Workgroup: Network Working Group
Internet-Draft: draft-hallambaker-mesh-rud
Published: 23 October 2022
Intended Status: Informational
Expires: 26 April 2023
Authors: P. M. Hallam-Baker
ThresholdSecrets.com

Mathematical Mesh 3.0 Part VI: Reliable User Datagram

Abstract

A presentation layer suitable for use in conjunction with HTTP and UDP transports is described.

https://mailarchive.ietf.org/arch/browse/mathmesh/Discussion of this draft should take place on the MathMesh mailing list (mathmesh@ietf.org), which is archived at .

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 https://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 26 April 2023.

Copyright Notice

Copyright (c) 2022 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 (https://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.

Table of Contents

1. Introduction
   1.1. Background
      1.1.1. Protocol Elements
      1.1.2. Application Interface
   1.2. Use Cases
      1.2.1. Network Agility
      1.2.2. Connection Transfer and Delegation
      1.2.3. Presence
      1.2.4. NAT Transit
      1.2.5. IPv6 Agility
      1.2.6. Tunneling
2. Definitions
   2.1. Requirements Language
   2.2. Defined Terms
   2.3. Related Specifications
   2.4. Implementation Status
3. Architecture
   3.1. Connections
   3.2. Ports
   3.3. Stream
      3.3.1. Transactional stream
      3.3.2. Asynchronous stream
   3.4. Credentials
      3.4.1. Device
      3.4.2. Principal
   3.5. Datagram Format
      3.5.1. Stream Identifier
      3.5.2. Nonce
      3.5.3. Ciphertext
      3.5.4. Authentication Tag
   3.6. Packet Format and Packetized Datagrams
      3.6.1. Packet Header
      3.6.2. Flow Control
   3.7. Connection Datagrams
4. Connection Establishment and Maintenance
   4.1. Key Exchange modes
      4.1.1. First Contact
      4.1.2. Immediate
      4.1.3. Deferred Trip
   4.2. Ticketed Connection
   4.3. Connection Forward Secrecy Rekeying
5. Stream Establishment and Maintenance
   5.1. Stream Creation and Binding
   5.2. Stream Close
   5.3. One Time Stream Identifier
   5.4. Stream Rekey

6. Schema
6.1. Connection Datagrams
6.2. Packet Extensions
7. Security Considerations
8. IANA Considerations
9. Acknowledgements
10. Normative References
11. Informative References

1. Introduction

The Reliable User Datagram (RUD) is a lightweight key exchange, authentication and encryption protocol that provides a transactional presentation when layered over a HTTP transport and transactional and streaming presentations when layered over a UDP transport.

The UDP binding of RUD provides much of the functionality normally provided through use of TCP, TLS, HTTP and Web Sockets with considerably less complexity. RUD does not attempt to reproduce all the functionality of these protocols, only the functionality required by Web Services and streaming applications is provided.

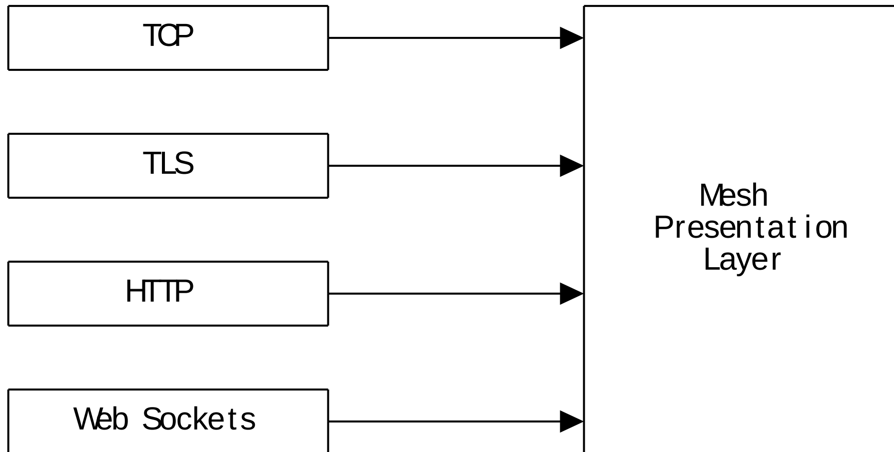

Figure 1: Mesh Presentation Layer provides a combination of necessary functionality from TCP, TLS, HTTP and Web Sockets.

As recognized in the design of QUIC, combining these protocols allows greater efficiency and security. But QUIC is the product of one set of design choices optimized to meet an important but specific set of needs. RUD makes a different set of choices to meet a different set of needs. A natural consequence of moving TCP functionality from a platform capability implemented in the OS kernel to an application level resource it that applications are now free to chose the transport and security capabilities that best fit their needs rather than being limited to a single size fits all.

The reduction in complexity afforded by combining four protocols into one allows entirely new communications patterns to be supported that are poorly supported by traditional approaches. HTTP is built on the Remote Procedure Call model of transactions that consist of a single request followed by a response. Using this communication pattern, a device attempting to synchronize a local store with a store at a remote service is required to periodically poll the service for updates:

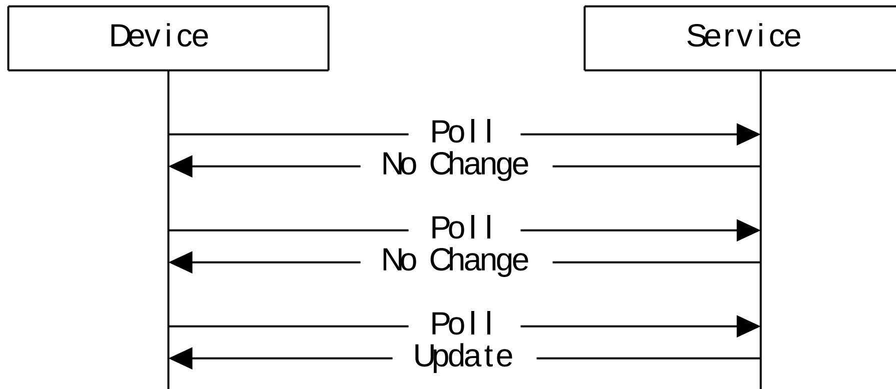

Figure 2: Limiting communications to a request/response communication requires inefficient polling for updates.

RUD allows a client device with this particular requirement to subscribe to a continuous stream of updates. There is no need for the device to poll the remote store for updates as it will be notified every time an update occurs.

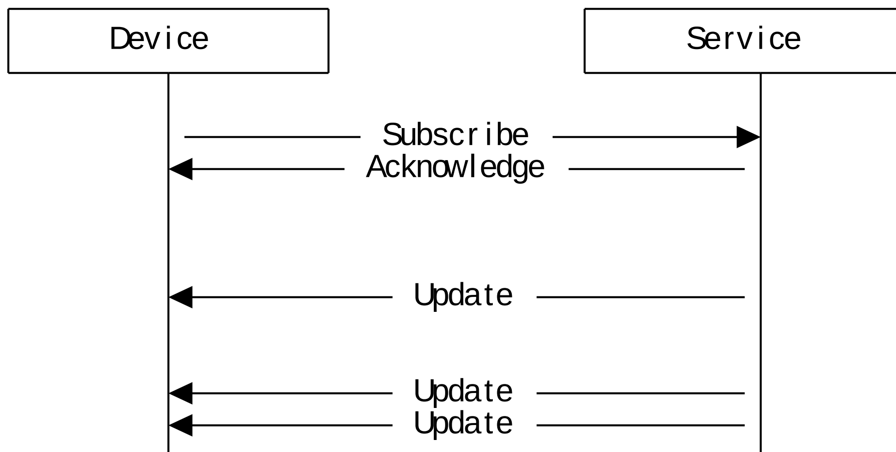

Figure 3: RUD supports asynchronous streaming of updates as they occur.

While it is possible to achieve the same functionality in TCP, this requires that the device and service maintain a TCP connection for the duration of the connection. This consumes communication resources and operating system resources.

As is shown in later sections, the 'blank slate' design approach of RUD allows completely new approaches to service discovery, network agility, presence, NAT tunneling and many other requirements that are critical in modern network use.

1.1. Background

Web Services have traditionally made use of HTTP bindings over TCP or TLS transports. While this approach is functional, it is less than satisfactory. The HTTP and TLS stacks are designed to meet the needs of Web browsing but fail to provide significant capabilities required by a service. Modern implementations of these stacks are large and cumbersome but little of that complexity addresses needs relevant to Web Services.

A basic requirement for a Web Service is the need to authenticate requests and the party making them. In particular, a Web Service requires the ability to authenticate the source of a request and verify that authentication of the request data is complete before acting on it. As its original name, 'Secure Socket Layer' implies, TLS is designed to present an abstract socket layer that hides this information from the application layer. In many large enterprise deployments, TLS processing and Web Service transactions are performed on separate machines and information from the TLS session is only visible to the transaction processor if special provision is made.

HTTP provides a wide range of protocol capabilities but almost none of this complexity is relevant to Web Services needs.

* Providing access to multiple Web Services through a single TCP/IP port.

* Delineating the beginning and end of request and response data.

Recognizing these needs, the DARE envelope format was originally conceived as a JSON equivalent of the XML Signature and Encryption protocols applied to provide message layer security in the WS-Security stack. Consideration of the resource requirements needed to verify each transaction request and sign each response led to the need for a key exchange mechanism and that this should be separate from the Mesh Service Protocol. This in turn led to the realization that in most circumstances, an AEAD scheme such as AES-GCM or AES-CFB would provide encryption with the same or less overhead than use of a MAC.

Having taken these considerations into account, a new lightweight envelope format was designed to allow authentication and encryption of Web Services messages as the payload of HTTP requests and responses. The success of this approach led to the realization that the same format could be modified to enable direct binding to UDP, allowing provision of HTTP support to be made optional.

1.1.1. Protocol Elements

A TCP connection provides a single pair of bidirectional communication streams between two devices connected to the network. While the TCP connection is symmetric from the application layer point of view, the HTTP presentation layer is not. HTTP limits communications to a request/response communication pattern in which the initiator device makes all the requests and the responder device makes all the responses.

RUD affords greater communication flexibility requiring distinctions to be made between terms which may be used interchangeably in the HTTP over TCP/IP case.

Connection An RUD connection is a bidirectional relationship between a pair of endpoints established through a cryptographic key exchange establishing a shared secret called the Connection Origin Key.

Stream An RUD stream is an individual communication channel established within a connection. An RUD connection comprises one or more streams.

Port An RUD port is a network interface through which data is exchanged. Currently only UDP and HTTP ports are supported. Each end of an RUD Connection accepts packets through one or more ports.

The distinction between connections and streams is analogous to that made between processes and threads. While any task that is performed by a separate thread may be performed by a separate process, use of threads consumes fewer platform resources and greater convenience to the programmer.

In the TCP/IP model, each stream is bound to a single pair of IP port. In the RUD model, any stream **MAY** make use of any available destination port to send traffic. This allows Alice to move from her home to her car to her office without dropping the connection to her conference call.

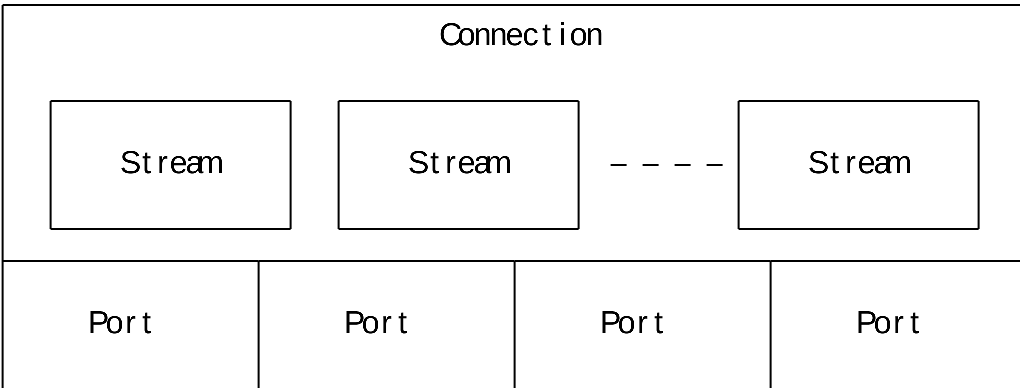

Figure 4: An RUD connection comprises one or more streams and one or more ports.

New streams may be created at any time by either party in a connection for many different reasons:

* To establish a client/service relationship to a different Web Service.
* To establish a new asynchronous streaming relationship
* To signal changes to the cryptographic context (e.g., forward secrecy rachet operation).
* Exhaustion of the datagram counter identifier space.
* To make use of traffic analysis avoidance techniques.
* To enable datagrams to be sent at different priorities.

A device connected to three separate Web Services provided by the same host establishes a separate stream for each service. This enables the device to present different application-level credentials top the different services. If permitted by the service, separate tasks on the same device **MAY** establish separate streams to the same service. This allows separate threads to wait on transactions taking a long time to complete without contention.

Each stream presents a sequence of datagrams spanning one or more packets according to a declared characteristic. The datagram represents the unit of payload data that is authenticated and thus the unit of payload data that is presented to the calling API.

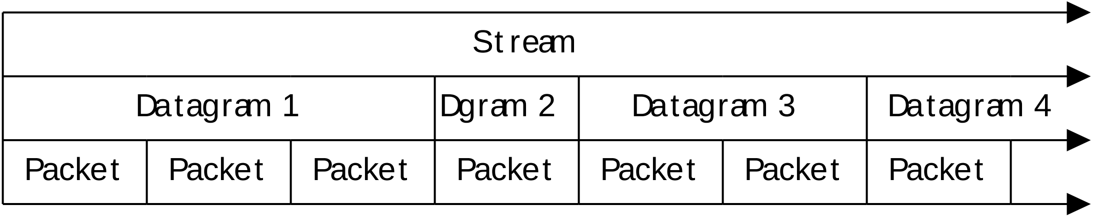

Figure 5: An RUD stream presents a sequence of datagrams, each of which may span multiple packets.

Two stream characteristics are currently defined, both of which are asymmetric.

Transactional Stream Each request sent in the forward direction is followed by a response returned in the reverse direction.

Asynchronous Stream A sequence of frames is received in the forward direction as directed by control messages sent in the reverse direction.

Thus, a bidirectional voice call between Alice and Bob requires a minimum of two Asynchronous Content streams, from Alice to Bob and from Bob to Alice.

1.1.2. Application Interface

The traditional TCP/IP programming API exposes only an undifferentiated stream of bytes to the application program context. Information such as packet latency, packet loss rate, error rate, etc. are only available through proprietary platform specific APIs, if at all.

RUD is designed to provide application programs with relevant connection information including:

- *Time since the most recent forward secrecy exchange
- *Average round trip latency (per port)
- *Packet loss (per port)

Providing this information allows applications to adapt to changing network conditions.

- *Forcing a forward secrecy exchange when a participant leaves a shared chat context
- *Making use of an outbound port with a different physical network connection.

*Establishing communication to a different host.

*Pre-empt (i.e. abandon) transfer of large datagrams in mid transfer.

The API exposes the following object classes:

Connection Reports the connection status, supports creation of new Streams.

Transactional Stream Initiator Supports the Post method comprising sending a request to the service to which the stream is bound and waiting for a response.

Transactional Stream Responder Is generated on receipt of a service request to which it is dispatched to obtain the service response.

Asynchronous Stream Sender Allows transmission of a sequence of datagrams to a receiver. Reports stream properties set by the receiver.

Asynchronous Stream Receiver Receives a continuous sequence of datagrams from the sender. Allows setting of stream properties.

Datagram Exposes the packet data relevant transmission information.

Stream Creation Ticket A stream creation ticket is a record created by an issuer device consisting of a stream identifier, a set of network endpoints, a shared secret and the issuer device's credential.

The details of the RUD API are out of scope for this specification.

1.2. Use Cases

Despite representing a modest incremental advance on the complexity of TCP and a substantial reduction in complexity compared to TLS, RUD provides support for a wide range of use cases outside the scope of either.

1.2.1. Network Agility

The separation of RUD streams from the network ports through which they are realized allows network agility to be supported without any additional complexity.

For example, Alice joins a conference call from her mobile phone in her home office. When she leaves for work in her car, the connection to her conference call is transferred seamlessly from hew home WiFi service to her mobile carrier.

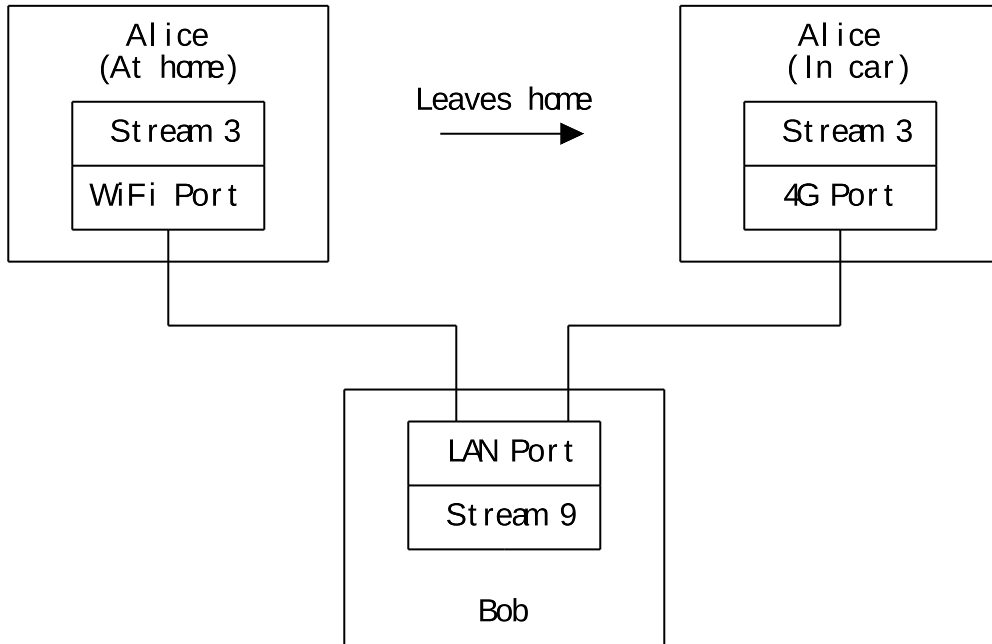

Figure 6: Alice maintains connection to her conference call when moving from her home to her office.

The same approach **MAY** be employed to provide fault tolerance in an enterprise server hosting environment.

1.2.2. Connection Transfer and Delegation

It is frequently desirable for a device to negotiate delivery of a service with one host that will ultimately be delivered by another. RUD supports the issue of a 'ticket' credential by a host providing a discovery service that **MAY** be used to establish a connection with the host that will deliver the requested service:

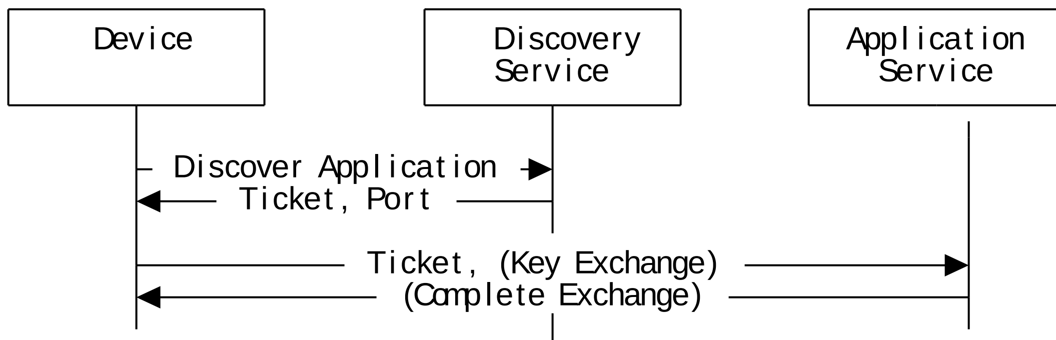

Figure 7: Connection establishment by means of a discovery service.

The same capability **MAY** be applied to enable hosts to direct connected devices to seek service from a different host, either immediately or in case of service interruption. This allows for greater flexibility in configuring systems to provide fault tolerance and to effect load balancing strategies.

1.2.3. Presence

Presence is a particular form of connection transfer. A presence service facilitates creation of a voice or video connection between the device used by the initiator to make the request to one of the multiple receiver devices on which it might be accepted.

Each receiver device maintains a connection to the presence service that is active whenever the receiver device is available to accept calls. To request a connection, the initiator device contacts the presence service which determines if the request is authorized and if so, returns the ticket and network endpoint information required to establish a connection.

For example, Bob's phone connects to the presence service and provides a ticket and network endpoint information to enable calls to be placed. When Alice attempts to place a call, her device contacts the presence service, which checks that Bob has authorized Alice to interrupt her work. Alice's device uses the information from the presence service to perform a forward secrecy operation against Bob's phone and thus achieve full end-to-end secure communication.

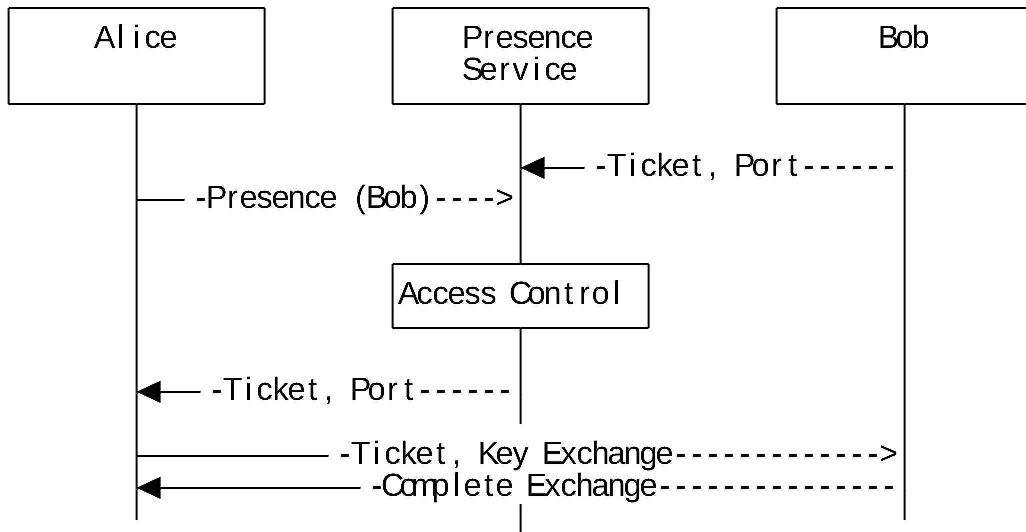

Figure 8: Alice contacts Bob through a presence service.

Contrawise, when Suzy spammer attempts to contact Bob to tell him the warranty on his rocket ship is about to expire, the presence service notes that Suzy spammer is not in Bob's contact list and refuses to allow the interruption. Bob's access control policy may allow Suzy spammer to make a contact exchange request, but it is rather likely that her reputation for wasting people's time will result in that request being rejected.

1.2.4. NAT Transit

The deployment of Network Address Translation (NAT) devices in the network represents a real world constraint that every modern network application has to work around if it is to be successful. NAT devices may be broadly regarded as belonging to two distinct categories.

Wide Cone NAT When a device inside the network attempts to establish a UDP connection to a remote device, the NAT selects an outbound UDP source port for forwarding. All inbound UDP packets specifying that port are forwarded to that device regardless of the source address.

Narrow Cone NAT + Firewall NAT All NAT devices that present a more restrictive policy. These include enterprise firewalls that require devices inside the network to explicitly request forwarding from external UDP ports (pinhole routing).

Since the protocols used to support NAT devices other than Wide Cone are typically proprietary, these are outside the scope of this specification.

One of the many inconveniences of being required to support NAT is that the external port binding may change without notice. In some cases, an external port is only kept open for as long as it is in use with timeouts of a second or less being common.

Enabling NAT transit during reconfiguration represents a special case of network agility. The network endpoint can change but the stream identifiers remain constant. A change to the NAT configuration causes inbound packets to be dropped temporarily but the stream is quickly re-established when an outbound packet is sent.

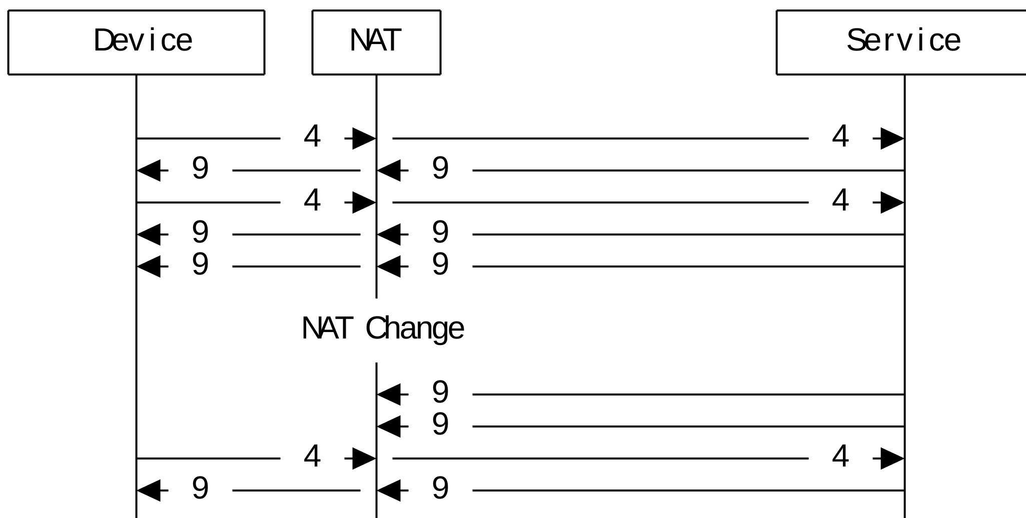

Figure 9: Wide cone NAT configuration.

The interaction of NAT transit with presence service presents a particular concern when interactions between the parties are infrequent.

For example, consider the case in which a digital camera is connected to a photo archive on a remote host that is behind a NAT device. This would commonly occur in situations in which Alice wants photographs to be automatically uploaded to the file storage in her house as soon as they are captured.

Rather than requiring the Archive device to maintain a constant RUD stream to every device that might possibly require access, the Archive device maintains a constant stream to the presence service which can then provide devices it has helped establish a connection with the necessary information to resume the stream:

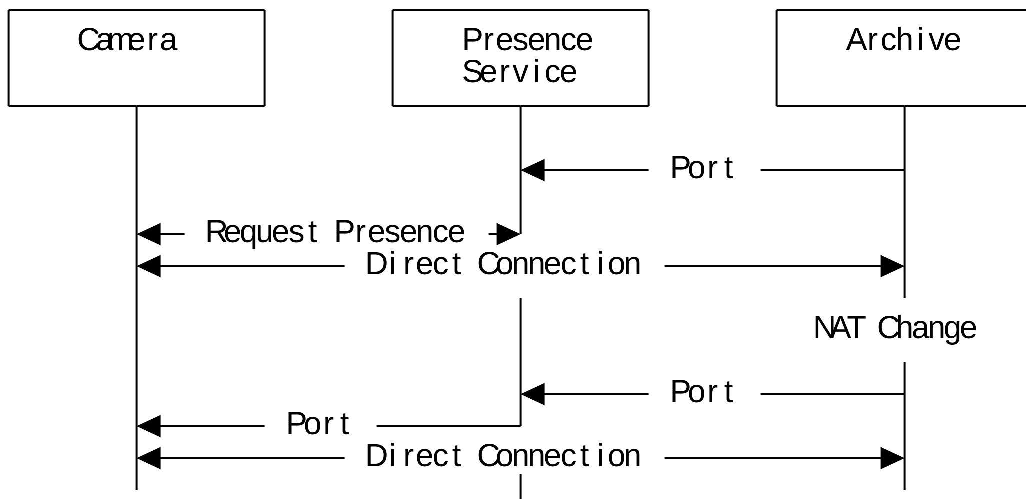

Figure 10: Brokered NAT transit.

Another important concern is the interaction between network agility and NAT transit. Mobile devices frequently operate inside and outside a NAT gated network. Care is required to ensure that a device that establishes a direct connection to another device within the same NAT gated local area network can maintain that connection when it moves to a different network.

1.2.5. IPv6 Agility

RUD makes no distinction between IPv4 and IPv6 transport. In normal circumstances, an application program does not need to be aware of either the initiator or receiver's address(es).

RUD connections **MAY** be established against either or both simultaneously. RUD is equally tolerant of 4-to-6 and 6-to-4 transmission gateways.

In the interest of reducing the impact of IPv4 address exhaustion, public facing RUD services **SHOULD** provide discovery and service to initiators that only have connection to an IPv6 network.

1.2.6. Tunneling

Establishment of an IP tunnel connecting one device to another is required for applications such as onion routing and VPN access.

While the broader requirements of these applications is outside the scope of RUD, consideration has been given for allowing this use case to be supported without causing 'packet inflation' in which every packet submitted as an input results in multiple packets being sent over the network.

Provision for onion routing is likely to be of particular interest. The RUD packet format allows for every bit of the UDP payload to be either encrypted data or an opaque authentication tag.

For example, an observer tapping the network at point A, point B or point C gains no information that allows them to determine that Alice is communicating with Bob. Even correlation of data from all three locations provides limited information through the timing of arrival and almost none that is likely to be of any benefit to the observer.

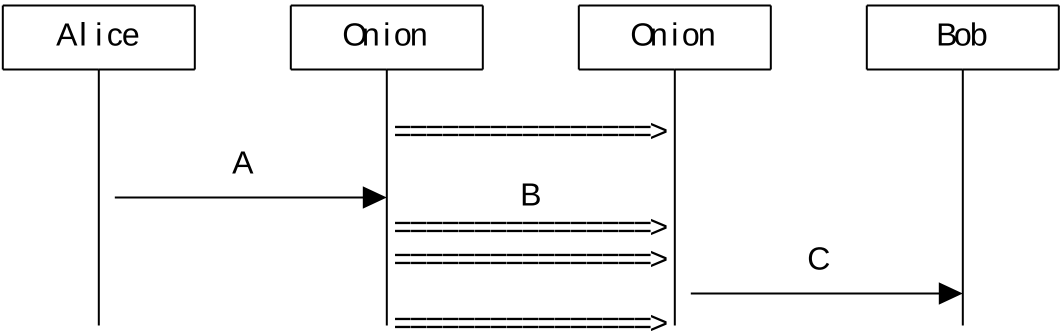

Figure 11: RUD Tunnelling.

2. Definitions

This section presents the related specifications and standard, the terms that are used as terms of art within the documents and the terms used as requirements language.

2.1. Requirements Language

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 [RFC2119].

2.2. Defined Terms

2.3. Related Specifications

2.4. Implementation Status

The implementation status of the reference code base is described in the companion document [draft-hallambaker-mesh-developer].

3. Architecture

Data Encoding

[This part of the specification describes the encodings to be used in the next release. The data encodings in the current reference code used to generate the examples do not match. These should be aligned with QUIC where appropriate.]

Encoding at the packet level presents different requirements to those at higher levels. These needs are better met by the compactness afforded by a place-value approach than the JSON/JSON-B Tag-Value encoding used in the upper layers of the Mesh specifications.

As at the application layer, the use of nested length encodings is avoided on account of the risk that bounds overflow vulnerabilities are introduced into applications.

Integer

Integer values are presented using the same variable length encoding described in section 16 [draft-ietf-quic-transport]

The first two most significant bits of the first byte are reserved to encode the base 2 logarithm of the integer encoding length in bytes. The integer value is encoded on the remaining bits, in network byte order.

2 Bit	Length	Usable Bits	Range
00	1	6	0-63
01	2	14	0-16383
10	4	30	0-1073741823
11	8	62	0-4611686018427387903

Table 1

Values do not need to be encoded on the minimum number of bytes necessary.

String

String values are Unicode strings encoded as a sequence of UTF-8 bytes preceded by an integer length specifier.

Binary data

Binary data consists of a sequence of opaque bytes encoded as the sequence of byte values preceded by an integer length specifier.

Packet Extension List

Packet extensions **MAY** be specified in any data section of a datagram.

Each extension is a tag-length-value sequence:

Tag A sequence of 1 to 16 octets, the leading octets of which are in the range 0-127 and the final octet of which is in the range 128-255.

Length The number of octets in the Value encoded as an Integer.

Value An opaque sequence of the specified number of octets.

The tag 255 is reserved to signal the end of the packet extension list.

Stream Identifier

Every RUD packet begins with a **stream identifier**. Stream identifiers consist of an opaque sequence of zero to 16 bytes.

Stream identifiers are used to allow their issuer to distinguish packets arriving from different streams in circumstances where the stream cannot be inferred from the port. Initiators and responders both issue stream identifiers for their own use during Connection Establishment. Additional stream identifiers **MAY** be issued as a means of defeating some forms of traffic analysis attack.

The semantics and encoding of stream identifiers including determining the length of the stream identifier is sole concern of the issuer which **MUST** ensure that it can correctly interpret the stream identifiers they issue for streams received on a specified port without ambiguity.

Stream identifiers are issued as binary data sequences specified in Connection Establishment packets and as Packet Extensions in Data Packets.

Issuers **MUST** accept every stream identifier issued as a valid identifier for that stream for the lifetime of the connection in which the stream was created.

3.1. Connections

An RUD connection is a bidirectional relationship between two devices secured by a *Connection Shared Secret* established through a key agreement between the devices. Once established a connection **MAY** be persisted for as long as the devices are willing to assign memory resources at each end. Unlike TCP connections, maintaining an RUD connection does not require communication overhead (but maintaining a NAT transit capability might).

3.2. Ports

An RUD port is a synonym for network endpoint. Only ports at which traffic is received are of concern.

Currently two port types are defined:

HTTP Web Service Endpoint A port type that is limited to supporting transactional streams

UDP Network Endpoint A network endpoint that receives Internet communications directed at a particular IP Address and Port.

The collection of ports associated with a connection **MAY** change over time. Connections **MAY** make use of multiple ports simultaneously.

For example, a virtual reality streaming game might establish a single connection to a game coordination host and make use of separate ports to achieve the desired latency and bandwidth characteristics.

3.3. Stream

An RUD stream is an asymmetric bidirectional channel that transmits a sequence of datagrams according to a defined characteristic. Two characteristics are currently defined:

Transactional Stream

Asynchronous Stream The stream characteristic and cryptographic parameters of a stream are immutable for the lifetime of a stream. To effect a forward secrecy exchange, a new stream is created and the existing stream closed.

Each stream is separately identified by one or more Stream Identifiers at each end specified by the party that receives data at that end.

Stream identifiers **MUST** be unique for the lifetime of a stream. Stream identifiers **MAY** be reused after the stream has been closed.

3.3.1. Transactional stream

Streams with a transactional characteristic are used to engage in a request/response communication pattern with a service specified during stream negotiation:

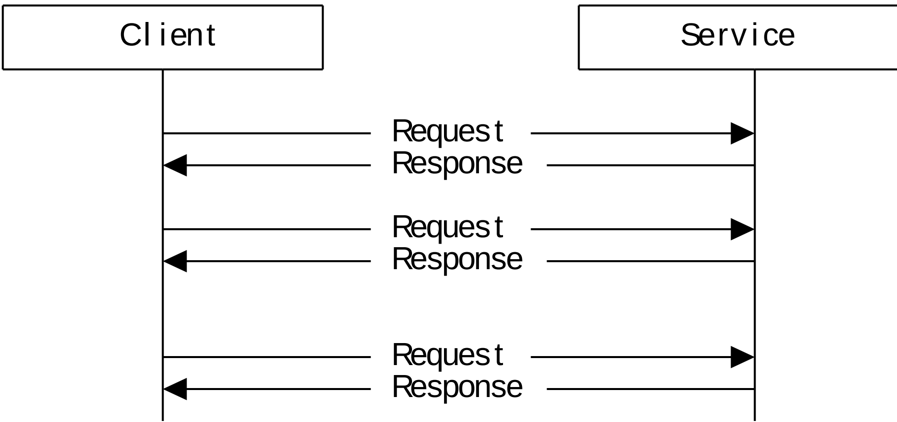

Figure 12: Transactional stream characteristic.

Each transactional stream is asymmetric, but a connection **MAY** contain separate transactional streams going in each direction.

For example, in a notary application, two hosts might offer notary service to the other through the same connection but through separate streams.

3.3.2. Asynchronous stream

Streams with an asynchronous characteristic deliver a stream of datagrams created by the stream originator to the stream receiver. Datagrams carrying control information sent in the reverse direction **MAY** be used to control the flow.

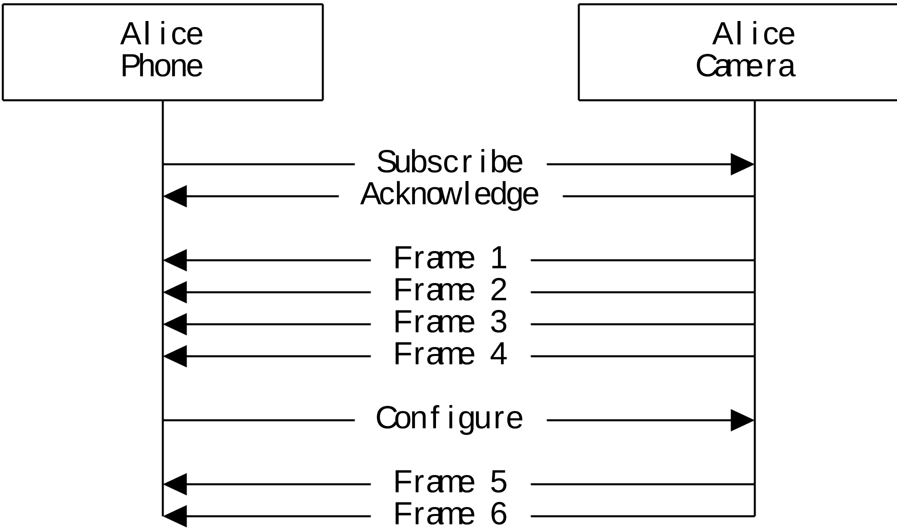

Figure 13: Asynchronous stream characteristic.

Asynchronous streams **MAY** be used for intermittent communications as well as real time streaming media. For example, a network server might use an asynchronous stream to log transaction data to another machine.

3.4. Credentials

RUD follows the Mesh approach of using separate credentials for devices and principals.

For example, the service offered by provider (@provider) is delivered by a fault tolerant collection of three separate hosts. Alice (@alice) can access those services through any one of her four personal devices.

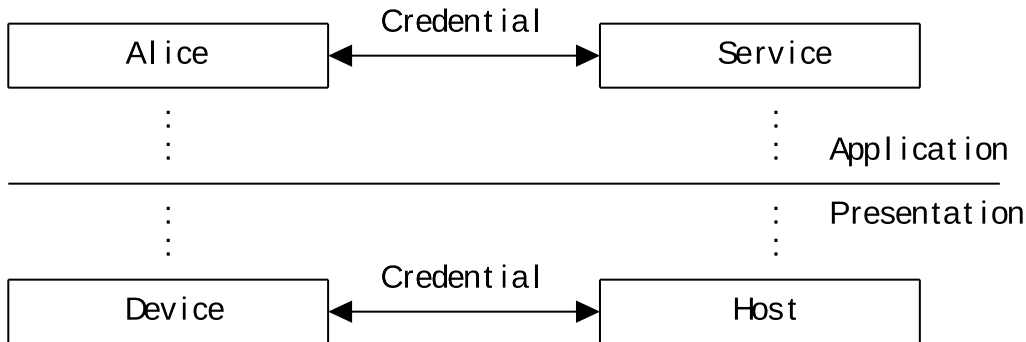

Figure 14: Mesh Layered Credential Model.

RUD makes use of the keys specified in device credentials to perform key agreement and enables user and service credential(s) to be presented for authentication and authorization of the principals.

3.4.1. Device

An RUD device credential is a credential that specifies a public key of one of the supported key agreement algorithms. Two forms of device credential are currently defined:

* Raw Key (e.g. X.448 key)

* Mesh Device Connection Assertion.

It might well be the case that it is more appropriate to pass the Device Connection Assertion as part of the Principal Credential.

3.4.2. Principal

Principal credentials are used to authenticate and authorize a device to act on behalf of the specified principal.

Presentation of Principal Credentials **MAY** be performed at the application layer (i.e. by means of credentials carried in an RUD payload) or as an RUD Packet Extension when a **stream service binding** is requested.

In either case, validation of Principal Credentials is an application concern and thus outside the scope of RUD.

Different streams **MAY** involve the use of different principal credentials. Alice might be authenticated through her Mesh profile in one stream, a PKIX client cert bound to a smartcard in another and a SAML authorization token in a third.

3.5. Datagram Format

An RUD Datagram is by definition the smallest unit of payload data passed from the application layer to the presentation layer or vice versa. A Datagram **MAY** be as large as a complete frame of 4K or 8K video or as small as a single key press or movement of a game controller.

Use of HTTP ports places no upper limit on the size of an RUD Datagram but UDP Datagrams are limited to a maximum number of packets.

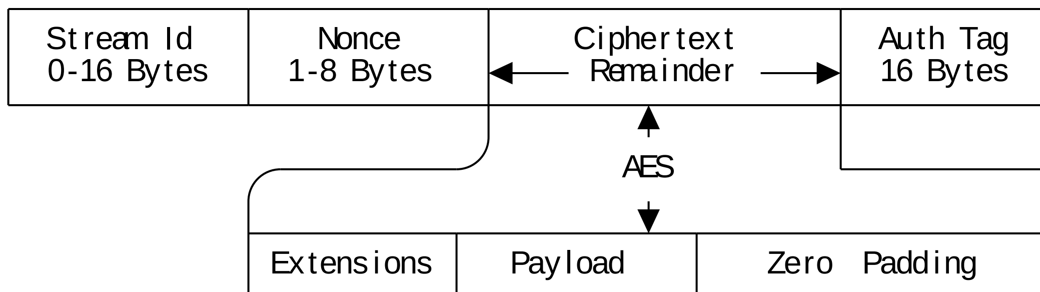

Figure 15: Datagram format.

Requests and Responses **MAY** be split into multiple Datagrams. In this case every datagram except the final one is marked with the continuation flag.

APIs **MAY** choose to present the individual datagrams to applications or reconstruct the complete datagram according to what best fits their circumstances.

3.5.1. Stream Identifier

A sequence of bytes as specified by the recipient during establishment of the connection or stream.

3.5.2. Nonce

The keys used to encrypt and authenticate each datagram are derived by applying a key derivation function to the Stream Primary Key, the Stream identifier and a nonce value specified in the datagram itself.

The nonce is the value of a monotonically increasing datagram counter.

3.5.3. Ciphertext

Ciphertext is the plaintext payload encrypted under a key derived from the active primary key and the initialization vector. Keys are only used to encrypt a single Datagram or continuation Datagram.

Plaintext of a data Datagram consists of a list of packet extension entries followed by a payload followed by zero padding.

3.5.4. Authentication Tag

Associated data is entire Datagram from the first byte to the start of the Initialization vector.

3.6. Packet Format and Packetized Datagrams

UDP provides unreliable transport of datagrams no larger than the maximum IP payload size of 64KB. In practice the largest datagram payload is typically limited to no more than 1260 bytes because the Ethernet specification is stuck in the Middle Ages. It is thus necessary to split large Datagrams into multiple packets and provide sufficient control information that the receiver can reassemble the packets in the right order and request any lost packets be re-sent.

Similar constraints are imposed by a wide range of wireless and wired physical and network transports.

When using Packetized Datagrams, each packet in the Datagram specifies the stream identifier and a packet identifier. Only the last packet in the Datagram presents an authentication tag.

Frame

Packets

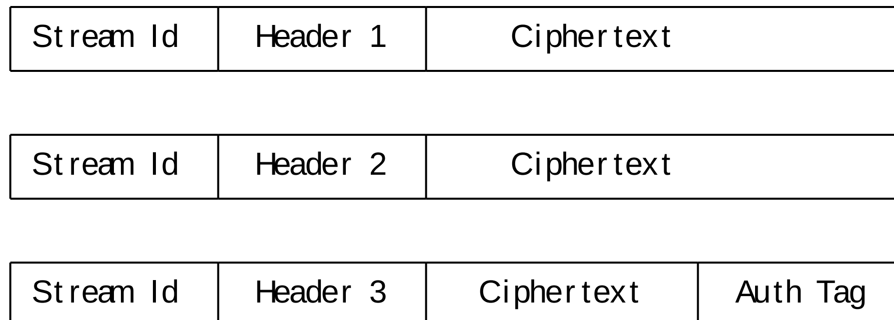

Figure 16: Datagram packetization across multiple packets.

RUD Datagram packetization is analogous to IP packet fragmentation except that it actually works.

3.6.1. Packet Header

The packet header block appears immediately after the Stream identifier. It specifies the information used to reassemble packets in order to form a complete datagram, to acknowledge packet receipt, and other control information.

Datagram index Integer counter beginning at 0 and incremented by 1 for each partial or final datagram sent in the stream.

Packet count Integer specifying the number of packets in the datagram

Packet sent index Integer counter beginning at 0 and incremented by 1 for each packet sent wrapping at 15 bits.

Retransmit index Integer incremented each time a packet is resent. Since the resent data is identical, it is only necessary to acknowledge data once even if multiple copies are received.

Continuation Datagram flag. Integer containing flags information. The low order bit (0) is 0 for a partial datagram and 1 for a final datagram.

Received index Integer index of the first packet being acknowledged.

Acknowledgements Bitfield containing the value 1 for packets that have been received and 0 otherwise.

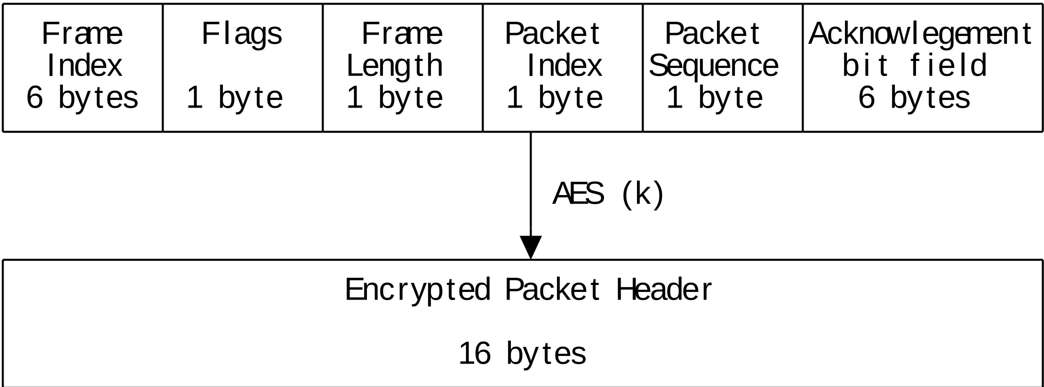

Figure 17: Flow control packet header

3.6.2. Flow Control

TCP flow control is designed to meet the needs of the day. Modern networks have very different needs.

Congestion avoidance is still important and implementations **MUST** take steps to avoid overburdening network resources.

May require additional packets that only contain flow control info. This **MAY** be used to communicate One time Stream IDs.

Congestion control limits **MAY** be specified for the connection as a whole and for individual ports. In situations where multiple Internet connections are available, applications **MUST** enforce flow control limits on each port.

[Here describe the feedback characteristics determined through experimentation.]

3.7. Connection Datagrams

Connection Datagrams are used during the key exchange used to establish a connection. Unlike data Datagrams in which have a single data section, all of which is encrypted, connection Datagrams may contain three separate types of data section:

Plaintext data Unencrypted data section. This is used to pass the information needed to perform the 'mezzanine' key exchange to the responder's credential and to present and respond to challenges.

Mezzanine data Data encrypted under 'mezzanine' key exchange authenticated by the responder's credential alone. This is used to pass information related to the initiator credential.

Ciphertext data
Data encrypted under the mutual key exchange authenticated to the credentials of the initiator and the responder.

The ClientCompleteDeferred Datagram contains all three data sections. The ciphertext data section is encapsulated inside the Mezzanine data section and is thus encrypted twice. Only the Mezzanine section is padded.

Stream Id	Plaintext	Mezzanine		
		Mezzanine Data	Ciphertext Data	Pad

Figure 18: Connection Datagram Format.

A connection Datagram **MAY** contain a payload. This is always carried in the last data section.

Connection Datagrams that only contain plaintext data **MUST NOT** contain confidential data. This requirement may be enforced in a future revision of the specification by limiting use to a 'service discovery' protocol.

4. Connection Establishment and Maintenance

4.1. Key Exchange modes

Three key exchange mechanisms are supported. The choice of mechanism depending on whether the initiator knows the credentials of the host and on whether the host is willing to perform a key exchange operation without challenge:

First Contact The only key exchange mode supported when the client does not know the responder credential. This is a two-round exchange in which the payload of the first round is limited to plaintext data.

Immediate A key exchange in which the initiator knows the responder and offers a complete mutual key exchange in the first message. If accepted by the responder, this exchange requires only a single round which **MAY** carry ciphertext payload.

Deferred A variation of the zero round trip exchange in which the responder returns a plaintext challenge to the initiator instead of immediately accepting the key exchange. This permits mitigation of denial-of-service attacks.

Unlike other RUD interactions, the response to a connection establishment request **MUST** be returned to the network endpoint from which it was received.

The InitiatorHello datagram and InitiatorExchange datagram are special in that they **MUST** be presented in a single packet and **MUST** use the Stream Identifier consisting of sixteen zero bytes.

4.1.1. First Contact

The first contact exchange comprises two round trips:

InitiatorHello [Plaintext] Stream ID, Payload

ResponderChallenge [Plaintext] Responder credential,

[Plaintext] Set of responder ephemeral keys

[Plaintext] Challenge

[Plaintext] Stream ID, Payload

InitiatorComplete [Plaintext] Responder key identifier

[Plaintext] Initiator Ephemeral

[Plaintext] Response to Challenge

[Mezzanine] Initiator Credential + Initiator key identifier

[Encrypted] Stream ID, Payload

Data [Encrypted] Stream ID, Payload

For example...

Key exchange example TBS

Since the InitiatorHello and InitiatorComplete payloads are always sent *en clair*, these are only available for use in querying the public capabilities exposed by the responder. For example, the set of services offered.

4.1.2. Immediate

The immediate exchange comprises a single round trip:

InitiatorExchange [Plaintext] Responder key identifier

[Plaintext] Initiator Ephemeral

[Mezzanine] Initiator Credential + Initiator key identifier

[Mezzanine] Stream ID, Payload

ResponderComplete [Mezzanine] Initiator key identifier

[Mezzanine] Responder Ephemeral

[Encrypted] Stream ID, Payload

Note that in this case the initial initiator payload is only encrypted, it is only authenticated to the responder since the responder has not contributed any form of challenge data to the initiator.

For example...

Key exchange example TBS

4.1.3. Deferred Trip

The deferred exchange is an alternative to the immediate exchange that is performed at the option of the responder. Instead of accepting the connection request immediately, the initiator defers acceptance of the request until the initiator has given a correct response to a challenge:

InitiatorExchange (As for the Immediate case)

ResponderDefer [Plaintext] Responder credential,

[Plaintext] Set of responder ephemeral keys

[Plaintext] Challenge

[Plaintext] Stream ID, Payload

InitiatorComplete (As for the First Contact case)

Key exchange example TBS

4.2. Ticketed Connection

For supporting presence and discovery services.

Ticket specifies a shared secret, session ID, network end point.

Example discovery, Alice's device connects to the provider's discovery service, is authenticated and receives a ticket to connect to a specific host. This allows a connection to be established to that host without the need to perform a connection key exchange.

Example presence, Alice wants to talk to Bob, contacts his discovery service, gets a ticket back allowing Bob to establish a direct, encrypted connection.

In either case, a client **SHOULD** perform a rekey operation as indicated by requirements for perfect forward secrecy, security policy, yada yada.

4.3. Connection Forward Secrecy Rekeying

Rekeying is only necessary to provide forward secrecy. The data encrypted under a single symmetric key is a small fraction of the maximum. But a connection that persists for several years still represents a liability.

[Should we reintroduce the mezzanine packet so we can use the remainder of a packet without rekeying?]

[Packet extension contains the rekey data]

5. Stream Establishment and Maintenance

5.1. Stream Creation and Binding

Streams are created by sending a packet containing a StreamClient or StreamReceiver Packet extension.

The Restart Packet extension is a special stream creation extension that causes the stream in which it is declared to be closed and a new stream opened. The Restart extension is used when restarting the stream datagram index and to change cryptographic parameters.

An Initiator **MUST** specify a stream creation extension in any connection packet containing either a ciphertext extensions or mezannine extensions section.

Senders **MUST NOT** specify more than one stream creation extension in a single datagram. Should a packet with multiple stream creation extensions be received, the recipient **MUST** reject the request.

When a packet contains a stream creation extension, all the extensions specified in that datagram are interpreted as applying to the stream being created. If the request is accepted, it is treated as if it is the first datagram in the newly created stream.

Certain packet extensions are only valid in stream creation.

SID Canonical Stream Identifier. Streams **MAY** be assigned any number of One time use IDs but there **MUST** be exactly one Stream Id.

PKIXC, PKIXO, MMMP, MMMC Used to specify additional stream credentials.

X448, (other ephemeral keys) Perform ephemeral key exchange.

Roll, Make use of the primary exchange key specified of the specified canonical stream ID (as assigned by the recipient).

Encrypt Set packet encryption parameters.

When a stream is created, it is typically bound to a service.

A host might offer Mesh, Callsign and Discovery services. After connecting to the host, a device requests a stream to each of the services in turn. Each stream **MAY** be separately authorized.

5.2. Stream Close

Streams **MAY** be closed when no longer needed. This allows reuse of the stream identifier(s) associated with the stream provided that no ambiguity can arise.

5.3. One Time Stream Identifier

In cases where defeating traffic analysis is of particular importance, a recipient **MAY** assign multiple identifiers to a single stream.

For example, a device connecting issues a set of three stream identifiers to a service. The device receives two packets which are acknowledged together with a further three one-time use identifiers.

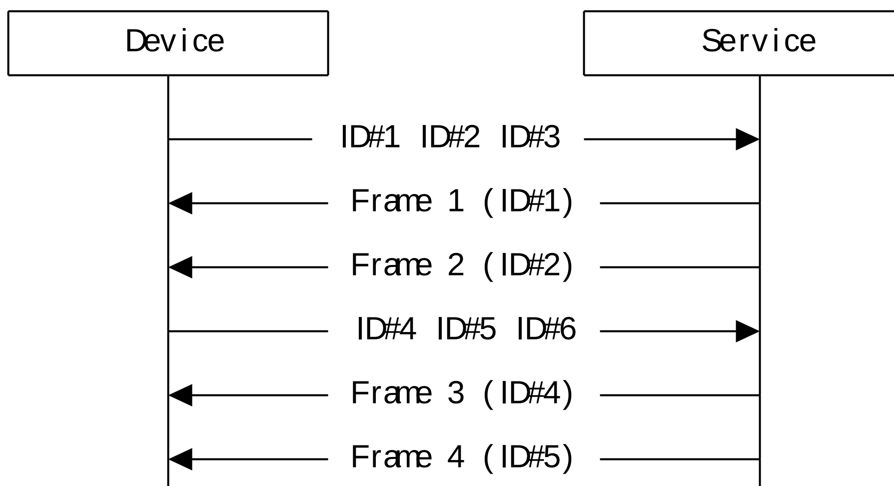

Figure 19: Use of one time Stream identifiers.

A block cipher **MAY** be employed to construct one time IDs that require no state to interpret. The issuer generates a fixed symmetric key. To create a new one time StreamID, the issuer encrypts a block containing a salt (e.g. a unique sequence number) and the stream ID to which the encrypted Id is to be mapped.

To interpret a one time ID, the service simply decrypts the encrypted form to recover the original stream ID, the salt is ignored.

Note that when using this approach it is absolutely critical that the cipher used is a block cipher and not a stream cipher or a block cipher used in a streaming mode (e.g. AES-GCM).

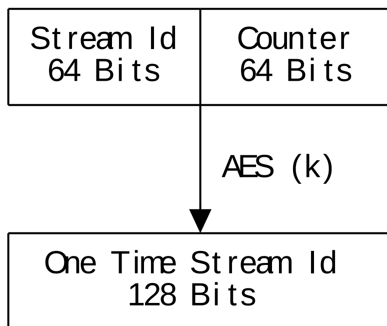

Figure 20: Use of a block cipher to create stateless stream identifiers.

5.4. Stream Rekey

Rekeying the stream requires us to change the stream ID. No other characteristics of the stream are changed.

Stream can rekey by specifying its own rekey data or say 'use the same keydata as this stream Id'.

Thus, a connection with four separate streams only needs to perform one rekey to refresh the crypto for all four.

6. Schema

6.1. Connection Datagrams

6.2. Packet Extensions

7. Security Considerations

8. IANA Considerations

This document requires no IANA actions.

9. Acknowledgements

10. Normative References

[draft-ietf-quic-transport] Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", Work in Progress, Internet-Draft, draft-ietf-quic-transport-34, 14 January 2021, <https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-34>.

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

11. Informative References

[draft-hallambaker-mesh-developer] Hallam-Baker, P., "Mathematical Mesh: Reference Implementation", Work in Progress, Internet-Draft, draft-hallambaker-mesh-developer-10, 27 July 2020, <https://datatracker.ietf.org/doc/html/draft-hallambaker-mesh-developer-10>.