2ping protocol

Line protocol version: 2.0
Document version: 20120422

Introduction

The 2ping protocol is a bi-directional ping system. This allows for several features, such as 3-way ping (akin to SYN, SYN/ACK, ACK), and the ability to determine whether packet loss was inbound or outbound. This is accomplished by the client and server each keeping state tables of needed incoming and outgoing requests. If packet loss occurs without a complete network failure, the client and server can eventually resume communication and compare notes about which sides received what requests.

Network transit

2ping operates over UDP, and is compatible with IPv4, IPv6 and possibly future Layer 3 protocols. The server/listener listens on port 15998. The client can use any high source port, and will send packets to the server on destination port 15998. The server will then respond to the client on the port the client originally used as a source port. The UDP message payload is currently completely binary data, and is designed with consistent and/or minimum space, consistent parsing, and forwards compatibility in mind. The protocol payload does not contain any IP-specific information, and is NAT/PAT safe.

Message format

The 2ping format is a variable length binary format, with the length determined by which opcodes are set. Opcode data lengths are in a standardized location, which allows for forward compatibility. A program utilizing the 2ping protocol can still parse a complete payload, even if the program does not understand a certain opcode.

All data is in network byte format: octets are most significant bit first, and multiple-octet sequences (flags and integers capable of being larger than 8 bits) are most significant octet first. An exception is the order of opcode data segments; data for opcodes is stored in sequence according to which flags are set, starting with the least significant bit.

Magic number 0x3250	2 octets, required
Checksum	2 octets, field required, checksum optional
Message ID	6 octets, required

Opcode flags	2 octets, required
Opcode data	Variable, zero or more octets, length depends on opcode flags
Padding	Variable, optional

Thus, the minimum valid length of a 2ping payload is 12 octets (2-octet magic number + 2-octet checksum field + 6-octet message ID + 2-octet opcode flags with no flags enabled).

2 octets of opcode flags allow for up to 16 flags. The final flag (0x8000) is a container for an extended segment format, and allows for a nearly unlimited number of available segments.

Padding may be added to a packet to bring it up to a desired minimum size. A 128 octet minimum packet size is recommended. Padding octets should be null character octets by default, but the contents of the padding is not important, as the packet parser should not use the contents of the padding for program operation.

Zero or more opcode data segments are appended to form the opcode data area. Each opcode data segment is comprised of the following:

Segment data length (not including length header itself)	2 octets, required
Segment data	Variable, zero or more octets, determined by segment data length header above

An opcode data segment may be between zero and 65,535 octets long, plus 2 octets for the segment header.

This layout is designed for forward compatibility with future protocol revisions. A 2ping implementation can simply stop processing once it reaches the last opcode flag it understands, or it can at least parse the opcodes it does not understand, as the opcode data segment header includes the length of the segment data -- using this information, the implementation can simply skip over the opcodes it does not understand.

If there is a numeric gap in the understood opcodes of a 2ping implementation, it must at least check for the opcode and parse the opcode data segment in order to skip over it. For example, a 2ping implementation might understand 0x0001, 0x0002 and 0x0008, but not 0x0004, 0x0010 or 0x0020. To be able to parse the data segment for 0x0008, it must still check whether 0x0004 is set and seek to its segment data length header, determine the length of the segment data for 0x0004, and skip over it to begin parsing 0x0008. Due to the standardized length headers, this is possible without knowing what 0x0004 does. Once it is finished parsing 0x0008, it may stop checking opcode flags altogether.

The minimum 2ping implementation must be able to understand and follow opcodes 0x0001 (reply requested) and 0x0002 (in reply to). Implementation of all other opcodes are optional (but highly recommended, as investigation opcodes are the core focus of 2ping).

Message IDs

Message IDs are 48 bits (6 octets) of data to identify a specific message. The message ID must be 48 bits of pseudo-random data; do not use incrementing numbers or a discernible pattern. Likewise, do not attempt to analyze or devise logic from the message IDs that a peer sends.

Message IDs are unique within a socket's 5-tuple (local address, local port, peer address, peer port, protocol). For example, when implementing a listener, care should be taken to differentiate between message ID 0xF1E2D3C4B5A6 sent from peer 10.0.0.10, and message ID 0xF1E2D3C4B5A6 sent from peer 172.16.0.10, or even between the same message ID sent from the same host on different source ports. While the specification's random ID requirement and keyspace size makes the chance of a collision from two different peers unlikely, it is still possible.

It is understandable that a 48-bit random unique ID is not easy to read from an end user's perspective. The implementation may therefore wish to keep a local mapping between message IDs and a more human-friendly identifier, such as an incrementing integer.

Checksum

The checksum allows for verification of packet integrity from unintentional transit errors. (If cryptographic verification is desired to prevent intentional tampering in transit, see opcode 0x0080 below.) While UDP includes its own checksum, it is optional in IPv4 and cannot be relied upon. (IPv6 makes the UDP checksum mandatory.) The 2ping protocol adopts the UDP method of computing checksums, as defined in RFC 768. The only difference between the two is 2ping checksums only the entire packet data payload (from the magic number to the end of the padding), while UDP checksums a pseudo-IP header, UDP header, and UDP data payload.

> Checksum is the 16-bit one's complement of the one's complement sum of the UDP data payload, padded with zero octets at the end (if necessary) to make a multiple of two octets. The checksum pad is not transmitted as part of the segment. While computing the checksum, the checksum field itself is replaced with zeros.
>
> If the computed checksum is zero, it is transmitted as all ones (the equivalent in one's complement arithmetic). An all zero transmitted checksum value means that the transmitter generated no checksum.

The checksum field is required, though an implementation is not required to perform or verify checksums. However, if verification of checksums are performed, the verifier must be able to recognize the zero transmitted checksum value from a peer that did not compute a checksum.

Checksum computation and verification attempts are strongly recommend for IPv4 packets, but should not be necessary for IPv6 packets, as verification is already performed at the UDPv6 level, and malformed packets should be discarded by the time they reach the userland.

Opcodes

Opcode data areas are only included when the corresponding opcode flag is set. Thus if no opcode flags are set, the opcode data area does not exist. If 0x0001 and 0x0020 flags are set, the entire opcode data area consists of the opcode header and segment data for 0x0001, followed by the opcode header and segment data for 0x0020.

0x0001 - Reply requested

No segment data	0 octets

The sending end requests a reply from the receiving end. If the receiving end does not send a reply back to the sending end, the sending end will consider it a lost packet. Note: No segment data is part of this opcode, but as all used opcode segments must have a header, there will still be a 2-octet length header, signifying zero octets of data.

0x0002 - In reply to

Replied message ID	6 octets, required

This opcode signifies the reply to a packet that requested a reply. The original sender's message ID is enclosed.

0x0004 - RTT enclosed

RTT in microseconds	4 octets, required

If this packet is a reply packet and an RTT is enclosed, the packet being replied to was a successful ping, and the RTT is the round trip time of the previous operation, in microseconds. Up to 2^32-1 microseconds are possible, approximately 71 minutes. In a typical 3-way ping scenario, this is only sent in the third message in the sequence, from the client to the server. Thus the server will know both the server->client->server ping RTT, as well as the original client->server->client ping RTT.

0x0008 - Investigation complete, originally replied to as requested

Number of message IDs enclosed	2 octets, required
Message ID	6 octets, optional
Message ID...	6 octets..., optional

This is a response to "0x0020 - Expected reply never received, please investigate", please continue reading below. The message IDs listed in this opcode are packets that the responder knows about, and had responded to. Since the requester never received the response, the requester can assume that the packet loss occurred inbound (relative to the requester).

In theory, the number of message IDs enclosed may be between zero and 65,535, but limits on the total octet length of the opcode data area will make this impossible to achieve. In addition, UDP, IP and Ethernet length restrictions will further limit the number of message IDs that can be enclosed. The number of message IDs enclosed may legally be zero, but if that is the case, it's better to just not set the 0x0008 opcode flag.

Investigation replies must not be unsolicited, and may only be sent in response to messages that requested a reply, and requested an investigation into specific message IDs. However, investigation responses by the requestee are not required, and can be ignored if packet payload space is limited or other problems arise. It is the responsibility of the requester to resend investigation requests if the requestee does not respond to them in its reply.

0x0010 - Investigation complete, request never received

Number of message IDs enclosed	2 octets, required
Message ID	6 octets, optional
Message ID...	6 octets..., optional

This is a response to "0x0020 - Expected reply never received, please investigate", please continue reading below. The message IDs listed in this opcode are packets that the responder does not know about. Since the responder never received the request, the requester can assume that the packet loss occurred outbound (relative to the requester).

In theory, the number of message IDs enclosed may be between zero and 65,535, but limits on the total octet length of the opcode data area will make this impossible to achieve. In addition, UDP, IP and Ethernet length restrictions will further limit the number of message IDs that can be enclosed. The number of message IDs enclosed may legally be zero, but if that is the case, it's better to just not set the 0x0010 opcode flag.

Investigation replies must not be unsolicited, and may only be sent in response to messages that requested a reply, and requested an investigation into specific message IDs. However, investigation responses by the requestee are not required, and can be ignored if packet payload space is limited or other problems arise. It is the responsibility of the requester to resend investigation requests if the requestee does not respond to them in its reply.

0x0020 - Expected reply never received, please investigate

Number of message IDs enclosed	2 octets, required
Message ID	6 octets, required
Message ID...	6 octets..., optional

If the requester sends a packet to a remote party and indicates a reply is requested (0x0001), and a reply is never received, the requester can use this opcode to inquire about what happened to it. If replies to this inquiry come back in either "0x0008 - Investigation complete, originally replied to as requested" or "0x0010 - Investigation complete, request never received" opcodes (as explained above), the requester can use this information to determine whether the packet loss occurred inbound or outbound relative to the requester.

In theory, the number of message IDs enclosed may be between zero and 65,535, but limits on the total octet length of the opcode data area will make this impossible to achieve. In addition, UDP, IP and Ethernet length restrictions will further limit the number of message IDs that can be enclosed. The number of message IDs enclosed may legally be zero, but if that is the case, it's better to just not set the 0x0020 opcode flag.

Care should be taken not to begin inquiring too quickly. UDP packets may arrive delayed or out of order, so 10 seconds should be a good amount of time to wait before inquiring.

The requester may send inquiries multiple times. The requestee is not required to respond to a specific inquiry immediately, as UDP payload space may be limited. Thus it is the responsibility of the requester to continue to send inquiries until it receives a response, so the requestee does not need to keep a state table of unreplied inquiries.

0x0040 - Courtesy message ID expiration

Number of message IDs enclosed	2 octets, required
Message ID	6 octets, required

Message ID...	6 octets..., optional

To facilitate bi-directional packet loss detection, it is necessary for each peer to maintain a set of state tables: messages expecting a reply and not yet received, and remote peer messages expecting a reply that the near end has replied to. Maintenance of the first cache is easy; simply remove a message ID once a reply or investigation has been received. However, the second case is trickier. In some cases, it may be possible to know if the far end has received an expected response and will not inquire about it. But in others, there may not be enough hints to be able to guarantee a peer will not ask for an investigation later.

In particular, in a 3-way ping (assuming "Peer 1" initiated the ping), Peer 2 can figure out that the near end is satisfied with the response to the first leg, since Peer 1 sent the third leg to Peer 2, and so Peer 2 can remove the first leg's message ID from its cache. However, Peer 1 cannot tell if Peer 2 received the third leg (a response to the second leg), since a response to the third leg does not exist. But Peer 2 could in the future inquire about the response to the second leg sometime in the future.

In this situation, it would be best to do a periodic cleanup of the state tables, but ideally the age of the message ID should be long enough (recommended default is 10 minutes), since it may take the peer awhile to request an investigation. In a session with one or two pings per second that's not a lot, but in a flood mode, thousands or millions of message IDs could get stored in the state table during that 10 minute period.

To combat this, the courtesy opcode contains a list of message IDs that a peer is no longer concerned about, and can be removed from the remote peer's cache. Courtesy opcodes can be sent any time a packet is being sent from one peer to another (unlike investigation requests, which may only be sent when requesting a reply, or investigation results, which are in the immediate response to the investigation request).

This opcode is optional (but recommended), and due to its opportunistic nature, it is not guaranteed the message will reach the remote peer. So it is still up to an implementation to do periodic maintenance on its caches, since it cannot rely on courtesy responses to prune its caches.

Peers may send unsolicited messages with courtesy opcodes (a message not requesting a reply, not replying to another message, and containing the courtesy opcode), but in the normal course of operation, it should not be necessary.

0x0080 - Message authentication code (MAC)

Digest type index	2 octets, required
Computed hash value	Length depends on digest type, required

The message authentication code field is allowed to provide cryptographic data integrity and authentication of the remote side. This helps to prevent injection and replay attacks during transit.

The payload data, combined with a shared secret key, is hashed and is verified by the other peer. The digest types supported are:

Index	Digest Type	Computed hash size
0	Private / locally reserved	Variable
1	HMAC-MD5	16 octets
2	HMAC-SHA1	20 octets
3	HMAC-SHA256	32 octets
4	HMAC-CRC32	4 octets

The two peers must use the same key and digest type. If a peer is instructed to hash its messages, it must not accept replies that are not hashed with the same digest type and key.

Index 0 may be used by clients for digest types not defined by this specification, and implementations must ignore index 0 hashes if a non-standard digest has not been defined locally. Additional digest types may be added to this specification in the future.

The entire payload is hashed, from the magic number to the padding, inclusive. When computing the MAC hash, the hash value itself is filled with zeroed octets, and replaced with the computed hash. The MAC hash is computed before the payload checksum, therefore the checksum area must also be zeroed before the MAC hash is computed.

Use of this opcode is optional and its parameters (whether to use hashing, the digest type and the shared secret) must be coordinated between the two sides ahead of time. An implementation is not required to implement all digest types.

HMAC-CRC32 is included in this specification mostly as a joke, and a programming exercise for the reader. Please don't use this (and expect any sort of cryptographic data integrity). Endianness is not specified by CRC32; use network byte order (big endian). The HMAC specification requires aligning the key to the digest function's block size; CRC32 does not use a block size, so assume a block size of 64 bytes (same as MD5, SHA1 and SHA256).

0x0100 - Host processing latency

Delay in microseconds	4 octets, required

The time between receiving a message packet and sending a reply packet may be non-trivial, due to host processing. This opcode can be used by an implementation, when sending a reply (0x0002), to inform a peer how much time progressed between a receive and a send. The peer may then subtract this latency time to better calculate network transit time. Up to 2^32-1 microseconds are possible, approximately 71 minutes.

The delta should be computed from immediately after the original message packet has been received, to as soon as possible before the reply packet has been sent. Note that due to optional MAC hash and checksum calculation processing, which must be computed after the rest of the opcode data area, there still could be some host processing latency not accounted for in the reported latency time.

0x8000 - Extended segments

This opcode data segment extends the 2ping format, allowing for a nearly unlimited number of available segments. Within the opcode segment data area is a series of sub-segments, referred to as extended segments, consisting of a 32-bit extID, a length of the extended segment data, and the extended segment data itself. Each part is built as so:

Extended segment ID	8 octets, required
Extended segment data length	2 octets, required
Extended segment data	Variable, zero or more octets, determined by extended segment data length header above

Multiple extended segment parts are chained together to form the data area of a 0x8000 opcode. Parsing this opcode's data area is similar to parsing the opcodes as a whole: read the extended segment ID and determine if it's a known ID. If so, parse its contents by reading the extended segment data length and extended segment data. If not, read the extended segment data length to determine how far to skip over the extended segment data. If no extended segments are to be included with the packet payload, do not set the opcode.

Defined extended segments

Extended segment IDs are 32 bits, allowing for approximately 4.2 billion different pieces of functionality. When defining a local extension, please choose a random ID to avoid the possibility of conflicting extensions. If the extension is useful to the public as a whole, please consider submitting it for inclusion in this specification.

The following are registered extended segments. An implementation is not required to implement any extended segments (or the extended segment opcode itself, for that matter), but you are strongly encouraged to do so.

0x3250564e - Program version

Program version text	Variable length

The human-readable text version of the program or firmware generating the packet, with optional information such as architecture, etc. An example could be "Network Tools 3.0-1distro2 (x86_64-linux)".

As the 2ping protocol is designed to be backwards and forwards compatible, this field must not be used by an implementation to determine functionality. It is recommended that this field be sent with every packet, but received segments should not be displayed to the user unless in a verbose/debug/etc mode.

0xa837b44e - Notice text

Notice text	Variable length

Arbitrary text to be sent with the packet. This text should be defined by the user by a program flag, UI option, etc. It is designed to be human-readable; do not use this segment to pass machine-parsed data between the client and listener. Instead, if extended data transfer is desired between the client and listener, simply define a new extended segment ID.

The implementation may display this text to the user, but it is not guaranteed the user will see it.

Procedures

In the following pseudocode examples, message IDs are represented as zero-padded incrementing numbers. This is for the purpose of identifying reply chains in these examples only. In real life, message IDs MUST be randomly-generated 48-bit (6 octet) identifiers. The program implementing the 2ping protocol may choose to locally associate a better identifier for the user (such as an incrementing integer), but the protocol message ID must be random and pseudo-unique.

As mentioned above, the simplest 2ping packet is a 12-octet payload: 2 octets for the magic number (always 0x3250), 2 octets for the checksum field, 6 octets for the message ID, and 2 blank opcode flag octets:

Example 1

```
CLIENT: 00000000a001, no opcodes
```

Of course, this isn't very useful, and would be analogous to a NOOP. The simplest ping would require the "reply requested" opcode:

Example 2

```
CLIENT: 00000000a001, reply requested
```

```
SERVER: 00000000b001, in reply to 00000000a001
```

A 3-way ping further extends this.

Example 3

```
CLIENT: 00000000a001, reply requested
SERVER: 00000000b001, in reply to 00000000a001, reply requested
CLIENT: 00000000a002, in reply to 00000000b001, successful ping rtt
12345 µs
```

The client successfully received the response from the server and was able to measure an RTT of 12345 µs. The client then sends a reply back to the server, referencing the original reply, and letting it know the RTT it measured. The server is then able to determine its RTT between the second and third leg (say, 11804 µs), and also know the RTT between the first and second leg.

Now let's say the reply to the original client ping never came back. The client can start inquiring about whether the server saw the original request, and the server can provide info at the same time it is responding to a new ping request:

Example 4

```
CLIENT: 00000000a001, reply requested
CLIENT: 00000000a002, reply requested, did not receive reply to
00000000a001
SERVER: 00000000b002, in reply to 00000000a002, reply requested,
received and replied to 00000000a001
CLIENT: 00000000a003, in reply to 00000000b002, successful ping rtt
12345 µs
```

Now the client can tell that the server received the original request, and replied, which suggests inbound packet loss. Note from the example message IDs, the server most likely originally replied to `00000000a001` with `00000000b001`. However, the specific message ID the server originally replied with is not important and not tracked, and hence is not indicated in the investigation reply, only that it was originally received and replied to.

Conversely, for a request for reply that the server never actually received:

Example 5

```
CLIENT: 00000000a001, reply requested
CLIENT: 00000000a002, reply requested, did not receive reply to
00000000a001
SERVER: 00000000b001, in reply to 00000000a002, reply requested, never
received 00000000a001
CLIENT: 00000000a003, in reply to 00000000b001, successful ping rtt
```

```
12345 µs
```

This suggests outbound packet loss. As mentioned before, the client should not immediately start asking about replies never received. An actual sequence of events may look something like this:

Example 6

```
CLIENT: 00000000a001, reply requested
CLIENT: 00000000a002, reply requested
CLIENT: 00000000a003, reply requested
SERVER: 00000000b002, in reply to 00000000a003, reply requested
CLIENT: 00000000a004, in reply to 00000000b002, successful ping rtt
12823 µs
... etc
CLIENT: 00000000a00a, reply requested, did not receive reply to
00000000a001 or 00000000a002
SERVER: 00000000b006, in reply to 00000000a00a, reply requested,
received and replied to 00000000a001, never received 00000000a002, did
not receive reply to 00000000b002
CLIENT: 00000000a00b, in reply to 00000000b006, successful ping rtt
13112 µs, received and replied to 00000000b002
```

The client waited a few seconds before asking about `00000000a001 and 00000000a002`. The server replied that it knew about `00000000a001`, but never received `00000000a002`, indicating there is some packet loss both inbound and outbound.

Also notice that in the same packet where the server replied with info about the client's lost packets inquiry, it also inquired about its own lost packet. In this case, the server never received the last segment of the 3-way ping, `00000000b002`. The client then tells the server it did originally receive and respond to the packet in question.

This raises an interesting point, that there is essentially no difference between a server and a client in 2ping. The "server" is expected to listen for the initial datagrams, and the "client" is expected to initiate ping requests at regular intervals. But if the "server" decides to randomly initiate a ping request of its own, the "client" is expected to respond appropriately, as a server would do.

Reference packet dumps

Legend

- Magic number: green
- Checksum: navy

- Message ID: orange
- Opcode flags: blue
- Opcode segment length header: red
- Opcode segment data: purple
- Padding: Optional, not included for readability

Example 1

```
CLIENT: 32 50 00 00 00 00 00 00 a0 01 00 00
```

Example 2

```
CLIENT: 32 50 00 00 00 00 00 00 a0 01 00 01 00 00
SERVER: 32 50 00 00 00 00 00 00 b0 01 00 02 00 06 00 00 00 00 a0 01
```

Example 3

```
CLIENT: 32 50 00 00 00 00 00 00 a0 01 00 01 00 00
SERVER: 32 50 00 00 00 00 00 00 b0 01 00 03 00 00 00 06 00 00 00 00 a0
01
CLIENT: 32 50 00 00 00 00 00 00 a0 02 00 06 00 06 00 00 00 00 b0 01 00
04 00 00 30 39
```

Example 4

```
CLIENT: 32 50 00 00 00 00 00 00 a0 01 00 01 00 00
CLIENT: 32 50 00 00 00 00 00 00 a0 02 00 21 00 00 00 08 00 01 00 00 00
00 a0 01
SERVER: 32 50 00 00 00 00 00 00 b0 02 00 0b 00 00 00 06 00 00 00 00 a0
02 00 08 00 01 00 00 00 00 a0 01
CLIENT: 32 50 00 00 00 00 00 00 a0 03 00 06 00 06 00 00 00 00 b0 02 00
04 00 00 30 39
```

Example 5

```
CLIENT: 32 50 00 00 00 00 00 00 a0 01 00 01 00 00
CLIENT: 32 50 00 00 00 00 00 00 a0 02 00 21 00 00 00 08 00 01 00 00 00
00 a0 01
SERVER: 32 50 00 00 00 00 00 00 b0 01 00 13 00 00 00 06 00 00 00 00 a0
02 00 08 00 01 00 00 00 00 a0 01
CLIENT: 32 50 00 00 00 00 00 00 a0 03 00 06 00 06 00 00 00 00 b0 01 00
04 00 00 30 39
```

Example 6

```
CLIENT: 32 50 00 00 00 00 00 00 a0 01 00 01 00 00
CLIENT: 32 50 00 00 00 00 00 00 a0 02 00 01 00 00
```

```
CLIENT: 32 50 00 00 00 00 00 00 a0 03 00 01 00 00
SERVER: 32 50 00 00 00 00 00 00 b0 02 00 03 00 00 00 06 00 00 00 00 a0
03
CLIENT: 32 50 00 00 00 00 00 00 a0 04 00 06 00 06 00 00 00 00 b0 02 00
04 00 00 32 17
...
CLIENT: 32 50 00 00 00 00 00 00 a0 0a 00 21 00 00 00 0e 00 02 00 00 00
00 a0 01 00 01 00 00 00 00 a0 02
SERVER: 32 50 00 00 00 00 00 00 b0 06 00 3b 00 00 00 06 00 00 00 00 a0
0a 00 08 00 01 00 00 00 00 a0 01 00 08 00 01 00 00 00 00 a0 02 00 08
00 01 00 00 00 00 b0 02
CLIENT: 32 50 00 00 00 00 00 00 a0 0b 00 0e 00 06 00 00 00 00 b0 06 00
04 00 00 33 38 00 08 00 01 00 00 00 00 b0 02
```

TODO: Compute checksums in examples. (Zeroed checksums are valid per the protocol, but should still be computed.)

Changelog

2.0 (20120422)

- Protocol versions 1.0 and 2.0 are backwards and forwards compatible with each other.
- Changed recommended default minimum packet size from 64 octets to 128 octets.
- Added an extended segment container at opcode 0x8000.
- Added the following registered extended segments:
    - 0x3250564e: Program version
    - 0xa837b44e: Notice text

1.0 (20101020)

- Finalized initial release.

Copyright