `tun` interfaces and the route tables.

A. The Universal TUN/TAP Driver

The HTun implementation utilizes the Universal TUN/TAP device driver [?]. The TUN/TAP driver provides a simple virtual network interface for use with HTun. It manages a logical device file (in Linux /dev/net/tun), and automatically brings up a new virtual network interface *each time* a program makes an open() system call on the file.

Each `tun` virtual interface is connected to the file descriptor obtained from the open() call that created the interface. When the kernel writes packets to the `tun` interface (for instance, when a program requests that data be sent), the program can use the read() system call to read the raw packet data from the associated file descriptor.

Conversely, when the program writes raw packets to the file descriptor using write(), the packets appear on the interface, as if they had just arrived on the interface.

The TUN/TAP driver provided the functionality necessary to create the virtual interface, allowing for an easy way to process packets written to and read from the virtual interface in user-space. Although we have chosen to use the TUN/TAP driver as our virtual network interface, this interface is available only for the platforms mentioned above, so implementors may choose to provide this functionality in some other way when writing client or server software for other platforms. The TUN/TAP driver allowed us to set up a point to point VPN without writing any kernel code.

B. Multiple client support

The HTun protocol was designed to allow multiple clients to be simultaneously connected to a given HTun server. This means that every data transfer request received by the server must be associated with a particular known client before the data is read, because the data must be written to the proper file descriptor in order to be delivered to the proper interface.

The way the protocol was initially designed, every POST request sent by a client contained its VPN IP address in the request-line. This design turned out to scale poorly when we decided that client should be able to connect to the server dynamically, so that the server need not know about the client in advance.

There were two possible solutions considered. Both involved identifying the client by a unique identifier, the hardware Ethernet address of a physical Ethernet adapter in the client machine.

The first possible solution was to change the format of the POST request so that the unique client key is given in *every* POST request. This way, the server would not need to maintain any connection state, and could just process each request separately.

However, we decided that the implementation of this protocol would be difficult, and the overhead involved in retrieving the client information for each and every request would be quite high, and would worsen the efficiency of the implementation. Implementing this efficiently is possible, however, and it is our goal to design the next version of the HTun protocol this way.

The second possible solution was to have the client supply its unique key to the server only upon connecting. The server would then associate the particular connected socket with the client connected to it. Because the HTun protocol uses HTTP persistent connections, this solution means that the client data will only have to be retrieved once per connection.

The only problem with this solution arises from the fact that when multiple clients connect to a proxy server and generate requests to the same server, most proxy servers interleave the two clients' requests into one server connection, thereby rendering the connection unusable by either client, or possibly allowing a malicious user with access to the same proxy as another user to hijack the other user's connection. However, most proxy servers allow access to only the local network, on which users should trust each other not to hijack each other's connections. Future versions of the protocol will fix this problem, though, as it is clearly a serious one.

C. Dynamic Client Support

The dynamic client support was an important feature of the server. The server reads its list of valid `iprange`s from a config file, and compares the list to the one sent by the client. The server simply manipulates the addresses and netmasks to quickly determine what addresses both sides have in common, and assigns the first ones that it has not already assigned.

D. Statefulness vs. Statelessness

A stateless protocol would theoretically be optimal, as it would allow many clients to connect to the same HTun server using the same proxy server. This would require the client to send its unique ID with each request, though, and would therefore require significant processing overhead for the server to match each incoming request with the appropriate data queues. This would also defeat the purpose of opening two channels with Protocol 2, because the proxy would interleave both channels into one open socket to the HTun server, thereby effectively turning it back into it a half duplex channel.

E. Support for Both Protocols

Since protocols 1 and 2 were were very similar, the HTun server and client support them both. The protocol the client will use is set in its configuration file. The client can also switch between protocol 1 and 2 while the VPN is up. This is possible due to the fact that the server maintains the crucial client data across disconnects; therefore, a client could simply disconnect and reconnect using a different protocol.

F. Coexisting With a Web Server

If the administrator of the HTun server machine wishes to run a webserver on port 80 as well, it is completely possible with HTun. Any requests it does not understand as HTun client requests are redirected to the webserver and port of choice (set in the configuration file), thereby allowing HTun to coexist with a real webserver.

G. Other Implementation Details Concerns

The HTun client can be configured to rewrite the system's routing table so that all non-local traffic will be routed through the `tun` device, so that the HTun server may receive the packets and possibly perform IP masquerading on it[1]. This enables the client to continue to talk to the other hosts on its local LAN using local routes, while all the external traffic is masqueraded.

A routing problem comes up, however, when the proxy server is not on the client's own subnet – that is, the client must use a router to get to the proxy server. In this case, the client user must set up a host route for the proxy server specifically. This will allow the client to communicate through the real gateway when attempting to talk to the proxy server, but through HTun when attempting to communicate elsewhere.

Currently the HTun client daemon can be configured to rewrite (and restore) the default route, whereas the configuration of a static route to the proxy server is left as the user's responsibility.

IX. Open Issues

.1 TCP-over-TCP Issues

Because HTun tunnels IP over HTTP, and HTTP is a TCP-based protocol, HTun effectively causes TCP to be tunneled over TCP whenever the user uses a TCP-based protocol over HTun.

[1]IP masquerading of routed data is not the job of HTun, but is easily accomplished using regular iptables masquerading rules in Linux

The TCP over TCP problem manifests itself when the lower IP layer experiences packet loss or delay. The lower TCP layer increases its retransmit timers, leaving the upper layer to retransmit faster than the lower layer is transmitting data. This upper layer retransmission is done despite the fact that TCP is operating over a reliable lower layer connection. This is known as the *meltdown effect*, and is described more fully by Olaf Titz [**?**].

There is no apparent solution to this problem other than changing the TCP timers for the upper layer without changing them for the lower layer, which is not possible due to the fact that both layers use the same TCP implementation.

A. Future design considerations

Recent research has identified a possible performance enhancement to the HTun protocol. Most proxy servers now support the CONNECT mechanism as a method to allow SSL communication between the client and the server. Simply put, the CONNECT method opens a fully bidirectional TCP channel between the client and the server, and allows arbitrary data to be sent back and forth.

For networks where the CONNECT method is supported, HTun could elect to use this channel with potentially significant improved performance, as well as data encryption features. On the contrary, when the CONNECT method is not supported by a proxy, HTun provides a mechanism for allowing network hosts to use SSL based applications in the Internet.

X. Conclusions

The ability to achieve complete IP connectivity in an efficient manner is critical to the success of many emerging network applications. This work presents a means to achieving this level of connectivity even when the available service is more restrictive. The HTun protocol successfully demonstrates the feasibility of this approach using HTTP proxies. The protocol and implementation provide a reasonably efficient and practical solution that works within the limitations of current protocols and proxy servers. In the future, these techniques could be used to design general network layer services over similar request-oriented applications.