TRAM
Internet-Draft
Obsoletes: 5389 (if approved)
Intended status: Standards Track
Expires: September 14, 2017

M. Petit-Huguenin
Impedance Mismatch
G. Salgueiro
J. Rosenberg
Cisco
D. Wing

R. Mahy
Plantronics
P. Matthews
Avaya

March 13, 2017

Session Traversal Utilities for NAT (STUN)
draft-ietf-tram-stunbis-11

Abstract

Session Traversal Utilities for NAT (STUN) is a protocol that serves as a tool for other protocols in dealing with Network Address Translator (NAT) traversal. It can be used by an endpoint to determine the IP address and port allocated to it by a NAT. It can also be used to check connectivity between two endpoints, and as a keep-alive protocol to maintain NAT bindings. STUN works with many existing NATs, and does not require any special behavior from them.

STUN is not a NAT traversal solution by itself. Rather, it is a tool to be used in the context of a NAT traversal solution.

This document obsoletes RFC 5389.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 14, 2017.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction 5
2. Overview of Operation 5
3. Terminology 8
4. Definitions 8
5. STUN Message Structure 10
6. Base Protocol Procedures 12
6.1. Forming a Request or an Indication 12
6.2. Sending the Request or Indication 13
6.2.1. Sending over UDP or DTLS-over-UDP 14
6.2.2. Sending over TCP or TLS-over-TCP 15
6.2.3. Sending over TLS-over-TCP or DTLS-over-UDP 16
6.3. Receiving a STUN Message 17
6.3.1. Processing a Request 17
6.3.1.1. Forming a Success or Error Response 18
6.3.1.2. Sending the Success or Error Response 19
6.3.2. Processing an Indication 19
6.3.3. Processing a Success Response 20
6.3.4. Processing an Error Response 20
7. FINGERPRINT Mechanism 21
8. DNS Discovery of a Server 21
8.1. STUN URI Scheme Semantics 22
9. Authentication and Message-Integrity Mechanisms 23
9.1. Short-Term Credential Mechanism 23
9.1.1. HMAC Key 23
9.1.2. Forming a Request or Indication 24
9.1.3. Receiving a Request or Indication 24
9.1.4. Receiving a Response 25
9.1.5. Sending Subsequent Requests 26
9.2. Long-Term Credential Mechanism 26
9.2.1. Bid Down Attack Prevention 27
9.2.2. HMAC Key 28

9.2.3. Forming a Request 28
9.2.3.1. First Request 29
9.2.3.2. Subsequent Requests 29
9.2.4. Receiving a Request 29
9.2.5. Receiving a Response 31
10. ALTERNATE-SERVER Mechanism 33
11. Backwards Compatibility with RFC 3489 34
12. Basic Server Behavior 34
13. STUN Usages 35
14. STUN Attributes 36
14.1. MAPPED-ADDRESS 37
14.2. XOR-MAPPED-ADDRESS 37
14.3. USERNAME 38
14.4. USERHASH 39
14.5. MESSAGE-INTEGRITY 39
14.6. MESSAGE-INTEGRITY-SHA256 40
14.7. FINGERPRINT 41
14.8. ERROR-CODE 41
14.9. REALM 43
14.10. NONCE 43
14.11. PASSWORD-ALGORITHMS 43
14.12. PASSWORD-ALGORITHM 44
14.13. UNKNOWN-ATTRIBUTES 45
14.14. SOFTWARE 45
14.15. ALTERNATE-SERVER 45
14.16. ALTERNATE-DOMAIN 45
15. Security Considerations 46
15.1. Attacks against the Protocol 46
15.1.1. Outside Attacks 46
15.1.2. Inside Attacks 47
15.2. Attacks Affecting the Usage 47
15.2.1. Attack I: Distributed DoS (DDoS) against a Target . 48
15.2.2. Attack II: Silencing a Client 48
15.2.3. Attack III: Assuming the Identity of a Client . . . 48
15.2.4. Attack IV: Eavesdropping 48
15.3. Hash Agility Plan 49
16. IAB Considerations 49
17. IANA Considerations 49
17.1. STUN Security Features Registry 50
17.2. STUN Methods Registry 50
17.3. STUN Attribute Registry 50
17.3.1. Updated Attributes 50
17.3.2. New Attributes 51
17.4. STUN Error Code Registry 51
17.5. Password Algorithm Registry 51
17.5.1. Password Algorithms 52
17.5.1.1. MD5 52
17.5.1.2. SHA256 52

17.6. STUN UDP and TCP Port Numbers 52
18. Changes since RFC 5389 52
19. References 53
19.1. Normative References 53
19.2. Informative References 55
Appendix A. C Snippet to Determine STUN Message Types 57
Appendix B. Test Vectors 58
B.1. Sample Request with Long-Term Authentication with MESSAGE-INTEGRITY-SHA256 and USERHASH 58
Appendix C. Release notes 60
C.1. Modifications between draft-ietf-tram-stunbis-11 and draft-ietf-tram-stunbis-10 60
C.2. Modifications between draft-ietf-tram-stunbis-10 and draft-ietf-tram-stunbis-09 60
C.3. Modifications between draft-ietf-tram-stunbis-09 and draft-ietf-tram-stunbis-08 60
C.4. Modifications between draft-ietf-tram-stunbis-09 and draft-ietf-tram-stunbis-08 61
C.5. Modifications between draft-ietf-tram-stunbis-08 and draft-ietf-tram-stunbis-07 61
C.6. Modifications between draft-ietf-tram-stunbis-07 and draft-ietf-tram-stunbis-06 62
C.7. Modifications between draft-ietf-tram-stunbis-06 and draft-ietf-tram-stunbis-05 62
C.8. Modifications between draft-ietf-tram-stunbis-05 and draft-ietf-tram-stunbis-04 62
C.9. Modifications between draft-ietf-tram-stunbis-04 and draft-ietf-tram-stunbis-03 62
C.10. Modifications between draft-ietf-tram-stunbis-03 and draft-ietf-tram-stunbis-02 63
C.11. Modifications between draft-ietf-tram-stunbis-02 and draft-ietf-tram-stunbis-01 63
C.12. Modifications between draft-ietf-tram-stunbis-01 and draft-ietf-tram-stunbis-00 64
C.13. Modifications between draft-salgueiro-tram-stunbis-02 and draft-ietf-tram-stunbis-00 64
C.14. Modifications between draft-salgueiro-tram-stunbis-02 and draft-salgueiro-tram-stunbis-01 64
C.15. Modifications between draft-salgueiro-tram-stunbis-01 and draft-salgueiro-tram-stunbis-00 65
Acknowledgements 65
Contributors 65
Authors' Addresses 66

1. Introduction

The protocol defined in this specification, Session Traversal Utilities for NAT, provides a tool for dealing with NATs. It provides a means for an endpoint to determine the IP address and port allocated by a NAT that corresponds to its private IP address and port. It also provides a way for an endpoint to keep a NAT binding alive. With some extensions, the protocol can be used to do connectivity checks between two endpoints [I-D.ietf-ice-rfc5245bis], or to relay packets between two endpoints [RFC5766].

In keeping with its tool nature, this specification defines an extensible packet format, defines operation over several transport protocols, and provides for two forms of authentication.

STUN is intended to be used in context of one or more NAT traversal solutions. These solutions are known as STUN usages. Each usage describes how STUN is utilized to achieve the NAT traversal solution. Typically, a usage indicates when STUN messages get sent, which optional attributes to include, what server is used, and what authentication mechanism is to be used. Interactive Connectivity Establishment (ICE) [I-D.ietf-ice-rfc5245bis] is one usage of STUN. SIP Outbound [RFC5626] is another usage of STUN. In some cases, a usage will require extensions to STUN. A STUN extension can be in the form of new methods, attributes, or error response codes. More information on STUN usages can be found in Section 13.

Implementations and deployments of a STUN usage using TLS or DTLS should follow the recommendations in [RFC7525].

2. Overview of Operation

This section is descriptive only.

```
                          /-----\
                        // STUN  \\
                       |   Server  |
                        \\       //
                          \-----/



                    +--------------+             Public Internet
................|     NAT 2    |......................
                    +--------------+



                    +--------------+             Private NET 2
................|     NAT 1    |......................
                    +--------------+



                          /-----\
                        // STUN  \\
                       |   Client  |
                        \\       //               Private NET 1
                          \-----/
```

Figure 1: One Possible STUN Configuration

One possible STUN configuration is shown in Figure 1. In this configuration, there are two entities (called STUN agents) that implement the STUN protocol. The lower agent in the figure is the client, and is connected to private network 1. This network connects to private network 2 through NAT 1. Private network 2 connects to the public Internet through NAT 2. The upper agent in the figure is the server, and resides on the public Internet.

STUN is a client-server protocol. It supports two types of transactions. One is a request/response transaction in which a client sends a request to a server, and the server returns a response. The second is an indication transaction in which either agent -- client or server -- sends an indication that generates no response. Both types of transactions include a transaction ID, which is a randomly selected 96-bit number. For request/response transactions, this transaction ID allows the client to associate the response with the request that generated it; for indications, the transaction ID serves as a debugging aid.

All STUN messages start with a fixed header that includes a method, a class, and the transaction ID. The method indicates which of the various requests or indications this is; this specification defines just one method, Binding, but other methods are expected to be defined in other documents. The class indicates whether this is a request, a success response, an error response, or an indication. Following the fixed header comes zero or more attributes, which are Type-Length-Value extensions that convey additional information for the specific message.

This document defines a single method called Binding. The Binding method can be used either in request/response transactions or in indication transactions. When used in request/response transactions, the Binding method can be used to determine the particular "binding" a NAT has allocated to a STUN client. When used in either request/response or in indication transactions, the Binding method can also be used to keep these "bindings" alive.

In the Binding request/response transaction, a Binding request is sent from a STUN client to a STUN server. When the Binding request arrives at the STUN server, it may have passed through one or more NATs between the STUN client and the STUN server (in Figure 1, there were two such NATs). As the Binding request message passes through a NAT, the NAT will modify the source transport address (that is, the source IP address and the source port) of the packet. As a result, the source transport address of the request received by the server will be the public IP address and port created by the NAT closest to the server. This is called a reflexive transport address. The STUN server copies that source transport address into an XOR-MAPPED-ADDRESS attribute in the STUN Binding response and sends the Binding response back to the STUN client. As this packet passes back through a NAT, the NAT will modify the destination transport address in the IP header, but the transport address in the XOR-MAPPED-ADDRESS attribute within the body of the STUN response will remain untouched. In this way, the client can learn its reflexive transport address allocated by the outermost NAT with respect to the STUN server.

In some usages, STUN must be multiplexed with other protocols (e.g., [I-D.ietf-ice-rfc5245bis], [RFC5626]). In these usages, there must be a way to inspect a packet and determine if it is a STUN packet or not. STUN provides three fields in the STUN header with fixed values that can be used for this purpose. If this is not sufficient, then STUN packets can also contain a FINGERPRINT value, which can further be used to distinguish the packets.

STUN defines a set of optional procedures that a usage can decide to use, called mechanisms. These mechanisms include DNS discovery, a redirection technique to an alternate server, a fingerprint attribute

for demultiplexing, and two authentication and message-integrity exchanges. The authentication mechanisms revolve around the use of a username, password, and message-integrity value. Two authentication mechanisms, the long-term credential mechanism and the short-term credential mechanism, are defined in this specification. Each usage specifies the mechanisms allowed with that usage.

In the long-term credential mechanism, the client and server share a pre-provisioned username and password and perform a digest challenge/response exchange inspired by (but differing in details) to the one defined for HTTP [RFC2617]. In the short-term credential mechanism, the client and the server exchange a username and password through some out-of-band method prior to the STUN exchange. For example, in the ICE usage [I-D.ietf-ice-rfc5245bis] the two endpoints use out-of-band signaling to exchange a username and password. These are used to integrity protect and authenticate the request and response. There is no challenge or nonce used.

3. Terminology

In this document, the key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" are to be interpreted as described in [RFC2119] and indicate requirement levels for compliant STUN implementations.

4. Definitions

STUN Agent: A STUN agent is an entity that implements the STUN protocol. The entity can be either a STUN client or a STUN server.

STUN Client: A STUN client is an entity that sends STUN requests and receives STUN responses. A STUN client can also send indications. In this specification, the terms STUN client and client are synonymous.

STUN Server: A STUN server is an entity that receives STUN requests and sends STUN responses. A STUN server can also send indications. In this specification, the terms STUN server and server are synonymous.

Transport Address: The combination of an IP address and port number (such as a UDP or TCP port number).

Reflexive Transport Address: A transport address learned by a client that identifies that client as seen by another host on an IP network, typically a STUN server. When there is an intervening NAT between the client and the other host, the reflexive transport

address represents the mapped address allocated to the client on the public side of the NAT. Reflexive transport addresses are learned from the mapped address attribute (MAPPED-ADDRESS or XOR-MAPPED-ADDRESS) in STUN responses.

Mapped Address: Same meaning as reflexive address. This term is retained only for historic reasons and due to the naming of the MAPPED-ADDRESS and XOR-MAPPED-ADDRESS attributes.

Long-Term Credential: A username and associated password that represent a shared secret between client and server. Long-term credentials are generally granted to the client when a subscriber enrolls in a service and persist until the subscriber leaves the service or explicitly changes the credential.

Long-Term Password: The password from a long-term credential.

Short-Term Credential: A temporary username and associated password that represent a shared secret between client and server. Short-term credentials are obtained through some kind of protocol mechanism between the client and server, preceding the STUN exchange. A short-term credential has an explicit temporal scope, which may be based on a specific amount of time (such as 5 minutes) or on an event (such as termination of a SIP dialog). The specific scope of a short-term credential is defined by the application usage.

Short-Term Password: The password component of a short-term credential.

STUN Indication: A STUN message that does not receive a response.

Attribute: The STUN term for a Type-Length-Value (TLV) object that can be added to a STUN message. Attributes are divided into two types: comprehension-required and comprehension-optional. STUN agents can safely ignore comprehension-optional attributes they don’t understand, but cannot successfully process a message if it contains comprehension-required attributes that are not understood.

RTO: Retransmission TimeOut, which defines the initial period of time between transmission of a request and the first retransmit of that request.

5. STUN Message Structure

STUN messages are encoded in binary using network-oriented format (most significant byte or octet first, also commonly known as big-endian). The transmission order is described in detail in Appendix B of [RFC0791]. Unless otherwise noted, numeric constants are in decimal (base 10).

All STUN messages MUST start with a 20-byte header followed by zero or more Attributes. The STUN header contains a STUN message type, magic cookie, transaction ID, and message length.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0|     STUN Message Type     |         Message Length        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Magic Cookie                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
|                     Transaction ID (96 bits)                  |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2: Format of STUN Message Header

The most significant 2 bits of every STUN message MUST be zeroes. This can be used to differentiate STUN packets from other protocols when STUN is multiplexed with other protocols on the same port.

The message type defines the message class (request, success response, failure response, or indication) and the message method (the primary function) of the STUN message. Although there are four message classes, there are only two types of transactions in STUN: request/response transactions (which consist of a request message and a response message) and indication transactions (which consist of a single indication message). Response classes are split into error and success responses to aid in quickly processing the STUN message.

The message type field is decomposed further into the following structure:

```
 0                 1
 2  3  4 5 6 7 8 9 0 1 2 3 4 5
+--+--+-+-+-+-+-+-+-+-+-+-+-+-+
|M |M |M|M|M|C|M|M|M|C|M|M|M|M|
|11|10|9|8|7|1|6|5|4|0|3|2|1|0|
+--+--+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 3: Format of STUN Message Type Field

Here the bits in the message type field are shown as most significant (M11) through least significant (M0). M11 through M0 represent a 12-bit encoding of the method. C1 and C0 represent a 2-bit encoding of the class. A class of 0b00 is a request, a class of 0b01 is an indication, a class of 0b10 is a success response, and a class of 0b11 is an error response. This specification defines a single method, Binding. The method and class are orthogonal, so that for each method, a request, success response, error response, and indication are possible for that method. Extensions defining new methods MUST indicate which classes are permitted for that method.

For example, a Binding request has class=0b00 (request) and method=0b000000000001 (Binding) and is encoded into the first 16 bits as 0x0001. A Binding response has class=0b10 (success response) and method=0b000000000001, and is encoded into the first 16 bits as 0x0101.

Note: This unfortunate encoding is due to assignment of values in [RFC3489] that did not consider encoding Indications, Success, and Errors using bit fields.

The magic cookie field MUST contain the fixed value 0x2112A442 in network byte order. In [RFC3489], this field was part of the transaction ID; placing the magic cookie in this location allows a server to detect if the client will understand certain attributes that were added in this revised specification. In addition, it aids in distinguishing STUN packets from packets of other protocols when STUN is multiplexed with those other protocols on the same port.

The transaction ID is a 96-bit identifier, used to uniquely identify STUN transactions. For request/response transactions, the transaction ID is chosen by the STUN client for the request and echoed by the server in the response. For indications, it is chosen by the agent sending the indication. It primarily serves to correlate requests with responses, though it also plays a small role in helping to prevent certain types of attacks. The server also uses the transaction ID as a key to identify each transaction uniquely across all clients. As such, the transaction ID MUST be uniformly and randomly chosen from the interval 0 .. 2**96-1, and SHOULD be

cryptographically random. Resends of the same request reuse the same transaction ID, but the client MUST choose a new transaction ID for new transactions unless the new request is bit-wise identical to the previous request and sent from the same transport address to the same IP address. Success and error responses MUST carry the same transaction ID as their corresponding request. When an agent is acting as a STUN server and STUN client on the same port, the transaction IDs in requests sent by the agent have no relationship to the transaction IDs in requests received by the agent.

The message length MUST contain the size, in bytes, of the message not including the 20-byte STUN header. Since all STUN attributes are padded to a multiple of 4 bytes, the last 2 bits of this field are always zero. This provides another way to distinguish STUN packets from packets of other protocols.

Following the STUN fixed portion of the header are zero or more attributes. Each attribute is TLV (Type-Length-Value) encoded. The details of the encoding, and of the attributes themselves are given in Section 14.

6. Base Protocol Procedures

This section defines the base procedures of the STUN protocol. It describes how messages are formed, how they are sent, and how they are processed when they are received. It also defines the detailed processing of the Binding method. Other sections in this document describe optional procedures that a usage may elect to use in certain situations. Other documents may define other extensions to STUN, by adding new methods, new attributes, or new error response codes.

6.1. Forming a Request or an Indication

When formulating a request or indication message, the agent MUST follow the rules in Section 5 when creating the header. In addition, the message class MUST be either "Request" or "Indication" (as appropriate), and the method must be either Binding or some method defined in another document.

The agent then adds any attributes specified by the method or the usage. For example, some usages may specify that the agent use an authentication method (Section 9) or the FINGERPRINT attribute (Section 7).

If the agent is sending a request, it SHOULD add a SOFTWARE attribute to the request. Agents MAY include a SOFTWARE attribute in indications, depending on the method. Extensions to STUN should discuss whether SOFTWARE is useful in new indications.

For the Binding method with no authentication, no attributes are required unless the usage specifies otherwise.

All STUN messages sent over UDP or DTLS-over-UDP [RFC6347] SHOULD be less than the path MTU, if known.

If the path MTU is unknown for UDP, messages SHOULD be the smaller of 576 bytes and the first-hop MTU for IPv4 [RFC1122] and 1280 bytes for IPv6 [RFC2460]. This value corresponds to the overall size of the IP packet. Consequently, for IPv4, the actual STUN message would need to be less than 548 bytes (576 minus 20-byte IP header, minus 8-byte UDP header, assuming no IP options are used).

If the path MTU is unknown for DTLS-over-UDP, the rules described in the previous paragraph need to be adjusted to take into account the size of the (13-byte) DTLS Record header, the MAC size, and the padding size.

STUN provides no ability to handle the case where the request is under the MTU but the response would be larger than the MTU. It is not envisioned that this limitation will be an issue for STUN. The MTU limitation is a SHOULD, and not a MUST, to account for cases where STUN itself is being used to probe for MTU characteristics [RFC5780]. Outside of this or similar applications, the MTU constraint MUST be followed.

6.2. Sending the Request or Indication

The agent then sends the request or indication. This document specifies how to send STUN messages over UDP, TCP, TLS-over-TCP, or DTLS-over-UDP; other transport protocols may be added in the future. The STUN usage must specify which transport protocol is used, and how the agent determines the IP address and port of the recipient. Section 8 describes a DNS-based method of determining the IP address and port of a server that a usage may elect to use. STUN may be used with anycast addresses, but only with UDP and in usages where authentication is not used.

At any time, a client MAY have multiple outstanding STUN requests with the same STUN server (that is, multiple transactions in progress, with different transaction IDs). Absent other limits to the rate of new transactions (such as those specified by ICE for connectivity checks or when STUN is run over TCP), a client SHOULD limit itself to ten outstanding transactions to the same server.

6.2.1. Sending over UDP or DTLS-over-UDP

When running STUN over UDP or STUN over DTLS-over-UDP [RFC7350], it is possible that the STUN message might be dropped by the network. Reliability of STUN request/response transactions is accomplished through retransmissions of the request message by the client application itself. STUN indications are not retransmitted; thus, indication transactions over UDP or DTLS-over-UDP are not reliable.

A client SHOULD retransmit a STUN request message starting with an interval of RTO ("Retransmission TimeOut"), doubling after each retransmission. The RTO is an estimate of the round-trip time (RTT), and is computed as described in [RFC6298], with two exceptions. First, the initial value for RTO SHOULD be greater than 500 ms. The exception cases for this "SHOULD" are when other mechanisms are used to derive congestion thresholds (such as the ones defined in ICE for fixed rate streams), or when STUN is used in non-Internet environments with known network capacities. In fixed-line access links, a value of 500 ms is RECOMMENDED. Second, the value of RTO SHOULD NOT be rounded up to the nearest second. Rather, a 1 ms accuracy SHOULD be maintained. As with TCP, the usage of Karn’s algorithm is RECOMMENDED [KARN87]. When applied to STUN, it means that RTT estimates SHOULD NOT be computed from STUN transactions that result in the retransmission of a request.

The value for RTO SHOULD be cached by a client after the completion of the transaction, and used as the starting value for RTO for the next transaction to the same server (based on equality of IP address). The value SHOULD be considered stale and discarded after 10 minutes without any transactions to the same server.

Retransmissions continue until a response is received, or until a total of Rc requests have been sent. Rc SHOULD be configurable and SHOULD have a default of 7. If, after the last request, a duration equal to Rm times the RTO has passed without a response (providing ample time to get a response if only this final request actually succeeds), the client SHOULD consider the transaction to have failed. Rm SHOULD be configurable and SHOULD have a default of 16. A STUN transaction over UDP or DTLS-over-UDP is also considered failed if there has been a hard ICMP error [RFC1122]. For example, assuming an RTO of 500ms, requests would be sent at times 0 ms, 500 ms, 1500 ms, 3500 ms, 7500 ms, 15500 ms, and 31500 ms. If the client has not received a response after 39500 ms, the client will consider the transaction to have timed out.

6.2.2. Sending over TCP or TLS-over-TCP

For TCP and TLS-over-TCP [RFC5246], the client opens a TCP connection to the server.

In some usages of STUN, STUN is sent as the only protocol over the TCP connection. In this case, it can be sent without the aid of any additional framing or demultiplexing. In other usages, or with other extensions, it may be multiplexed with other data over a TCP connection. In that case, STUN MUST be run on top of some kind of framing protocol, specified by the usage or extension, which allows for the agent to extract complete STUN messages and complete application layer messages. The STUN service running on the well-known port or ports discovered through the DNS procedures in Section 8 is for STUN alone, and not for STUN multiplexed with other data. Consequently, no framing protocols are used in connections to those servers. When additional framing is utilized, the usage will specify how the client knows to apply it and what port to connect to. For example, in the case of ICE connectivity checks, this information is learned through out-of-band negotiation between client and server.

Reliability of STUN over TCP and TLS-over-TCP is handled by TCP itself, and there are no retransmissions at the STUN protocol level. However, for a request/response transaction, if the client has not received a response by Ti seconds after it sent the SYN to establish the connection, it considers the transaction to have timed out. Ti SHOULD be configurable and SHOULD have a default of 39.5s. This value has been chosen to equalize the TCP and UDP timeouts for the default initial RTO.

In addition, if the client is unable to establish the TCP connection, or the TCP connection is reset or fails before a response is received, any request/response transaction in progress is considered to have failed.

The client MAY send multiple transactions over a single TCP (or TLS-over-TCP) connection, and it MAY send another request before receiving a response to the previous. The client SHOULD keep the connection open until it:

- o has no further STUN requests or indications to send over that connection, and

- o has no plans to use any resources (such as a mapped address (MAPPED-ADDRESS or XOR-MAPPED-ADDRESS) or relayed address [RFC5766]) that were learned though STUN requests sent over that connection, and

- o if multiplexing other application protocols over that port, has finished using that other application, and

- o if using that learned port with a remote peer, has established communications with that remote peer, as is required by some TCP NAT traversal techniques (e.g., [RFC6544]).

At the server end, the server SHOULD keep the connection open, and let the client close it, unless the server has determined that the connection has timed out (for example, due to the client disconnecting from the network). Bindings learned by the client will remain valid in intervening NATs only while the connection remains open. Only the client knows how long it needs the binding. The server SHOULD NOT close a connection if a request was received over that connection for which a response was not sent. A server MUST NOT ever open a connection back towards the client in order to send a response. Servers SHOULD follow best practices regarding connection management in cases of overload.

6.2.3. Sending over TLS-over-TCP or DTLS-over-UDP

When STUN is run by itself over TLS-over-TCP or DTLS-over-UDP, the TLS_DHE_RSA_WITH_AES_128_GCM_SHA256 and TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 cipher suites MUST be implemented and other cipher suites MAY be implemented. Perfect Forward Secrecy (PFS) cipher suites MUST be preferred over non-PFS cipher suites. Cipher suites with known weaknesses, such as those based on (single) DES and RC4, MUST NOT be used. Implementations MUST disable TLS-level compression.

When it receives the TLS Certificate message, the client SHOULD verify the certificate and inspect the site identified by the certificate. If the certificate is invalid or revoked, or if it does not identify the appropriate party, the client MUST NOT send the STUN message or otherwise proceed with the STUN transaction. The client MUST verify the identity of the server. To do that, it follows the identification procedures defined in [RFC6125]. Alternatively, a client MAY be configured with a set of domains or IP addresses that are trusted; if a certificate is received that identifies one of those domains or IP addresses, the client considers the identity of the server to be verified.

When STUN is run multiplexed with other protocols over a TLS-over-TCP connection or a DTLS-over-UDP association, the mandatory ciphersuites and TLS handling procedures operate as defined by those protocols.

6.3. Receiving a STUN Message

This section specifies the processing of a STUN message. The processing specified here is for STUN messages as defined in this specification; additional rules for backwards compatibility are defined in Section 11. Those additional procedures are optional, and usages can elect to utilize them. First, a set of processing operations is applied that is independent of the class. This is followed by class-specific processing, described in the subsections that follow.

When a STUN agent receives a STUN message, it first checks that the message obeys the rules of Section 5. It checks that the first two bits are 0, that the magic cookie field has the correct value, that the message length is sensible, and that the method value is a supported method. It checks that the message class is allowed for the particular method. If the message class is "Success Response" or "Error Response", the agent checks that the transaction ID matches a transaction that is still in progress. If the FINGERPRINT extension is being used, the agent checks that the FINGERPRINT attribute is present and contains the correct value. If any errors are detected, the message is silently discarded. In the case when STUN is being multiplexed with another protocol, an error may indicate that this is not really a STUN message; in this case, the agent should try to parse the message as a different protocol.

The STUN agent then does any checks that are required by a authentication mechanism that the usage has specified (see Section 9).

Once the authentication checks are done, the STUN agent checks for unknown attributes and known-but-unexpected attributes in the message. Unknown comprehension-optional attributes MUST be ignored by the agent. Known-but-unexpected attributes SHOULD be ignored by the agent. Unknown comprehension-required attributes cause processing that depends on the message class and is described below.

At this point, further processing depends on the message class of the request.

6.3.1. Processing a Request

If the request contains one or more unknown comprehension-required attributes, the server replies with an error response with an error code of 420 (Unknown Attribute), and includes an UNKNOWN-ATTRIBUTES attribute in the response that lists the unknown comprehension-required attributes.

The server then does any additional checking that the method or the specific usage requires. If all the checks succeed, the server formulates a success response as described below.

When run over UDP or DTLS-over-UDP, a request received by the server could be the first request of a transaction, or a retransmission. The server MUST respond to retransmissions such that the following property is preserved: if the client receives the response to the retransmission and not the response that was sent to the original request, the overall state on the client and server is identical to the case where only the response to the original retransmission is received, or where both responses are received (in which case the client will use the first). The easiest way to meet this requirement is for the server to remember all transaction IDs received over UDP or DTLS-over-UDP and their corresponding responses in the last 40 seconds. However, this requires the server to hold state, and will be inappropriate for any requests which are not authenticated. Another way is to reprocess the request and recompute the response. The latter technique MUST only be applied to requests that are idempotent (a request is considered idempotent when the same request can be safely repeated without impacting the overall state of the system) and result in the same success response for the same request. The Binding method is considered to be idempotent. Note that there are certain rare network events that could cause the reflexive transport address value to change, resulting in a different mapped address in different success responses. Extensions to STUN MUST discuss the implications of request retransmissions on servers that do not store transaction state.

6.3.1.1. Forming a Success or Error Response

When forming the response (success or error), the server follows the rules of Section 6. The method of the response is the same as that of the request, and the message class is either "Success Response" or "Error Response".

For an error response, the server MUST add an ERROR-CODE attribute containing the error code specified in the processing above. The reason phrase is not fixed, but SHOULD be something suitable for the error code. For certain errors, additional attributes are added to the message. These attributes are spelled out in the description where the error code is specified. For example, for an error code of 420 (Unknown Attribute), the server MUST include an UNKNOWN-ATTRIBUTES attribute. Certain authentication errors also cause attributes to be added (see Section 9). Extensions may define other errors and/or additional attributes to add in error cases.

If the server authenticated the request using an authentication mechanism, then the server SHOULD add the appropriate authentication attributes to the response (see Section 9).

The server also adds any attributes required by the specific method or usage. In addition, the server SHOULD add a SOFTWARE attribute to the message.

For the Binding method, no additional checking is required unless the usage specifies otherwise. When forming the success response, the server adds a XOR-MAPPED-ADDRESS attribute to the response, where the contents of the attribute are the source transport address of the request message. For UDP or DTLS-over-UDP this is the source IP address and source UDP port of the request message. For TCP and TLS-over-TCP, this is the source IP address and source TCP port of the TCP connection as seen by the server.

6.3.1.2. Sending the Success or Error Response

The response (success or error) is sent over the same transport as the request was received on. If the request was received over UDP or DTLS-over-UDP the destination IP address and port of the response are the source IP address and port of the received request message, and the source IP address and port of the response are equal to the destination IP address and port of the received request message. If the request was received over TCP or TLS-over-TCP, the response is sent back on the same TCP connection as the request was received on.

6.3.2. Processing an Indication

If the indication contains unknown comprehension-required attributes, the indication is discarded and processing ceases.

The agent then does any additional checking that the method or the specific usage requires. If all the checks succeed, the agent then processes the indication. No response is generated for an indication.

For the Binding method, no additional checking or processing is required, unless the usage specifies otherwise. The mere receipt of the message by the agent has refreshed the "bindings" in the intervening NATs.

Since indications are not re-transmitted over UDP or DTLS-over-UDP (unlike requests), there is no need to handle re-transmissions of indications at the sending agent.

6.3.3. Processing a Success Response

If the success response contains unknown comprehension-required attributes, the response is discarded and the transaction is considered to have failed.

The client then does any additional checking that the method or the specific usage requires. If all the checks succeed, the client then processes the success response.

For the Binding method, the client checks that the XOR-MAPPED-ADDRESS attribute is present in the response. The client checks the address family specified. If it is an unsupported address family, the attribute SHOULD be ignored. If it is an unexpected but supported address family (for example, the Binding transaction was sent over IPv4, but the address family specified is IPv6), then the client MAY accept and use the value.

6.3.4. Processing an Error Response

If the error response contains unknown comprehension-required attributes, or if the error response does not contain an ERROR-CODE attribute, then the transaction is simply considered to have failed.

The client then does any processing specified by the authentication mechanism (see Section 9). This may result in a new transaction attempt.

The processing at this point depends on the error code, the method, and the usage; the following are the default rules:

- If the error code is 300 through 399, the client SHOULD consider the transaction as failed unless the ALTERNATE-SERVER extension is being used. See Section 10.

- If the error code is 400 through 499, the client declares the transaction failed; in the case of 420 (Unknown Attribute), the response should contain a UNKNOWN-ATTRIBUTES attribute that gives additional information.

- If the error code is 500 through 599, the client MAY resend the request; clients that do so MUST limit the number of times they do this.

Any other error code causes the client to consider the transaction failed.

7. FINGERPRINT Mechanism

This section describes an optional mechanism for STUN that aids in distinguishing STUN messages from packets of other protocols when the two are multiplexed on the same transport address. This mechanism is optional, and a STUN usage must describe if and when it is used. The FINGERPRINT mechanism is not backwards compatible with RFC3489, and cannot be used in environments where such compatibility is required.

In some usages, STUN messages are multiplexed on the same transport address as other protocols, such as the Real Time Transport Protocol (RTP). In order to apply the processing described in Section 6, STUN messages must first be separated from the application packets.

Section 5 describes three fixed fields in the STUN header that can be used for this purpose. However, in some cases, these three fixed fields may not be sufficient.

When the FINGERPRINT extension is used, an agent includes the FINGERPRINT attribute in messages it sends to another agent. Section 14.7 describes the placement and value of this attribute.

When the agent receives what it believes is a STUN message, then, in addition to other basic checks, the agent also checks that the message contains a FINGERPRINT attribute and that the attribute contains the correct value. Section 6.3 describes when in the overall processing of a STUN message the FINGERPRINT check is performed. This additional check helps the agent detect messages of other protocols that might otherwise seem to be STUN messages.

8. DNS Discovery of a Server

This section describes an optional procedure for STUN that allows a client to use DNS to determine the IP address and port of a server. A STUN usage must describe if and when this extension is used. To use this procedure, the client must know a STUN URI [RFC7064]; the usage must also describe how the client obtains this URI. Hard-coding a STUN URI into software is NOT RECOMMENDED in case the domain name is lost or needs to change for legal or other reasons.

When a client wishes to locate a STUN server on the public Internet that accepts Binding request/response transactions, the STUN URI scheme is "stun". When it wishes to locate a STUN server that accepts Binding request/response transactions over a TLS, or DTLS session, the URI scheme is "stuns".

The syntax of the "stun" and "stuns" URIs are defined in Section 3.1 of [RFC7064]. STUN usages MAY define additional URI schemes.

8.1. STUN URI Scheme Semantics

If the <host> part contains an IP address, then this IP address is used directly to contact the server. A "stuns" URI containing an IP address MUST be rejected, unless the domain name is provided by the same mechanism that provided the STUN URI, and that domain name can be passed to the verification code.

If the URI does not contain an IP address, the domain name contained in the <host> part is resolved to a transport address using the SRV procedures specified in [RFC2782]. The DNS SRV service name is the content of the <scheme> part. The protocol in the SRV lookup is the transport protocol the client will run STUN over: "udp" for UDP and "tcp" for TCP.

The procedures of RFC 2782 are followed to determine the server to contact. RFC 2782 spells out the details of how a set of SRV records is sorted and then tried. However, RFC 2782 only states that the client should "try to connect to the (protocol, address, service)" without giving any details on what happens in the event of failure. When following these procedures, if the STUN transaction times out without receipt of a response, the client SHOULD retry the request to the next server in the ordered defined by RFC 2782. Such a retry is only possible for request/response transmissions, since indication transactions generate no response or timeout.

The default port for STUN requests is 3478, for both TCP and UDP. The default port for STUN over TLS and STUN over DTLS requests is 5349. Servers can run STUN over DTLS on the same port as STUN over UDP if the server software supports determining whether the initial message is a DTLS or STUN message. Servers can run STUN over TLS on the same port as STUN over TCP if the server software supports determining whether the initial message is a TLS or STUN message.

Administrators of STUN servers SHOULD use these ports in their SRV records for UDP and TCP. In all cases, the port in DNS MUST reflect the one on which the server is listening.

If no SRV records were found, the client performs an A or AAAA record lookup of the domain name. The result will be a list of IP addresses, each of which can be contacted at the default port using UDP or TCP, independent of the STUN usage. For usages that require TLS, the client connects to one of the IP addresses using the default STUN over TLS port. For usages that require DTLS, the client connects to one of the IP addresses using the default STUN over DTLS port.

9. Authentication and Message-Integrity Mechanisms

This section defines two mechanisms for STUN that a client and server can use to provide authentication and message integrity; these two mechanisms are known as the short-term credential mechanism and the long-term credential mechanism. These two mechanisms are optional, and each usage must specify if and when these mechanisms are used. Consequently, both clients and servers will know which mechanism (if any) to follow based on knowledge of which usage applies. For example, a STUN server on the public Internet supporting ICE would have no authentication, whereas the STUN server functionality in an agent supporting connectivity checks would utilize short-term credentials. An overview of these two mechanisms is given in Section 2.

Each mechanism specifies the additional processing required to use that mechanism, extending the processing specified in Section 6. The additional processing occurs in three different places: when forming a message, when receiving a message immediately after the basic checks have been performed, and when doing the detailed processing of error responses.

9.1. Short-Term Credential Mechanism

The short-term credential mechanism assumes that, prior to the STUN transaction, the client and server have used some other protocol to exchange a credential in the form of a username and password. This credential is time-limited. The time limit is defined by the usage. As an example, in the ICE usage [I-D.ietf-ice-rfc5245bis], the two endpoints use out-of-band signaling to agree on a username and password, and this username and password are applicable for the duration of the media session.

This credential is used to form a message-integrity check in each request and in many responses. There is no challenge and response as in the long-term mechanism; consequently, replay is prevented by virtue of the time-limited nature of the credential.

9.1.1. HMAC Key

For short-term credentials the HMAC key is defined as follow:

key = OpaqueString(password)

where the OpaqueString profile is defined in [RFC7613].

9.1.2. Forming a Request or Indication

For a request or indication message, the agent MUST include the USERNAME, MESSAGE-INTEGRITY-SHA256, and MESSAGE-INTEGRITY attributes in the message unless the agent knows from an external indication which message integrity algorithm is supported by both agents. In this case either MESSAGE-INTEGRITY or MESSAGE-INTEGRITY-SHA256 MUST be included in addition to USERNAME. The HMAC for the MESSAGE-INTEGRITY attribute is computed as described in Section 14.5 and the HMAC for the MESSAGE-INTEGRITY-SHA256 attributes is computed as described in Section 14.6. Note that the password is never included in the request or indication.

9.1.3. Receiving a Request or Indication

After the agent has done the basic processing of a message, the agent performs the checks listed below in order specified:

- o If the message does not contain 1) a MESSAGE-INTEGRITY or a MESSAGE-INTEGRITY-SHA256 attribute and 2) a USERNAME attribute:
  - * If the message is a request, the server MUST reject the request with an error response. This response MUST use an error code of 400 (Bad Request).
  - * If the message is an indication, the agent MUST silently discard the indication.
- o If the USERNAME does not contain a username value currently valid within the server:
  - * If the message is a request, the server MUST reject the request with an error response. This response MUST use an error code of 401 (Unauthenticated).
  - * If the message is an indication, the agent MUST silently discard the indication.
- o If the MESSAGE-INTEGRITY-SHA256 attribute is present compute the value for the message integrity as described in Section 14.6, using the password associated with the username. If the MESSAGE-INTEGRITY-SHA256 attribute is not present, and using the same password, compute the value for the message integrity as described in Section 14.5. If the resulting value does not match the contents of the corresponding attribute (MESSAGE-INTEGRITY-SHA256 or MESSAGE-INTEGRITY):

* If the message is a request, the server MUST reject the request with an error response. This response MUST use an error code of 401 (Unauthenticated).

* If the message is an indication, the agent MUST silently discard the indication.

If these checks pass, the agent continues to process the request or indication. Any response generated by a server to a request that contains a MESSAGE-INTEGRITY-SHA256 attribute MUST include the MESSAGE-INTEGRITY-SHA256 attribute, computed using the password utilized to authenticate the request. Any response generated by a server to a request that contains only a MESSAGE-INTEGRITY attribute MUST include the MESSAGE-INTEGRITY attribute, computed using the password utilized to authenticate the request. This means that only one of these attributes can appear in a response. The response MUST NOT contain the USERNAME attribute.

If any of the checks fail, a server MUST NOT include a MESSAGE-INTEGRITY-SHA256, MESSAGE-INTEGRITY, or USERNAME attribute in the error response. This is because, in these failure cases, the server cannot determine the shared secret necessary to compute the MESSAGE-INTEGRITY-SHA256 or MESSAGE-INTEGRITY attributes.

9.1.4. Receiving a Response

The client looks for the MESSAGE-INTEGRITY or the MESSAGE-INTEGRITY-SHA256 attribute in the response. If present, the client computes the message integrity over the response as defined in Section 14.5 or Section 14.6, respectively, using the same password it utilized for the request. If the resulting value matches the contents of the MESSAGE-INTEGRITY or MESSAGE-INTEGRITY-SHA256 attribute, respectively, the response is considered authenticated. If the value does not match, or if both MESSAGE-INTEGRITY and MESSAGE-INTEGRITY-SHA256 were absent, the processing depends on the request been sent over a reliable or an unreliable transport.

If the request was sent over an unreliable transport, the response MUST be discarded, as if it was never received. This means that retransmits, if applicable, will continue. If all the reponses received are discarded then instead of signalling a timeout after ending the transaction the layer MUST signal that an attack took place.

If the request was sent over a reliable transport, the response MUST be discarded and the layer MUST immediately end the transaction and signal that an attack took place.

If the client only sent only one of MESSAGE-INTEGRITY or MESSAGE-INTEGRITY-SHA256 attributes in the request (because of the external indication in section Section 9.2.3, or this being a subsequent request as defined in Section 9.1.5) the algorithm in the response has to match otherwise the response MUST be discarded.

9.1.5. Sending Subsequent Requests

A client sending subsequent requests to the same server MUST send only the MESSAGE-INTEGRITY-SHA256 or the MESSAGE-INTEGRITY attribute that matches the attribute that was received in the response to the initial request. Here same server means same IP address and port number, not just the same URI or SRV lookup result.

9.2. Long-Term Credential Mechanism

The long-term credential mechanism relies on a long-term credential, in the form of a username and password that are shared between client and server. The credential is considered long-term since it is assumed that it is provisioned for a user, and remains in effect until the user is no longer a subscriber of the system, or is changed. This is basically a traditional "log-in" username and password given to users.

Because these usernames and passwords are expected to be valid for extended periods of time, replay prevention is provided in the form of a digest challenge. In this mechanism, the client initially sends a request, without offering any credentials or any integrity checks. The server rejects this request, providing the user a realm (used to guide the user or agent in selection of a username and password) and a nonce. The nonce provides the replay protection. It is a cookie, selected by the server, and encoded in such a way as to indicate a duration of validity or client identity from which it is valid. The client retries the request, this time including its username and the realm, and echoing the nonce provided by the server. The client also includes a message-integrity, which provides an HMAC over the entire request, including the nonce. The server validates the nonce and checks the message integrity. If they match, the request is authenticated. If the nonce is no longer valid, it is considered "stale", and the server rejects the request, providing a new nonce.

In subsequent requests to the same server, the client reuses the nonce, username, realm, and password it used previously. In this way, subsequent requests are not rejected until the nonce becomes invalid by the server, in which case the rejection provides a new nonce to the client.

Note that the long-term credential mechanism cannot be used to protect indications, since indications cannot be challenged. Usages utilizing indications must either use a short-term credential or omit authentication and message integrity for them.

To indicate that it supports this specification, a server MUST prepend the NONCE attribute value with the character string composed of "obMatJos2" concatenated with the Base64 encoding of the 24 bit STUN Security Features as defined in Section 17.1. The 24 bit Security Feature set is encoded as a 24 bit integer in network order. If no security features are used, then the value 0 MUST be encoded instead. For the remainder of this document the term "nonce cookie" will refer to the complete 13 character string prepended to the NONCE attribute value.

Since the long-term credential mechanism is susceptible to offline dictionary attacks, deployments SHOULD utilize passwords that are difficult to guess. In cases where the credentials are not entered by the user, but are rather placed on a client device during device provisioning, the password SHOULD have at least 128 bits of randomness. In cases where the credentials are entered by the user, they should follow best current practices around password structure.

9.2.1. Bid Down Attack Prevention

This document introduces two new security features that provide the ability to choose the algorithm used for password protection as well as the ability to use an anonymous username. Both of these capabilities are optional in order to remain backwards compatible with previous versions of the STUN protocol.

These new capabilities are subject to bid down attacks whereby an attacker in the message path can remove these capabilities and force weaker security properties. To prevent these kinds of attacks from going undetected, the nonce is enhanced with additional information.

The value of the "nonce cookie" will vary based on the specific STUN Security Features bit values selected. When this document makes reference to the "nonce cookie" in a section discussing a specific STUN Security Feature it is understood that the corresponding STUN Security Feature bit in the "nonce cookie" is set to 1.

For example, in Section 9.2.4 discussing the PASSWORD-ALGORITHMS security feature, it is implied that the "Password algorithms" bit, as defined in Section 17.1, is set to 1 in the "nonce cookie".

9.2.2. HMAC Key

For long-term credentials that do not use a different algorithm, as specified by the PASSWORD-ALGORITHM attribute, the key is 16 bytes:

```
key = MD5(username ":" realm ":" OpaqueString(password))
```

Where MD5 is defined in [RFC1321] and the OpaqueString profile is defined in [RFC7613].

The 16-byte key is formed by taking the MD5 hash of the result of concatenating the following five fields: (1) the username, with any quotes and trailing nulls removed, as taken from the USERNAME attribute (in which case OpaqueString has already been applied); (2) a single colon; (3) the realm, with any quotes and trailing nulls removed; (4) a single colon; and (5) the password, with any trailing nulls removed and after processing using OpaqueString. For example, if the username was ’user’, the realm was ’realm’, and the password was ’pass’, then the 16-byte HMAC key would be the result of performing an MD5 hash on the string ’user:realm:pass’, the resulting hash being 0x8493fbc53ba582fb4c044c456bdc40eb.

The structure of the key when used with long-term credentials facilitates deployment in systems that also utilize SIP. Typically, SIP systems utilizing SIP’s digest authentication mechanism do not actually store the password in the database. Rather, they store a value called H(A1), which is equal to the key defined above.

When a PASSWORD-ALGORITHM is used, the key length and algorithm to use are described in Section 17.5.1.

9.2.3. Forming a Request

There are two cases when forming a request. In the first case, this is the first request from the client to the server (as identified by its IP address and port). In the second case, the client is submitting a subsequent request once a previous request/response transaction has completed successfully. Forming a request as a consequence of a 401 or 438 error response is covered in Section 9.2.5 and is not considered a "subsequent request" and thus does not utilize the rules described in Section 9.2.3.2.

The difference between a first request and a subsequent request is the presence or absence of some attributes, so omitting or including them is a MUST.

9.2.3.1. First Request

If the client has not completed a successful request/response transaction with the server (as identified by hostname, if the DNS procedures of Section 8 are used, else IP address if not), it MUST omit the USERNAME, USERHASH, MESSAGE-INTEGRITY, MESSAGE-INTEGRITY-SHA256, REALM, NONCE, PASSWORD-ALGORITHMS, and PASSWORD-ALGORITHM attributes. In other words, the very first request is sent as if there were no authentication or message integrity applied.

9.2.3.2. Subsequent Requests

Once a request/response transaction has completed successfully, the client will have been presented a realm and nonce by the server, and selected a username and password with which it authenticated. The client SHOULD cache the username, password, realm, and nonce for subsequent communications with the server. When the client sends a subsequent request, it MUST include either the USERNAME or USERHASH, REALM, NONCE, and PASSWORD-ALGORITHM attributes with these cached values. It MUST include a MESSAGE-INTEGRITY attribute or a MESSAGE-INTEGRITY-SHA256 attribute, computed as described in Section 14.5 and Section 14.6 using the cached password. The choice between the two attributes depends on the attribute received in the response to the first request.

9.2.4. Receiving a Request

After the server has done the basic processing of a request, it performs the checks listed below in the order specified:

- If the message does not contain a MESSAGE-INTEGRITY or MESSAGE-INTEGRITY-SHA256 attribute, the server MUST generate an error response with an error code of 401 (Unauthenticated). This response MUST include a REALM value. It is RECOMMENDED that the REALM value be the domain name of the provider of the STUN server. The response MUST include a NONCE, selected by the server. The server MUST ensure that the same NONCE cannot be selected for clients that use different IP addresses and/or different ports. The server MAY support alternate password algorithms, in which case it can list them in preferential order in a PASSWORD-ALGORITHMS attribute. If the server adds a PASSWORD-ALGORITHMS attribute it MUST set the STUN Security Feature "Password algorithms" bit set to 1. The server MAY support anonymous username, in which case it MUST set the STUN Security Feature "Anonymous username" bit set to 1. The response SHOULD NOT contain a USERNAME, USERHASH, MESSAGE-INTEGRITY or MESSAGE-INTEGRITY-SHA256 attribute.

Note: Sharing a NONCE is no longer permitted, so trying to share one will result in a wasted transaction.

- If the message contains a MESSAGE-INTEGRITY or a MESSAGE-INTEGRITY-SHA256 attribute, but is missing either the USERNAME or USERHASH, REALM, or NONCE attribute, the server MUST generate an error response with an error code of 400 (Bad Request). This response SHOULD NOT include a USERNAME, USERHASH, NONCE, or REALM. The response cannot contain a MESSAGE-INTEGRITY or MESSAGE-INTEGRITY-SHA256 attribute, as the attributes required to generate them are missing.

- If the NONCE attribute starts with the "nonce cookie" with the STUN Security Feature "Password algorithm" bit set to 1 but PASSWORD-ALGORITHMS does not match the value sent in the response that sent this NONCE, then the server MUST generate an error response with an error code of 400 (Bad Request).

- If the NONCE attribute starts with the "nonce cookie" with the STUN Security Feature "Password algorithm" bit set to 1 but the request contains neither PASSWORD-ALGORITHMS nor PASSWORD-ALGORITHM, then the request is processed as though PASSWORD-ALGORITHM were MD5 (Note that if the original PASSWORD-ALGORITHMS attribute did not contain MD5, this will result in a 400 Bad Request in a later step below).

- If the NONCE attribute starts with the "nonce cookie" with the STUN Security Feature "Password algorithm" bit set to 1 but only one of PASSWORD-ALGORITHM or PASSWORD-ALGORITHMS is present, then the server MUST generate an error response with an error code of 400 (Bad Request).

- If the NONCE attribute starts with the "nonce cookie" with the STUN Security Feature "Password algorithm" bit set to 1 but PASSWORD-ALGORITHM does not match one of the entries in PASSWORD-ALGORITHMS, then the server MUST generate an error response with an error code of 400 (Bad Request).

- If the NONCE is no longer valid and at the same time the MESSAGE-INTEGRITY or a MESSAGE-INTEGRITY-SHA256 attribute is invalid, the server MUST generate an error response with an error code of 401. This response MUST include NONCE, REALM, and PASSWORD-ALGORITHMS attributes and SHOULD NOT include the USERNAME or USERHASH attribute. The response MAY include a MESSAGE-INTEGRITY or MESSAGE-INTEGRITY-SHA256 attribute, using the previous NONCE to calculate it.

- o If the NONCE is no longer valid, the server MUST generate an error response with an error code of 438 (Stale Nonce). This response MUST include NONCE, REALM, and PASSWORD-ALGORITHMS attributes and SHOULD NOT include the USERNAME, USERHASH attribute, The response MAY include a MESSAGE-INTEGRITY or MESSAGE-INTEGRITY-SHA256 attribute, using the previous NONCE to calculate it. Servers can invalidate nonces in order to provide additional security. See Section 4.3 of [RFC2617] for guidelines.

- o If the username in the USERNAME or USERHASH attribute is not valid, the server MUST generate an error response with an error code of 401 (Unauthenticated). This response MUST include a REALM value. It is RECOMMENDED that the REALM value be the domain name of the provider of the STUN server. The response MUST include a NONCE, selected by the server. The response MUST include a PASSWORD-ALGORITHMS attribute. The response SHOULD NOT contain a USERNAME, USERHASH attribute. The response MAY include a MESSAGE-INTEGRITY or MESSAGE-INTEGRITY-SHA256 attribute, using the previous password to calculate it.

- o If the MESSAGE-INTEGRITY-SHA256 attribute is present compute the value for the message integrity as described in Section 14.6, using the password associated with the username. Else, using the same password, compute the value for the message integrity as described in Section 14.5. If the resulting value does not match the contents of the MESSAGE-INTEGRITY attribute or the MESSAGE-INTEGRITY-SHA256 attribute, the server MUST reject the request with an error response. This response MUST use an error code of 401 (Unauthenticated). It MUST include REALM and NONCE attributes and SHOULD NOT include the USERNAME, USERHASH, MESSAGE-INTEGRITY, or MESSAGE-INTEGRITY-SHA256 attribute.

If these checks pass, the server continues to process the request. Any response generated by the server MUST include MESSAGE-INTEGRITY-SHA256 attribute, computed using the username and password utilized to authenticate the request, unless the request was processed as though PASSWORD-ALGORITHM was MD5 (because the request contained neither PASSWORD-ALGORITHMS nor PASSWORD-ALGORITHM). In that case the MESSAGE-INTEGRITY attribute MUST be used instead of the MESSAGE-INTEGRITY-SHA256 attribute. The REALM, NONCE, USERNAME and USERHASH attributes SHOULD NOT be included.

9.2.5. Receiving a Response

If the response is an error response with an error code of 401 (Unauthenticated) or 438 (Stale Nonce), the client MUST test if the NONCE attribute value starts with the "nonce cookie". If the test succeeds and the "nonce cookie" has the STUN Security Feature

"Password algorithm" bit set to 1 but no PASSWORD-ALGORITHMS attribute is present, then the client MUST NOT retry the request with a new transaction. If the test succeeds and the "nonce cookie" has the STUN Security Feature "Username anonymity" bit set to 1 but no USERHASH attribute is present, then the client MUST NOT retry the request with a new transaction.

If the response is an error response with an error code of 401 (Unauthenticated), the client SHOULD retry the request with a new transaction. This request MUST contain a USERNAME or a USERHASH, determined by the client as the appropriate username for the REALM from the error response. If the "nonce cookie" was present and had the STUN Security Feature "Username anonymity" bit set to 1 then the USERHASH attribute MUST be used, else the USERNAME attribute MUST be used. The request MUST contain the REALM, copied from the error response. The request MUST contain the NONCE, copied from the error response. If the response contains a PASSWORD-ALGORITHMS attribute, the request MUST contain the PASSWORD-ALGORITHMS attribute with the same content. If the response contains a PASSWORD-ALGORITHMS attribute, and this attribute contains at least one algorithm that is supported by the client then the request MUST contain a PASSWORD-ALGORITHM attribute with the first algorithm supported on the list. If the response contains a PASSWORD-ALGORITHMS attribute, and this attribute does not contain any algorithm that is supported by the client, then the client MUST NOT retry the request with a new transaction. The client MUST NOT perform this retry if it is not changing the USERNAME or USERHASH or REALM or its associated password, from the previous attempt.

If the response is an error response with an error code of 438 (Stale Nonce), the client MUST retry the request, using the new NONCE attribute supplied in the 438 (Stale Nonce) response. This retry MUST also include either the USERNAME or USERHASH, REALM and either the MESSAGE-INTEGRITY or MESSAGE-INTEGRITY-SHA256 attributes.

For all other responses, if the NONCE attribute starts with the "nonce cookie" with the STUN Security Feature "Password algorithm" bit set to 1 but PASSWORD-ALGORITHMS is not present, the response MUST be ignored. For all other responses, if the NONCE attribute starts with the "nonce cookie" with the STUN Security Feature "User anonymity" bit set to 1 but USERHASH is not present, the response MUST be ignored.

If the response is an error response with an error code of 400, and does not contains either MESSAGE-INTEGRITY or MESSAGE-INTEGRITY-SHA256 attribute then the response MUST be discarded, as if it was never received. This means that retransmits, if applicable, will continue.

The client looks for the MESSAGE-INTEGRITY or MESSAGE-INTEGRITY-SHA256 attribute in the response (either success or failure). If present, the client computes the message integrity over the response as defined in Section 14.5 or Section 14.6, using the same password it utilized for the request. If the resulting value matches the contents of the MESSAGE-INTEGRITY or MESSAGE-INTEGRITY-SHA256 attribute, the response is considered authenticated. If the value does not match, or if both MESSAGE-INTEGRITY and MESSAGE-INTEGRITY-SHA256 were absent, the processing depends on the request been sent over a reliable or an unreliable transport.

If the request was sent over an unreliable transport, the response MUST be discarded, as if it was never received. This means that retransmits, if applicable, will continue. If all the reponses received are discarded then instead of signalling a timeout after ending the transaction the layer MUST signal that an attack took place.

If the request was sent over a reliable transport, the response MUST be discarded and the layer MUST immediately end the transaction and signal that an attack took place.

If the response contains a PASSWORD-ALGORITHMS attribute, the subsequent request MUST be authenticated using MESSAGE-INTEGRITY-SHA256 only.

10. ALTERNATE-SERVER Mechanism

This section describes a mechanism in STUN that allows a server to redirect a client to another server. This extension is optional, and a usage must define if and when this extension is used.

A server using this extension redirects a client to another server by replying to a request message with an error response message with an error code of 300 (Try Alternate). The server MUST include an ALTERNATE-SERVER attribute in the error response. The error response message MAY be authenticated; however, there are uses cases for ALTERNATE-SERVER where authentication of the response is not possible or practical. If the transaction uses TLS or DTLS and if the transaction is authenticated by a MESSAGE-INTEGRITY-SHA256 attribute and if the server wants to redirect to a server that uses a different certificate, then it MUST include an ALTERNATE-DOMAIN attribute containing the subjectAltName of that certificate.

A client using this extension handles a 300 (Try Alternate) error code as follows. The client looks for an ALTERNATE-SERVER attribute in the error response. If one is found, then the client considers the current transaction as failed, and reattempts the request with

the server specified in the attribute, using the same transport protocol used for the previous request. That request, if authenticated, MUST utilize the same credentials that the client would have used in the request to the server that performed the redirection. If the transport protocol uses TLS or DTLS, then the client looks for an ALTERNATE-DOMAIN attribute. If the attribute is found, the domain MUST be used to validate the certificate using the recommendations in [RFC6125]. If the attribute is not found, the same domain that was used for the original request MUST be used to validate the certificate. If the client has been redirected to a server on which it has already tried this request within the last five minutes, it MUST ignore the redirection and consider the transaction to have failed. This prevents infinite ping-ponging between servers in case of redirection loops.

11. Backwards Compatibility with RFC 3489

In addition to the backward compatibility already described in Section 12 of [RFC5389], DTLS MUST NOT be used with STUN [RFC3489] (also referred to as "classic STUN"). Any STUN request or indication without the magic cookie (see Section 6 of [RFC5389]) over DTLS MUST always result in an error.

12. Basic Server Behavior

This section defines the behavior of a basic, stand-alone STUN server. A basic STUN server provides clients with server reflexive transport addresses by receiving and replying to STUN Binding requests.

The STUN server MUST support the Binding method. It SHOULD NOT utilize the short-term or long-term credential mechanism. This is because the work involved in authenticating the request is more than the work in simply processing it. It SHOULD NOT utilize the ALTERNATE-SERVER mechanism for the same reason. It MUST support UDP and TCP. It MAY support STUN over TCP/TLS or STUN over UDP/DTLS; however, DTLS and TLS provide minimal security benefits in this basic mode of operation. It MAY utilize the FINGERPRINT mechanism but MUST NOT require it. Since the stand-alone server only runs STUN, FINGERPRINT provides no benefit. Requiring it would break compatibility with RFC 3489, and such compatibility is desirable in a stand-alone server. Stand-alone STUN servers SHOULD support backwards compatibility with [RFC3489] clients, as described in Section 11.

It is RECOMMENDED that administrators of STUN servers provide DNS entries for those servers as described in Section 8.

A basic STUN server is not a solution for NAT traversal by itself. However, it can be utilized as part of a solution through STUN usages. This is discussed further in Section 13.

13. STUN Usages

STUN by itself is not a solution to the NAT traversal problem. Rather, STUN defines a tool that can be used inside a larger solution. The term "STUN usage" is used for any solution that uses STUN as a component.

A STUN usage defines how STUN is actually utilized -- when to send requests, what to do with the responses, and which optional procedures defined here (or in an extension to STUN) are to be used. A usage would also define:

- Which STUN methods are used.
- What transports are used. If DTLS-over-UDP is used then implementing the denial-of-service countermeasure described in Section 4.2.1 of [RFC6347] is mandatory.
- What authentication and message-integrity mechanisms are used.
- The considerations around manual vs. automatic key derivation for the integrity mechanism, as discussed in [RFC4107].
- What mechanisms are used to distinguish STUN messages from other messages. When STUN is run over TCP, a framing mechanism may be required.
- How a STUN client determines the IP address and port of the STUN server.
- Whether backwards compatibility to RFC 3489 is required.
- What optional attributes defined here (such as FINGERPRINT and ALTERNATE-SERVER) or in other extensions are required.
- If MESSAGE-INTEGRITY-256 truncation is permitted, and the limits permitted for truncation.

In addition, any STUN usage must consider the security implications of using STUN in that usage. A number of attacks against STUN are known (see the Security Considerations section in this document), and any usage must consider how these attacks can be thwarted or mitigated.

Finally, a usage must consider whether its usage of STUN is an example of the Unilateral Self-Address Fixing approach to NAT traversal, and if so, address the questions raised in RFC 3424 [RFC3424].

14. STUN Attributes

After the STUN header are zero or more attributes. Each attribute MUST be TLV encoded, with a 16-bit type, 16-bit length, and value. Each STUN attribute MUST end on a 32-bit boundary. As mentioned above, all fields in an attribute are transmitted most significant bit first.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Type                  |            Length             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Value (variable)                ....
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4: Format of STUN Attributes

The value in the length field MUST contain the length of the Value part of the attribute, prior to padding, measured in bytes. Since STUN aligns attributes on 32-bit boundaries, attributes whose content is not a multiple of 4 bytes are padded with 1, 2, or 3 bytes of padding so that its value contains a multiple of 4 bytes. The padding bits are ignored, and may be any value.

Any attribute type MAY appear more than once in a STUN message. Unless specified otherwise, the order of appearance is significant: only the first occurrence needs to be processed by a receiver, and any duplicates MAY be ignored by a receiver.

To allow future revisions of this specification to add new attributes if needed, the attribute space is divided into two ranges. Attributes with type values between 0x0000 and 0x7FFF are comprehension-required attributes, which means that the STUN agent cannot successfully process the message unless it understands the attribute. Attributes with type values between 0x8000 and 0xFFFF are comprehension-optional attributes, which means that those attributes can be ignored by the STUN agent if it does not understand them.

The set of STUN attribute types is maintained by IANA. The initial set defined by this specification is found in Section 17.3.

The rest of this section describes the format of the various attributes defined in this specification.

14.1. MAPPED-ADDRESS

The MAPPED-ADDRESS attribute indicates a reflexive transport address of the client. It consists of an 8-bit address family and a 16-bit port, followed by a fixed-length value representing the IP address. If the address family is IPv4, the address MUST be 32 bits. If the address family is IPv6, the address MUST be 128 bits. All fields must be in network byte order.

The format of the MAPPED-ADDRESS attribute is:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 0 0|    Family     |           Port                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
|                 Address (32 bits or 128 bits)                 |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 5: Format of MAPPED-ADDRESS Attribute

The address family can take on the following values:

0x01:IPv4
0x02:IPv6

The first 8 bits of the MAPPED-ADDRESS MUST be set to 0 and MUST be ignored by receivers. These bits are present for aligning parameters on natural 32-bit boundaries.

This attribute is used only by servers for achieving backwards compatibility with [RFC3489] clients.

14.2. XOR-MAPPED-ADDRESS

The XOR-MAPPED-ADDRESS attribute is identical to the MAPPED-ADDRESS attribute, except that the reflexive transport address is obfuscated through the XOR function.

The format of the XOR-MAPPED-ADDRESS is:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 0 0|    Family     |         X-Port                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                X-Address (Variable)
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 6: Format of XOR-MAPPED-ADDRESS Attribute

The Family represents the IP address family, and is encoded identically to the Family in MAPPED-ADDRESS.

X-Port is computed by taking the mapped port in host byte order, XOR'ing it with the most significant 16 bits of the magic cookie, and then the converting the result to network byte order. If the IP address family is IPv4, X-Address is computed by taking the mapped IP address in host byte order, XOR'ing it with the magic cookie, and converting the result to network byte order. If the IP address family is IPv6, X-Address is computed by taking the mapped IP address in host byte order, XOR'ing it with the concatenation of the magic cookie and the 96-bit transaction ID, and converting the result to network byte order.

The rules for encoding and processing the first 8 bits of the attribute's value, the rules for handling multiple occurrences of the attribute, and the rules for processing address families are the same as for MAPPED-ADDRESS.

Note: XOR-MAPPED-ADDRESS and MAPPED-ADDRESS differ only in their encoding of the transport address. The former encodes the transport address by exclusive-or'ing it with the magic cookie. The latter encodes it directly in binary. RFC 3489 originally specified only MAPPED-ADDRESS. However, deployment experience found that some NATs rewrite the 32-bit binary payloads containing the NAT's public IP address, such as STUN's MAPPED-ADDRESS attribute, in the well-meaning but misguided attempt at providing a generic ALG function. Such behavior interferes with the operation of STUN and also causes failure of STUN's message-integrity checking.

14.3. USERNAME

The USERNAME attribute is used for message integrity. It identifies the username and password combination used in the message-integrity check.

The value of USERNAME is a variable-length value. It MUST contain a UTF-8 [RFC3629] encoded sequence of less than 513 bytes, and MUST have been processed using the OpaqueString profile [RFC7613].

14.4. USERHASH

The USERHASH attribute is used as a replacement for the USERNAME attribute when username anonymity is supported.

The value of USERHASH has a fixed length of 32 bytes. The username MUST have been processed using the OpaqueString profile [RFC7613] before hashing.

The following is the operation that the client will perform to hash the username:

```
userhash = SHA256(username ":" realm)
```

14.5. MESSAGE-INTEGRITY

The MESSAGE-INTEGRITY attribute contains an HMAC-SHA1 [RFC2104] of the STUN message. The MESSAGE-INTEGRITY attribute can be present in any STUN message type. Since it uses the SHA1 hash, the HMAC will be at 20 bytes.

The text used as input to HMAC is the STUN message, including the header, up to and including the attribute preceding the MESSAGE-INTEGRITY attribute. With the exception of the MESSAGE-INTEGRITY-SHA256 and FINGERPRINT attributes, which appear after MESSAGE-INTEGRITY, agents MUST ignore all other attributes that follow MESSAGE-INTEGRITY.

The key for the HMAC depends on which credential mechanism is in use. Section 9.1.1 defines the key for the short-term credential mechanism and Section 9.2.2 defines the key for the long-term credential mechanism. Other credential mechanisms MUST define the key that is used for the HMAC.

Based on the rules above, the hash used to construct MESSAGE-INTEGRITY includes the length field from the STUN message header. Prior to performing the hash, the MESSAGE-INTEGRITY attribute MUST be inserted into the message (with dummy content). The length MUST then be set to point to the length of the message up to, and including, the MESSAGE-INTEGRITY attribute itself, but excluding any attributes after it. Once the computation is performed, the value of the MESSAGE-INTEGRITY attribute can be filled in, and the value of the length in the STUN header can be set to its correct value -- the length of the entire message. Similarly, when validating the

MESSAGE-INTEGRITY, the length field should be adjusted to point to the end of the MESSAGE-INTEGRITY attribute prior to calculating the HMAC. Such adjustment is necessary when attributes, such as FINGERPRINT, appear after MESSAGE-INTEGRITY.

14.6. MESSAGE-INTEGRITY-SHA256

The MESSAGE-INTEGRITY-SHA256 attribute contains an HMAC-SHA-256 [RFC2104] of the STUN message. The MESSAGE-INTEGRITY-SHA256 attribute can be present in any STUN message type. Since it uses the SHA256 hash, the HMAC will be at most 32 bytes. The HMAC MUST NOT be truncated below a minimum size of 16 bytes. If truncation is employed then the HMAC size MUST be a multiple of 4. Truncation MUST be done by stripping off the final bytes. STUN Usages can define their own truncation limits, as long as they adhere to the guidelines specificed above. STUN Usages that do not define truncation limits MUST NOT use truncation at all.

The text used as input to HMAC is the STUN message, including the header, up to and including the attribute preceding the MESSAGE-INTEGRITY-SHA256 attribute. With the exception of the FINGERPRINT attribute, which appears after MESSAGE-INTEGRITY-SHA256, agents MUST ignore all other attributes that follow MESSAGE-INTEGRITY-SHA256.

The key for the HMAC depends on which credential mechanism is in use. Section 9.1.1 defines the key for the short-term credential mechanism and Section 9.2.2 defines the key for the long-term credential mechanism. Other credential mechanism MUST define the key that is used for the HMAC.

Based on the rules above, the hash used to construct MESSAGE-INTEGRITY-SHA256 includes the length field from the STUN message header. Prior to performing the hash, the MESSAGE-INTEGRITY-SHA256 attribute MUST be inserted into the message (with dummy content). The length MUST then be set to point to the length of the message up to, and including, the MESSAGE-INTEGRITY-SHA256 attribute itself, but excluding any attributes after it. Once the computation is performed, the value of the MESSAGE-INTEGRITY-SHA256 attribute can be filled in, and the value of the length in the STUN header can be set to its correct value -- the length of the entire message. Similarly, when validating the MESSAGE-INTEGRITY-SHA256, the length field should be adjusted to point to the end of the MESSAGE-INTEGRITY-SHA256 attribute prior to calculating the HMAC. Such adjustment is necessary when attributes, such as FINGERPRINT, appear after MESSAGE-INTEGRITY-SHA256.

14.7. FINGERPRINT

The FINGERPRINT attribute MAY be present in all STUN messages. The value of the attribute is computed as the CRC-32 of the STUN message up to (but excluding) the FINGERPRINT attribute itself, XOR'ed with the 32-bit value 0x5354554e (the XOR helps in cases where an application packet is also using CRC-32 in it). The 32-bit CRC is the one defined in ITU V.42 [ITU.V42.2002], which has a generator polynomial of x32+x26+x23+x22+x16+x12+x11+x10+x8+x7+x5+x4+x2+x+1. See the sample code for the CRC-32 in Section 8 of [RFC1952].

When present, the FINGERPRINT attribute MUST be the last attribute in the message, and thus will appear after MESSAGE-INTEGRITY.

The FINGERPRINT attribute can aid in distinguishing STUN packets from packets of other protocols. See Section 7.

As with MESSAGE-INTEGRITY, the CRC used in the FINGERPRINT attribute covers the length field from the STUN message header. Therefore, this value must be correct and include the CRC attribute as part of the message length, prior to computation of the CRC. When using the FINGERPRINT attribute in a message, the attribute is first placed into the message with a dummy value, then the CRC is computed, and then the value of the attribute is updated. If the MESSAGE-INTEGRITY attribute is also present, then it must be present with the correct message-integrity value before the CRC is computed, since the CRC is done over the value of the MESSAGE-INTEGRITY attribute as well.

14.8. ERROR-CODE

The ERROR-CODE attribute is used in error response messages. It contains a numeric error code value in the range of 300 to 699 plus a textual reason phrase encoded in UTF-8 [RFC3629], and is consistent in its code assignments and semantics with SIP [RFC3261] and HTTP [RFC2616]. The reason phrase is meant for user consumption, and can be anything appropriate for the error code. Recommended reason phrases for the defined error codes are included in the IANA registry for error codes. The reason phrase MUST be a UTF-8 [RFC3629] encoded sequence of less than 128 characters (which can be as long as 763 bytes).

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Reserved, should be 0         |Class|     Number    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Reason Phrase (variable)                                ..
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 7: ERROR-CODE Attribute

To facilitate processing, the class of the error code (the hundreds digit) is encoded separately from the rest of the code, as shown in Figure 7.

The Reserved bits SHOULD be 0, and are for alignment on 32-bit boundaries. Receivers MUST ignore these bits. The Class represents the hundreds digit of the error code. The value MUST be between 3 and 6. The Number represents the error code modulo 100, and its value MUST be between 0 and 99.

The following error codes, along with their recommended reason phrases, are defined:

300 Try Alternate: The client should contact an alternate server for this request. This error response MUST only be sent if the request included either a USERNAME or USERHASH attribute and a valid MESSAGE-INTEGRITY attribute; otherwise, it MUST NOT be sent and error code 400 (Bad Request) is suggested. This error response MUST be protected with the MESSAGE-INTEGRITY attribute, and receivers MUST validate the MESSAGE-INTEGRITY of this response before redirecting themselves to an alternate server.

Note: Failure to generate and validate message integrity for a 300 response allows an on-path attacker to falsify a 300 response thus causing subsequent STUN messages to be sent to a victim.

400 Bad Request: The request was malformed. The client SHOULD NOT retry the request without modification from the previous attempt. The server may not be able to generate a valid MESSAGE-INTEGRITY for this error, so the client MUST NOT expect a valid MESSAGE-INTEGRITY attribute on this response.

401 Unauthenticated: The request did not contain the correct credentials to proceed. The client should retry the request with proper credentials.

420 Unknown Attribute: The server received a STUN packet containing a comprehension-required attribute that it did not understand.

The server MUST put this unknown attribute in the UNKNOWN-ATTRIBUTE attribute of its error response.

438 Stale Nonce: The NONCE used by the client was no longer valid. The client should retry, using the NONCE provided in the response.

500 Server Error: The server has suffered a temporary error. The client should try again.

14.9. REALM

The REALM attribute may be present in requests and responses. It contains text that meets the grammar for "realm-value" as described in [RFC3261] but without the double quotes and their surrounding whitespace. That is, it is an unquoted realm-value (and is therefore a sequence of qdtext or quoted-pair). It MUST be a UTF-8 [RFC3629] encoded sequence of less than 128 characters (which can be as long as 763 bytes), and MUST have been processed using the OpaqueString profile [RFC7613].

Presence of the REALM attribute in a request indicates that long-term credentials are being used for authentication. Presence in certain error responses indicates that the server wishes the client to use a long-term credential for authentication.

14.10. NONCE

The NONCE attribute may be present in requests and responses. It contains a sequence of qdtext or quoted-pair, which are defined in RFC 3261 [RFC3261]. Note that this means that the NONCE attribute will not contain actual quote characters. See [RFC2617], Section 4.3, for guidance on selection of nonce values in a server. It MUST be less than 128 characters (which can be as long as 763 bytes).

14.11. PASSWORD-ALGORITHMS

The PASSWORD-ALGORITHMS attribute may be present in requests and responses. It contains the list of algorithms that the server can use to derive the long-term password.

The set of known algorithms is maintained by IANA. The initial set defined by this specification is found in Section 17.5.

The attribute contains a list of algorithm numbers and variable length parameters. The algorithm number is a 16-bit value as defined in Section 17.5. The parameters start with the actual length of the parameters as a 16-bit value, followed by the parameters that are

specific to each algorithm. The parameters are padded to a 32-bit boundary, in the same manner as an attribute.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Algorithm 1           | Algorithm 1 Parameters Length |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Algorithm 1 Parameters (variable)
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Algorithm 2           | Algorithm 2 Parameters Length |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Algorithm 2 Parameter (variable)
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               ...
```

Figure 8: Format of PASSWORD-ALGORITHMS Attribute

14.12. PASSWORD-ALGORITHM

The PASSWORD-ALGORITHM attribute is present only in requests. It contains the algorithms that the server must use to derive the long-term password.

The set of known algorithms is maintained by IANA. The initial set defined by this specification is found in Section 17.5.

The attribute contains an algorithm number and variable length parameters. The algorithm number is a 16-bit value as defined in Section 17.5. The parameters starts with the actual length of the parameters as a 16-bit value, followed by the parameters that are specific to the algorithm. The parameters are padded to a 32-bit boundary, in the same manner as an attribute.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Algorithm            |  Algorithm Parameters Length  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Algorithm Parameters (variable)
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 9: Format of PASSWORD-ALGORITHM Attribute

14.13. UNKNOWN-ATTRIBUTES

The UNKNOWN-ATTRIBUTES attribute is present only in an error response when the response code in the ERROR-CODE attribute is 420.

The attribute contains a list of 16-bit values, each of which represents an attribute type that was not understood by the server.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Attribute 1 Type         |       Attribute 2 Type        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Attribute 3 Type         |       Attribute 4 Type    ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 10: Format of UNKNOWN-ATTRIBUTES Attribute

Note: In [RFC3489], this field was padded to 32 by duplicating the last attribute. In this version of the specification, the normal padding rules for attributes are used instead.

14.14. SOFTWARE

The SOFTWARE attribute contains a textual description of the software being used by the agent sending the message. It is used by clients and servers. Its value SHOULD include manufacturer and version number. The attribute has no impact on operation of the protocol, and serves only as a tool for diagnostic and debugging purposes. The value of SOFTWARE is variable length. It MUST be a UTF-8 [RFC3629] encoded sequence of less than 128 characters (which can be as long as 763 bytes).

14.15. ALTERNATE-SERVER

The alternate server represents an alternate transport address identifying a different STUN server that the STUN client should try.

It is encoded in the same way as MAPPED-ADDRESS, and thus refers to a single server by IP address. The IP address family MUST be identical to that of the source IP address of the request.

14.16. ALTERNATE-DOMAIN

The alternate domain represents the domain name that is used to verify the IP address in the ALTERNATE-SERVER attribute when the transport protocol uses TLS or DTLS.

The value of ALTERNATE-DOMAIN is variable length. It MUST be a UTF-8 [RFC3629] encoded sequence of less than 128 characters (which can be as long as 763 bytes).

15. Security Considerations

15.1. Attacks against the Protocol

15.1.1. Outside Attacks

An attacker can try to modify STUN messages in transit, in order to cause a failure in STUN operation. These attacks are detected for both requests and responses through the message-integrity mechanism, using either a short-term or long-term credential. Of course, once detected, the manipulated packets will be dropped, causing the STUN transaction to effectively fail. This attack is possible only by an on-path attacker.

An attacker that can observe, but not modify, STUN messages in-transit (for example, an attacker present on a shared access medium, such as Wi-Fi), can see a STUN request, and then immediately send a STUN response, typically an error response, in order to disrupt STUN processing. This attack is also prevented for messages that utilize MESSAGE-INTEGRITY. However, some error responses, those related to authentication in particular, cannot be protected by MESSAGE-INTEGRITY. When STUN itself is run over a secure transport protocol (e.g., TLS), these attacks are completely mitigated.

Depending on the STUN usage, these attacks may be of minimal consequence and thus do not require message integrity to mitigate. For example, when STUN is used to a basic STUN server to discover a server reflexive candidate for usage with ICE, authentication and message integrity are not required since these attacks are detected during the connectivity check phase. The connectivity checks themselves, however, require protection for proper operation of ICE overall. As described in Section 13, STUN usages describe when authentication and message integrity are needed.

Since STUN uses the HMAC of a shared secret for authentication and integrity protection, it is subject to offline dictionary attacks. When authentication is utilized, it SHOULD be with a strong password that is not readily subject to offline dictionary attacks. Protection of the channel itself, using TLS or DTLS, mitigates these attacks.

STUN supports both MESSAGE-INTEGRITY and MESSAGE-INTEGRITY-SHA256, which is subject to bid down attacks by an on-path attacker. Protection of the channel itself, using TLS or DTLS, mitigates these

attacks. Timely removal of the support of MESSAGE-INTEGRITY in a future version of STUN is necessary.

15.1.2. Inside Attacks

A rogue client may try to launch a DoS attack against a server by sending it a large number of STUN requests. Fortunately, STUN requests can be processed statelessly by a server, making such attacks hard to launch.

A rogue client may use a STUN server as a reflector, sending it requests with a falsified source IP address and port. In such a case, the response would be delivered to that source IP and port. There is no amplification of the number of packets with this attack (the STUN server sends one packet for each packet sent by the client), though there is a small increase in the amount of data, since STUN responses are typically larger than requests. This attack is mitigated by ingress source address filtering.

Revealing the specific software version of the agent through the SOFTWARE attribute might allow them to become more vulnerable to attacks against software that is known to contain security holes. Implementers SHOULD make usage of the SOFTWARE attribute a configurable option.

15.2. Attacks Affecting the Usage

This section lists attacks that might be launched against a usage of STUN. Each STUN usage must consider whether these attacks are applicable to it, and if so, discuss counter-measures.

Most of the attacks in this section revolve around an attacker modifying the reflexive address learned by a STUN client through a Binding request/response transaction. Since the usage of the reflexive address is a function of the usage, the applicability and remediation of these attacks are usage-specific. In common situations, modification of the reflexive address by an on-path attacker is easy to do. Consider, for example, the common situation where STUN is run directly over UDP. In this case, an on-path attacker can modify the source IP address of the Binding request before it arrives at the STUN server. The STUN server will then return this IP address in the XOR-MAPPED-ADDRESS attribute to the client, and send the response back to that (falsified) IP address and port. If the attacker can also intercept this response, it can direct it back towards the client. Protecting against this attack by using a message-integrity check is impossible, since a message-integrity value cannot cover the source IP address, since the intervening NAT must be able to modify this value. Instead, one

solution to preventing the attacks listed below is for the client to verify the reflexive address learned, as is done in ICE [I-D.ietf-ice-rfc5245bis]. Other usages may use other means to prevent these attacks.

15.2.1. Attack I: Distributed DoS (DDoS) against a Target

In this attack, the attacker provides one or more clients with the same faked reflexive address that points to the intended target. This will trick the STUN clients into thinking that their reflexive addresses are equal to that of the target. If the clients hand out that reflexive address in order to receive traffic on it (for example, in SIP messages), the traffic will instead be sent to the target. This attack can provide substantial amplification, especially when used with clients that are using STUN to enable multimedia applications. However, it can only be launched against targets for which packets from the STUN server to the target pass through the attacker, limiting the cases in which it is possible.

15.2.2. Attack II: Silencing a Client

In this attack, the attacker provides a STUN client with a faked reflexive address. The reflexive address it provides is a transport address that routes to nowhere. As a result, the client won’t receive any of the packets it expects to receive when it hands out the reflexive address. This exploitation is not very interesting for the attacker. It impacts a single client, which is frequently not the desired target. Moreover, any attacker that can mount the attack could also deny service to the client by other means, such as preventing the client from receiving any response from the STUN server, or even a DHCP server. As with the attack in Section 15.2.1, this attack is only possible when the attacker is on path for packets sent from the STUN server towards this unused IP address.

15.2.3. Attack III: Assuming the Identity of a Client

This attack is similar to attack II. However, the faked reflexive address points to the attacker itself. This allows the attacker to receive traffic that was destined for the client.

15.2.4. Attack IV: Eavesdropping

In this attack, the attacker forces the client to use a reflexive address that routes to itself. It then forwards any packets it receives to the client. This attack would allow the attacker to observe all packets sent to the client. However, in order to launch the attack, the attacker must have already been able to observe packets from the client to the STUN server. In most cases (such as

when the attack is launched from an access network), this means that the attacker could already observe packets sent to the client. This attack is, as a result, only useful for observing traffic by attackers on the path from the client to the STUN server, but not generally on the path of packets being routed towards the client.

15.3. Hash Agility Plan

This specification uses both HMAC-SHA-1 and HMAC-SHA-256 for computation of the message integrity. If, at a later time, HMAC-SHA-256 is found to be compromised, the following is the remedy that will be applied.

We will define a STUN extension that introduces a new message-integrity attribute, computed using a new hash. Clients would be required to include both the new and old message-integrity attributes in their requests or indications. A new server will utilize the new message-integrity attribute, and an old one, the old. After a transition period where mixed implementations are in deployment, the old message-integrity attribute will be deprecated by another specification, and clients will cease including it in requests.

After a transition period, a new document updating this document will remove the usage of HMAC-SHA-1 for computation of the message-integrity.

16. IAB Considerations

The IAB has studied the problem of Unilateral Self-Address Fixing (UNSAF), which is the general process by which a client attempts to determine its address in another realm on the other side of a NAT through a collaborative protocol reflection mechanism ([RFC3424]). STUN can be used to perform this function using a Binding request/response transaction if one agent is behind a NAT and the other is on the public side of the NAT.

The IAB has suggested that protocols developed for this purpose document a specific set of considerations. Because some STUN usages provide UNSAF functions (such as ICE [I-D.ietf-ice-rfc5245bis]), and others do not (such as SIP Outbound [RFC5626]), answers to these considerations need to be addressed by the usages themselves.

17. IANA Considerations

17.1. STUN Security Features Registry

A STUN Security Feature set is a 24 bit value.

IANA is requested to create a new registry containing the STUN Security Features that are protected by the bid down attack prevention mechanism described in section Section 9.2.1.

The initial STUN Security Features are:

0x000001: Password algorithms
0x000002: Username anonymity

New Security Features are assigned by a Standard Action [RFC5226].

17.2. STUN Methods Registry

IANA is requested to update the reference from RFC 5389 to RFC-to-be for the following STUN methods:

0x000: (Reserved)
0x001: Binding
0x002: (Reserved; was SharedSecret)

17.3. STUN Attribute Registry

17.3.1. Updated Attributes

IANA is requested to update the reference from RFC 5389 to RFC-to-be for the following STUN methods:

```
Comprehension-required range (0x0000-0x7FFF):
0x0000: (Reserved)
0x0001: MAPPED-ADDRESS
0x0002: (Reserved; was RESPONSE-ADDRESS)
0x0003: (Reserved; was CHANGE-REQUEST)
0x0004: (Reserved; was SOURCE-ADDRESS)
0x0005: (Reserved; was CHANGED-ADDRESS)
0x0006: USERNAME
0x0007: (Reserved; was PASSWORD)
0x0008: MESSAGE-INTEGRITY
0x0009: ERROR-CODE
0x000A: UNKNOWN-ATTRIBUTES
0x000B: (Reserved; was REFLECTED-FROM)
0x0014: REALM
0x0015: NONCE
0x0020: XOR-MAPPED-ADDRESS

Comprehension-optional range (0x8000-0xFFFF)
0x8022: SOFTWARE
0x8023: ALTERNATE-SERVER
0x8028: FINGERPRINT
```

17.3.2. New Attributes

IANA is requested to add the following attribute to the STUN Attribute Registry:

```
Comprehension-required range (0x0000-0x7FFF):
0xXXXX: MESSAGE-INTEGRITY-SHA256
0xXXXX: PASSWORD-ALGORITHM
0xXXXX: USERHASH

Comprehension-optional range (0x8000-0xFFFF)
0xXXXX: PASSSORD-ALGORITHMS
0xXXXX: ALTERNATE-DOMAIN
```

17.4. STUN Error Code Registry

IANA is requested to update the reference from RFC 5389 to RFC-to-be for the Error Codes given in Section 14.8.

17.5. Password Algorithm Registry

IANA is requested to create a new registry for Password Algorithm.

A Password Algorithm is a hex number in the range 0x0000 - 0xFFFF.

The initial Password Algorithms are:

0x0001: MD5
0x0002: SHA256

Password Algorithms in the first half of the range (0x0000 - 0x7FFF) are assigned by IETF Review [RFC5226]. Password Algorithms in the second half of the range (0x8000 - 0xFFFF) are assigned by Designated Expert [RFC5226].

17.5.1. Password Algorithms

17.5.1.1. MD5

This password algorithm is taken from [RFC1321].

The key length is 20 bytes and the parameters value is empty.

Note: This algorithm MUST only be used for compatibility with legacy systems.

```
key = MD5(username ":" realm ":" OpaqueString(password))
```

17.5.1.2. SHA256

This password algorithm is taken from [RFC7616].

The key length is 32 bytes and the parameters value is empty.

```
key = SHA256(username ":" realm ":" OpaqueString(password))
```

17.6. STUN UDP and TCP Port Numbers

IANA is requested to update the reference from RFC 5389 to RFC-to-be for the following ports:

stun	3478/tcp	Session Traversal Utilities for NAT (STUN) port
stun	3478/udp	Session Traversal Utilities for NAT (STUN) port
stuns	5349/tcp	Session Traversal Utilities for NAT (STUN) port

18. Changes since RFC 5389

This specification obsoletes [RFC5389]. This specification differs from RFC 5389 in the following ways:

- Added support for DTLS-over-UDP (RFC 6347).

- Made clear that the RTO is considered stale if there is no transactions with the server.

- o Aligned the RTO calculation with RFC 6298.
- o Updated the cipher suites for TLS.
- o Added support for STUN URI (RFC 7064).
- o Added support for SHA256 message integrity.
- o Updated the PRECIS support to RFC 7613.
- o Added protocol and registry to choose the password encryption algorithm.
- o Added support for anonymous username.
- o Added protocol and registry for preventing biddown attacks.
- o Sharing a NONCE is no longer permitted.
- o Added the possibility of using a domain name in the alternate server mechanism.
- o Added more C snippets.
- o Added test vector.

19. References

19.1. Normative References

[ITU.V42.2002]
International Telecommunications Union, "Error-correcting Procedures for DCEs Using Asynchronous-to-Synchronous Conversion", ITU-T Recommendation V.42, 2002.

[RFC0791] Postel, J., "Internet Protocol", STD 5, RFC 791, DOI 10.17487/RFC0791, September 1981, <http://www.rfc-editor.org/info/rfc791>.

[RFC1122] Braden, R., Ed., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, DOI 10.17487/RFC1122, October 1989, <http://www.rfc-editor.org/info/rfc1122>.

[RFC1321] Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321, DOI 10.17487/RFC1321, April 1992, <http://www.rfc-editor.org/info/rfc1321>.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <http://www.rfc-editor.org/info/rfc2104>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC2460] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", RFC 2460, DOI 10.17487/RFC2460, December 1998, <http://www.rfc-editor.org/info/rfc2460>.

[RFC2617] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication", RFC 2617, DOI 10.17487/RFC2617, June 1999, <http://www.rfc-editor.org/info/rfc2617>.

[RFC2782] Gulbrandsen, A., Vixie, P., and L. Esibov, "A DNS RR for specifying the location of services (DNS SRV)", RFC 2782, DOI 10.17487/RFC2782, February 2000, <http://www.rfc-editor.org/info/rfc2782>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <http://www.rfc-editor.org/info/rfc3629>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <http://www.rfc-editor.org/info/rfc5246>.

[RFC6125] Saint-Andre, P. and J. Hodges, "Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)", RFC 6125, DOI 10.17487/RFC6125, March 2011, <http://www.rfc-editor.org/info/rfc6125>.

[RFC6298] Paxson, V., Allman, M., Chu, J., and M. Sargent, "Computing TCP’s Retransmission Timer", RFC 6298, DOI 10.17487/RFC6298, June 2011, <http://www.rfc-editor.org/info/rfc6298>.

[RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <http://www.rfc-editor.org/info/rfc6347>.

[RFC7064] Nandakumar, S., Salgueiro, G., Jones, P., and M. Petit-Huguenin, "URI Scheme for the Session Traversal Utilities for NAT (STUN) Protocol", RFC 7064, DOI 10.17487/RFC7064, November 2013, <http://www.rfc-editor.org/info/rfc7064>.

[RFC7350] Petit-Huguenin, M. and G. Salgueiro, "Datagram Transport Layer Security (DTLS) as Transport for Session Traversal Utilities for NAT (STUN)", RFC 7350, DOI 10.17487/RFC7350, August 2014, <http://www.rfc-editor.org/info/rfc7350>.

[RFC7613] Saint-Andre, P. and A. Melnikov, "Preparation, Enforcement, and Comparison of Internationalized Strings Representing Usernames and Passwords", RFC 7613, DOI 10.17487/RFC7613, August 2015, <http://www.rfc-editor.org/info/rfc7613>.

19.2. Informative References

[I-D.ietf-ice-rfc5245bis] Keranen, A. and J. Rosenberg, "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal", draft-ietf-ice-rfc5245bis-01 (work in progress), December 2015.

[KARN87] Karn, P. and C. Partridge, "Improving Round-Trip Time Estimates in Reliable Transport Protocols", SIGCOMM 1987, August 1987.

[RFC1952] Deutsch, P., "GZIP file format specification version 4.3", RFC 1952, DOI 10.17487/RFC1952, May 1996, <http://www.rfc-editor.org/info/rfc1952>.

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, DOI 10.17487/RFC2616, June 1999, <http://www.rfc-editor.org/info/rfc2616>.

[RFC3261] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and E. Schooler, "SIP: Session Initiation Protocol", RFC 3261, DOI 10.17487/RFC3261, June 2002, <http://www.rfc-editor.org/info/rfc3261>.

[RFC3424] Daigle, L., Ed. and IAB, "IAB Considerations for UNilateral Self-Address Fixing (UNSAF) Across Network Address Translation", RFC 3424, DOI 10.17487/RFC3424, November 2002, <http://www.rfc-editor.org/info/rfc3424>.

[RFC3489] Rosenberg, J., Weinberger, J., Huitema, C., and R. Mahy, "STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)", RFC 3489, DOI 10.17487/RFC3489, March 2003, <http://www.rfc-editor.org/info/rfc3489>.

[RFC4107] Bellovin, S. and R. Housley, "Guidelines for Cryptographic Key Management", BCP 107, RFC 4107, DOI 10.17487/RFC4107, June 2005, <http://www.rfc-editor.org/info/rfc4107>.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, DOI 10.17487/RFC5226, May 2008, <http://www.rfc-editor.org/info/rfc5226>.

[RFC5389] Rosenberg, J., Mahy, R., Matthews, P., and D. Wing, "Session Traversal Utilities for NAT (STUN)", RFC 5389, DOI 10.17487/RFC5389, October 2008, <http://www.rfc-editor.org/info/rfc5389>.

[RFC5626] Jennings, C., Ed., Mahy, R., Ed., and F. Audet, Ed., "Managing Client-Initiated Connections in the Session Initiation Protocol (SIP)", RFC 5626, DOI 10.17487/RFC5626, October 2009, <http://www.rfc-editor.org/info/rfc5626>.

[RFC5766] Mahy, R., Matthews, P., and J. Rosenberg, "Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)", RFC 5766, DOI 10.17487/RFC5766, April 2010, <http://www.rfc-editor.org/info/rfc5766>.

[RFC5769] Denis-Courmont, R., "Test Vectors for Session Traversal Utilities for NAT (STUN)", RFC 5769, DOI 10.17487/RFC5769, April 2010, <http://www.rfc-editor.org/info/rfc5769>.

[RFC5780] MacDonald, D. and B. Lowekamp, "NAT Behavior Discovery Using Session Traversal Utilities for NAT (STUN)", RFC 5780, DOI 10.17487/RFC5780, May 2010, <http://www.rfc-editor.org/info/rfc5780>.

[RFC6544] Rosenberg, J., Keranen, A., Lowekamp, B., and A. Roach, "TCP Candidates with Interactive Connectivity Establishment (ICE)", RFC 6544, DOI 10.17487/RFC6544, March 2012, <http://www.rfc-editor.org/info/rfc6544>.

[RFC7525] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 7525, DOI 10.17487/RFC7525, May 2015, <http://www.rfc-editor.org/info/rfc7525>.

[RFC7616] Shekh-Yusef, R., Ed., Ahrens, D., and S. Bremer, "HTTP Digest Access Authentication", RFC 7616, DOI 10.17487/RFC7616, September 2015, <http://www.rfc-editor.org/info/rfc7616>.

Appendix A. C Snippet to Determine STUN Message Types

Given a 16-bit STUN message type value in host byte order in msg_type parameter, below are C macros to determine the STUN message types:

```
<CODE BEGINS>
#define IS_REQUEST(msg_type)       (((msg_type) & 0x0110) == 0x0000)
#define IS_INDICATION(msg_type)    (((msg_type) & 0x0110) == 0x0010)
#define IS_SUCCESS_RESP(msg_type)  (((msg_type) & 0x0110) == 0x0100)
#define IS_ERR_RESP(msg_type)      (((msg_type) & 0x0110) == 0x0110)
<CODE ENDS>
```

A function to convert method and class into a message type:

```
<CODE BEGINS>
int type(int method, int cls) {
  return (method & 0x0F80) << 9 | (method & 0x0070) << 5
       | (method & 0x000F) | (cls & 0x0002) << 8
       | (cls & 0x0001) << 4;
  }
<CODE ENDS>
```

A function to extract the method from the message type:

```
<CODE BEGINS>
int method(int type) {
  return (type & 0x3E00) >> 2 | (type & 0x00E0) >> 1
       | (type & 0x000F);
  }
<CODE ENDS>
```

A function to extract the class from the message type:

```
<CODE BEGINS>
int cls(int type) {
  return (type & 0x0100) >> 7 | (type & 0x0010) >> 4;
  }
<CODE ENDS>
```

Appendix B. Test Vectors

This section augments the list of test vectors defined in [RFC5769] with MESSAGE-INTEGRITY-SHA256. All the formats and definitions listed in Section 2 of [RFC5769] apply here.

B.1. Sample Request with Long-Term Authentication with MESSAGE-INTEGRITY-SHA256 and USERHASH

This request uses the following parameters:

Username: "<U+30DE><U+30C8><U+30EA><U+30C3><U+30AF><U+30B9>" (without quotes) unaffected by OpaqueString [RFC7613] processing

Password: "The<U+00AD>M<U+00AA>tr<U+2168>" and "TheMatrIX" (without quotes) respectively before and after OpaqueString processing

Nonce: "obMatJos2AAACf//499k954d6OL34oL9FSTvy64sA" (without quotes)

Realm: "example.org" (without quotes)

```
00 01 00 9c      Request type and message length
21 12 a4 42      Magic cookie
78 ad 34 33   }
c6 ad 72 c0   }  Transaction ID
29 da 41 2e   }
XX XX 00 20      USERHASH attribute header
4a 3c f3 8f   }
ef 69 92 bd   }
a9 52 c6 78   }
04 17 da 0f   }  Userhash value (32 bytes)
24 81 94 15   }
56 9e 60 b2   }
05 c4 6e 41   }
40 7f 17 04   }
00 15 00 29      NONCE attribute header
6f 62 4d 61   }
74 4a 6f 73   }
32 41 41 41   }
43 66 2f 2f   }
34 39 39 6b   }  Nonce value and padding (3 bytes)
39 35 34 64   }
36 4f 4c 33   }
34 6f 4c 39   }
46 53 54 76   }
79 36 34 73   }
41 00 00 00   }
00 14 00 0b      REALM attribute header
65 78 61 6d   }
70 6c 65 2e   }  Realm value (11 bytes) and padding (1 byte)
6f 72 67 00   }
XX XX 00 20      MESSAGE-INTEGRITY-SHA256 attribute header
c4 ec a2 b6   }
24 6f 26 be   }
bc 2f 77 49   }
07 c2 00 a3   }  HMAC-SHA256 value
76 c7 c2 8e   }
b4 d1 26 60   }
bb fe 9f 28   }
0e 85 71 f2   }
```

Note: Before publication, the XX XX placeholder must be replaced by the value assigned to MESSAGE-INTEGRITY-SHA256 and USERHASH by IANA. The MESSAGE-INTEGRITY-SHA256 attribute value will need to be updated after this.

Appendix C. Release notes

This section must be removed before publication as an RFC.

C.1. Modifications between draft-ietf-tram-stunbis-11 and draft-ietf-tram-stunbis-10

- Made clear that the same HMAC than received in response of short term credential must be used for subsequent transactions.
- s/URL/URI/
- The "nonce cookie" is now mandatory to signal that SHA256 must be used in the next transaction.
- s/SHA1/SHA256/
- Changed co-author affiliation.

C.2. Modifications between draft-ietf-tram-stunbis-10 and draft-ietf-tram-stunbis-09

- Removed the reserved value in the security registry, as it does not make sense in a bitset.
- Updated change list.
- Updated the minimum trancation size for M-I-256 to 16 bytes.
- Changed the truncation order to match RFC 7518.
- Fixed bugs in truncation boundary text.
- Stated that STUN Usages have to explicitly state that they can use truncation.
- Removed truncation from the MESSAGE-INTEGRITY attribute.
- Add reference to C code in RFC 1952.
- Replaced RFC 2818 reference to RFC 6125.

C.3. Modifications between draft-ietf-tram-stunbis-09 and draft-ietf-tram-stunbis-08

- Removed the reserved value in the security registry, as it does not make sense in a bitset.

- o Updated change list.
- o Updated the minimum trancation size for M-I-256 to 16 bytes.
- o Changed the truncation order to match RFC 7518.
- o Fixed bugs in truncation boundary text.
- o Stated that STUN Usages have to explicitly state that they can use truncation.
- o Removed truncation from the MESSAGE-INTEGRITY attrbute.
- o Add reference to C code in RFC 1952.
- o Replaced RFC 2818 reference to RFC 6125.

C.4. Modifications between draft-ietf-tram-stunbis-09 and draft-ietf-tram-stunbis-08

- o Packets discarded in a reliable or unreliable transaction triggers an attack error instead of a timeout error. An attack error on a reliable transport is signaled immediately instead of waiting for the timeout.
- o Explicitly state that a received 400 response without authentication will be dropped until timeout.
- o Clarify the SHOULD omit/include rules in LTCM.
- o If the nonce and the hmac are both invalid, then a 401 is sent instead of a 438.
- o The 401 and 438 error response to subsequent requests may use the previous NONCE/password to authenticate, if they are still available.
- o Change "401 Unauthorized" to "401 Unauthenticated"
- o Make clear that in some cases it is impossible to add a MI or MI2 even if the text says SHOULD NOT.

C.5. Modifications between draft-ietf-tram-stunbis-08 and draft-ietf-tram-stunbis-07

- o Updated list of changes since RFC 5389.
- o More examples are automatically generated.

- o Message integrity truncation is fixed at a multiple of 4 bytes, because the padding will not decrease by more than this.

- o USERHASH contains the 32 bytes of the hash, not a character string.

- o Updated the example to use the USERHASH attribuet and the modified NONCE attribute.

- o Updated ICEbis reference.

C.6. Modifications between draft-ietf-tram-stunbis-07 and draft-ietf-tram-stunbis-06

- o Add USERHASH attribute to carry the hashed version of the username.

- o Add IANA registry and nonce encoding for Security Features that need to be protected from bid down attacks.

- o Modified MESSAGE-INTEGRITY and MESSAGE-INTEGRITY-SHA256 to support truncation limits (pending cryptographic review),

C.7. Modifications between draft-ietf-tram-stunbis-06 and draft-ietf-tram-stunbis-05

- o Changed I-D references to RFC references.

- o Changed CHANGE-ADDRESS to CHANGE-REQUEST (Errata #4233).

- o Added test vector for MESSAGE-INTEGRITY-SHA256.

- o Address additional review comments from Jonathan Lennox and Brandon Williams.

C.8. Modifications between draft-ietf-tram-stunbis-05 and draft-ietf-tram-stunbis-04

- o Address review comments from Jonathan Lennox and Brandon Williams.

C.9. Modifications between draft-ietf-tram-stunbis-04 and draft-ietf-tram-stunbis-03

- o Remove SCTP.

- o Remove DANE.

- o s/MESSAGE-INTEGRITY2/MESSAGE-INTEGRITY-SHA256/

- o Remove Salted SHA256 password hash.
- o The RTO delay between transactions is removed.
- o Make clear that reusing NONCE will trigger a wasted round trip.

C.10. Modifications between draft-ietf-tram-stunbis-03 and draft-ietf-tram-stunbis-02

- o SCTP prefix is now 0b00000101 instead of 0x11.
- o Add SCTP at various places it was needed.
- o Update the hash agility plan to take in account HMAC-SHA-256.
- o Adds the bid down attack on message-integrity in the security section.

C.11. Modifications between draft-ietf-tram-stunbis-02 and draft-ietf-tram-stunbis-01

- o STUN hash algorithm agility (currently only SHA-1 is allowed).
- o Clarify terminology, text and guidance for STUN fragmentation.
- o Clarify whether it’s valid to share nonces across TURN allocations.
- o Prevent the server to allocate the same NONCE to clients with different IP address and/or different port. This prevent sharing the nonce between TURN allocations in TURN.
- o Add reference to draft-ietf-uta-tls-bcp
- o Add a new attribute ALTERNATE-DOMAIN to verify the certificate of the ALTERNATE-SERVER after a 300 over (D)TLS.
- o The RTP delay between transactions applies only to parallel transactions, not to serial transactions. That prevents a 3RTT delay between the first transaction and the second transaction with long term authentication.
- o Add text saying ORIGIN can increase a request size beyond the MTU and so require an SCTPoUDP transport.
- o Move the Acknowledgments and Contributor sections to the end of the document, in accordance with RFC 7322 section 4.

C.12. Modifications between draft-ietf-tram-stunbis-01 and draft-ietf-tram-stunbis-00

- o Add negotiation mechanism for new password algorithms.
- o Describe the MESSAGE-INTEGRITY/MESSAGE-INTEGRITY2 protocol.
- o Add support for SCTP to solve the fragmentation problem.
- o Merge RFC 7350:
  - * Split the "Sending over..." sections in 3.
  - * Add DTLS-over-UDP as transport.
  - * Update the cipher suites and cipher/compression restrictions.
  - * A stuns uri with an IP address is rejected.
  - * Replace most of the RFC 3489 compatibility by a reference to the section in RFC 5389.
  - * Update the STUN Usages list with transport applicability.
- o Merge RFC 7064:
  - * DNS discovery is done from the URI.
  - * Reorganized the text about default ports.
- o Add more C snippets.
- o Make clear that the cached RTO is discarded only if there is no new transations for 10 minutes.

C.13. Modifications between draft-salgueiro-tram-stunbis-02 and draft-ietf-tram-stunbis-00

- o Draft adopted as WG item.

C.14. Modifications between draft-salgueiro-tram-stunbis-02 and draft-salgueiro-tram-stunbis-01

- o Add definition of MESSAGE-INTEGRITY2.
- o Update text and reference from RFC 2988 to RFC 6298.
- o s/The IAB has mandated/The IAB has suggested/ (Errata #3737).

- o Fix the figure for the UNKNOWN-ATTRIBUTES (Errata #2972).
- o Fix section number and make clear that the original domain name is used for the server certificate verification. This is consistent with what RFC 5922 (section 4) is doing. (Errata #2010)
- o Remove text transitioning from RFC 3489.
- o Add definition of MESSAGE-INTEGRITY2.
- o Update text and reference from RFC 2988 to RFC 6298.
- o s/The IAB has mandated/The IAB has suggested/ (Errata #3737).
- o Fix the figure for the UNKNOWN-ATTRIBUTES (Errata #2972).
- o Fix section number and make clear that the original domain name is used for the server certificate verification. This is consistent with what RFC 5922 (section 4) is doing. (Errata #2010)

C.15. Modifications between draft-salgueiro-tram-stunbis-01 and draft-salgueiro-tram-stunbis-00

- o Restore the RFC 5389 text.
- o Add list of open issues.

Acknowledgements

Thanks to Michael Tuexen, Tirumaleswar Reddy, Oleg Moskalenko, Simon Perreault, Benjamin Schwartz, Rifaat Shekh-Yusef, Alan Johnston, Jonathan Lennox, Brandon Williams, Olle Johansson, Martin Thomson, and Mihaly Meszaros for the comments, suggestions, and questions that helped improve this document.

The authors of RFC 5389 would like to thank Cedric Aoun, Pete Cordell, Cullen Jennings, Bob Penfield, Xavier Marjou, Magnus Westerlund, Miguel Garcia, Bruce Lowekamp, and Chris Sullivan for their comments, and Baruch Sterman and Alan Hawrylyshen for initial implementations. Thanks for Leslie Daigle, Allison Mankin, Eric Rescorla, and Henning Schulzrinne for IESG and IAB input on this work.

Contributors

Christian Huitema and Joel Weinberger were original co-authors of RFC 3489.

Authors' Addresses

Marc Petit-Huguenin
Impedance Mismatch

Email: marc@petit-huguenin.org

Gonzalo Salgueiro
Cisco
7200-12 Kit Creek Road
Research Triangle Park, NC 27709
US

Email: gsalguei@cisco.com

Jonathan Rosenberg
Cisco
Edison, NJ
US

Email: jdrosen@cisco.com
URI: http://www.jdrosen.net

Dan Wing

Email: dwing-ietf@fuggles.com

Rohan Mahy
Plantronics
345 Encinal Street
Santa Cruz, CA 95060
US

Email: rohan@ekabal.com

Philip Matthews
Avaya
1135 Innovation Drive
Ottawa, Ontario K2K 3G7
Canada

Phone: +1 613 592 4343 x224
Email: philip_matthews@magma.ca

TRAM WG
Internet-Draft
Intended status: Standards Track
Expires: May 3, 2017

T. Reddy, Ed.
Cisco Systems, Inc.
A. Johnston, Ed.
Unaffiliated
P. Matthews
Alcatel-Lucent
J. Rosenberg
jdrosen.net
October 30, 2016

Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)
draft-ietf-tram-turnbis-09

Abstract

If a host is located behind a NAT, then in certain situations it can be impossible for that host to communicate directly with other hosts (peers). In these situations, it is necessary for the host to use the services of an intermediate node that acts as a communication relay. This specification defines a protocol, called TURN (Traversal Using Relays around NAT), that allows the host to control the operation of the relay and to exchange packets with its peers using the relay. TURN differs from some other relay control protocols in that it allows a client to communicate with multiple peers using a single relay address.

The TURN protocol was designed to be used as part of the ICE (Interactive Connectivity Establishment) approach to NAT traversal, though it also can be used without ICE.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on May 3, 2017.

Copyright Notice

Copyright (c) 2016 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction 4
2. Overview of Operation 6
  2.1. Transports 8
  2.2. Allocations 9
  2.3. Permissions 11
  2.4. Send Mechanism 12
  2.5. Channels 14
  2.6. Unprivileged TURN Servers 16
  2.7. Avoiding IP Fragmentation 16
  2.8. RTP Support 18
  2.9. Discovery of TURN server 18
    2.9.1. TURN URI Scheme Semantics 18
3. Terminology 18
4. General Behavior 20
5. Allocations 22
6. Creating an Allocation 24
  6.1. Sending an Allocate Request 24
  6.2. Receiving an Allocate Request 25
  6.3. Receiving an Allocate Success Response 30
  6.4. Receiving an Allocate Error Response 31
7. Refreshing an Allocation 33
  7.1. Sending a Refresh Request 33
  7.2. Receiving a Refresh Request 34
  7.3. Receiving a Refresh Response 34
8. Permissions 35
9. CreatePermission 36
  9.1. Forming a CreatePermission Request 36
  9.2. Receiving a CreatePermission Request 36
  9.3. Receiving a CreatePermission Response 37
10. Send and Data Methods 37
  10.1. Forming a Send Indication 37

10.2. Receiving a Send Indication 38
10.3. Receiving a UDP Datagram 39
10.4. Receiving a Data Indication with DATA attribute 39
10.5. Receiving an ICMP Packet 40
10.6. Receiving a Data Indication with an ICMP attribute . . . 40
11. Channels 41
11.1. Sending a ChannelBind Request 43
11.2. Receiving a ChannelBind Request 43
11.3. Receiving a ChannelBind Response 44
11.4. The ChannelData Message 45
11.5. Sending a ChannelData Message 45
11.6. Receiving a ChannelData Message 46
11.7. Relaying Data from the Peer 47
12. Packet Translations 47
12.1. IPv4-to-IPv6 Translations 47
12.2. IPv6-to-IPv6 Translations 48
12.3. IPv6-to-IPv4 Translations 49
13. IP Header Fields 50
14. New STUN Methods 52
15. New STUN Attributes 52
15.1. CHANNEL-NUMBER 53
15.2. LIFETIME 53
15.3. XOR-PEER-ADDRESS 53
15.4. DATA 53
15.5. XOR-RELAYED-ADDRESS 53
15.6. REQUESTED-ADDRESS-FAMILY 54
15.7. EVEN-PORT 54
15.8. REQUESTED-TRANSPORT 55
15.9. DONT-FRAGMENT 55
15.10. RESERVATION-TOKEN 55
15.11. ADDITIONAL-ADDRESS-FAMILY 56
15.12. ADDRESS-ERROR-CODE Attribute 56
15.13. ICMP Attribute 57
16. New STUN Error Response Codes 57
17. Detailed Example 58
18. Security Considerations 65
18.1. Outsider Attacks 65
18.1.1. Obtaining Unauthorized Allocations 65
18.1.2. Offline Dictionary Attacks 66
18.1.3. Faked Refreshes and Permissions 66
18.1.4. Fake Data 66
18.1.5. Impersonating a Server 67
18.1.6. Eavesdropping Traffic 67
18.1.7. TURN Loop Attack 68
18.2. Firewall Considerations 69
18.2.1. Faked Permissions 69
18.2.2. Blacklisted IP Addresses 70
18.2.3. Running Servers on Well-Known Ports 70

18.3. Insider Attacks 70
18.3.1. DoS against TURN Server 70
18.3.2. Anonymous Relaying of Malicious Traffic 71
18.3.3. Manipulating Other Allocations 71
18.4. Tunnel Amplification Attack 71
18.5. Other Considerations 72
19. IANA Considerations 72
20. IAB Considerations 73
21. Changes since RFC 5766 75
22. Acknowledgements 75
23. References 76
23.1. Normative References 76
23.2. Informative References 77
Authors' Addresses 80

1. Introduction

A host behind a NAT may wish to exchange packets with other hosts, some of which may also be behind NATs. To do this, the hosts involved can use "hole punching" techniques (see [RFC5128]) in an attempt discover a direct communication path; that is, a communication path that goes from one host to another through intervening NATs and routers, but does not traverse any relays.

As described in [RFC5128] and [RFC4787], hole punching techniques will fail if both hosts are behind NATs that are not well behaved. For example, if both hosts are behind NATs that have a mapping behavior of "address-dependent mapping" or "address- and port-dependent mapping", then hole punching techniques generally fail.

When a direct communication path cannot be found, it is necessary to use the services of an intermediate host that acts as a relay for the packets. This relay typically sits in the public Internet and relays packets between two hosts that both sit behind NATs.

This specification defines a protocol, called TURN, that allows a host behind a NAT (called the TURN client) to request that another host (called the TURN server) act as a relay. The client can arrange for the server to relay packets to and from certain other hosts (called peers) and can control aspects of how the relaying is done. The client does this by obtaining an IP address and port on the server, called the relayed transport address. When a peer sends a packet to the relayed transport address, the server relays the packet to the client. When the client sends a data packet to the server, the server relays it to the appropriate peer using the relayed transport address as the source.

A client using TURN must have some way to communicate the relayed transport address to its peers, and to learn each peer’s IP address and port (more precisely, each peer’s server-reflexive transport address, see Section 2). How this is done is out of the scope of the TURN protocol. One way this might be done is for the client and peers to exchange email messages. Another way is for the client and its peers to use a special-purpose "introduction" or "rendezvous" protocol (see [RFC5128] for more details).

If TURN is used with ICE [RFC5245], then the relayed transport address and the IP addresses and ports of the peers are included in the ICE candidate information that the rendezvous protocol must carry. For example, if TURN and ICE are used as part of a multimedia solution using SIP [RFC3261], then SIP serves the role of the rendezvous protocol, carrying the ICE candidate information inside the body of SIP messages. If TURN and ICE are used with some other rendezvous protocol, then [I-D.rosenberg-mmusic-ice-nonsip] provides guidance on the services the rendezvous protocol must perform.

Though the use of a TURN server to enable communication between two hosts behind NATs is very likely to work, it comes at a high cost to the provider of the TURN server, since the server typically needs a high-bandwidth connection to the Internet . As a consequence, it is best to use a TURN server only when a direct communication path cannot be found. When the client and a peer use ICE to determine the communication path, ICE will use hole punching techniques to search for a direct path first and only use a TURN server when a direct path cannot be found.

TURN was originally invented to support multimedia sessions signaled using SIP. Since SIP supports forking, TURN supports multiple peers per relayed transport address; a feature not supported by other approaches (e.g., SOCKS [RFC1928]). However, care has been taken to make sure that TURN is suitable for other types of applications.

TURN was designed as one piece in the larger ICE approach to NAT traversal. Implementors of TURN are urged to investigate ICE and seriously consider using it for their application. However, it is possible to use TURN without ICE.

TURN is an extension to the STUN (Session Traversal Utilities for NAT) protocol [RFC5389]. Most, though not all, TURN messages are STUN-formatted messages. A reader of this document should be familiar with STUN.

2. Overview of Operation

This section gives an overview of the operation of TURN. It is non-normative.

In a typical configuration, a TURN client is connected to a private network [RFC1918] and through one or more NATs to the public Internet. On the public Internet is a TURN server. Elsewhere in the Internet are one or more peers with which the TURN client wishes to communicate. These peers may or may not be behind one or more NATs. The client uses the server as a relay to send packets to these peers and to receive packets from these peers.

```
                                          Peer A
                                          Server-Reflexive    +---------+
                                          Transport Address   |         |
                                          192.0.2.150:32102   |         |
                                              |               /|         |
                            TURN              |            / ^|  Peer A |
    Client's                Server            |           /  ||         |
    Host Transport          Transport         |         //   ||         |
    Address                 Address           |       //     |+---------+
   10.1.1.2:49721       192.0.2.15:3478       |+-+  //     Peer A
            |               |                ||N| /       Host Transport
            |   +-+         |                ||A|/        Address
            |   | |         |                v|T|     192.168.100.2:49582
            |   | |         |                /+-+
 +---------+|   | |         |+---------+   /              +---------+
 |         ||   |N|         ||         | //               |         |
 | TURN    |v   | |         v| TURN    |/                 |         |
 | Client  |----|A|----------| Server  |------------------|  Peer B |
 |         |    | |^         |         |^                 ^|         |
 |         |    |T||         |         ||                 ||         |
 +---------+    | ||         +---------+|                 |+---------+
                | ||                     |                 |
                | ||                     |                 |
                +-+|                     |                 |
                   |                     |                 |
                   |                     |                 |
             Client's                    |            Peer B
             Server-Reflexive    Relayed             Transport
             Transport Address   Transport Address   Address
             192.0.2.1:7000      192.0.2.15:50000     192.0.2.210:49191
```

Figure 1

Figure 1 shows a typical deployment. In this figure, the TURN client and the TURN server are separated by a NAT, with the client on the

private side and the server on the public side of the NAT. This NAT is assumed to be a "bad" NAT; for example, it might have a mapping property of "address-and-port-dependent mapping" (see [RFC4787]).

The client talks to the server from a (IP address, port) combination called the client's HOST TRANSPORT ADDRESS. (The combination of an IP address and port is called a TRANSPORT ADDRESS.)

The client sends TURN messages from its host transport address to a transport address on the TURN server that is known as the TURN SERVER TRANSPORT ADDRESS. The client learns the TURN server transport address through some unspecified means (e.g., configuration), and this address is typically used by many clients simultaneously.

Since the client is behind a NAT, the server sees packets from the client as coming from a transport address on the NAT itself. This address is known as the client's SERVER-REFLEXIVE transport address; packets sent by the server to the client's server-reflexive transport address will be forwarded by the NAT to the client's host transport address.

The client uses TURN commands to create and manipulate an ALLOCATION on the server. An allocation is a data structure on the server. This data structure contains, amongst other things, the RELAYED TRANSPORT ADDRESS for the allocation. The relayed transport address is the transport address on the server that peers can use to have the server relay data to the client. An allocation is uniquely identified by its relayed transport address.

Once an allocation is created, the client can send application data to the server along with an indication of to which peer the data is to be sent, and the server will relay this data to the appropriate peer. The client sends the application data to the server inside a TURN message; at the server, the data is extracted from the TURN message and sent to the peer in a UDP datagram. In the reverse direction, a peer can send application data in a UDP datagram to the relayed transport address for the allocation; the server will then encapsulate this data inside a TURN message and send it to the client along with an indication of which peer sent the data. Since the TURN message always contains an indication of which peer the client is communicating with, the client can use a single allocation to communicate with multiple peers.

When the peer is behind a NAT, then the client must identify the peer using its server-reflexive transport address rather than its host transport address. For example, to send application data to Peer A in the example above, the client must specify 192.0.2.150:32102 (Peer

A's server-reflexive transport address) rather than 192.168.100.2:49582 (Peer A's host transport address).

Each allocation on the server belongs to a single client and has exactly one relayed transport address that is used only by that allocation. Thus, when a packet arrives at a relayed transport address on the server, the server knows for which client the data is intended.

The client may have multiple allocations on a server at the same time.

2.1. Transports

TURN, as defined in this specification, always uses UDP between the server and the peer. However, this specification allows the use of any one of UDP, TCP, Transport Layer Security (TLS) over TCP or Datagram Transport Layer Security (DTLS) over UDP to carry the TURN messages between the client and the server.

```
+----------------------------+---------------------+
| TURN client to TURN server | TURN server to peer |
+----------------------------+---------------------+
|            UDP             |         UDP         |
|            TCP             |         UDP         |
|        TLS-over-TCP        |         UDP         |
|       DTLS-over-UDP        |         UDP         |
+----------------------------+---------------------+
```

If TCP or TLS-over-TCP is used between the client and the server, then the server will convert between these transports and UDP transport when relaying data to/from the peer.

Since this version of TURN only supports UDP between the server and the peer, it is expected that most clients will prefer to use UDP between the client and the server as well. That being the case, some readers may wonder: Why also support TCP and TLS-over-TCP?

TURN supports TCP transport between the client and the server because some firewalls are configured to block UDP entirely. These firewalls block UDP but not TCP, in part because TCP has properties that make the intention of the nodes being protected by the firewall more obvious to the firewall. For example, TCP has a three-way handshake that makes in clearer that the protected node really wishes to have that particular connection established, while for UDP the best the firewall can do is guess which flows are desired by using filtering rules. Also, TCP has explicit connection teardown; while for UDP, the firewall has to use timers to guess when the flow is finished.

TURN supports TLS-over-TCP transport and DTLS-over-UDP transport between the client and the server because (D)TLS provides additional security properties not provided by TURN's default digest authentication; properties that some clients may wish to take advantage of. In particular, (D)TLS provides a way for the client to ascertain that it is talking to the correct server, and provides for confidentiality of TURN control messages. TURN does not require (D)TLS because the overhead of using (D)TLS is higher than that of digest authentication; for example, using (D)TLS likely means that most application data will be doubly encrypted (once by (D)TLS and once to ensure it is still encrypted in the UDP datagram).

There is an extension to TURN for TCP transport between the server and the peers [RFC6062]. For this reason, allocations that use UDP between the server and the peers are known as UDP allocations, while allocations that use TCP between the server and the peers are known as TCP allocations. This specification describes only UDP allocations.

In some applications for TURN, the client may send and receive packets other than TURN packets on the host transport address it uses to communicate with the server. This can happen, for example, when using TURN with ICE. In these cases, the client can distinguish TURN packets from other packets by examining the source address of the arriving packet: those arriving from the TURN server will be TURN packets.

2.2. Allocations

To create an allocation on the server, the client uses an Allocate transaction. The client sends an Allocate request to the server, and the server replies with an Allocate success response containing the allocated relayed transport address. The client can include attributes in the Allocate request that describe the type of allocation it desires (e.g., the lifetime of the allocation). Since relaying data has security implications, the server requires that the client authenticate itself, typically using STUN's long-term credential mechanism, to show that it is authorized to use the server.

Once a relayed transport address is allocated, a client must keep the allocation alive. To do this, the client periodically sends a Refresh request to the server. TURN deliberately uses a different method (Refresh rather than Allocate) for refreshes to ensure that the client is informed if the allocation vanishes for some reason.

The frequency of the Refresh transaction is determined by the lifetime of the allocation. The default lifetime of an allocation is

10 minutes -- this value was chosen to be long enough so that refreshing is not typically a burden on the client, while expiring allocations where the client has unexpectedly quit in a timely manner. However, the client can request a longer lifetime in the Allocate request and may modify its request in a Refresh request, and the server always indicates the actual lifetime in the response. The client must issue a new Refresh transaction within "lifetime" seconds of the previous Allocate or Refresh transaction. Once a client no longer wishes to use an allocation, it should delete the allocation using a Refresh request with a requested lifetime of 0.

Both the server and client keep track of a value known as the 5-TUPLE. At the client, the 5-tuple consists of the client’s host transport address, the server transport address, and the transport protocol used by the client to communicate with the server. At the server, the 5-tuple value is the same except that the client’s host transport address is replaced by the client’s server-reflexive address, since that is the client’s address as seen by the server.

Both the client and the server remember the 5-tuple used in the Allocate request. Subsequent messages between the client and the server use the same 5-tuple. In this way, the client and server know which allocation is being referred to. If the client wishes to allocate a second relayed transport address, it must create a second allocation using a different 5-tuple (e.g., by using a different client host address or port).

NOTE: While the terminology used in this document refers to 5-tuples, the TURN server can store whatever identifier it likes that yields identical results. Specifically, an implementation may use a file-descriptor in place of a 5-tuple to represent a TCP connection.

```
  TURN                                  TURN           Peer          Peer
  client                                server          A             B
    |-- Allocate request --------------->|             |             |
    |                                    |             |             |
    |<--------------- Allocate failure --|             |             |
    |                 (401 Unauthorized) |             |             |
    |                                    |             |             |
    |-- Allocate request --------------->|             |             |
    |                                    |             |             |
    |<---------- Allocate success resp --|             |             |
    |            (192.0.2.15:50000)      |             |             |
    //                                  //             //            //
    |                                    |             |             |
    |-- Refresh request ---------------->|             |             |
    |                                    |             |             |
    |<----------- Refresh success resp --|             |             |
    |                                    |             |             |
```

Figure 2

In Figure 2, the client sends an Allocate request to the server without credentials. Since the server requires that all requests be authenticated using STUN's long-term credential mechanism, the server rejects the request with a 401 (Unauthorized) error code. The client then tries again, this time including credentials (not shown). This time, the server accepts the Allocate request and returns an Allocate success response containing (amongst other things) the relayed transport address assigned to the allocation. Sometime later, the client decides to refresh the allocation and thus sends a Refresh request to the server. The refresh is accepted and the server replies with a Refresh success response.

2.3. Permissions

To ease concerns amongst enterprise IT administrators that TURN could be used to bypass corporate firewall security, TURN includes the notion of permissions. TURN permissions mimic the address-restricted filtering mechanism of NATs that comply with [RFC4787].

An allocation can have zero or more permissions. Each permission consists of an IP address and a lifetime. When the server receives a UDP datagram on the allocation's relayed transport address, it first checks the list of permissions. If the source IP address of the datagram matches a permission, the application data is relayed to the client, otherwise the UDP datagram is silently discarded.

A permission expires after 5 minutes if it is not refreshed, and there is no way to explicitly delete a permission. This behavior was selected to match the behavior of a NAT that complies with [RFC4787].

The client can install or refresh a permission using either a CreatePermission request or a ChannelBind request. Using the CreatePermission request, multiple permissions can be installed or refreshed with a single request -- this is important for applications that use ICE. For security reasons, permissions can only be installed or refreshed by transactions that can be authenticated; thus, Send indications and ChannelData messages (which are used to send data to peers) do not install or refresh any permissions.

Note that permissions are within the context of an allocation, so adding or expiring a permission in one allocation does not affect other allocations.

2.4. Send Mechanism

There are two mechanisms for the client and peers to exchange application data using the TURN server. The first mechanism uses the Send and Data methods, the second way uses channels. Common to both ways is the ability of the client to communicate with multiple peers using a single allocated relayed transport address; thus, both ways include a means for the client to indicate to the server which peer should receive the data, and for the server to indicate to the client which peer sent the data.

The Send mechanism uses Send and Data indications. Send indications are used to send application data from the client to the server, while Data indications are used to send application data from the server to the client.

When using the Send mechanism, the client sends a Send indication to the TURN server containing (a) an XOR-PEER-ADDRESS attribute specifying the (server-reflexive) transport address of the peer and (b) a DATA attribute holding the application data. When the TURN server receives the Send indication, it extracts the application data from the DATA attribute and sends it in a UDP datagram to the peer, using the allocated relay address as the source address. Note that there is no need to specify the relayed transport address, since it is implied by the 5-tuple used for the Send indication.

In the reverse direction, UDP datagrams arriving at the relayed transport address on the TURN server are converted into Data indications and sent to the client, with the server-reflexive transport address of the peer included in an XOR-PEER-ADDRESS attribute and the data itself in a DATA attribute. Since the relayed

transport address uniquely identified the allocation, the server knows which client should receive the data.

Some ICMP (Internet Control Message Protocol) packets arriving at the relayed transport address on the TURN server may be converted into Data indications and sent to the client, with the transport address of the peer included in an XOR-PEER-ADDRESS attribute and the ICMP type and code in a ICMP attribute. Data indications containing the XOR-PEER-ADDRESS and ICMP attribute are also sent when using the channel mechanism.

Send and Data indications cannot be authenticated, since the long-term credential mechanism of STUN does not support authenticating indications. This is not as big an issue as it might first appear, since the client-to-server leg is only half of the total path to the peer. Applications that want proper security should encrypt the data sent between the client and a peer.

Because Send indications are not authenticated, it is possible for an attacker to send bogus Send indications to the server, which will then relay these to a peer. To partly mitigate this attack, TURN requires that the client install a permission towards a peer before sending data to it using a Send indication.

```
  TURN                                  TURN           Peer          Peer
  client                                server          A             B
    |                                     |             |             |
    |-- CreatePermission req (Peer A) -->|             |             |
    |<-- CreatePermission success resp --|             |             |
    |                                     |             |             |
    |--- Send ind (Peer A)-------------->|             |             |
    |                                     |=== data ===>|             |
    |                                     |             |             |
    |                                     |<== data ====|             |
    |<-------------- Data ind (Peer A) --|             |             |
    |                                     |             |             |
    |                                     |             |             |
    |--- Send ind (Peer B)-------------->|             |             |
    |                                     | dropped     |             |
    |                                     |             |             |
    |                                     |<== data ==================|
    |                             dropped |             |             |
    |                                     |             |             |
```

Figure 3

In Figure 3, the client has already created an allocation and now wishes to send data to its peers. The client first creates a

permission by sending the server a CreatePermission request specifying Peer A's (server-reflexive) IP address in the XOR-PEER-ADDRESS attribute; if this was not done, the server would not relay data between the client and the server. The client then sends data to Peer A using a Send indication; at the server, the application data is extracted and forwarded in a UDP datagram to Peer A, using the relayed transport address as the source transport address. When a UDP datagram from Peer A is received at the relayed transport address, the contents are placed into a Data indication and forwarded to the client. Later, the client attempts to exchange data with Peer B; however, no permission has been installed for Peer B, so the Send indication from the client and the UDP datagram from the peer are both dropped by the server.

2.5. Channels

For some applications (e.g., Voice over IP), the 36 bytes of overhead that a Send indication or Data indication adds to the application data can substantially increase the bandwidth required between the client and the server. To remedy this, TURN offers a second way for the client and server to associate data with a specific peer.

This second way uses an alternate packet format known as the ChannelData message. The ChannelData message does not use the STUN header used by other TURN messages, but instead has a 4-byte header that includes a number known as a channel number. Each channel number in use is bound to a specific peer and thus serves as a shorthand for the peer's host transport address.

To bind a channel to a peer, the client sends a ChannelBind request to the server, and includes an unbound channel number and the transport address of the peer. Once the channel is bound, the client can use a ChannelData message to send the server data destined for the peer. Similarly, the server can relay data from that peer towards the client using a ChannelData message.

Channel bindings last for 10 minutes unless refreshed -- this lifetime was chosen to be longer than the permission lifetime. Channel bindings are refreshed by sending another ChannelBind request rebinding the channel to the peer. Like permissions (but unlike allocations), there is no way to explicitly delete a channel binding; the client must simply wait for it to time out.

```
  TURN                                  TURN           Peer          Peer
  client                                server          A             B
    |                                     |             |             |
    |-- ChannelBind req ---------------->|             |             |
    | (Peer A to 0x4001)                  |             |             |
    |                                     |             |             |
    |<---------- ChannelBind succ resp --|             |             |
    |                                     |             |             |
    |-- [0x4001] data ------------------>|             |             |
    |                                     |=== data ===>|             |
    |                                     |             |             |
    |                                     |<== data ====|             |
    |<------------------ [0x4001] data --|             |             |
    |                                     |             |             |
    |--- Send ind (Peer A)-------------->|             |             |
    |                                     |=== data ===>|             |
    |                                     |             |             |
    |                                     |<== data ====|             |
    |<------------------ [0x4001] data --|             |             |
    |                                     |             |             |
```

Figure 4

Figure 4 shows the channel mechanism in use. The client has already created an allocation and now wishes to bind a channel to Peer A. To do this, the client sends a ChannelBind request to the server, specifying the transport address of Peer A and a channel number (0x4001). After that, the client can send application data encapsulated inside ChannelData messages to Peer A: this is shown as "[0x4001] data" where 0x4001 is the channel number. When the ChannelData message arrives at the server, the server transfers the data to a UDP datagram and sends it to Peer A (which is the peer bound to channel number 0x4001).

In the reverse direction, when Peer A sends a UDP datagram to the relayed transport address, this UDP datagram arrives at the server on the relayed transport address assigned to the allocation. Since the UDP datagram was received from Peer A, which has a channel number assigned to it, the server encapsulates the data into a ChannelData message when sending the data to the client.

Once a channel has been bound, the client is free to intermix ChannelData messages and Send indications. In the figure, the client later decides to use a Send indication rather than a ChannelData message to send additional data to Peer A. The client might decide to do this, for example, so it can use the DONT-FRAGMENT attribute (see the next section). However, once a channel is bound, the server will always use a ChannelData message, as shown in the call flow.

Note that ChannelData messages can only be used for peers to which the client has bound a channel. In the example above, Peer A has been bound to a channel, but Peer B has not, so application data to and from Peer B would use the Send mechanism.

2.6. Unprivileged TURN Servers

This version of TURN is designed so that the server can be implemented as an application that runs in user space under commonly available operating systems without requiring special privileges. This design decision was made to make it easy to deploy a TURN server: for example, to allow a TURN server to be integrated into a peer-to-peer application so that one peer can offer NAT traversal services to another peer.

This design decision has the following implications for data relayed by a TURN server:

- o The value of the Diffserv field may not be preserved across the server;
- o The Time to Live (TTL) field may be reset, rather than decremented, across the server;
- o The Explicit Congestion Notification (ECN) field may be reset by the server;
- o There is no end-to-end fragmentation, since the packet is re-assembled at the server.

Future work may specify alternate TURN semantics that address these limitations.

2.7. Avoiding IP Fragmentation

For reasons described in [Frag-Harmful], applications, especially those sending large volumes of data, should try hard to avoid having their packets fragmented. Applications using TCP can more or less ignore this issue because fragmentation avoidance is now a standard part of TCP, but applications using UDP (and thus any application using this version of TURN) must handle fragmentation avoidance themselves.

The application running on the client and the peer can take one of two approaches to avoid IP fragmentation.

The first approach is to avoid sending large amounts of application data in the TURN messages/UDP datagrams exchanged between the client

and the peer. This is the approach taken by most VoIP (Voice-over-IP) applications. In this approach, the application exploits the fact that the IP specification [RFC0791] specifies that IP packets up to 576 bytes should never need to be fragmented.

The exact amount of application data that can be included while avoiding fragmentation depends on the details of the TURN session between the client and the server: whether UDP, TCP, or (D)TLS transport is used, whether ChannelData messages or Send/Data indications are used, and whether any additional attributes (such as the DONT-FRAGMENT attribute) are included. Another factor, which is hard to determine, is whether the MTU is reduced somewhere along the path for other reasons, such as the use of IP-in-IP tunneling.

As a guideline, sending a maximum of 500 bytes of application data in a single TURN message (by the client on the client-to-server leg) or a UDP datagram (by the peer on the peer-to-server leg) will generally avoid IP fragmentation. To further reduce the chance of fragmentation, it is recommended that the client use ChannelData messages when transferring significant volumes of data, since the overhead of the ChannelData message is less than Send and Data indications.

The second approach the client and peer can take to avoid fragmentation is to use a path MTU discovery algorithm to determine the maximum amount of application data that can be sent without fragmentation. The classic path MTU discovery algorithm defined in [RFC1191] may not be able to discover the MTU of the transmission path between the client and the peer since:

- a probe packet with DF bit set to test a path for a larger MTU can be dropped by routers, or

- ICMP error messages can be dropped by middle boxes.

As a result, the client and server need to use a path MTU discovery algorithm that does not require ICMP messages. The Packetized Path MTU Discovery algorithm defined in [RFC4821] is one such algorithm.

[I-D.ietf-tram-stun-pmtud] is an implementation of [RFC4821] that is using STUN to discover the PMTUD, and so may be a suitable approach to be used in conjunction with a TURN server, together with the DONT-FRAGMENT attribute. When the client includes the DONT-FRAGMENT attribute in a Send indication, this tells the server to set the DF bit in the resulting UDP datagram that it sends to the peer. Since some servers may be unable to set the DF bit, the client should also include this attribute in the Allocate request -- any server that

does not support the DONT-FRAGMENT attribute will indicate this by rejecting the Allocate request.

2.8. RTP Support

One of the envisioned uses of TURN is as a relay for clients and peers wishing to exchange real-time data (e.g., voice or video) using RTP. To facilitate the use of TURN for this purpose, TURN includes some special support for older versions of RTP.

Old versions of RTP [RFC3550] required that the RTP stream be on an even port number and the associated RTP Control Protocol (RTCP) stream, if present, be on the next highest port. To allow clients to work with peers that still require this, TURN allows the client to request that the server allocate a relayed transport address with an even port number, and to optionally request the server reserve the next-highest port number for a subsequent allocation.

2.9. Discovery of TURN server

Methods of TURN server discovery, including using anycast, are described in [I-D.ietf-tram-turn-server-discovery]. The syntax of the "turn" and "turn" URIs are defined in Section 3.1 of [RFC7065].

2.9.1. TURN URI Scheme Semantics

The "turn" and "turns" URI schemes are used to designate a TURN server (also known as a relay) on Internet hosts accessible using the TURN protocol. The TURN protocol supports sending messages over UDP, TCP, TLS-over-TCP or DTLS-over-UDP. The "turns" URI scheme MUST be used when TURN is run over TLS-over-TCP or in DTLS-over-UDP, and the "turn" scheme MUST be used otherwise. The required <host> part of the "turn" URI denotes the TURN server host. The <port> part, if present, denotes the port on which the TURN server is awaiting connection requests. If it is absent, the default port is 3478 for both UDP and TCP. The default port for TURN over TLS and TURN over DTLS is 5349.

3. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

Readers are expected to be familiar with [RFC5389] and the terms defined there.

The following terms are used in this document:

TURN: The protocol spoken between a TURN client and a TURN server. It is an extension to the STUN protocol [RFC5389]. The protocol allows a client to allocate and use a relayed transport address.

TURN client: A STUN client that implements this specification.

TURN server: A STUN server that implements this specification. It relays data between a TURN client and its peer(s).

Peer: A host with which the TURN client wishes to communicate. The TURN server relays traffic between the TURN client and its peer(s). The peer does not interact with the TURN server using the protocol defined in this document; rather, the peer receives data sent by the TURN server and the peer sends data towards the TURN server.

Transport Address: The combination of an IP address and a port.

Host Transport Address: A transport address on a client or a peer.

Server-Reflexive Transport Address: A transport address on the "public side" of a NAT. This address is allocated by the NAT to correspond to a specific host transport address.

Relayed Transport Address: A transport address on the TURN server that is used for relaying packets between the client and a peer. A peer sends to this address on the TURN server, and the packet is then relayed to the client.

TURN Server Transport Address: A transport address on the TURN server that is used for sending TURN messages to the server. This is the transport address that the client uses to communicate with the server.

Peer Transport Address: The transport address of the peer as seen by the server. When the peer is behind a NAT, this is the peer’s server-reflexive transport address.

Allocation: The relayed transport address granted to a client through an Allocate request, along with related state, such as permissions and expiration timers.

5-tuple: The combination (client IP address and port, server IP address and port, and transport protocol (currently one of UDP, TCP, or (D)TLS)) used to communicate between the client and the server. The 5-tuple uniquely identifies this communication stream. The 5-tuple also uniquely identifies the Allocation on the server.

Channel: A channel number and associated peer transport address. Once a channel number is bound to a peer's transport address, the client and server can use the more bandwidth-efficient ChannelData message to exchange data.

Permission: The IP address and transport protocol (but not the port) of a peer that is permitted to send traffic to the TURN server and have that traffic relayed to the TURN client. The TURN server will only forward traffic to its client from peers that match an existing permission.

Realm: A string used to describe the server or a context within the server. The realm tells the client which username and password combination to use to authenticate requests.

Nonce: A string chosen at random by the server and included in the message-digest. To prevent reply attacks, the server should change the nonce regularly.

4. General Behavior

This section contains general TURN processing rules that apply to all TURN messages.

TURN is an extension to STUN. All TURN messages, with the exception of the ChannelData message, are STUN-formatted messages. All the base processing rules described in [RFC5389] apply to STUN-formatted messages. This means that all the message-forming and message-processing descriptions in this document are implicitly prefixed with the rules of [RFC5389].

[RFC5389] specifies an authentication mechanism called the long-term credential mechanism. TURN servers and clients MUST implement this mechanism. The server MUST demand that all requests from the client be authenticated using this mechanism, or that a equally strong or stronger mechanism for client authentication is used.

Note that the long-term credential mechanism applies only to requests and cannot be used to authenticate indications; thus, indications in TURN are never authenticated. If the server requires requests to be authenticated, then the server's administrator MUST choose a realm value that will uniquely identify the username and password combination that the client must use, even if the client uses multiple servers under different administrations. The server's administrator MAY choose to allocate a unique username to each client, or MAY choose to allocate the same username to more than one client (for example, to all clients from the same department or company). For each allocation, the server SHOULD generate a new

random nonce when the allocation is first attempted following the randomness recommendations in [RFC4086] and SHOULD expire the nonce at least once every hour during the lifetime of the allocation.

All requests after the initial Allocate must use the same username as that used to create the allocation, to prevent attackers from hijacking the client's allocation. Specifically, if the server requires the use of the long-term credential mechanism, and if a non-Allocate request passes authentication under this mechanism, and if the 5-tuple identifies an existing allocation, but the request does not use the same username as used to create the allocation, then the request MUST be rejected with a 441 (Wrong Credentials) error.

When a TURN message arrives at the server from the client, the server uses the 5-tuple in the message to identify the associated allocation. For all TURN messages (including ChannelData) EXCEPT an Allocate request, if the 5-tuple does not identify an existing allocation, then the message MUST either be rejected with a 437 Allocation Mismatch error (if it is a request) or silently ignored (if it is an indication or a ChannelData message). A client receiving a 437 error response to a request other than Allocate MUST assume the allocation no longer exists.

[RFC5389] defines a number of attributes, including the SOFTWARE and FINGERPRINT attributes. The client SHOULD include the SOFTWARE attribute in all Allocate and Refresh requests and MAY include it in any other requests or indications. The server SHOULD include the SOFTWARE attribute in all Allocate and Refresh responses (either success or failure) and MAY include it in other responses or indications. The client and the server MAY include the FINGERPRINT attribute in any STUN-formatted messages defined in this document.

TURN does not use the backwards-compatibility mechanism described in [RFC5389].

TURN, as defined in this specification, supports both IPv4 and IPv6. IPv6 support in TURN includes IPv4-to-IPv6, IPv6-to-IPv6, and IPv6-to-IPv4 relaying. The REQUESTED-ADDRESS-FAMILY attribute allows a client to explicitly request the address type the TURN server will allocate (e.g., an IPv4-only node may request the TURN server to allocate an IPv6 address). The ADDITIONAL-ADDRESS-FAMILY attribute allows a client to request the server to allocate one IPv4 and one IPv6 relay address in a single Allocate request. This saves local ports on the client and reduces the number of messages sent between the client and the TURN server.

By default, TURN runs on the same ports as STUN: 3478 for TURN over UDP and TCP, and 5349 for TURN over (D)TLS. However, TURN has its

own set of Service Record (SRV) names: "turn" for UDP and TCP, and "turns" for (D)TLS. Either the SRV procedures or the ALTERNATE-SERVER procedures, both described in Section 6, can be used to run TURN on a different port.

To ensure interoperability, a TURN server MUST support the use of UDP transport between the client and the server, and SHOULD support the use of TCP and (D)TLS transport.

When UDP transport is used between the client and the server, the client will retransmit a request if it does not receive a response within a certain timeout period. Because of this, the server may receive two (or more) requests with the same 5-tuple and same transaction id. STUN requires that the server recognize this case and treat the request as idempotent (see [RFC5389]). Some implementations may choose to meet this requirement by remembering all received requests and the corresponding responses for 40 seconds. Other implementations may choose to reprocess the request and arrange that such reprocessing returns essentially the same response. To aid implementors who choose the latter approach (the so-called "stateless stack approach"), this specification includes some implementation notes on how this might be done. Implementations are free to choose either approach or choose some other approach that gives the same results.

When TCP transport is used between the client and the server, it is possible that a bit error will cause a length field in a TURN packet to become corrupted, causing the receiver to lose synchronization with the incoming stream of TURN messages. A client or server that detects a long sequence of invalid TURN messages over TCP transport SHOULD close the corresponding TCP connection to help the other end detect this situation more rapidly.

To mitigate either intentional or unintentional denial-of-service attacks against the server by clients with valid usernames and passwords, it is RECOMMENDED that the server impose limits on both the number of allocations active at one time for a given username and on the amount of bandwidth those allocations can use. The server should reject new allocations that would exceed the limit on the allowed number of allocations active at one time with a 486 (Allocation Quota Exceeded) (see Section 6.2), and should discard application data traffic that exceeds the bandwidth quota.

5. Allocations

All TURN operations revolve around allocations, and all TURN messages are associated with an allocation. An allocation conceptually consists of the following state data:

- o the relayed transport address;
- o the 5-tuple: (client's IP address, client's port, server IP address, server port, transport protocol);
- o the authentication information;
- o the time-to-expiry;
- o a list of permissions;
- o a list of channel to peer bindings.

The relayed transport address is the transport address allocated by the server for communicating with peers, while the 5-tuple describes the communication path between the client and the server. On the client, the 5-tuple uses the client's host transport address; on the server, the 5-tuple uses the client's server-reflexive transport address.

Both the relayed transport address and the 5-tuple MUST be unique across all allocations, so either one can be used to uniquely identify the allocation.

The authentication information (e.g., username, password, realm, and nonce) is used to both verify subsequent requests and to compute the message integrity of responses. The username, realm, and nonce values are initially those used in the authenticated Allocate request that creates the allocation, though the server can change the nonce value during the lifetime of the allocation using a 438 (Stale Nonce) reply. Note that, rather than storing the password explicitly, for security reasons, it may be desirable for the server to store the key value, which is an secure hash over the username, realm, and password (see [I-D.ietf-tram-stunbis]).

The time-to-expiry is the time in seconds left until the allocation expires. Each Allocate or Refresh transaction sets this timer, which then ticks down towards 0. By default, each Allocate or Refresh transaction resets this timer to the default lifetime value of 600 seconds (10 minutes), but the client can request a different value in the Allocate and Refresh request. Allocations can only be refreshed using the Refresh request; sending data to a peer does not refresh an allocation. When an allocation expires, the state data associated with the allocation can be freed.

The list of permissions is described in Section 8 and the list of channels is described in Section 11.

6. Creating an Allocation

An allocation on the server is created using an Allocate transaction.

6.1. Sending an Allocate Request

The client forms an Allocate request as follows.

The client first picks a host transport address. It is RECOMMENDED that the client pick a currently unused transport address, typically by allowing the underlying OS to pick a currently unused port for a new socket.

The client then picks a transport protocol to use between the client and the server. The transport protocol MUST be one of UDP, TCP, TLS-over-TCP or DTLS-over-UDP. Since this specification only allows UDP between the server and the peers, it is RECOMMENDED that the client pick UDP unless it has a reason to use a different transport. One reason to pick a different transport would be that the client believes, either through configuration or by experiment, that it is unable to contact any TURN server using UDP. See Section 2.1 for more discussion.

The client also picks a server transport address, which SHOULD be done as follows. The client uses the procedures described in [I-D.ietf-tram-turn-server-discovery] to discover a TURN server and TURN server resolution mechanism defined in [RFC5928] to get a list of server transport addresses that can be tried to create a TURN allocation.

The client MUST include a REQUESTED-TRANSPORT attribute in the request. This attribute specifies the transport protocol between the server and the peers (note that this is NOT the transport protocol that appears in the 5-tuple). In this specification, the REQUESTED-TRANSPORT type is always UDP. This attribute is included to allow future extensions to specify other protocols.

If the client wishes to obtain a relayed transport address of a specific address type then it includes a REQUESTED-ADDRESS-FAMILY attribute in the request. This attribute indicates the specific address type the client wishes the TURN server to allocate. Clients MUST NOT include more than one REQUESTED-ADDRESS-FAMILY attribute in an Allocate request. Clients MUST NOT include a REQUESTED-ADDRESS-FAMILY attribute in an Allocate request that contains a RESERVATION-TOKEN attribute, for the reasons outlined in [RFC6156].

If the client wishes to obtain one IPv6 and one IPv4 relayed transport addresses then it includes an ADDITIONAL-ADDRESS-FAMILY

attribute in the request. This attribute specifies that the server must allocate both address types. The attribute value in the ADDITIONAL-ADDRESS-FAMILY MUST be set to 0x02 (IPv6 address family). Clients MUST NOT include REQUESTED-ADDRESS-FAMILY and ADDITIONAL-ADDRESS-FAMILY attributes in the same request. Clients MUST NOT include ADDITIONAL-ADDRESS-FAMILY attribute in a Allocate request that contains a RESERVATION-TOKEN attribute. Clients MUST NOT include ADDITIONAL-ADDRESS-FAMILY attribute in a Allocate request that contains a EVEN-PORT attribute with the R bit set to 1.

If the client wishes the server to initialize the time-to-expiry field of the allocation to some value other than the default lifetime, then it MAY include a LIFETIME attribute specifying its desired value. This is just a hint, and the server may elect to use a different value. Note that the server will ignore requests to initialize the field to less than the default value.

If the client wishes to later use the DONT-FRAGMENT attribute in one or more Send indications on this allocation, then the client SHOULD include the DONT-FRAGMENT attribute in the Allocate request. This allows the client to test whether this attribute is supported by the server.

If the client requires the port number of the relayed transport address be even, the client includes the EVEN-PORT attribute. If this attribute is not included, then the port can be even or odd. By setting the R bit in the EVEN-PORT attribute to 1, the client can request that the server reserve the next highest port number (on the same IP address) for a subsequent allocation. If the R bit is 0, no such request is made.

The client MAY also include a RESERVATION-TOKEN attribute in the request to ask the server to use a previously reserved port for the allocation. If the RESERVATION-TOKEN attribute is included, then the client MUST omit the EVEN-PORT attribute.

Once constructed, the client sends the Allocate request on the 5-tuple.

6.2. Receiving an Allocate Request

When the server receives an Allocate request, it performs the following checks:

1. The server MUST require that the request be authenticated. This authentication MUST be done using the long-term credential mechanism of [RFC5389] unless the client and server agree to use

another mechanism through some procedure outside the scope of this document.

2. The server checks if the 5-tuple is currently in use by an existing allocation. If yes, the server rejects the request with a 437 (Allocation Mismatch) error.

3. The server checks if the request contains a REQUESTED-TRANSPORT attribute. If the REQUESTED-TRANSPORT attribute is not included or is malformed, the server rejects the request with a 400 (Bad Request) error. Otherwise, if the attribute is included but specifies a protocol other that UDP, the server rejects the request with a 442 (Unsupported Transport Protocol) error.

4. The request may contain a DONT-FRAGMENT attribute. If it does, but the server does not support sending UDP datagrams with the DF bit set to 1 (see Section 13), then the server treats the DONT-FRAGMENT attribute in the Allocate request as an unknown comprehension-required attribute.

5. The server checks if the request contains a RESERVATION-TOKEN attribute. If yes, and the request also contains an EVEN-PORT or REQUESTED-ADDRESS-FAMILY or ADDITIONAL-ADDRESS-FAMILY attribute, the server rejects the request with a 400 (Bad Request) error. Otherwise, it checks to see if the token is valid (i.e., the token is in range and has not expired and the corresponding relayed transport address is still available). If the token is not valid for some reason, the server rejects the request with a 508 (Insufficient Capacity) error.

6. The server checks if the request contains both REQUESTED-ADDRESS-FAMILY and ADDITIONAL-ADDRESS-FAMILY attributes, then the server rejects the request with a 400 (Bad Request) error.

7. If the server does not support the address family requested by the client in REQUESTED-ADDRESS-FAMILY or is disabled by local policy, it MUST generate an Allocate error response, and it MUST include an ERROR-CODE attribute with the 440 (Address Family not Supported) response code. If the REQUESTED-ADDRESS-FAMILY attribute is absent, the server MUST allocate an IPv4 relayed transport address for the TURN client.

8. The server checks if the request contains an EVEN-PORT attribute with the R bit set to 1. If yes, and the request also contains an ADDITIONAL- ADDRESS-FAMILY attribute, the server rejects the request with a 400 (Bad Request) error. Otherwise, the server checks if it can satisfy the request (i.e., can allocate a relayed transport address as described below). If the server

cannot satisfy the request, then the server rejects the request with a 508 (Insufficient Capacity) error.

9. The server checks if the request contains an ADDITIONAL-ADDRESS-FAMILY attribute. If yes, and the attribute value is 0x01 (IPv4 address family), then the server rejects the request with a 400 (Bad Request) error. Otherwise, and the server checks if it can allocate relayed transport addresses of both address types. If the server cannot satisfy the request, then the server rejects the request with a 508 (Insufficient Capacity) error. If the server can partially meet the request, i.e. if it can only allocate one relayed transport address of a specific address type, then it includes ADDRESS-ERROR-CODE attribute in the response to inform the client the reason for partial failure of the request. The error code value signaled in the ADDRESS-ERROR-CODE attribute could be 440 (Address Family not Supported) or 508 (Insufficient Capacity).

10. At any point, the server MAY choose to reject the request with a 486 (Allocation Quota Reached) error if it feels the client is trying to exceed some locally defined allocation quota. The server is free to define this allocation quota any way it wishes, but SHOULD define it based on the username used to authenticate the request, and not on the client’s transport address.

11. Also at any point, the server MAY choose to reject the request with a 300 (Try Alternate) error if it wishes to redirect the client to a different server. The use of this error code and attribute follow the specification in [RFC5389].

If all the checks pass, the server creates the allocation. The 5-tuple is set to the 5-tuple from the Allocate request, while the list of permissions and the list of channels are initially empty.

The server chooses a relayed transport address for the allocation as follows:

- If the request contains a RESERVATION-TOKEN attribute, the server uses the previously reserved transport address corresponding to the included token (if it is still available). Note that the reservation is a server-wide reservation and is not specific to a particular allocation, since the Allocate request containing the RESERVATION-TOKEN uses a different 5-tuple than the Allocate request that made the reservation. The 5-tuple for the Allocate request containing the RESERVATION-TOKEN attribute can be any allowed 5-tuple; it can use a different client IP address and port, a different transport protocol, and even different server IP

address and port (provided, of course, that the server IP address and port are ones on which the server is listening for TURN requests).

- o If the request contains an EVEN-PORT attribute with the R bit set to 0, then the server allocates a relayed transport address with an even port number.

- o If the request contains an EVEN-PORT attribute with the R bit set to 1, then the server looks for a pair of port numbers N and N+1 on the same IP address, where N is even. Port N is used in the current allocation, while the relayed transport address with port N+1 is assigned a token and reserved for a future allocation. The server MUST hold this reservation for at least 30 seconds, and MAY choose to hold longer (e.g., until the allocation with port N expires). The server then includes the token in a RESERVATION-TOKEN attribute in the success response.

- o Otherwise, the server allocates any available relayed transport address.

In all cases, the server SHOULD only allocate ports from the range 49152 - 65535 (the Dynamic and/or Private Port range [Port-Numbers]), unless the TURN server application knows, through some means not specified here, that other applications running on the same host as the TURN server application will not be impacted by allocating ports outside this range. This condition can often be satisfied by running the TURN server application on a dedicated machine and/or by arranging that any other applications on the machine allocate ports before the TURN server application starts. In any case, the TURN server SHOULD NOT allocate ports in the range 0 - 1023 (the Well-Known Port range) to discourage clients from using TURN to run standard services.

NOTE: The use of randomized port assignments to avoid certain types of attacks is described in [RFC6056]. It is RECOMMENDED that a TURN server implement a randomized port assignment algorithm from [RFC6056]. This is especially applicable to servers that choose to pre-allocate a number of ports from the underlying OS and then later assign them to allocations; for example, a server may choose this technique to implement the EVEN-PORT attribute.

The server determines the initial value of the time-to-expiry field as follows. If the request contains a LIFETIME attribute, then the server computes the minimum of the client’s proposed lifetime and the server’s maximum allowed lifetime. If this computed value is greater than the default lifetime, then the server uses the computed lifetime

as the initial value of the time-to-expiry field. Otherwise, the server uses the default lifetime. It is RECOMMENDED that the server use a maximum allowed lifetime value of no more than 3600 seconds (1 hour). Servers that implement allocation quotas or charge users for allocations in some way may wish to use a smaller maximum allowed lifetime (perhaps as small as the default lifetime) to more quickly remove orphaned allocations (that is, allocations where the corresponding client has crashed or terminated or the client connection has been lost for some reason). Also, note that the time-to-expiry is recomputed with each successful Refresh request, and thus the value computed here applies only until the first refresh.

Once the allocation is created, the server replies with a success response. The success response contains:

- An XOR-RELAYED-ADDRESS attribute containing the relayed transport address.

- A LIFETIME attribute containing the current value of the time-to-expiry timer.

- A RESERVATION-TOKEN attribute (if a second relayed transport address was reserved).

- An XOR-MAPPED-ADDRESS attribute containing the client’s IP address and port (from the 5-tuple).

  NOTE: The XOR-MAPPED-ADDRESS attribute is included in the response as a convenience to the client. TURN itself does not make use of this value, but clients running ICE can often need this value and can thus avoid having to do an extra Binding transaction with some STUN server to learn it.

The response (either success or error) is sent back to the client on the 5-tuple.

NOTE: When the Allocate request is sent over UDP, section 7.3.1 of [RFC5389] requires that the server handle the possible retransmissions of the request so that retransmissions do not cause multiple allocations to be created. Implementations may achieve this using the so-called "stateless stack approach" as follows. To detect retransmissions when the original request was successful in creating an allocation, the server can store the transaction id that created the request with the allocation data and compare it with incoming Allocate requests on the same 5-tuple. Once such a request is detected, the server can stop parsing the request and immediately generate a success response. When building this response, the value of the LIFETIME attribute

can be taken from the time-to-expiry field in the allocate state data, even though this value may differ slightly from the LIFETIME value originally returned. In addition, the server may need to store an indication of any reservation token returned in the original response, so that this may be returned in any retransmitted responses.

For the case where the original request was unsuccessful in creating an allocation, the server may choose to do nothing special. Note, however, that there is a rare case where the server rejects the original request but accepts the retransmitted request (because conditions have changed in the brief intervening time period). If the client receives the first failure response, it will ignore the second (success) response and believe that an allocation was not created. An allocation created in this matter will eventually timeout, since the client will not refresh it. Furthermore, if the client later retries with the same 5-tuple but different transaction id, it will receive a 437 (Allocation Mismatch), which will cause it to retry with a different 5-tuple. The server may use a smaller maximum lifetime value to minimize the lifetime of allocations "orphaned" in this manner.

6.3. Receiving an Allocate Success Response

If the client receives an Allocate success response, then it MUST check that the mapped address and the relayed transport address are part of an address family that the client understands and is prepared to handle. If these two addresses are not part of an address family which the client is prepared to handle, then the client MUST delete the allocation (Section 7) and MUST NOT attempt to create another allocation on that server until it believes the mismatch has been fixed.

Otherwise, the client creates its own copy of the allocation data structure to track what is happening on the server. In particular, the client needs to remember the actual lifetime received back from the server, rather than the value sent to the server in the request. The client must also remember the 5-tuple used for the request and the username and password it used to authenticate the request to ensure that it reuses them for subsequent messages. The client also needs to track the channels and permissions it establishes on the server.

The client will probably wish to send the relayed transport address to peers (using some method not specified here) so the peers can communicate with it. The client may also wish to use the server-reflexive address it receives in the XOR-MAPPED-ADDRESS attribute in its ICE processing.

6.4. Receiving an Allocate Error Response

If the client receives an Allocate error response, then the processing depends on the actual error code returned:

- (Request timed out): There is either a problem with the server, or a problem reaching the server with the chosen transport. The client considers the current transaction as having failed but MAY choose to retry the Allocate request using a different transport (e.g., TCP instead of UDP).

- 300 (Try Alternate): The server would like the client to use the server specified in the ALTERNATE-SERVER attribute instead. The client considers the current transaction as having failed, but SHOULD try the Allocate request with the alternate server before trying any other servers (e.g., other servers discovered using the SRV procedures). When trying the Allocate request with the alternate server, the client follows the ALTERNATE-SERVER procedures specified in [RFC5389].

- 400 (Bad Request): The server believes the client’s request is malformed for some reason. The client considers the current transaction as having failed. The client MAY notify the user or operator and SHOULD NOT retry the request with this server until it believes the problem has been fixed.

- 401 (Unauthorized): If the client has followed the procedures of the long-term credential mechanism and still gets this error, then the server is not accepting the client’s credentials. In this case, the client considers the current transaction as having failed and SHOULD notify the user or operator. The client SHOULD NOT send any further requests to this server until it believes the problem has been fixed.

- 403 (Forbidden): The request is valid, but the server is refusing to perform it, likely due to administrative restrictions. The client considers the current transaction as having failed. The client MAY notify the user or operator and SHOULD NOT retry the same request with this server until it believes the problem has been fixed.

- 420 (Unknown Attribute): If the client included a DONT-FRAGMENT attribute in the request and the server rejected the request with a 420 error code and listed the DONT-FRAGMENT attribute in the UNKNOWN-ATTRIBUTES attribute in the error response, then the client now knows that the server does not support the DONT-FRAGMENT attribute. The client considers the current transaction

as having failed but MAY choose to retry the Allocate request without the DONT-FRAGMENT attribute.

- o 437 (Allocation Mismatch): This indicates that the client has picked a 5-tuple that the server sees as already in use. One way this could happen is if an intervening NAT assigned a mapped transport address that was used by another client that recently crashed. The client considers the current transaction as having failed. The client SHOULD pick another client transport address and retry the Allocate request (using a different transaction id). The client SHOULD try three different client transport addresses before giving up on this server. Once the client gives up on the server, it SHOULD NOT try to create another allocation on the server for 2 minutes.

- o 438 (Stale Nonce): See the procedures for the long-term credential mechanism [RFC5389].

- o 440 (Address Family not Supported): The server does not support the address family requested by the client. If the client receives an Allocate error response with the 440 (Unsupported Address Family) error code, the client MUST NOT retry the request.

- o 441 (Wrong Credentials): The client should not receive this error in response to a Allocate request. The client MAY notify the user or operator and SHOULD NOT retry the same request with this server until it believes the problem has been fixed.

- o 442 (Unsupported Transport Address): The client should not receive this error in response to a request for a UDP allocation. The client MAY notify the user or operator and SHOULD NOT reattempt the request with this server until it believes the problem has been fixed.

- o 486 (Allocation Quota Reached): The server is currently unable to create any more allocations with this username. The client considers the current transaction as having failed. The client SHOULD wait at least 1 minute before trying to create any more allocations on the server.

- o 508 (Insufficient Capacity): The server has no more relayed transport addresses available, or has none with the requested properties, or the one that was reserved is no longer available. The client considers the current operation as having failed. If the client is using either the EVEN-PORT or the RESERVATION-TOKEN attribute, then the client MAY choose to remove or modify this attribute and try again immediately. Otherwise, the client SHOULD

wait at least 1 minute before trying to create any more allocations on this server.

An unknown error response MUST be handled as described in [RFC5389].

7. Refreshing an Allocation

A Refresh transaction can be used to either (a) refresh an existing allocation and update its time-to-expiry or (b) delete an existing allocation.

If a client wishes to continue using an allocation, then the client MUST refresh it before it expires. It is suggested that the client refresh the allocation roughly 1 minute before it expires. If a client no longer wishes to use an allocation, then it SHOULD explicitly delete the allocation. A client MAY refresh an allocation at any time for other reasons.

7.1. Sending a Refresh Request

If the client wishes to immediately delete an existing allocation, it includes a LIFETIME attribute with a value of 0. All other forms of the request refresh the allocation.

When refreshing a dual allocation, the client includes REQUESTED-ADDRESS-FAMILY attribute indicating the address family type that should be refreshed. If no REQUESTED-ADDRESS-FAMILY is included then the request should be treated as applying to all current allocations. The client MUST only include family types it previously allocated and has not yet deleted. This process can also be used to delete an allocation of a specific address type, by setting the lifetime of that refresh request to 0. Deleting a single allocation destroys any permissions or channels associated with that particular allocation; it MUST NOT affect any permissions or channels associated with allocations for the other address family.

The Refresh transaction updates the time-to-expiry timer of an allocation. If the client wishes the server to set the time-to-expiry timer to something other than the default lifetime, it includes a LIFETIME attribute with the requested value. The server then computes a new time-to-expiry value in the same way as it does for an Allocate transaction, with the exception that a requested lifetime of 0 causes the server to immediately delete the allocation.

7.2. Receiving a Refresh Request

When the server receives a Refresh request, it processes it as per Section 4 plus the specific rules mentioned here.

If the server receives a Refresh Request with an REQUESTED-ADDRESS-FAMILY attribute and the attribute value does not match the address family of the allocation, the server MUST reply with a 443 (Peer Address Family Mismatch) Refresh error response.

The server computes a value called the "desired lifetime" as follows: if the request contains a LIFETIME attribute and the attribute value is 0, then the "desired lifetime" is 0. Otherwise, if the request contains a LIFETIME attribute, then the server computes the minimum of the client’s requested lifetime and the server’s maximum allowed lifetime. If this computed value is greater than the default lifetime, then the "desired lifetime" is the computed value. Otherwise, the "desired lifetime" is the default lifetime.

Subsequent processing depends on the "desired lifetime" value:

- o If the "desired lifetime" is 0, then the request succeeds and the allocation is deleted.

- o If the "desired lifetime" is non-zero, then the request succeeds and the allocation’s time-to-expiry is set to the "desired lifetime".

If the request succeeds, then the server sends a success response containing:

- o A LIFETIME attribute containing the current value of the time-to-expiry timer.

  NOTE: A server need not do anything special to implement idempotency of Refresh requests over UDP using the "stateless stack approach". Retransmitted Refresh requests with a non-zero "desired lifetime" will simply refresh the allocation. A retransmitted Refresh request with a zero "desired lifetime" will cause a 437 (Allocation Mismatch) response if the allocation has already been deleted, but the client will treat this as equivalent to a success response (see below).

7.3. Receiving a Refresh Response

If the client receives a success response to its Refresh request with a non-zero lifetime, it updates its copy of the allocation data structure with the time-to-expiry value contained in the response.

If the client receives a 437 (Allocation Mismatch) error response to a request to delete the allocation, then the allocation no longer exists and it should consider its request as having effectively succeeded.

8. Permissions

For each allocation, the server keeps a list of zero or more permissions. Each permission consists of an IP address and an associated time-to-expiry. While a permission exists, all peers using the IP address in the permission are allowed to send data to the client. The time-to-expiry is the number of seconds until the permission expires. Within the context of an allocation, a permission is uniquely identified by its associated IP address.

By sending either CreatePermission requests or ChannelBind requests, the client can cause the server to install or refresh a permission for a given IP address. This causes one of two things to happen:

- o If no permission for that IP address exists, then a permission is created with the given IP address and a time-to-expiry equal to Permission Lifetime.

- o If a permission for that IP address already exists, then the time-to-expiry for that permission is reset to Permission Lifetime.

The Permission Lifetime MUST be 300 seconds (= 5 minutes).

Each permission’s time-to-expiry decreases down once per second until it reaches 0; at which point, the permission expires and is deleted.

CreatePermission and ChannelBind requests may be freely intermixed on a permission. A given permission may be initially installed and/or refreshed with a CreatePermission request, and then later refreshed with a ChannelBind request, or vice versa.

When a UDP datagram arrives at the relayed transport address for the allocation, the server extracts the source IP address from the IP header. The server then compares this address with the IP address associated with each permission in the list of permissions for the allocation. If no match is found, relaying is not permitted, and the server silently discards the UDP datagram. If an exact match is found, then the permission check is considered to have succeeded and the server continues to process the UDP datagram as specified elsewhere (Section 10.3). Note that only addresses are compared and port numbers are not considered.

The permissions for one allocation are totally unrelated to the permissions for a different allocation. If an allocation expires, all its permissions expire with it.

NOTE: Though TURN permissions expire after 5 minutes, many NATs deployed at the time of publication expire their UDP bindings considerably faster. Thus, an application using TURN will probably wish to send some sort of keep-alive traffic at a much faster rate. Applications using ICE should follow the keep-alive guidelines of ICE [RFC5245], and applications not using ICE are advised to do something similar.

9. CreatePermission

TURN supports two ways for the client to install or refresh permissions on the server. This section describes one way: the CreatePermission request.

A CreatePermission request may be used in conjunction with either the Send mechanism in Section 10 or the Channel mechanism in Section 11.

9.1. Forming a CreatePermission Request

The client who wishes to install or refresh one or more permissions can send a CreatePermission request to the server.

When forming a CreatePermission request, the client MUST include at least one XOR-PEER-ADDRESS attribute, and MAY include more than one such attribute. The IP address portion of each XOR-PEER-ADDRESS attribute contains the IP address for which a permission should be installed or refreshed. The port portion of each XOR-PEER-ADDRESS attribute will be ignored and can be any arbitrary value. The various XOR-PEER-ADDRESS attributes can appear in any order. The client MUST only include XOR-PEER-ADDRESS attributes with addresses of the same address family as that of the relayed transport address for the allocation. For dual allocations obtained using the ADDITIONAL-FAMILY-ADDRESS attribute, the client can include XOR-PEER-ADDRESS attributes with addresses of IPv4 and IPv6 address families.

9.2. Receiving a CreatePermission Request

When the server receives the CreatePermission request, it processes as per Section 4 plus the specific rules mentioned here.

The message is checked for validity. The CreatePermission request MUST contain at least one XOR-PEER-ADDRESS attribute and MAY contain multiple such attributes. If no such attribute exists, or if any of these attributes are invalid, then a 400 (Bad Request) error is

returned. If the request is valid, but the server is unable to satisfy the request due to some capacity limit or similar, then a 508 (Insufficient Capacity) error is returned.

If an XOR-PEER-ADDRESS attribute contains an address of an address family that is not the same as that of the relayed transport address for the allocation, the server MUST generate an error response with the 443 (Peer Address Family Mismatch) response code.

The server MAY impose restrictions on the IP address allowed in the XOR-PEER-ADDRESS attribute -- if a value is not allowed, the server rejects the request with a 403 (Forbidden) error.

If the message is valid and the server is capable of carrying out the request, then the server installs or refreshes a permission for the IP address contained in each XOR-PEER-ADDRESS attribute as described in Section 8. The port portion of each attribute is ignored and may be any arbitrary value.

The server then responds with a CreatePermission success response. There are no mandatory attributes in the success response.

> NOTE: A server need not do anything special to implement idempotency of CreatePermission requests over UDP using the "stateless stack approach". Retransmitted CreatePermission requests will simply refresh the permissions.

9.3. Receiving a CreatePermission Response

If the client receives a valid CreatePermission success response, then the client updates its data structures to indicate that the permissions have been installed or refreshed.

10. Send and Data Methods

TURN supports two mechanisms for sending and receiving data from peers. This section describes the use of the Send and Data mechanisms, while Section 11 describes the use of the Channel mechanism.

10.1. Forming a Send Indication

The client can use a Send indication to pass data to the server for relaying to a peer. A client may use a Send indication even if a channel is bound to that peer. However, the client MUST ensure that there is a permission installed for the IP address of the peer to which the Send indication is being sent; this prevents a third party from using a TURN server to send data to arbitrary destinations.

When forming a Send indication, the client MUST include an XOR-PEER-ADDRESS attribute and a DATA attribute. The XOR-PEER-ADDRESS attribute contains the transport address of the peer to which the data is to be sent, and the DATA attribute contains the actual application data to be sent to the peer.

The client MAY include a DONT-FRAGMENT attribute in the Send indication if it wishes the server to set the DF bit on the UDP datagram sent to the peer.

10.2. Receiving a Send Indication

When the server receives a Send indication, it processes as per Section 4 plus the specific rules mentioned here.

The message is first checked for validity. The Send indication MUST contain both an XOR-PEER-ADDRESS attribute and a DATA attribute. If one of these attributes is missing or invalid, then the message is discarded. Note that the DATA attribute is allowed to contain zero bytes of data.

The Send indication may also contain the DONT-FRAGMENT attribute. If the server is unable to set the DF bit on outgoing UDP datagrams when this attribute is present, then the server acts as if the DONT-FRAGMENT attribute is an unknown comprehension-required attribute (and thus the Send indication is discarded).

The server also checks that there is a permission installed for the IP address contained in the XOR-PEER-ADDRESS attribute. If no such permission exists, the message is discarded. Note that a Send indication never causes the server to refresh the permission.

The server MAY impose restrictions on the IP address and port values allowed in the XOR-PEER-ADDRESS attribute -- if a value is not allowed, the server silently discards the Send indication.

If everything is OK, then the server forms a UDP datagram as follows:

- o the source transport address is the relayed transport address of the allocation, where the allocation is determined by the 5-tuple on which the Send indication arrived;

- o the destination transport address is taken from the XOR-PEER-ADDRESS attribute;

- o the data following the UDP header is the contents of the value field of the DATA attribute.

The handling of the DONT-FRAGMENT attribute (if present), is described in Section 13.

The resulting UDP datagram is then sent to the peer.

10.3. Receiving a UDP Datagram

When the server receives a UDP datagram at a currently allocated relayed transport address, the server looks up the allocation associated with the relayed transport address. The server then checks to see whether the set of permissions for the allocation allow the relaying of the UDP datagram as described in Section 8.

If relaying is permitted, then the server checks if there is a channel bound to the peer that sent the UDP datagram (see Section 11). If a channel is bound, then processing proceeds as described in Section 11.7.

If relaying is permitted but no channel is bound to the peer, then the server forms and sends a Data indication. The Data indication MUST contain both an XOR-PEER-ADDRESS and a DATA attribute. The DATA attribute is set to the value of the 'data octets' field from the datagram, and the XOR-PEER-ADDRESS attribute is set to the source transport address of the received UDP datagram. The Data indication is then sent on the 5-tuple associated with the allocation.

10.4. Receiving a Data Indication with DATA attribute

When the client receives a Data indication with DATA attribute, it checks that the Data indication contains an XOR-PEER-ADDRESS attribute, and discards the indication if it does not. The client SHOULD also check that the XOR-PEER-ADDRESS attribute value contains an IP address with which the client believes there is an active permission, and discard the Data indication otherwise. Note that the DATA attribute is allowed to contain zero bytes of data.

> NOTE: The latter check protects the client against an attacker who somehow manages to trick the server into installing permissions not desired by the client.

If the Data indication passes the above checks, the client delivers the data octets inside the DATA attribute to the application, along with an indication that they were received from the peer whose transport address is given by the XOR-PEER-ADDRESS attribute.

10.5. Receiving an ICMP Packet

When the server receives an ICMP packet, the server verifies that the type is either 3, 11 or 12 for an ICMPv4 [RFC0792] packet or either 1, 2, or 3 for an ICMPv6 [RFC4443] packet. It also verifies that the IP packet in the ICMP packet payload contains a UDP header. If either of these conditions fail, then the ICMP packet is silently dropped.

The server looks up the allocation whose relayed transport address corresponds to the encapsulated packet's source IP address and UDP port. If no such allocation exists, the packet is silently dropped. The server then checks to see whether the set of permissions for the allocation allows the relaying of the ICMP packet. For ICMP packets, the source IP address MUST NOT be checked against the permissions list as it would be for UDP packets. Instead, the server extracts the destination IP address from the encapsulated IP header. The server then compares this address with the IP address associated with each permission in the list of permissions for the allocation. If no match is found, relaying is not permitted, and the server silently discards the ICMP packet. Note that only addresses are compared and port numbers are not considered.

If relaying is permitted then the server forms and sends a Data indication. The Data indication MUST contain both an XOR-PEER-ADDRESS and an ICMP attribute. The ICMP attribute is set to the value of the type and code fields from the ICMP packet. The IP address portion of XOR-PEER-ADDRESS attribute is set to the destination IP address in the encapsulated IP header. At the time of writing of this specification, Socket APIs on some operating systems do not deliver the destination port in the encapsulated UDP header to applications without superuser privileges. If destination port in the encapsulated UDP header is available to the server then the port portion of XOR-PEER-ADDRESS attribute is set to the destination port otherwise the port portion is set to 0. The Data indication is then sent on the 5-tuple associated with the allocation.

10.6. Receiving a Data Indication with an ICMP attribute

When the client receives a Data indication with an ICMP attribute, it checks that the Data indication contains an XOR-PEER-ADDRESS attribute, and discards the indication if it does not. The client SHOULD also check that the XOR-PEER-ADDRESS attribute value contains an IP address with an active permission, and discard the Data indication otherwise.

If the Data indication passes the above checks, the client signals the application of the error condition, along with an indication that

it was received from the peer whose transport address is given by the XOR-PEER-ADDRESS attribute. The application can make sense of the meaning of the type and code values in the ICMP attribute by using the family field in the XOR-PEER-ADDRESS attribute.

11. Channels

Channels provide a way for the client and server to send application data using ChannelData messages, which have less overhead than Send and Data indications.

The ChannelData message (see Section 11.4) starts with a two-byte field that carries the channel number. The values of this field are allocated as follows:

0x0000 through 0x3FFF: These values can never be used for channel numbers.

0x4000 through 0x7FFF: These values are the allowed channel numbers (16,384 possible values).

0x8000 through 0xFFFF: These values are reserved for future use.

Because of this division, ChannelData messages can be distinguished from STUN-formatted messages (e.g., Allocate request, Send indication, etc.) by examining the first two bits of the message:

0b00: STUN-formatted message (since the first two bits of a STUN-formatted message are always zero).

0b01: ChannelData message (since the channel number is the first field in the ChannelData message and channel numbers fall in the range 0x4000 - 0x7FFF).

0b10: Reserved

0b11: Reserved

The reserved values may be used in the future to extend the range of channel numbers. Thus, an implementation MUST NOT assume that a TURN message always starts with a 0 bit.

Channel bindings are always initiated by the client. The client can bind a channel to a peer at any time during the lifetime of the allocation. The client may bind a channel to a peer before exchanging data with it, or after exchanging data with it (using Send and Data indications) for some time, or may choose never to bind a

channel to it. The client can also bind channels to some peers while not binding channels to other peers.

Channel bindings are specific to an allocation, so that the use of a channel number or peer transport address in a channel binding in one allocation has no impact on their use in a different allocation. If an allocation expires, all its channel bindings expire with it.

A channel binding consists of:

- o a channel number;
- o a transport address (of the peer); and
- o A time-to-expiry timer.

Within the context of an allocation, a channel binding is uniquely identified either by the channel number or by the peer's transport address. Thus, the same channel cannot be bound to two different transport addresses, nor can the same transport address be bound to two different channels.

A channel binding lasts for 10 minutes unless refreshed. Refreshing the binding (by the server receiving a ChannelBind request rebinding the channel to the same peer) resets the time-to-expiry timer back to 10 minutes.

When the channel binding expires, the channel becomes unbound. Once unbound, the channel number can be bound to a different transport address, and the transport address can be bound to a different channel number. To prevent race conditions, the client MUST wait 5 minutes after the channel binding expires before attempting to bind the channel number to a different transport address or the transport address to a different channel number.

When binding a channel to a peer, the client SHOULD be prepared to receive ChannelData messages on the channel from the server as soon as it has sent the ChannelBind request. Over UDP, it is possible for the client to receive ChannelData messages from the server before it receives a ChannelBind success response.

In the other direction, the client MAY elect to send ChannelData messages before receiving the ChannelBind success response. Doing so, however, runs the risk of having the ChannelData messages dropped by the server if the ChannelBind request does not succeed for some reason (e.g., packet lost if the request is sent over UDP, or the server being unable to fulfill the request). A client that wishes to

be safe should either queue the data or use Send indications until the channel binding is confirmed.

11.1. Sending a ChannelBind Request

A channel binding is created or refreshed using a ChannelBind transaction. A ChannelBind transaction also creates or refreshes a permission towards the peer (see Section 8).

To initiate the ChannelBind transaction, the client forms a ChannelBind request. The channel to be bound is specified in a CHANNEL-NUMBER attribute, and the peer’s transport address is specified in an XOR-PEER-ADDRESS attribute. Section 11.2 describes the restrictions on these attributes. The client MUST only include an XOR-PEER-ADDRESS attribute with an address of the same address family as that of the relayed transport address for the allocation.

Rebinding a channel to the same transport address that it is already bound to provides a way to refresh a channel binding and the corresponding permission without sending data to the peer. Note however, that permissions need to be refreshed more frequently than channels.

11.2. Receiving a ChannelBind Request

When the server receives a ChannelBind request, it processes as per Section 4 plus the specific rules mentioned here.

The server checks the following:

- o The request contains both a CHANNEL-NUMBER and an XOR-PEER-ADDRESS attribute;

- o The channel number is in the range 0x4000 through 0x7FFE (inclusive);

- o The channel number is not currently bound to a different transport address (same transport address is OK);

- o The transport address is not currently bound to a different channel number.

- o If the XOR-PEER-ADDRESS attribute contains an address of an address family that is not the same as that of the relayed transport address for the allocation, the server MUST generate an error response with the 443 (Peer Address Family Mismatch) response code.

If any of these tests fail, the server replies with a 400 (Bad Request) error.

The server MAY impose restrictions on the IP address and port values allowed in the XOR-PEER-ADDRESS attribute -- if a value is not allowed, the server rejects the request with a 403 (Forbidden) error.

If the request is valid, but the server is unable to fulfill the request due to some capacity limit or similar, the server replies with a 508 (Insufficient Capacity) error.

Otherwise, the server replies with a ChannelBind success response. There are no required attributes in a successful ChannelBind response.

If the server can satisfy the request, then the server creates or refreshes the channel binding using the channel number in the CHANNEL-NUMBER attribute and the transport address in the XOR-PEER-ADDRESS attribute. The server also installs or refreshes a permission for the IP address in the XOR-PEER-ADDRESS attribute as described in Section 8.

> NOTE: A server need not do anything special to implement idempotency of ChannelBind requests over UDP using the "stateless stack approach". Retransmitted ChannelBind requests will simply refresh the channel binding and the corresponding permission. Furthermore, the client must wait 5 minutes before binding a previously bound channel number or peer address to a different channel, eliminating the possibility that the transaction would initially fail but succeed on a retransmission.

11.3. Receiving a ChannelBind Response

When the client receives a ChannelBind success response, it updates its data structures to record that the channel binding is now active. It also updates its data structures to record that the corresponding permission has been installed or refreshed.

If the client receives a ChannelBind failure response that indicates that the channel information is out-of-sync between the client and the server (e.g., an unexpected 400 "Bad Request" response), then it is RECOMMENDED that the client immediately delete the allocation and start afresh with a new allocation.

11.4. The ChannelData Message

The ChannelData message is used to carry application data between the client and the server. It has the following format:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Channel Number        |            Length             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
/                       Application Data                        /
/                                                               /
|                                                               |
|                               +-------------------------------+
|                               |
+-------------------------------+
```

The Channel Number field specifies the number of the channel on which the data is traveling, and thus the address of the peer that is sending or is to receive the data.

The Length field specifies the length in bytes of the application data field (i.e., it does not include the size of the ChannelData header). Note that 0 is a valid length.

The Application Data field carries the data the client is trying to send to the peer, or that the peer is sending to the client.

11.5. Sending a ChannelData Message

Once a client has bound a channel to a peer, then when the client has data to send to that peer it may use either a ChannelData message or a Send indication; that is, the client is not obligated to use the channel when it exists and may freely intermix the two message types when sending data to the peer. The server, on the other hand, MUST use the ChannelData message if a channel has been bound to the peer. The server uses a Data indication to signal the XOR-PEER-ADDRESS and ICMP attributes to the client even if a channel has been bound to the peer.

The fields of the ChannelData message are filled in as described in Section 11.4.

Over TCP and TLS-over-TCP, the ChannelData message MUST be padded to a multiple of four bytes in order to ensure the alignment of subsequent messages. The padding is not reflected in the length field of the ChannelData message, so the actual size of a ChannelData

message (including padding) is (4 + Length) rounded up to the nearest multiple of 4. Over UDP, the padding is not required but MAY be included.

The ChannelData message is then sent on the 5-tuple associated with the allocation.

11.6. Receiving a ChannelData Message

The receiver of the ChannelData message uses the first two bits to distinguish it from STUN-formatted messages, as described above. If the message uses a value in the reserved range (0x8000 through 0xFFFF), then the message is silently discarded.

If the ChannelData message is received in a UDP datagram, and if the UDP datagram is too short to contain the claimed length of the ChannelData message (i.e., the UDP header length field value is less than the ChannelData header length field value + 4 + 8), then the message is silently discarded.

If the ChannelData message is received over TCP or over TLS-over-TCP, then the actual length of the ChannelData message is as described in Section 11.5.

If the ChannelData message is received on a channel that is not bound to any peer, then the message is silently discarded.

On the client, it is RECOMMENDED that the client discard the ChannelData message if the client believes there is no active permission towards the peer. On the server, the receipt of a ChannelData message MUST NOT refresh either the channel binding or the permission towards the peer.

On the server, if no errors are detected, the server relays the application data to the peer by forming a UDP datagram as follows:

- o the source transport address is the relayed transport address of the allocation, where the allocation is determined by the 5-tuple on which the ChannelData message arrived;

- o the destination transport address is the transport address to which the channel is bound;

- o the data following the UDP header is the contents of the data field of the ChannelData message.

The resulting UDP datagram is then sent to the peer. Note that if the Length field in the ChannelData message is 0, then there will be

no data in the UDP datagram, but the UDP datagram is still formed and sent.

11.7. Relaying Data from the Peer

When the server receives a UDP datagram on the relayed transport address associated with an allocation, the server processes it as described in Section 10.3. If that section indicates that a ChannelData message should be sent (because there is a channel bound to the peer that sent to the UDP datagram), then the server forms and sends a ChannelData message as described in Section 11.5.

When the server receives an ICMP packet, the server processes it as described in Section 10.5. A Data indication MUST be sent regardless if there is a channel bound to the peer that was the destination of the UDP datagram that triggered the reception of the ICMP packet.

12. Packet Translations

As discussed in Section 2.6, translations in TURN are designed so that a TURN server can be implemented as an application that runs in userland under commonly available operating systems and that does not require special privileges. The translations specified in the following sections follow this principle.

The descriptions below have two parts: a preferred behavior and an alternate behavior. The server SHOULD implement the preferred behavior. Otherwise, the server MUST implement the alternate behavior and MUST NOT do anything else for the reasons detailed in [RFC6145].

12.1. IPv4-to-IPv6 Translations

Traffic Class

Preferred behavior: As specified in Section 4 of [RFC6145].

Alternate behavior: The relay sets the Traffic Class to the default value for outgoing packets.

Flow Label

Preferred behavior: The relay sets the Flow label to 0. The relay can choose to set the Flow label to a different value if it supports the IPv6 Flow Label field[RFC3697].

Alternate behavior: the relay sets the Flow label to the default value for outgoing packets.

Hop Limit

Preferred behavior: As specified in Section 4 of [RFC6145].

Alternate behavior: The relay sets the Hop Limit to the default value for outgoing packets.

Fragmentation

Preferred behavior: As specified in Section 4 of [RFC6145].

Alternate behavior: The relay assembles incoming fragments. The relay follows its default behavior to send outgoing packets.

For both preferred and alternate behavior, the DONT-FRAGMENT attribute MUST be ignored by the server.

Extension Headers

Preferred behavior: The relay sends outgoing packet without any IPv6 extension headers, with the exception of the Fragmentation header as described above.

Alternate behavior: Same as preferred.

12.2. IPv6-to-IPv6 Translations

Flow Label

The relay should consider that it is handling two different IPv6 flows. Therefore, the Flow label [RFC3697] SHOULD NOT be copied as part of the translation.

Preferred behavior: The relay sets the Flow label to 0. The relay can choose to set the Flow label to a different value if it supports the IPv6 Flow Label field[RFC3697].

Alternate behavior: The relay sets the Flow label to the default value for outgoing packets.

Hop Limit

Preferred behavior: The relay acts as a regular router with respect to decrementing the Hop Limit and generating an ICMPv6 error if it reaches zero.

Alternate behavior: The relay sets the Hop Limit to the default value for outgoing packets.

Fragmentation

Preferred behavior: If the incoming packet did not include a Fragment header and the outgoing packet size does not exceed the outgoing link’s MTU, the relay sends the outgoing packet without a Fragment header.

If the incoming packet did not include a Fragment header and the outgoing packet size exceeds the outgoing link’s MTU, the relay drops the outgoing packet and send an ICMP message of type 2 code 0 ("Packet too big") to the sender of the incoming packet. If the packet is being sent to the peer, the relay reduces the MTU reported in the ICMP message by 48 bytes to allow room for the overhead of a Data indication.

If the incoming packet included a Fragment header and the outgoing packet size (with a Fragment header included) does not exceed the outgoing link’s MTU, the relay sends the outgoing packet with a Fragment header. The relay sets the fields of the Fragment header as appropriate for a packet originating from the server.

If the incoming packet included a Fragment header and the outgoing packet size exceeds the outgoing link’s MTU, the relay MUST fragment the outgoing packet into fragments of no more than 1280 bytes. The relay sets the fields of the Fragment header as appropriate for a packet originating from the server.

Alternate behavior: The relay assembles incoming fragments. The relay follows its default behavior to send outgoing packets.

For both preferred and alternate behavior, the DONT-FRAGMENT attribute MUST be ignored by the server.

Extension Headers

Preferred behavior: The relay sends outgoing packet without any IPv6 extension headers, with the exception of the Fragmentation header as described above.

Alternate behavior: Same as preferred.

12.3. IPv6-to-IPv4 Translations

Type of Service and Precedence

Preferred behavior: As specified in Section 5 of [RFC6145].

Alternate behavior: The relay sets the Type of Service and Precedence to the default value for outgoing packets.

Time to Live

Preferred behavior: As specified in Section 5 of [RFC6145].

Alternate behavior: The relay sets the Time to Live to the default value for outgoing packets.

Fragmentation

Preferred behavior: As specified in Section 5 of [RFC6145]. Additionally, when the outgoing packet's size exceeds the outgoing link's MTU, the relay needs to generate an ICMP error (ICMPv6 Packet Too Big) reporting the MTU size. If the packet is being sent to the peer, the relay SHOULD reduce the MTU reported in the ICMP message by 48 bytes to allow room for the overhead of a Data indication.

Alternate behavior: The relay assembles incoming fragments. The relay follows its default behavior to send outgoing packets.

For both preferred and alternate behavior, the DONT-FRAGMENT attribute MUST be ignored by the server.

13. IP Header Fields

This section describes how the server sets various fields in the IP header when relaying between the client and the peer or vice versa. The descriptions in this section apply: (a) when the server sends a UDP datagram to the peer, or (b) when the server sends a Data indication or ChannelData message to the client over UDP transport. The descriptions in this section do not apply to TURN messages sent over TCP or TLS transport from the server to the client.

The descriptions below have two parts: a preferred behavior and an alternate behavior. The server SHOULD implement the preferred behavior, but if that is not possible for a particular field, then it SHOULD implement the alternative behavior.

Time to Live (TTL) field

Preferred Behavior: If the incoming value is 0, then the drop the incoming packet. Otherwise, set the outgoing Time to Live/Hop Count to one less than the incoming value.

Alternate Behavior: Set the outgoing value to the default for outgoing packets.

Differentiated Services Code Point (DSCP) field [RFC2474]

Preferred Behavior: Set the outgoing value to the incoming value, unless the server includes a differentiated services classifier and marker [RFC2474].

Alternate Behavior: Set the outgoing value to a fixed value, which by default is Best Effort unless configured otherwise.

In both cases, if the server is immediately adjacent to a differentiated services classifier and marker, then DSCP MAY be set to any arbitrary value in the direction towards the classifier.

Explicit Congestion Notification (ECN) field [RFC3168]

Preferred Behavior: Set the outgoing value to the incoming value, UNLESS the server is doing Active Queue Management, the incoming ECN field is ECT(1) (=0b01) or ECT(0) (=0b10), and the server wishes to indicate that congestion has been experienced, in which case set the outgoing value to CE (=0b11).

Alternate Behavior: Set the outgoing value to Not-ECT (=0b00).

IPv4 Fragmentation fields

Preferred Behavior: When the server sends a packet to a peer in response to a Send indication containing the DONT-FRAGMENT attribute, then set the DF bit in the outgoing IP header to 1. In all other cases when sending an outgoing packet containing application data (e.g., Data indication, ChannelData message, or DONT-FRAGMENT attribute not included in the Send indication), copy the DF bit from the DF bit of the incoming packet that contained the application data.

Set the other fragmentation fields (Identification, More Fragments, Fragment Offset) as appropriate for a packet originating from the server.

Alternate Behavior: As described in the Preferred Behavior, except always assume the incoming DF bit is 0.

In both the Preferred and Alternate Behaviors, the resulting packet may be too large for the outgoing link. If this is the case, then the normal fragmentation rules apply [RFC1122].

IPv4 Options

Preferred Behavior: The outgoing packet is sent without any IPv4 options.

Alternate Behavior: Same as preferred.

14. New STUN Methods

This section lists the codepoints for the new STUN methods defined in this specification. See elsewhere in this document for the semantics of these new methods.

```
0x003  :  Allocate          (only request/response semantics defined)
0x004  :  Refresh           (only request/response semantics defined)
0x006  :  Send              (only indication semantics defined)
0x007  :  Data              (only indication semantics defined)
0x008  :  CreatePermission  (only request/response semantics defined
0x009  :  ChannelBind       (only request/response semantics defined)
```

15. New STUN Attributes

This STUN extension defines the following new attributes:

```
0x000C: CHANNEL-NUMBER
0x000D: LIFETIME
0x0010: Reserved (was BANDWIDTH)
0x0012: XOR-PEER-ADDRESS
0x0013: DATA
0x0016: XOR-RELAYED-ADDRESS
0x0017: REQUESTED-ADDRESS-FAMILY
0x0018: EVEN-PORT
0x0019: REQUESTED-TRANSPORT
0x001A: DONT-FRAGMENT
0x0021: Reserved (was TIMER-VAL)
0x0022: RESERVATION-TOKEN
TBD-CA: ADDITIONAL-ADDRESS-FAMILY
TBD-CA: ADDRESS-ERROR-CODE
TBD-CA: ICMP
```

Some of these attributes have lengths that are not multiples of 4. By the rules of STUN, any attribute whose length is not a multiple of

4 bytes MUST be immediately followed by 1 to 3 padding bytes to ensure the next attribute (if any) would start on a 4-byte boundary (see [RFC5389]).

15.1. CHANNEL-NUMBER

The CHANNEL-NUMBER attribute contains the number of the channel. The value portion of this attribute is 4 bytes long and consists of a 16-bit unsigned integer, followed by a two-octet RFFU (Reserved For Future Use) field, which MUST be set to 0 on transmission and MUST be ignored on reception.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Channel Number         |         RFFU = 0              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

15.2. LIFETIME

The LIFETIME attribute represents the duration for which the server will maintain an allocation in the absence of a refresh. The value portion of this attribute is 4-bytes long and consists of a 32-bit unsigned integral value representing the number of seconds remaining until expiration.

15.3. XOR-PEER-ADDRESS

The XOR-PEER-ADDRESS specifies the address and port of the peer as seen from the TURN server. (For example, the peer’s server-reflexive transport address if the peer is behind a NAT.) It is encoded in the same way as XOR-MAPPED-ADDRESS [RFC5389].

15.4. DATA

The DATA attribute is present in all Send and Data indications. The value portion of this attribute is variable length and consists of the application data (that is, the data that would immediately follow the UDP header if the data was been sent directly between the client and the peer). If the length of this attribute is not a multiple of 4, then padding must be added after this attribute.

15.5. XOR-RELAYED-ADDRESS

The XOR-RELAYED-ADDRESS is present in Allocate responses. It specifies the address and port that the server allocated to the client. It is encoded in the same way as XOR-MAPPED-ADDRESS [RFC5389].

15.6. REQUESTED-ADDRESS-FAMILY

This attribute is used by clients to request the allocation of a specific address type from a server. The following is the format of the REQUESTED-ADDRESS-FAMILY attribute. Note that TURN attributes are TLV (Type-Length-Value) encoded, with a 16-bit type, a 16-bit length, and a variable-length value.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Type                  |            Length             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Family    |            Reserved                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Type: the type of the REQUESTED-ADDRESS-FAMILY attribute is 0x0017. As specified in [RFC5389], attributes with values between 0x0000 and 0x7FFF are comprehension-required, which means that the client or server cannot successfully process the message unless it understands the attribute.

Length: this 16-bit field contains the length of the attribute in bytes. The length of this attribute is 4 bytes.

Family: there are two values defined for this field and specified in [RFC5389], Section 15.1: 0x01 for IPv4 addresses and 0x02 for IPv6 addresses.

Reserved: at this point, the 24 bits in the Reserved field MUST be set to zero by the client and MUST be ignored by the server.

The REQUEST-ADDRESS-TYPE attribute MAY only be present in Allocate requests.

15.7. EVEN-PORT

This attribute allows the client to request that the port in the relayed transport address be even, and (optionally) that the server reserve the next-higher port number. The value portion of this attribute is 1 byte long. Its format is:

```
 0
 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+
|R|    RFFU     |
+-+-+-+-+-+-+-+-+
```

The value contains a single 1-bit flag:

R: If 1, the server is requested to reserve the next-higher port number (on the same IP address) for a subsequent allocation. If 0, no such reservation is requested.

The other 7 bits of the attribute's value must be set to zero on transmission and ignored on reception.

Since the length of this attribute is not a multiple of 4, padding must immediately follow this attribute.

15.8. REQUESTED-TRANSPORT

This attribute is used by the client to request a specific transport protocol for the allocated transport address. The value of this attribute is 4 bytes with the following format:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Protocol   |                    RFFU                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The Protocol field specifies the desired protocol. The codepoints used in this field are taken from those allowed in the Protocol field in the IPv4 header and the NextHeader field in the IPv6 header [Protocol-Numbers]. This specification only allows the use of codepoint 17 (User Datagram Protocol).

The RFFU field MUST be set to zero on transmission and MUST be ignored on reception. It is reserved for future uses.

15.9. DONT-FRAGMENT

This attribute is used by the client to request that the server set the DF (Don't Fragment) bit in the IP header when relaying the application data onward to the peer. This attribute has no value part and thus the attribute length field is 0.

15.10. RESERVATION-TOKEN

The RESERVATION-TOKEN attribute contains a token that uniquely identifies a relayed transport address being held in reserve by the server. The server includes this attribute in a success response to tell the client about the token, and the client includes this attribute in a subsequent Allocate request to request the server use that relayed transport address for the allocation.

The attribute value is 8 bytes and contains the token value.

15.11. ADDITIONAL-ADDRESS-FAMILY

This attribute is used by clients to request the allocation of a IPv4 and IPv6 address type from a server. It is encoded in the same way as REQUESTED-ADDRESS-FAMILY Section 15.6. The ADDITIONAL-ADDRESS-FAMILY attribute MAY be present in Allocate request. The attribute value of 0x02 (IPv6 address) is the only valid value in Allocate request.

15.12. ADDRESS-ERROR-CODE Attribute

This attribute is used by servers to signal the reason for not allocating the requested address family. The following is the format of the ADDRESS-ERROR-CODE attribute.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Type                  |            Length             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Family       |    Rsvd                 |Class|     Number    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Reason Phrase (variable)                                ..
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Type: the type of the ADDRESS-ERROR-CODE attribute is TBD-CA. As specified in [RFC5389], attributes with values between 0x8000 and 0xFFFF are comprehension-optional, which means that the client or server can safely ignore the attribute if they don’t understand it.

Length: this 16-bit field contains the length of the attribute in bytes.

Family: there are two values defined for this field and specified in [RFC5389], Section 15.1: 0x01 for IPv4 addresses and 0x02 for IPv6 addresses.

Reserved: at this point, the 13 bits in the Reserved field MUST be set to zero by the client and MUST be ignored by the server.

Class: The Class represents the hundreds digit of the error code and is defined in section 15.6 of [RFC5389].

Number: this 8-bit field contains the reason server cannot allocate one of the requested address types. The error code values could be either 440 (unsupported address family) or 508 (insufficient capacity). The number representation is defined in section 15.6 of [RFC5389].

Reason Phrase: The recommended reason phrases for error codes 440 and 508 are explained in Section 16.

The ADDRESS-ERROR-CODE attribute MAY only be present in Allocate responses.

15.13. ICMP Attribute

This attribute is used by servers to signal the reason an UDP packet was dropped. The following is the format of the ICMP attribute.

```
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Reserved                     |  Type       |  Code           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Reserved: This field MUST be set to 0 when sent, and MUST be ignored when received.

Type: The field contains the value in the ICMP type. Its interpretation depends whether the ICMP was received over IPv4 or IPv6.

Code: The field contains the value in the ICMP code. Its interpretation depends whether the ICMP was received over IPv4 or IPv6.

16. New STUN Error Response Codes

This document defines the following new error response codes:

403 (Forbidden): The request was valid but cannot be performed due to administrative or similar restrictions.

437 (Allocation Mismatch): A request was received by the server that requires an allocation to be in place, but no allocation exists, or a request was received that requires no allocation, but an allocation exists.

440 (Address Family not Supported): The server does not support the address family requested by the client.

441 (Wrong Credentials): The credentials in the (non-Allocate) request do not match those used to create the allocation.

442 (Unsupported Transport Protocol): The Allocate request asked the server to use a transport protocol between the server and the peer that the server does not support. NOTE: This does NOT refer to the transport protocol used in the 5-tuple.

443 (Peer Address Family Mismatch). A peer address is part of a different address family than that of the relayed transport address of the allocation.

486 (Allocation Quota Reached): No more allocations using this username can be created at the present time.

508 (Insufficient Capacity): The server is unable to carry out the request due to some capacity limit being reached. In an Allocate response, this could be due to the server having no more relayed transport addresses available at that time, having none with the requested properties, or the one that corresponds to the specified reservation token is not available.

17. Detailed Example

This section gives an example of the use of TURN, showing in detail the contents of the messages exchanged. The example uses the network diagram shown in the Overview (Figure 1).

For each message, the attributes included in the message and their values are shown. For convenience, values are shown in a human-readable format rather than showing the actual octets; for example, "XOR-RELAYED-ADDRESS=192.0.2.15:9000" shows that the XOR-RELAYED-ADDRESS attribute is included with an address of 192.0.2.15 and a port of 9000, here the address and port are shown before the xor-ing is done. For attributes with string-like values (e.g., SOFTWARE="Example client, version 1.03" and NONCE="adl7W7PeDU4hKE72jdaQvbAMcr6h39sm"), the value of the attribute is shown in quotes for readability, but these quotes do not appear in the actual value.

```
  TURN                                  TURN           Peer          Peer
  client                                server          A             B
    |                                     |             |             |
    |--- Allocate request --------------->|             |             |
    |    Transaction-Id=0xA56250D3F17ABE679422DE85      |             |
    |    SOFTWARE="Example client, version 1.03"        |             |
    |    LIFETIME=3600 (1 hour)           |             |             |
    |    REQUESTED-TRANSPORT=17 (UDP)     |             |             |
    |    DONT-FRAGMENT                    |             |             |
    |                                     |             |             |
    |<-- Allocate error response --------|             |             |
    |    Transaction-Id=0xA56250D3F17ABE679422DE85      |             |
    |    SOFTWARE="Example server, version 1.17"        |             |
    |    ERROR-CODE=401 (Unauthorized)    |             |             |
    |    REALM="example.com"              |             |             |
    |    NONCE="adl7W7PeDU4hKE72jdaQvbAMcr6h39sm"       |             |
    |                                     |             |             |
    |--- Allocate request --------------->|             |             |
    |    Transaction-Id=0xC271E932AD7446A32C234492      |             |
    |    SOFTWARE="Example client 1.03"   |             |             |
    |    LIFETIME=3600 (1 hour)           |             |             |
    |    REQUESTED-TRANSPORT=17 (UDP)     |             |             |
    |    DONT-FRAGMENT                    |             |             |
    |    USERNAME="George"                |             |             |
    |    REALM="example.com"              |             |             |
    |    NONCE="adl7W7PeDU4hKE72jdaQvbAMcr6h39sm"       |             |
    |    MESSAGE-INTEGRITY=...            |             |             |
    |                                     |             |             |
    |<-- Allocate success response ------|             |             |
    |    Transaction-Id=0xC271E932AD7446A32C234492      |             |
    |    SOFTWARE="Example server, version 1.17"        |             |
    |    LIFETIME=1200 (20 minutes)       |             |             |
    |    XOR-RELAYED-ADDRESS=192.0.2.15:50000           |             |
    |    XOR-MAPPED-ADDRESS=192.0.2.1:7000              |             |
    |    MESSAGE-INTEGRITY=...            |             |             |
```

The client begins by selecting a host transport address to use for the TURN session; in this example, the client has selected 10.1.1.2:49721 as shown in Figure 1. The client then sends an Allocate request to the server at the server transport address. The client randomly selects a 96-bit transaction id of 0xA56250D3F17ABE679422DE85 for this transaction; this is encoded in the transaction id field in the fixed header. The client includes a SOFTWARE attribute that gives information about the client’s software; here the value is "Example client, version 1.03" to indicate that this is version 1.03 of something called the Example client. The client includes the LIFETIME attribute because it wishes the allocation to have a longer lifetime than the default of 10

minutes; the value of this attribute is 3600 seconds, which corresponds to 1 hour. The client must always include a REQUESTED-TRANSPORT attribute in an Allocate request and the only value allowed by this specification is 17, which indicates UDP transport between the server and the peers. The client also includes the DONT-FRAGMENT attribute because it wishes to use the DONT-FRAGMENT attribute later in Send indications; this attribute consists of only an attribute header, there is no value part. We assume the client has not recently interacted with the server, thus the client does not include USERNAME, REALM, NONCE, or MESSAGE-INTEGRITY attribute. Finally, note that the order of attributes in a message is arbitrary (except for the MESSAGE-INTEGRITY and FINGERPRINT attributes) and the client could have used a different order.

Servers require any request to be authenticated. Thus, when the server receives the initial Allocate request, it rejects the request because the request does not contain the authentication attributes. Following the procedures of the long-term credential mechanism of STUN [RFC5389], the server includes an ERROR-CODE attribute with a value of 401 (Unauthorized), a REALM attribute that specifies the authentication realm used by the server (in this case, the server’s domain "example.com"), and a nonce value in a NONCE attribute. The server also includes a SOFTWARE attribute that gives information about the server’s software.

The client, upon receipt of the 401 error, re-attempts the Allocate request, this time including the authentication attributes. The client selects a new transaction id, and then populates the new Allocate request with the same attributes as before. The client includes a USERNAME attribute and uses the realm value received from the server to help it determine which value to use; here the client is configured to use the username "George" for the realm "example.com". The client also includes the REALM and NONCE attributes, which are just copied from the 401 error response. Finally, the client includes a MESSAGE-INTEGRITY attribute as the last attribute in the message, whose value is a Hashed Message Authentication Code - Secure Hash Algorithm 1 (HMAC-SHA1) hash over the contents of the message (shown as just "..." above); this HMAC-SHA1 computation includes a password value. Thus, an attacker cannot compute the message integrity value without somehow knowing the secret password.

The server, upon receipt of the authenticated Allocate request, checks that everything is OK, then creates an allocation. The server replies with an Allocate success response. The server includes a LIFETIME attribute giving the lifetime of the allocation; here, the server has reduced the client’s requested 1-hour lifetime to just 20 minutes, because this particular server doesn’t allow lifetimes

longer than 20 minutes. The server includes an XOR-RELAYED-ADDRESS attribute whose value is the relayed transport address of the allocation. The server includes an XOR-MAPPED-ADDRESS attribute whose value is the server-reflexive address of the client; this value is not used otherwise in TURN but is returned as a convenience to the client. The server includes a MESSAGE-INTEGRITY attribute to authenticate the response and to ensure its integrity; note that the response does not contain the USERNAME, REALM, and NONCE attributes. The server also includes a SOFTWARE attribute.

```
TURN                                  TURN           Peer           Peer
client                                server          A              B
  |--- CreatePermission request ------>|             |              |
  |    Transaction-Id=0xE5913A8F460956CA277D3319     |              |
  |    XOR-PEER-ADDRESS=192.0.2.150:0  |             |              |
  |    USERNAME="George"               |             |              |
  |    REALM="example.com"             |             |              |
  |    NONCE="adl7W7PeDU4hKE72jdaQvbAMcr6h39sm"      |              |
  |    MESSAGE-INTEGRITY=...           |             |              |
  |                                    |             |              |
  |<-- CreatePermission success resp.--|             |              |
  |    Transaction-Id=0xE5913A8F460956CA277D3319     |              |
  |    MESSAGE-INTEGRITY=...           |             |              |
```

The client then creates a permission towards Peer A in preparation for sending it some application data. This is done through a CreatePermission request. The XOR-PEER-ADDRESS attribute contains the IP address for which a permission is established (the IP address of peer A); note that the port number in the attribute is ignored when used in a CreatePermission request, and here it has been set to 0; also, note how the client uses Peer A’s server-reflexive IP address and not its (private) host address. The client uses the same username, realm, and nonce values as in the previous request on the allocation. Though it is allowed to do so, the client has chosen not to include a SOFTWARE attribute in this request.

The server receives the CreatePermission request, creates the corresponding permission, and then replies with a CreatePermission success response. Like the client, the server chooses not to include the SOFTWARE attribute in its reply. Again, note how success responses contain a MESSAGE-INTEGRITY attribute (assuming the server uses the long-term credential mechanism), but no USERNAME, REALM, and NONCE attributes.

```
  TURN                                  TURN           Peer          Peer
  client                                server          A             B
    |--- Send indication --------------->|             |             |
    |    Transaction-Id=0x1278E9ACA2711637EF7D3328     |             |
    |    XOR-PEER-ADDRESS=192.0.2.150:32102            |             |
    |    DONT-FRAGMENT                   |             |             |
    |    DATA=...                        |             |             |
    |                                    |-- UDP dgm ->|             |
    |                                    |  data=...   |             |
    |                                    |             |             |
    |                                    |<- UDP dgm --|             |
    |                                    |  data=...   |             |
    |<-- Data indication ----------------|             |             |
    |    Transaction-Id=0x8231AE8F9242DA9FF287FEFF     |             |
    |    XOR-PEER-ADDRESS=192.0.2.150:32102            |             |
    |    DATA=...                        |             |             |
```

The client now sends application data to Peer A using a Send indication. Peer A’s server-reflexive transport address is specified in the XOR-PEER-ADDRESS attribute, and the application data (shown here as just "...") is specified in the DATA attribute. The client is doing a form of path MTU discovery at the application layer and thus specifies (by including the DONT-FRAGMENT attribute) that the server should set the DF bit in the UDP datagram to send to the peer. Indications cannot be authenticated using the long-term credential mechanism of STUN, so no MESSAGE-INTEGRITY attribute is included in the message. An application wishing to ensure that its data is not altered or forged must integrity-protect its data at the application level.

Upon receipt of the Send indication, the server extracts the application data and sends it in a UDP datagram to Peer A, with the relayed transport address as the source transport address of the datagram, and with the DF bit set as requested. Note that, had the client not previously established a permission for Peer A’s server-reflexive IP address, then the server would have silently discarded the Send indication instead.

Peer A then replies with its own UDP datagram containing application data. The datagram is sent to the relayed transport address on the server. When this arrives, the server creates a Data indication containing the source of the UDP datagram in the XOR-PEER-ADDRESS attribute, and the data from the UDP datagram in the DATA attribute. The resulting Data indication is then sent to the client.

```
  TURN                                  TURN           Peer          Peer
  client                                server          A             B
    |--- ChannelBind request ----------->|             |             |
    |    Transaction-Id=0x6490D3BC175AFF3D84513212     |             |
    |    CHANNEL-NUMBER=0x4000           |             |             |
    |    XOR-PEER-ADDRESS=192.0.2.210:49191            |             |
    |    USERNAME="George"               |             |             |
    |    REALM="example.com"             |             |             |
    |    NONCE="adl7W7PeDU4hKE72jdaQvbAMcr6h39sm"      |             |
    |    MESSAGE-INTEGRITY=...           |             |             |
    |                                    |             |             |
    |<-- ChannelBind success response ---|             |             |
    |    Transaction-Id=0x6490D3BC175AFF3D84513212     |             |
    |    MESSAGE-INTEGRITY=...           |             |             |
```

The client now binds a channel to Peer B, specifying a free channel number (0x4000) in the CHANNEL-NUMBER attribute, and Peer B's transport address in the XOR-PEER-ADDRESS attribute. As before, the client re-uses the username, realm, and nonce from its last request in the message.

Upon receipt of the request, the server binds the channel number to the peer, installs a permission for Peer B's IP address, and then replies with ChannelBind success response.

```
  TURN                                  TURN           Peer          Peer
  client                                server          A             B
    |--- ChannelData -------------------->|             |             |
    |    Channel-number=0x4000           |--- UDP datagram --------->|
    |    Data=...                        |    Data=...               |
    |                                    |             |             |
    |                                    |<-- UDP datagram ----------|
    |                                    |    Data=... |             |
    |<-- ChannelData --------------------|             |             |
    |    Channel-number=0x4000           |             |             |
    |    Data=...                        |             |             |
```

The client now sends a ChannelData message to the server with data destined for Peer B. The ChannelData message is not a STUN message, and thus has no transaction id. Instead, it has only three fields: a channel number, data, and data length; here the channel number field is 0x4000 (the channel the client just bound to Peer B). When the server receives the ChannelData message, it checks that the channel is currently bound (which it is) and then sends the data onward to Peer B in a UDP datagram, using the relayed transport address as the source transport address and 192.0.2.210:49191 (the value of the XOR-PEER-ADDRESS attribute in the ChannelBind request) as the destination transport address.

Later, Peer B sends a UDP datagram back to the relayed transport address. This causes the server to send a ChannelData message to the client containing the data from the UDP datagram. The server knows to which client to send the ChannelData message because of the relayed transport address at which the UDP datagram arrived, and knows to use channel 0x4000 because this is the channel bound to 192.0.2.210:49191. Note that if there had not been any channel number bound to that address, the server would have used a Data indication instead.

```
  TURN                                  TURN           Peer           Peer
  client                                server          A              B
    |--- Refresh request --------------->|             |              |
    |    Transaction-Id=0x0864B3C27ADE9354B4312414     |              |
    |    SOFTWARE="Example client 1.03"  |             |              |
    |    USERNAME="George"               |             |              |
    |    REALM="example.com"             |             |              |
    |    NONCE="adl7W7PeDU4hKE72jdaQvbAMcr6h39sm"      |              |
    |    MESSAGE-INTEGRITY=...           |             |              |
    |                                    |             |              |
    |<-- Refresh error response ---------|             |              |
    |    Transaction-Id=0x0864B3C27ADE9354B4312414     |              |
    |    SOFTWARE="Example server, version 1.17"       |              |
    |    ERROR-CODE=438 (Stale Nonce)    |             |              |
    |    REALM="example.com"             |             |              |
    |    NONCE="npSw1Xw239bBwGYhjNWgz2yH47sxB2j"       |              |
    |                                    |             |              |
    |--- Refresh request --------------->|             |              |
    |    Transaction-Id=0x427BD3E625A85FC731DC4191     |              |
    |    SOFTWARE="Example client 1.03"  |             |              |
    |    USERNAME="George"               |             |              |
    |    REALM="example.com"             |             |              |
    |    NONCE="npSw1Xw239bBwGYhjNWgz2yH47sxB2j"       |              |
    |    MESSAGE-INTEGRITY=...           |             |              |
    |                                    |             |              |
    |<-- Refresh success response -------|             |              |
    |    Transaction-Id=0x427BD3E625A85FC731DC4191     |              |
    |    SOFTWARE="Example server, version 1.17"       |              |
    |    LIFETIME=600 (10 minutes)       |             |              |
```

Sometime before the 20 minute lifetime is up, the client refreshes the allocation. This is done using a Refresh request. As before, the client includes the latest username, realm, and nonce values in the request. The client also includes the SOFTWARE attribute, following the recommended practice of always including this attribute in Allocate and Refresh messages. When the server receives the Refresh request, it notices that the nonce value has expired, and so replies with 438 (Stale Nonce) error given a new nonce value. The

client then reattempts the request, this time with the new nonce value. This second attempt is accepted, and the server replies with a success response. Note that the client did not include a LIFETIME attribute in the request, so the server refreshes the allocation for the default lifetime of 10 minutes (as can be seen by the LIFETIME attribute in the success response).

18. Security Considerations

This section considers attacks that are possible in a TURN deployment, and discusses how they are mitigated by mechanisms in the protocol or recommended practices in the implementation.

Most of the attacks on TURN are mitigated by the server requiring requests be authenticated. Thus, this specification requires the use of authentication. The mandatory-to-implement mechanism is the long-term credential mechanism of STUN. Other authentication mechanisms of equal or stronger security properties may be used. However, it is important to ensure that they can be invoked in an inter-operable way.

18.1. Outsider Attacks

Outsider attacks are ones where the attacker has no credentials in the system, and is attempting to disrupt the service seen by the client or the server.

18.1.1. Obtaining Unauthorized Allocations

An attacker might wish to obtain allocations on a TURN server for any number of nefarious purposes. A TURN server provides a mechanism for sending and receiving packets while cloaking the actual IP address of the client. This makes TURN servers an attractive target for attackers who wish to use it to mask their true identity.

An attacker might also wish to simply utilize the services of a TURN server without paying for them. Since TURN services require resources from the provider, it is anticipated that their usage will come with a cost.

These attacks are prevented using the long-term credential mechanism, which allows the TURN server to determine the identity of the requestor and whether the requestor is allowed to obtain the allocation.

18.1.2. Offline Dictionary Attacks

The long-term credential mechanism used by TURN is subject to offline dictionary attacks. An attacker that is capable of eavesdropping on a message exchange between a client and server can determine the password by trying a number of candidate passwords and seeing if one of them is correct. This attack works when the passwords are low entropy, such as a word from the dictionary. This attack can be mitigated by using strong passwords with large entropy. In situations where even stronger mitigation is required, (D)TLS transport between the client and the server can be used.

18.1.3. Faked Refreshes and Permissions

An attacker might wish to attack an active allocation by sending it a Refresh request with an immediate expiration, in order to delete it and disrupt service to the client. This is prevented by authentication of refreshes. Similarly, an attacker wishing to send CreatePermission requests to create permissions to undesirable destinations is prevented from doing so through authentication. The motivations for such an attack are described in Section 18.2.

18.1.4. Fake Data

An attacker might wish to send data to the client or the peer, as if they came from the peer or client, respectively. To do that, the attacker can send the client a faked Data Indication or ChannelData message, or send the TURN server a faked Send Indication or ChannelData message.

Since indications and ChannelData messages are not authenticated, this attack is not prevented by TURN. However, this attack is generally present in IP-based communications and is not substantially worsened by TURN. Consider a normal, non-TURN IP session between hosts A and B. An attacker can send packets to B as if they came from A by sending packets towards A with a spoofed IP address of B. This attack requires the attacker to know the IP addresses of A and B. With TURN, an attacker wishing to send packets towards a client using a Data indication needs to know its IP address (and port), the IP address and port of the TURN server, and the IP address and port of the peer (for inclusion in the XOR-PEER-ADDRESS attribute). To send a fake ChannelData message to a client, an attacker needs to know the IP address and port of the client, the IP address and port of the TURN server, and the channel number. This particular combination is mildly more guessable than in the non-TURN case.

These attacks are more properly mitigated by application-layer authentication techniques. In the case of real-time traffic, usage of SRTP [RFC3711] prevents these attacks.

In some situations, the TURN server may be situated in the network such that it is able to send to hosts to which the client cannot directly send. This can happen, for example, if the server is located behind a firewall that allows packets from outside the firewall to be delivered to the server, but not to other hosts behind the firewall. In these situations, an attacker could send the server a Send indication with an XOR-PEER-ADDRESS attribute containing the transport address of one of the other hosts behind the firewall. If the server was to allow relaying of traffic to arbitrary peers, then this would provide a way for the attacker to attack arbitrary hosts behind the firewall.

To mitigate this attack, TURN requires that the client establish a permission to a host before sending it data. Thus, an attacker can only attack hosts with which the client is already communicating, unless the attacker is able to create authenticated requests. Furthermore, the server administrator may configure the server to restrict the range of IP addresses and ports to which it will relay data. To provide even greater security, the server administrator can require that the client use (D)TLS for all communication between the client and the server.

18.1.5. Impersonating a Server

When a client learns a relayed address from a TURN server, it uses that relayed address in application protocols to receive traffic. Therefore, an attacker wishing to intercept or redirect that traffic might try to impersonate a TURN server and provide the client with a faked relayed address.

This attack is prevented through the long-term credential mechanism, which provides message integrity for responses in addition to verifying that they came from the server. Furthermore, an attacker cannot replay old server responses as the transaction id in the STUN header prevents this. Replay attacks are further thwarted through frequent changes to the nonce value.

18.1.6. Eavesdropping Traffic

TURN concerns itself primarily with authentication and message integrity. Confidentiality is only a secondary concern, as TURN control messages do not include information that is particularly sensitive. The primary protocol content of the messages is the IP

address of the peer. If it is important to prevent an eavesdropper on a TURN connection from learning this, TURN can be run over (D)TLS.

Confidentiality for the application data relayed by TURN is best provided by the application protocol itself, since running TURN over (D)TLS does not protect application data between the server and the peer. If confidentiality of application data is important, then the application should encrypt or otherwise protect its data. For example, for real-time media, confidentiality can be provided by using SRTP.

18.1.7. TURN Loop Attack

An attacker might attempt to cause data packets to loop indefinitely between two TURN servers. The attack goes as follows. First, the attacker sends an Allocate request to server A, using the source address of server B. Server A will send its response to server B, and for the attack to succeed, the attacker must have the ability to either view or guess the contents of this response, so that the attacker can learn the allocated relayed transport address. The attacker then sends an Allocate request to server B, using the source address of server A. Again, the attacker must be able to view or guess the contents of the response, so it can send learn the allocated relayed transport address. Using the same spoofed source address technique, the attacker then binds a channel number on server A to the relayed transport address on server B, and similarly binds the same channel number on server B to the relayed transport address on server A. Finally, the attacker sends a ChannelData message to server A.

The result is a data packet that loops from the relayed transport address on server A to the relayed transport address on server B, then from server B’s transport address to server A’s transport address, and then around the loop again.

This attack is mitigated as follows. By requiring all requests to be authenticated and/or by randomizing the port number allocated for the relayed transport address, the server forces the attacker to either intercept or view responses sent to a third party (in this case, the other server) so that the attacker can authenticate the requests and learn the relayed transport address. Without one of these two measures, an attacker can guess the contents of the responses without needing to see them, which makes the attack much easier to perform. Furthermore, by requiring authenticated requests, the server forces the attacker to have credentials acceptable to the server, which turns this from an outsider attack into an insider attack and allows the attack to be traced back to the client initiating it.

The attack can be further mitigated by imposing a per-username limit on the bandwidth used to relay data by allocations owned by that username, to limit the impact of this attack on other allocations. More mitigation can be achieved by decrementing the TTL when relaying data packets (if the underlying OS allows this).

18.2. Firewall Considerations

A key security consideration of TURN is that TURN should not weaken the protections afforded by firewalls deployed between a client and a TURN server. It is anticipated that TURN servers will often be present on the public Internet, and clients may often be inside enterprise networks with corporate firewalls. If TURN servers provide a ’backdoor’ for reaching into the enterprise, TURN will be blocked by these firewalls.

TURN servers therefore emulate the behavior of NAT devices that implement address-dependent filtering [RFC4787], a property common in many firewalls as well. When a NAT or firewall implements this behavior, packets from an outside IP address are only allowed to be sent to an internal IP address and port if the internal IP address and port had recently sent a packet to that outside IP address. TURN servers introduce the concept of permissions, which provide exactly this same behavior on the TURN server. An attacker cannot send a packet to a TURN server and expect it to be relayed towards the client, unless the client has tried to contact the attacker first.

It is important to note that some firewalls have policies that are even more restrictive than address-dependent filtering. Firewalls can also be configured with address- and port-dependent filtering, or can be configured to disallow inbound traffic entirely. In these cases, if a client is allowed to connect the TURN server, communications to the client will be less restrictive than what the firewall would normally allow.

18.2.1. Faked Permissions

In firewalls and NAT devices, permissions are granted implicitly through the traversal of a packet from the inside of the network towards the outside peer. Thus, a permission cannot, by definition, be created by any entity except one inside the firewall or NAT. With TURN, this restriction no longer holds. Since the TURN server sits outside the firewall, at attacker outside the firewall can now send a message to the TURN server and try to create a permission for itself.

This attack is prevented because all messages that create permissions (i.e., ChannelBind and CreatePermission) are authenticated.

18.2.2. Blacklisted IP Addresses

Many firewalls can be configured with blacklists that prevent a client behind the firewall from sending packets to, or receiving packets from, ranges of blacklisted IP addresses. This is accomplished by inspecting the source and destination addresses of packets entering and exiting the firewall, respectively.

This feature is also present in TURN, since TURN servers are allowed to arbitrarily restrict the range of addresses of peers that they will relay to.

18.2.3. Running Servers on Well-Known Ports

A malicious client behind a firewall might try to connect to a TURN server and obtain an allocation which it then uses to run a server. For example, a client might try to run a DNS server or FTP server.

This is not possible in TURN. A TURN server will never accept traffic from a peer for which the client has not installed a permission. Thus, peers cannot just connect to the allocated port in order to obtain the service.

18.3. Insider Attacks

In insider attacks, a client has legitimate credentials but defies the trust relationship that goes with those credentials. These attacks cannot be prevented by cryptographic means but need to be considered in the design of the protocol.

18.3.1. DoS against TURN Server

A client wishing to disrupt service to other clients might obtain an allocation and then flood it with traffic, in an attempt to swamp the server and prevent it from servicing other legitimate clients. This is mitigated by the recommendation that the server limit the amount of bandwidth it will relay for a given username. This won’t prevent a client from sending a large amount of traffic, but it allows the server to immediately discard traffic in excess.

Since each allocation uses a port number on the IP address of the TURN server, the number of allocations on a server is finite. An attacker might attempt to consume all of them by requesting a large number of allocations. This is prevented by the recommendation that the server impose a limit of the number of allocations active at a time for a given username.

18.3.2. Anonymous Relaying of Malicious Traffic

TURN servers provide a degree of anonymization. A client can send data to peers without revealing its own IP address. TURN servers may therefore become attractive vehicles for attackers to launch attacks against targets without fear of detection. Indeed, it is possible for a client to chain together multiple TURN servers, such that any number of relays can be used before a target receives a packet.

Administrators who are worried about this attack can maintain logs that capture the actual source IP and port of the client, and perhaps even every permission that client installs. This will allow for forensic tracing to determine the original source, should it be discovered that an attack is being relayed through a TURN server.

18.3.3. Manipulating Other Allocations

An attacker might attempt to disrupt service to other users of the TURN server by sending Refresh requests or CreatePermission requests that (through source address spoofing) appear to be coming from another user of the TURN server. TURN prevents this by requiring that the credentials used in CreatePermission, Refresh, and ChannelBind messages match those used to create the initial allocation. Thus, the fake requests from the attacker will be rejected.

18.4. Tunnel Amplification Attack

An attacker might attempt to cause data packets to loop numerous times between a TURN server and a tunnel between IPv4 and IPv6. The attack goes as follows.

Suppose an attacker knows that a tunnel endpoint will forward encapsulated packets from a given IPv6 address (this doesn’t necessarily need to be the tunnel endpoint’s address). Suppose he then spoofs two packets from this address:

1. An Allocate request asking for a v4 address, and

2. A ChannelBind request establishing a channel to the IPv4 address of the tunnel endpoint

Then he has set up an amplification attack:

- The TURN relay will re-encapsulate IPv6 UDP data in v4 and send it to the tunnel endpoint

o The tunnel endpoint will de-encapsulate packets from the v4 interface and send them to v6

So if the attacker sends a packet of the following form...

```
IPv6: src=2001:DB9::1 dst=2001:DB8::2
UDP:  <ports>
TURN: <channel id>
IPv6: src=2001:DB9::1 dst=2001:DB8::2
UDP:  <ports>
TURN: <channel id>
IPv6: src=2001:DB9::1 dst=2001:DB8::2
UDP:  <ports>
TURN: <channel id>
...
```

Then the TURN relay and the tunnel endpoint will send it back and forth until the last TURN header is consumed, at which point the TURN relay will send an empty packet, which the tunnel endpoint will drop.

The amplification potential here is limited by the MTU, so it’s not huge: IPv6+UDP+TURN takes 334 bytes, so a four-to-one amplification out of a 1500-byte packet is possible. But the attacker could still increase traffic volume by sending multiple packets or by establishing multiple channels spoofed from different addresses behind the same tunnel endpoint.

The attack is mitigated as follows. It is RECOMMENDED that TURN relays not accept allocation or channel binding requests from addresses known to be tunneled, and that they not forward data to such addresses. In particular, a TURN relay MUST NOT accept Teredo or 6to4 addresses in these requests.

18.5. Other Considerations

Any relay addresses learned through an Allocate request will not operate properly with IPsec Authentication Header (AH) [RFC4302] in transport or tunnel mode. However, tunnel-mode IPsec Encapsulating Security Payload (ESP) [RFC4303] should still operate.

19. IANA Considerations

Since TURN is an extension to STUN [RFC5389], the methods, attributes, and error codes defined in this specification are new methods, attributes, and error codes for STUN. IANA has added these new protocol elements to the IANA registry of STUN protocol elements.

The codepoints for the new STUN methods defined in this specification are listed in Section 14.

The codepoints for the new STUN attributes defined in this specification are listed in Section 15.

The codepoints for the new STUN error codes defined in this specification are listed in Section 16.

IANA has allocated the SRV service name of "turn" for TURN over UDP or TCP, and the service name of "turns" for TURN over (D)TLS.

IANA has created a registry for TURN channel numbers, initially populated as follows:

- o 0x0000 through 0x3FFF: Reserved and not available for use, since they conflict with the STUN header.

- o 0x4000 through 0x7FFF: A TURN implementation is free to use channel numbers in this range.

- o 0x8000 through 0xFFFF: Unassigned.

Any change to this registry must be made through an IETF Standards Action.

[Paragraphs in braces should be removed by the RFC Editor upon publication]

[The ADDITIONAL-ADDRESS-FAMILY, ADDRESS-ERROR-CODE and ICMP attributes requires that IANA allocate a value in the "STUN attributes Registry" from the comprehension- optional range (0x8000-0xFFFF), to be replaced for TBD-CA throughout this document]

[The SendErr method requires that IANA allocate a value in the "STUN Methods Registry" from the range (0x000-0x7FF), to be replaced for TBD-DA throughout this document]

20. IAB Considerations

The IAB has studied the problem of "Unilateral Self Address Fixing" (UNSAF), which is the general process by which a client attempts to determine its address in another realm on the other side of a NAT through a collaborative protocol-reflection mechanism [RFC3424]. The TURN extension is an example of a protocol that performs this type of function. The IAB has mandated that any protocols developed for this purpose document a specific set of considerations. These

considerations and the responses for TURN are documented in this section.

Consideration 1: Precise definition of a specific, limited-scope problem that is to be solved with the UNSAF proposal. A short-term fix should not be generalized to solve other problems. Such generalizations lead to the prolonged dependence on and usage of the supposed short-term fix -- meaning that it is no longer accurate to call it "short-term".

Response: TURN is a protocol for communication between a relay (= TURN server) and its client. The protocol allows a client that is behind a NAT to obtain and use a public IP address on the relay. As a convenience to the client, TURN also allows the client to determine its server-reflexive transport address.

Consideration 2: Description of an exit strategy/transition plan. The better short-term fixes are the ones that will naturally see less and less use as the appropriate technology is deployed.

Response: TURN will no longer be needed once there are no longer any NATs. Unfortunately, as of the date of publication of this document, it no longer seems very likely that NATs will go away any time soon. However, the need for TURN will also decrease as the number of NATs with the mapping property of Endpoint-Independent Mapping [RFC4787] increases.

Consideration 3: Discussion of specific issues that may render systems more "brittle". For example, approaches that involve using data at multiple network layers create more dependencies, increase debugging challenges, and make it harder to transition.

Response: TURN is "brittle" in that it requires the NAT bindings between the client and the server to be maintained unchanged for the lifetime of the allocation. This is typically done using keep-alives. If this is not done, then the client will lose its allocation and can no longer exchange data with its peers.

Consideration 4: Identify requirements for longer-term, sound technical solutions; contribute to the process of finding the right longer-term solution.

Response: The need for TURN will be reduced once NATs implement the recommendations for NAT UDP behavior documented in [RFC4787]. Applications are also strongly urged to use ICE [RFC5245] to communicate with peers; though ICE uses TURN, it does so only as a last resort, and uses it in a controlled manner.

Consideration 5: Discussion of the impact of the noted practical issues with existing deployed NATs and experience reports.

Response: Some NATs deployed today exhibit a mapping behavior other than Endpoint-Independent mapping. These NATs are difficult to work with, as they make it difficult or impossible for protocols like ICE to use server-reflexive transport addresses on those NATs. A client behind such a NAT is often forced to use a relay protocol like TURN because "UDP hole punching" techniques [RFC5128] do not work.

21. Changes since RFC 5766

This section lists the major changes in the TURN protocol from the original [RFC5766] specification.

- o IPv6 support.
- o REQUESTED-ADDRESS-FAMILY, ADDITIONAL-ADDRESS-FAMILY, AND ADDRESS-ERRR-CODE attributes.
- o 440 (Address Family not Supported) and 443 (Peer Address Family Mismatch) responses.
- o Description of the tunnel amplification attack.
- o DTLS support.
- o More details on packet translations.
- o Add support for receiving ICMP packets.
- o Updates PMTUD.

22. Acknowledgements

Most of the text in this note comes from the original TURN specification, [RFC5766]. The authors would like to thank Rohan Mahy co-author of orginal TURN specification and everyone who had contributed to that document.

Thanks to Justin Uberti, Pal Martinsen, Oleg Moskalenko, Aijun Wang and Simon Perreault for their help on SSODA mechanism. Authors would like to thank Gonzalo Salgueiro, Simon Perreault, Jonathan Lennox and Oleg Moskalenko for comments and review. The authors would like to thank Marc for his contributions to the text.

23. References

23.1. Normative References

[RFC5389] Rosenberg, J., Mahy, R., Matthews, P., and D. Wing, "Session Traversal Utilities for NAT (STUN)", RFC 5389, DOI 10.17487/RFC5389, October 2008, <http://www.rfc-editor.org/info/rfc5389>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC2474] Nichols, K., Blake, S., Baker, F., and D. Black, "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers", RFC 2474, DOI 10.17487/RFC2474, December 1998, <http://www.rfc-editor.org/info/rfc2474>.

[RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, DOI 10.17487/RFC3168, September 2001, <http://www.rfc-editor.org/info/rfc3168>.

[RFC1122] Braden, R., Ed., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, DOI 10.17487/RFC1122, October 1989, <http://www.rfc-editor.org/info/rfc1122>.

[RFC6145] Li, X., Bao, C., and F. Baker, "IP/ICMP Translation Algorithm", RFC 6145, DOI 10.17487/RFC6145, April 2011, <http://www.rfc-editor.org/info/rfc6145>.

[RFC3697] Rajahalme, J., Conta, A., Carpenter, B., and S. Deering, "IPv6 Flow Label Specification", RFC 3697, DOI 10.17487/RFC3697, March 2004, <http://www.rfc-editor.org/info/rfc3697>.

[RFC7065] Petit-Huguenin, M., Nandakumar, S., Salgueiro, G., and P. Jones, "Traversal Using Relays around NAT (TURN) Uniform Resource Identifiers", RFC 7065, DOI 10.17487/RFC7065, November 2013, <http://www.rfc-editor.org/info/rfc7065>.

[RFC0792] Postel, J., "Internet Control Message Protocol", STD 5, RFC 792, DOI 10.17487/RFC0792, September 1981, <http://www.rfc-editor.org/info/rfc792>.

[RFC4443] Conta, A., Deering, S., and M. Gupta, Ed., "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification", RFC 4443, DOI 10.17487/RFC4443, March 2006, <http://www.rfc-editor.org/info/rfc4443>.

[I-D.ietf-tram-stunbis] Petit-Huguenin, M., Salgueiro, G., Rosenberg, J., Wing, D., Mahy, R., and P. Matthews, "Session Traversal Utilities for NAT (STUN)", draft-ietf-tram-stunbis-08 (work in progress), June 2016.

23.2. Informative References

[RFC1191] Mogul, J. and S. Deering, "Path MTU discovery", RFC 1191, DOI 10.17487/RFC1191, November 1990, <http://www.rfc-editor.org/info/rfc1191>.

[RFC0791] Postel, J., "Internet Protocol", STD 5, RFC 791, DOI 10.17487/RFC0791, September 1981, <http://www.rfc-editor.org/info/rfc791>.

[RFC1918] Rekhter, Y., Moskowitz, B., Karrenberg, D., de Groot, G., and E. Lear, "Address Allocation for Private Internets", BCP 5, RFC 1918, DOI 10.17487/RFC1918, February 1996, <http://www.rfc-editor.org/info/rfc1918>.

[RFC3424] Daigle, L., Ed. and IAB, "IAB Considerations for UNilateral Self-Address Fixing (UNSAF) Across Network Address Translation", RFC 3424, DOI 10.17487/RFC3424, November 2002, <http://www.rfc-editor.org/info/rfc3424>.

[RFC4787] Audet, F., Ed. and C. Jennings, "Network Address Translation (NAT) Behavioral Requirements for Unicast UDP", BCP 127, RFC 4787, DOI 10.17487/RFC4787, January 2007, <http://www.rfc-editor.org/info/rfc4787>.

[RFC5245] Rosenberg, J., "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols", RFC 5245, DOI 10.17487/RFC5245, April 2010, <http://www.rfc-editor.org/info/rfc5245>.

[RFC6062] Perreault, S., Ed. and J. Rosenberg, "Traversal Using Relays around NAT (TURN) Extensions for TCP Allocations", RFC 6062, DOI 10.17487/RFC6062, November 2010, <http://www.rfc-editor.org/info/rfc6062>.

[RFC6156] Camarillo, G., Novo, O., and S. Perreault, Ed., "Traversal Using Relays around NAT (TURN) Extension for IPv6", RFC 6156, DOI 10.17487/RFC6156, April 2011, <http://www.rfc-editor.org/info/rfc6156>.

[RFC6056] Larsen, M. and F. Gont, "Recommendations for Transport-Protocol Port Randomization", BCP 156, RFC 6056, DOI 10.17487/RFC6056, January 2011, <http://www.rfc-editor.org/info/rfc6056>.

[RFC5128] Srisuresh, P., Ford, B., and D. Kegel, "State of Peer-to-Peer (P2P) Communication across Network Address Translators (NATs)", RFC 5128, DOI 10.17487/RFC5128, March 2008, <http://www.rfc-editor.org/info/rfc5128>.

[RFC1928] Leech, M., Ganis, M., Lee, Y., Kuris, R., Koblas, D., and L. Jones, "SOCKS Protocol Version 5", RFC 1928, DOI 10.17487/RFC1928, March 1996, <http://www.rfc-editor.org/info/rfc1928>.

[RFC3550] Schulzrinne, H., Casner, S., Frederick, R., and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications", STD 64, RFC 3550, DOI 10.17487/RFC3550, July 2003, <http://www.rfc-editor.org/info/rfc3550>.

[RFC3711] Baugher, M., McGrew, D., Naslund, M., Carrara, E., and K. Norrman, "The Secure Real-time Transport Protocol (SRTP)", RFC 3711, DOI 10.17487/RFC3711, March 2004, <http://www.rfc-editor.org/info/rfc3711>.

[RFC4302] Kent, S., "IP Authentication Header", RFC 4302, DOI 10.17487/RFC4302, December 2005, <http://www.rfc-editor.org/info/rfc4302>.

[RFC4303] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC 4303, DOI 10.17487/RFC4303, December 2005, <http://www.rfc-editor.org/info/rfc4303>.

[RFC4821] Mathis, M. and J. Heffner, "Packetization Layer Path MTU Discovery", RFC 4821, DOI 10.17487/RFC4821, March 2007, <http://www.rfc-editor.org/info/rfc4821>.

[RFC3261] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and E. Schooler, "SIP: Session Initiation Protocol", RFC 3261, DOI 10.17487/RFC3261, June 2002, <http://www.rfc-editor.org/info/rfc3261>.

[I-D.rosenberg-mmusic-ice-nonsip]
Rosenberg, J., "Guidelines for Usage of Interactive Connectivity Establishment (ICE) by non Session Initiation Protocol (SIP) Protocols", draft-rosenberg-mmusic-ice-nonsip-01 (work in progress), July 2008.

[I-D.ietf-tram-stun-pmtud]
Petit-Huguenin, M. and G. Salgueiro, "Path MTU Discovery Using Session Traversal Utilities for NAT (STUN)", draft-ietf-tram-stun-pmtud-03 (work in progress), October 2016.

[I-D.ietf-tram-turn-server-discovery]
Patil, P., Reddy, T., and D. Wing, "TURN Server Auto Discovery", draft-ietf-tram-turn-server-discovery-10 (work in progress), October 2016.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <http://www.rfc-editor.org/info/rfc4086>.

[RFC5766] Mahy, R., Matthews, P., and J. Rosenberg, "Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)", RFC 5766, DOI 10.17487/RFC5766, April 2010, <http://www.rfc-editor.org/info/rfc5766>.

[RFC5928] Petit-Huguenin, M., "Traversal Using Relays around NAT (TURN) Resolution Mechanism", RFC 5928, DOI 10.17487/RFC5928, August 2010, <http://www.rfc-editor.org/info/rfc5928>.

[Port-Numbers]
"IANA Port Numbers Registry", 2005, <http://www.iana.org/assignments/port-numbers>.

[Frag-Harmful]
"Fragmentation Considered Harmful", <Proc. SIGCOMM ’87, vol. 17, No. 5, October 1987>.

[Protocol-Numbers]
"IANA Protocol Numbers Registry", 2005, <http://www.iana.org/assignments/protocol-numbers>.

Authors' Addresses

Tirumaleswar Reddy (editor)
Cisco Systems, Inc.
Cessna Business Park, Varthur Hobl
Sarjapur Marathalli Outer Ring Road
Bangalore, Karnataka 560103
India

Email: tireddy@cisco.com

Alan Johnston (editor)
Unaffiliated
Bellevue, WA
USA

Email: alan.b.johnston@gmail.com

Philip Matthews
Alcatel-Lucent
600 March Road
Ottawa, Ontario
Canada

Email: philip_matthews@magma.ca

Jonathan Rosenberg
jdrosen.net
Edison, NJ
USA

Email: jdrosen@jdrosen.net
URI: http://www.jdrosen.net