

1 2015/08/24  
 2  
 3  
 4  
 5  
 6 Abstract  
 7  
 8  
 9  
 10  
 11  
 12  
 13  
 14  
 15  
 16  
 17  
 18  
 19  
 20  
 21  
 22  
 23  
 24  
 25  
 26  
 27  
 28  
 29  
 30  
 31  
 32  
 33  
 34  
 35  
 36  
 37  
 38  
 39  
 40  
 41  
 42  
 43  
 44  
 45  
 46  
 47  
 48  
 49  
 50  
 51  
 52  
 53  
 54  
 55  
 56  
 57  
 58  
 59  
 60  
 61  
 62  
 63  
 64  
 65

## The PROXY protocol Versions 1 & 2

Willy Tarreau  
 HAPROXY Technologies

The PROXY protocol provides a convenient way to safely transport connection information such as a client's address across multiple layers of NAT or TCP proxies. It is designed to require little changes to existing components and to limit the performance impact caused by the processing of the transported information.

### Revision history

2010/10/29 - first version  
 2011/03/20 - update: implementation and security considerations  
 2012/06/21 - add support for binary format  
 2012/11/19 - final review and fixes  
 2014/05/18 - modify and extend PROXY protocol version 2  
 2014/06/11 - fix example code to consider ver+cmd merge  
 2014/06/14 - fix v2 header check in example code, and update Forwarded spec  
 2014/07/12 - update list of implementations (add Squid)  
 2015/05/02 - update list of implementations and format of the TLV add-ons

### 1. Background

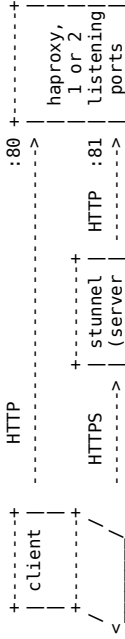
Relaying TCP connections through proxies generally involves a loss of the original TCP connection parameters such as source and destination addresses, ports, and so on. Some protocols make it a little bit easier to transfer such information. For SMTP, Postfix authors have proposed the XCLIENT protocol [1] which received broad adoption and is particularly suited to mail exchanges. For HTTP, there is the "Forwarded" extension [2], which aims at replacing the omnipresent "X-Forwarded-For" header which carries information about the original source address, and the less common X-Original-To which carries information about the destination address.

However, both mechanisms require a knowledge of the underlying protocol to be implemented in intermediaries.

Then comes a new class of products which we'll call "dumb proxies", not because they don't do anything, but because they're processing protocol-agnostic data. Both Stunnel[3] and Stud[4] are examples of such "dumb proxies". They talk raw TCP on one side, and raw SSL on the other one, and do that reliably, without any knowledge of what protocol is transported on top of the connection. Haproxy running in pure TCP mode obviously falls into that category as well.

The problem with such a proxy when it is combined with another one such as haproxy, is to adapt it to talk the higher level protocol. A patch is available for Stunnel to make it capable of inserting an X-Forwarded-For header in the first HTTP request of each incoming connection. Haproxy is able not to add another one when the connection comes from Stunnel, so that it's possible to hide it from the servers.

The typical architecture becomes the following one :



66  
 67  
 68  
 69  
 70  
 71  
 72  
 73  
 74  
 75  
 76  
 77  
 78  
 79  
 80  
 81  
 82  
 83  
 84  
 85  
 86  
 87  
 88  
 89  
 90  
 91  
 92  
 93  
 94  
 95  
 96  
 97  
 98  
 99  
 100  
 101  
 102  
 103  
 104  
 105  
 106  
 107  
 108  
 109  
 110  
 111  
 112  
 113  
 114  
 115  
 116  
 117  
 118  
 119  
 120  
 121  
 122  
 123  
 124  
 125  
 126  
 127  
 128  
 129  
 130

The problem appears when haproxy runs with keep-alive on the side towards the client. The Stunnel patch will only add the X-Forwarded-For header to the first request of each connection and all subsequent requests will not have it. One solution could be to improve the patch to make it support keep-alive and parse all forwarded data, whether they're announced with a Content-Length or with a Transfer-Encoding, taking care of special methods such as HEAD which announce data without transferring them, etc... In fact, it would require implementing a full HTTP stack in Stunnel. It would then become a lot more complex, a lot less reliable and would not anymore be the "dumb proxy" that fits every purposes.

In practice, we don't need to add a header for each request because we'll emit the exact same information every time : the information related to the client side connection. We could then cache that information in haproxy and use it for every other request. But that becomes dangerous and is still limited to HTTP only.

Another approach consists in prepending each connection with a header reporting the characteristics of the other side's connection. This method is simpler to implement, does not require any protocol-specific knowledge on either side, and completely fits the purpose since what is desired precisely is to know the other side's connection endpoints. It is easy to perform for the sender (just send a short header once the connection is established) and to parse for the receiver (simply perform one read() on the incoming connection to fill in addresses after an accept). The protocol used to carry connection information across proxies was thus called the PROXY protocol.

### 2. The PROXY protocol header

This document uses a few terms that are worth explaining here :

- "connection initiator" is the party requesting a new connection
- "connection target" is the party accepting a connection request
- "client" is the party for which a connection was requested
- "server" is the party to which the client desired to connect
- "proxy" is the party intercepting and relaying the connection from the client to the server.
- "sender" is the party sending data over a connection.
- "receiver" is the party receiving data from the sender.
- "header" or "PROXY protocol header" is the block of connection information the connection initiator prepends at the beginning of a connection, which makes it the sender from the protocol point of view.

The PROXY protocol's goal is to fill the server's internal structures with the information collected by the proxy that the server would have been able to get by itself if the client was connecting directly to the server instead of via a proxy. The information carried by the protocol are the ones the server would get using getsockname() and getpeername() :

- address family (AF\_INET for IPv4, AF\_INET6 for IPv6, AF\_UNIX)
- socket protocol (SOCK\_STREAM for TCP, SOCK\_DGRAM for UDP)
- layer 3 source and destination addresses
- layer 4 source and destination ports if any

Unlike the XCLIENT protocol, the PROXY protocol was designed with limited extensibility in order to help the receiver parse it very fast. Version 1 was focused on keeping it human-readable for better debugging possibilities, which is always desirable for early adoption when few implementations exist. Version 2 adds support for a binary encoding of the header which is much more efficient to produce and to parse, especially when dealing with IPv6 addresses that are expensive to emit in ASCII form and to parse.

In both cases, the protocol simply consists in an easily parsable header placed

131 by the connection initiator at the beginning of each connection. The protocol  
 132 is intentionally stateless in that it does not expect the sender to wait for  
 133 the receiver before sending the header, nor the receiver to send anything back.  
 134

135 This specification supports two header formats, a human-readable format which  
 136 is the only format supported in version 1 of the protocol, and a binary format  
 137 which is only supported in version 2. Both formats were designed to ensure that  
 138 the header cannot be confused with common higher level protocols such as HTTP,  
 139 SSL/TLS, FTP or SMTP, and that both formats are easily distinguishable one from  
 140 each other for the receiver.  
 141

142 Version 1 senders MAY only produce the human-readable header format. Version 2  
 143 senders MAY only produce the binary header format. Version 1 receivers MUST at  
 144 least implement the human-readable header format. Version 2 receivers MUST at  
 145 least implement the binary header format, and it is recommended that they also  
 146 implement the human-readable header format for better interoperability and ease  
 147 of upgrade when facing version 1 senders.  
 148

149 Both formats are designed to fit in the smallest TCP segment that any TCP/IP  
 150 host is required to support (576 - 40 = 536 bytes). This ensures that the whole  
 151 header will always be delivered at once when the socket buffers are still empty  
 152 at the beginning of a connection. The sender must always ensure that the header  
 153 is sent at once, so that the transport layer maintains atomicity along the path  
 154 to the receiver. The receiver may be tolerant to partial headers or may simply  
 155 drop the connection when receiving a partial header. Recommendation is to be  
 156 tolerant, but implementation constraints may not always easily permit this. It  
 157 is important to note that nothing forces any intermediary to forward the whole  
 158 header at once, because TCP is a streaming protocol which may be processed one  
 159 byte at a time if desired, causing the header to be fragmented when reaching  
 160 the receiver. But due to the places where such a protocol is used, the above  
 161 simplification generally is acceptable because the risk of crossing such a  
 162 device handling one byte at a time is close to zero.  
 163

164 The receiver MUST NOT start processing the connection before it receives a  
 165 complete and valid PROXY protocol header. This is particularly important for  
 166 protocols where the receiver is expected to speak first (eg: SMTP, FTP or SSH).  
 167 The receiver may apply a short timeout and decide to abort the connection if  
 168 the protocol header is not seen within a few seconds (at least 3 seconds to  
 169 cover a TCP retransmit).  
 170

171 The receiver MUST be configured to only receive the protocol described in this  
 172 specification and MUST not try to guess whether the protocol header is present  
 173 or not. This means that the protocol explicitly prevents port sharing between  
 174 public and private access. Otherwise it would open a major security breach by  
 175 allowing untrusted parties to spoof their connection addresses. The receiver  
 176 SHOULD ensure proper access filtering so that only trusted proxies are allowed  
 177 to use this protocol.  
 178

179 Some proxies are smart enough to understand transported protocols and to reuse  
 180 idle server connections for multiple messages. This typically happens in HTTP  
 181 where requests from multiple clients may be sent over the same connection. Such  
 182 proxies MUST NOT implement this protocol on multiplexed connections because the  
 183 receiver would use the address advertised in the PROXY header as the address of  
 184 all forwarded requests's senders. In fact, such proxies are not dumb proxies,  
 185 and since they do have a complete understanding of the transported protocol,  
 186 they MUST use the facilities provided by this protocol to present the client's  
 187 address.  
 188

## 2.1. Human-readable header format (Version 1)

192 This is the format specified in version 1 of the protocol. It consists in one  
 193 line of ASCII text matching exactly the following block, sent immediately and  
 194 at once upon the connection establishment and prepended before any data flowing  
 195 from the sender to the receiver :

- a string identifying the protocol : "PROXY" ( \x50 \x52 \x4F \x58 \x59 )  
 Seeing this string indicates that this is version 1 of the protocol.
- exactly one space : " " ( \x20 )
- a string indicating the proxied INET protocol and family. As of version 1,  
 only "TCP4" ( \x54 \x43 \x50 \x34 ) for TCP over IPv4, and "TCP6"  
 ( \x54 \x43 \x50 \x36 ) for TCP over IPv6 are allowed. Other, unsupported,  
 or unknown protocols must be reported with the name "UNKNOWN" ( \x55 \x4E  
 \x4B \x4E \x4F \x57 \x4E ). For "UNKNOWN", the rest of the line before the  
 CRLF may be omitted by the sender, and the receiver must ignore anything  
 presented before the CRLF is found. Note that an earlier version of this  
 specification suggested to use this when sending health checks, but this  
 causes issues with servers that reject the "UNKNOWN" keyword. Thus is it  
 now recommended not to send "UNKNOWN" when the connection is expected to  
 be accepted, but only when it is not possible to correctly fill the PROXY  
 line.
- exactly one space : " " ( \x20 )
- the layer 3 source address in its canonical format. IPv4 addresses must be  
 indicated as a series of exactly 4 integers in the range [0..255] inclusive  
 written in decimal representation separated by exactly one dot between each  
 other. Heading zeroes are not permitted in front of numbers in order to  
 avoid any possible confusion with octal numbers. IPv6 addresses must be  
 indicated as series of 4 hexadecimal digits (upper or lower case) delimited  
 by colons between each other, with the acceptance of one double colon  
 sequence to replace the largest acceptable range of consecutive zeroes. The  
 total number of decoded bits must exactly be 128. The advertised protocol  
 family dictates what format to use.
- exactly one space : " " ( \x20 )
- the layer 3 destination address in its canonical format. It is the same  
 format as the layer 3 source address and matches the same family.
- exactly one space : " " ( \x20 )
- the TCP source port represented as a decimal integer in the range  
 [0..65535] inclusive. Heading zeroes are not permitted in front of numbers  
 in order to avoid any possible confusion with octal numbers.
- exactly one space : " " ( \x20 )
- the TCP destination port represented as a decimal integer in the range  
 [0..65535] inclusive. Heading zeroes are not permitted in front of numbers  
 in order to avoid any possible confusion with octal numbers.
- the CRLF sequence ( \x0D \x0A )

The maximum line lengths the receiver must support including the CRLF are :

- TCP/IPv4 :  
 "PROXY TCP4 255.255.255.255 255.255.255.255 65535 65535\r\n"  
 => 5 + 1 + 4 + 1 + 15 + 1 + 15 + 1 + 5 + 1 + 5 + 2 = 56 chars
- TCP/IPv6 :  
 "PROXY TCP6 ffff:f...f:ffff ffff:f...f:ffff 65535 65535\r\n"  
 => 5 + 1 + 4 + 1 + 39 + 1 + 39 + 1 + 5 + 1 + 5 + 2 = 104 chars
- unknown connection (short form) :  
 "PROXY UNKNOWN\r\n"  
 => 5 + 1 + 7 + 2 = 15 chars

261 - worst case (optional fields set to 0xff) :  
 262 "PROXY UNKNOWN ffff:f...f:ffff ffff:f...f:ffff 65535 65535\r\n"  
 263 => 5 + 1 + 7 + 1 + 39 + 1 + 39 + 1 + 5 + 1 + 5 + 2 = 107 chars  
 264

265 So a 108-byte buffer is always enough to store all the line and a trailing zero  
 266 for string processing.

267 The receiver must wait for the CRLF sequence before starting to decode the  
 268 addresses in order to ensure they are complete and properly parsed. If the CRLF  
 269 sequence is not found in the first 107 characters, the receiver should declare  
 270 the line invalid. A receiver may reject an incomplete line which does not  
 271 contain the CRLF sequence in the first atomic read operation. The receiver must  
 272 not tolerate a single CR or LF character to end the line when a complete CRLF  
 273 sequence is expected.

274 Any sequence which does not exactly match the protocol must be discarded and  
 275 cause the receiver to abort the connection. It is recommended to abort the  
 276 connection as soon as possible so that the sender gets a chance to notice the  
 277 anomaly and log it.

280 If the announced transport protocol is "UNKNOWN", then the receiver knows that  
 281 the sender speaks the correct PROXY protocol with the appropriate version, and  
 282 SHOULD accept the connection and use the real connection's parameters as if  
 283 there were no PROXY protocol header on the wire. However, senders SHOULD not  
 284 use the "UNKNOWN" protocol when they are the initiators of outgoing connections  
 285 because some receivers may reject them. When a load balancing proxy has to send  
 286 health checks to a server, it SHOULD build a valid PROXY line which it will  
 287 fill with a getsockname()/getpeername() pair indicating the addresses used. It  
 288 is important to understand that doing so is not appropriate when some source  
 289 address translation is performed between the sender and the receiver.

292 An example of such a line before an HTTP request would look like this (CR  
 293 marked as "\r" and LF marked as "\n") :

```
294 PROXY TCP4 192.168.0.1 192.168.0.11 56324 443\r\n
295 GET / HTTP/1.1\r\n
296 Host: 192.168.0.11\r\n
297 \r\n
```

299 For the sender, the header line is easy to put into the output buffers once the  
 300 connection is established. Note that since the line is always shorter than an  
 301 MSS, the sender is guaranteed to always be able to emit it at once and should  
 302 not even bother handling partial sends. For the receiver, once the header is  
 303 parsed, it is easy to skip it from the input buffers. Please consult section 9  
 304 for implementation suggestions.

## 2.2. Binary header format (version 2)

309 Producing human-readable IPv6 addresses and parsing them is very inefficient,  
 310 due to the multiple possible representation formats and the handling of compact  
 311 address format. It was also not possible to specify address families outside  
 312 IPv4/IPv6 nor non-TCP protocols. Another drawback of the human-readable format  
 313 is the fact that implementations need to parse all characters to find the  
 314 trailing CRLF, which makes it harder to read only the exact bytes count. Last,  
 315 the UNKNOWN address type has not always been accepted by servers as a valid  
 316 protocol because of its imprecise meaning.

319 Version 2 of the protocol thus introduces a new binary format which remains  
 320 distinguishable from version 1 and from other commonly used protocols. It was  
 321 specially designed in order to be incompatible with a wide range of protocols  
 322 and to be rejected by a number of common implementations of these protocols  
 323 when unexpectedly presented (please see section 7). Also for better processing  
 324 efficiency, IPv4 and IPv6 addresses are respectively aligned on 4 and 16 bytes  
 325 boundaries.

326 The binary header format starts with a constant 12 bytes block containing the  
 327 protocol signature :

```
328 \x0D \x0A \x0D \x0A \x00 \x0D \x0A \x51 \x55 \x49 \x54 \x0A
```

331 Note that this block contains a null byte at the 5th position, so it must not  
 332 be handled as a null-terminated string.

333 The next byte (the 13th one) is the protocol version and command.

335 The highest four bits contains the version. As of this specification, it must  
 336 always be sent as \x2 and the receiver must only accept this value.

339 The lowest four bits represents the command :

- \x0 : LOCAL : the connection was established on purpose by the proxy without being relayed. The connection endpoints are the sender and the receiver. Such connections exist when the proxy sends health-checks to the server. The receiver must accept this connection as valid and must use the real connection endpoints and discard the protocol block including the family which is ignored.

- \x1 : PROXY : the connection was established on behalf of another node, and reflects the original connection endpoints. The receiver must then use the information provided in the protocol block to get original the address.
- other values are unassigned and must not be emitted by senders. Receivers must drop connections presenting unexpected values here.

The 14th byte contains the transport protocol and address family. The highest 4 bits contain the address family, the lowest 4 bits contain the protocol.

The address family maps to the original socket family without necessarily matching the values internally used by the system. It may be one of :

- 0x0 : AF\_UNSPEC : the connection is forwarded for an unknown, unspecified or unsupported protocol. The sender should use this family when sending LOCAL commands or when dealing with unsupported protocol families. The receiver is free to accept the connection anyway and use the real endpoint addresses or to reject it. The receiver should ignore address information.
- 0x1 : AF\_INET : the forwarded connection uses the AF\_INET address family (IPv4). The addresses are exactly 4 bytes each in network byte order, followed by transport protocol information (typically ports).
- 0x2 : AF\_INET6 : the forwarded connection uses the AF\_INET6 address family (IPv6). The addresses are exactly 16 bytes each in network byte order, followed by transport protocol information (typically ports).
- 0x3 : AF\_UNIX : the forwarded connection uses the AF\_UNIX address family (UNIX). The addresses are exactly 108 bytes each.
- other values are unspecified and must not be emitted in version 2 of this protocol and must be rejected as invalid by receivers.

The transport protocol is specified in the lowest 4 bits of the 14th byte :

- 0x0 : UNSPEC : the connection is forwarded for an unknown, unspecified or unsupported protocol. The sender should use this family when sending LOCAL commands or when dealing with unsupported protocol families. The receiver is free to accept the connection anyway and use the real endpoint addresses or to reject it. The receiver should ignore address information.
- 0x1 : STREAM : the forwarded connection uses a SOCK\_STREAM protocol (eg: TCP or UNIX\_STREAM). When used with AF\_INET/AF\_INET6 (TCP), the addresses

391 are followed by the source and destination ports represented on 2 bytes  
 392 each in network byte order.  
 393  
 394 - 0x2 : DGRAM : the forwarded connection uses a SOCK\_DGRAM protocol (eg:  
 395 UDP or UNIX\_DGRAM). When used with AF\_INET/AF\_INET6 (UDP), the addresses  
 396 are followed by the source and destination ports represented on 2 bytes  
 397 each in network byte order.  
 398  
 399 - other values are unspecified and must not be emitted in version 2 of this  
 400 protocol and must be rejected as invalid by receivers.  
 401  
 402 In practice, the following protocol bytes are expected :  
 403  
 404 - \x00 : UNSPEC : the connection is forwarded for an unknown, unspecified  
 405 or unsupported protocol. The sender should use this family when sending  
 406 LOCAL commands or when dealing with unsupported protocol families. When  
 407 used with a LOCAL command, the receiver must accept the connection and  
 408 ignore any address information. For other commands, the receiver is free  
 409 to accept the connection anyway and use the real endpoints addresses or to  
 410 reject the connection. The receiver should ignore address information.  
 411  
 412 - \x11 : TCP over IPv4 : the forwarded connection uses TCP over the AF\_INET  
 413 protocol family. Address length is 2\*4 + 2\*2 = 12 bytes.  
 414  
 415 - \x12 : UDP over IPv4 : the forwarded connection uses UDP over the AF\_INET  
 416 protocol family. Address length is 2\*4 + 2\*2 = 12 bytes.  
 417  
 418 - \x21 : TCP over IPv6 : the forwarded connection uses TCP over the AF\_INET6  
 419 protocol family. Address length is 2\*16 + 2\*2 = 36 bytes.  
 420  
 421 - \x22 : UDP over IPv6 : the forwarded connection uses UDP over the AF\_INET6  
 422 protocol family. Address length is 2\*16 + 2\*2 = 36 bytes.  
 423  
 424 - \x31 : UNIX stream : the forwarded connection uses SOCK\_STREAM over the  
 425 AF\_UNIX protocol family. Address length is 2\*108 = 216 bytes.  
 426  
 427 - \x32 : UNIX datagram : the forwarded connection uses SOCK\_DGRAM over the  
 428 AF\_UNIX protocol family. Address length is 2\*108 = 216 bytes.  
 429

430 Only the UNSPEC protocol byte (\x00) is mandatory. A receiver is not required  
 431 to implement other ones, provided that it automatically falls back to the  
 432 UNSPEC mode for the valid combinations above that it does not support.  
 433

434 The 15th and 16th bytes is the address length in bytes in network endian order.  
 435 It is used so that the receiver knows how many address bytes to skip even when  
 436 it does not implement the presented protocol. Thus the length of the protocol  
 437 header in bytes is always exactly 16 + this value. When a sender presents a  
 438 LOCAL connection, it should not present any address so it sets this field to  
 439 zero. Receivers MUST always consider this field to skip the appropriate number  
 440 of bytes and must not assume zero is presented for LOCAL connections. When a  
 441 receiver accepts an incoming connection showing an UNSPEC address family or  
 442 protocol, it may or may not decide to log the address information if present.  
 443

444 So the 16-byte version 2 header can be described this way :

```

445 struct proxy_hdr_v2 {
446     uint8_t sig[12]; /* hex 0D 0A 0D 0A 00 0D 0A 51 55 49 54 0A */
447     uint8_t ver_cmd; /* protocol version and command */
448     uint8_t fam; /* protocol family and address */
449     uint16_t len; /* number of following bytes part of the header */
450 };
  
```

451 Starting from the 17th byte, addresses are presented in network byte order.  
 452 The address order is always the same :

- source layer 3 address in network byte order
- destination layer 3 address in network byte order
- source layer 4 address if any, in network byte order (port)
- destination layer 4 address if any, in network byte order (port)

The address block may directly be sent from or received into the following union which makes it easy to cast from/to the relevant socket native structs depending on the address type :

```

455 union proxy_addr {
456     struct { /* for TCP/UDP over IPv4, len = 12 */
457         uint32_t src_addr;
458         uint32_t dst_addr;
459         uint16_t src_port;
460         uint16_t dst_port;
461     } ipv4_addr;
462     struct { /* for TCP/UDP over IPv6, len = 36 */
463         uint8_t src_addr[16];
464         uint8_t dst_addr[16];
465         uint16_t src_port;
466         uint16_t dst_port;
467     } ipv6_addr;
468     struct { /* for AF_UNIX sockets, len = 216 */
469         uint8_t src_addr[108];
470         uint8_t dst_addr[108];
471         uint8_t src_addr;
472     } unix_addr;
473 };
  
```

The sender must ensure that all the protocol header is sent at once. This block is always smaller than an MSS, so there is no reason for it to be segmented at the beginning of the connection. The receiver should also process the header at once. The receiver must not start to parse an address before the whole address block is received. The receiver must also reject incoming connections containing partial protocol headers.

A receiver may be configured to support both version 1 and version 2 of the protocol. Identifying the protocol version is easy :

- if the incoming byte count is 16 or above and the 13 first bytes match the protocol signature block followed by the protocol version 2 :

```

    \x0D\x0A\x0D\x0A\x00\x0D\x0A\x51\x55\x49\x54\x0A\x02
  
```

- otherwise, if the incoming byte count is 8 or above, and the 5 first characters match the ASCII representation of "PROXY" then the protocol must be parsed as version 1 :

```

    \x50\x52\x4F\x58\x59
  
```

- otherwise the protocol is not covered by this specification and the connection must be dropped.

If the length specified in the PROXY protocol header indicates that additional bytes are part of the header beyond the address information, a receiver may choose to skip over and ignore those bytes, or attempt to interpret those bytes.

The information in those bytes will be arranged in Type-Length-Value (TLV vectors) in the following format. The first byte is the Type of the vector. The second two bytes represent the length in bytes of the value (not included the Type and Length bytes), and following the length field is the number of bytes specified by the length.

```

    struct pp2_tlv {
        uint8_t type;
  
```

```

521         uint8_t length_hi;
522         uint8_t length_lo;
523         uint8_t value[0];
524     };
525
526     The following types have already been registered for the <type> field :
527
528     #define PP2_TYPE_ALPN      0x01
529     #define PP2_TYPE_AUTHORITY 0x02
530     #define PP2_TYPE_SSL       0x20
531     #define PP2_SUBTYPE_SSL_VERSION 0x21
532     #define PP2_SUBTYPE_SSL_CN  0x22
533     #define PP2_TYPE_NETNS      0x30
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555

```

The following types have already been registered for the <type> field :

```

527
528     #define PP2_TYPE_ALPN      0x01
529     #define PP2_TYPE_AUTHORITY 0x02
530     #define PP2_TYPE_SSL       0x20
531     #define PP2_SUBTYPE_SSL_VERSION 0x21
532     #define PP2_SUBTYPE_SSL_CN  0x22
533     #define PP2_TYPE_NETNS      0x30
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555

```

#### 2.2.1. The PP2\_TYPE\_SSL type and subtypes

For the type PP2\_TYPE\_SSL, the value is itself a defined like this :

```

540     struct pp2_tlv_ssl {
541         uint8_t client;
542         uint32_t verify;
543         struct pp2_tlv sub_tlv[0];
544     };
545
546
547
548
549
550
551
552
553
554
555

```

The <verify> field will be zero if the client presented a certificate and it was successfully verified, and non-zero otherwise.

The <client> field is made of a bit field from the following values, indicating which element is present :

```

552     #define PP2_CLIENT_SSL      0x01
553     #define PP2_CLIENT_CERT_CONN 0x02
554     #define PP2_CLIENT_CERT_SESS 0x04
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585

```

Note, that each of these elements may lead to extra data being appended to this TLV using a second level of TLV encapsulation. It is thus possible to find multiple TLV values after this field. The total length of the pp2\_tlv\_ssl TLV will reflect this.

The PP2\_CLIENT\_SSL flag indicates that the client connected over SSL/TLS. When this field is present, the string representation of the TLS version is appended at the end of the field in the TLV format using the type PP2\_SUBTYPE\_SSL\_VERSION.

PP2\_CLIENT\_CERT\_CONN indicates that the client provided a certificate over the current connection. PP2\_CLIENT\_CERT\_SESS indicates that the client provided a certificate at least once over the TLS session this connection belongs to.

In all cases, the string representation (in UTF8) of the Common Name field (OID: 2.5.4.3) of the client certificate's DistinguishedName, is appended using the TLV format and the type PP2\_SUBTYPE\_SSL\_CN.

#### 2.2.2. The PP2\_TYPE\_NETNS type

The type PP2\_TYPE\_NETNS defines the value as the string representation of the namespace's name.

### 3. Implementations

Haproxy 1.5 implements version 1 of the PROXY protocol on both sides :  
 - the listening sockets accept the protocol when the "accept-proxy" setting is passed to the "bind" keyword. Connections accepted on such listeners will behave just as if the source really was the one advertised in the

```

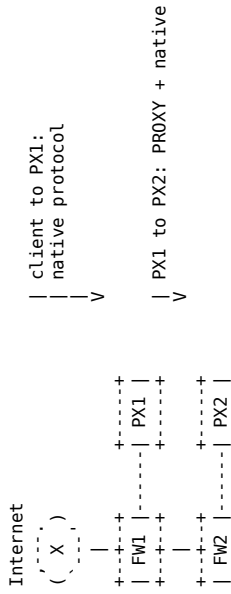
586     protocol. This is true for logging, ACLs, content filtering, transparent
587     proxying, etc...
588
589     - the protocol may be used to connect to servers if the "send-proxy" setting
590     is present on the "server" line. It is enabled on a per-server basis, so it
591     is possible to have it enabled for remote servers only and still have local
592     ones behave differently. If the incoming connection was accepted with the
593     "accept-proxy", then the relayed information is the one advertised in this
594     connection's PROXY line.
595
596     - Haproxy 1.5 also implements version 2 of the PROXY protocol as a sender. In
597     addition, a TLV with limited, optional, SSL information has been added.
598
599     Stunnel added support for version 1 of the protocol for outgoing connections in
600     version 4.45.
601
602     Stud added support for version 1 of the protocol for outgoing connections on
603     2011/06/29.
604
605     Postfix added support for version 1 of the protocol for incoming connections
606     in smtpd and postscreen in version 2.10.
607
608     A patch is available for Stud[5] to implement version 1 of the protocol on
609     incoming connections.
610
611     Support for versions 1 and 2 of the protocol was added to Varnish 4.1 [6].
612
613     Exim added support for version 1 and version 2 of the protocol for incoming
614     connections on 2014/05/13, and will be released as part of version 4.83.
615
616     Squid added support for versions 1 and 2 of the protocol in version 3.5 [7].
617
618     Jetty 9.3.0 supports protocol version 1.
619
620     The protocol is simple enough that it is expected that other implementations
621     will appear, especially in environments such as SMTP, IMAP, FTP, RDP where the
622     client's address is an important piece of information for the server and some
623     intermediaries. In fact, several proprietary deployments have already done so
624     on FTP and SMTP servers.
625
626     Proxy developers are encouraged to implement this protocol, because it will
627     make their products much more transparent in complex infrastructures, and will
628     get rid of a number of issues related to logging and access control.
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650

```

#### 4. Architectural benefits

##### 4.1. Multiple layers

Using the PROXY protocol instead of transparent proxy provides several benefits in multiple-layer infrastructures. The first immediate benefit is that it becomes possible to chain multiple layers of proxies and always present the original IP address. for instance, let's consider the following 2-layer proxy architecture :





## 7. Future developments

It is possible that the protocol may slightly evolve to present other information such as the incoming network interface, or the origin addresses in case of network address translation happening before the first proxy, but this is not identified as a requirement right now. Some deep thinking has been spent on this and it appears that trying to add a few more information open a Pandora box with many information from MAC addresses to SSL client certificates, which would make the protocol much more complex. So at this point it is not planned. Suggestions on improvements are welcome.

## 8. Contacts and links

Please use w@lwt.eu to send any comments to the author.

The following links were referenced in the document.

- [1] [http://www.postfix.org/XCLIENT\\_README.html](http://www.postfix.org/XCLIENT_README.html)
- [2] <http://tools.ietf.org/html/rfc7239>
- [3] <http://www.stunnel.org/>
- [4] <https://github.com/bumptech/stud>
- [5] <https://github.com/bumptech/stud/pull/81>
- [6] [https://www.varnish-cache.org/docs/trunk/phk/ssl\\_again.html](https://www.varnish-cache.org/docs/trunk/phk/ssl_again.html)
- [7] <http://wiki.squid-cache.org/Squid-3.5>

## 9. Sample code

The code below is an example of how a receiver may deal with both versions of the protocol header for TCP over IPv4 or IPv6. The function is supposed to be called upon a read event. Addresses may be directly copied into their final memory location since they're transported in network byte order. The sending side is even simpler and can easily be deduced from this sample code.

```
struct sockaddr_storage from; /* already filled by accept() */
struct sockaddr_storage to; /* already filled by getsockname() */
const char v2sig[12] = "\x00\x0A\x0D\x0A\x00\x0D\x0A\x51\x55\x49\x54\x0A";

/* returns 0 if needs to poll, <0 upon error or >0 if it did the job */
int read_evt(int fd)
{
    union {
        struct {
            char line[108];
        } v1;
        struct {
            uint8_t sig[12];
            uint8_t ver_cmd;
            uint8_t fam;
            uint16_t len;
        } v2;
    };

    struct { /* for TCP/UDP over IPv4, len = 12 */
        uint32_t src_addr;
        uint32_t dst_addr;
        uint16_t src_port;
        uint16_t dst_port;
    } ip4;

    struct { /* for TCP/UDP over IPv6, len = 36 */
        uint8_t src_addr[16];
        uint8_t dst_addr[16];
        uint16_t src_port;
        uint16_t dst_port;
    } ip6;

    struct { /* for AF_UNIX sockets, len = 216 */
```

```
        uint8_t src_addr[108];
        uint8_t dst_addr[108];
    } unx;
    } addr;
    } v2;
    } hdr;

    int size, ret;

    do {
        ret = recv(fd, &hdr, sizeof(hdr), MSG_PEEK);
    } while (ret == -1 && errno == EINTR);

    if (ret == -1)
        return (errno == EAGAIN) ? 0 : -1;

    if (ret >= 16 && memcmp(&hdr.v2, v2sig, 12) == 0 &&
        (hdr.v2.ver_cmd & 0xF0) == 0x20) {
        size = 16 + hdr.v2.len;
        if (ret < size)
            return -1; /* truncated or too large header */

        switch (hdr.v2.ver_cmd & 0xF) {
        case 0x01: /* PROXY command */
            switch (hdr.v2.fam) {
            case 0x11: /* TCPv4 */
                ((struct sockaddr_in *)&from)->sin_family = AF_INET;
                ((struct sockaddr_in *)&from)->sin_addr.s_addr =
                    hdr.v2.addr.ip4.src_addr;
                ((struct sockaddr_in *)&from)->sin_port =
                    hdr.v2.addr.ip4.src_port;
                ((struct sockaddr_in *)&to)->sin_family = AF_INET;
                ((struct sockaddr_in *)&to)->sin_addr.s_addr =
                    hdr.v2.addr.ip4.dst_addr;
                ((struct sockaddr_in *)&to)->sin_port =
                    hdr.v2.addr.ip4.dst_port;
                goto done;
            case 0x21: /* TCPv6 */
                ((struct sockaddr_in6 *)&from)->sin6_family = AF_INET6;
                memcpy(&((struct sockaddr_in6 *)&from)->sin6_addr,
                    hdr.v2.addr.ip6.src_addr, 16);
                ((struct sockaddr_in6 *)&from)->sin6_port =
                    hdr.v2.addr.ip6.src_port;
                ((struct sockaddr_in6 *)&to)->sin6_family = AF_INET6;
                memcpy(&((struct sockaddr_in6 *)&to)->sin6_addr,
                    hdr.v2.addr.ip6.dst_addr, 16);
                ((struct sockaddr_in6 *)&to)->sin6_port =
                    hdr.v2.addr.ip6.dst_port;
                goto done;
            }
            /* unsupported protocol, keep local connection address */
            break;
        case 0x00: /* LOCAL command */
            /* keep local connection address for LOCAL */
            break;
        default:
            return -1; /* not a supported command */
        }
    } else if (ret >= 8 && memcmp(hdr.v1.line, "PROXY", 5) == 0) {
        char *end = strchr(hdr.v1.line, '\r', ret - 1);
        if (end || end[1] != '\n')
            return -1; /* partial or invalid header */
        *end = '\0'; /* terminate the string to ease parsing */
        size = end + 2 - hdr.v1.line; /* skip header + CRLF */
    }
```

```
911      /* parse the V1 header using favorite address parsers like inet_pton.
912      * return -1 upon error, or simply fall through to accept.
913      */
914    }
915    else {
916      /* Wrong protocol */
917      return -1;
918    }
919  }
920  done:
921    /* we need to consume the appropriate amount of data from the socket */
922    do {
923      ret = recv(fd, &hdr, size, 0);
924    } while (ret == -1 && errno == EINTR);
925    return (ret >= 0) ? 1 : -1;
926  }
```