kinesin-rdt

Like QUIC but worse

Overview

- Provides a stream abstraction (like TCP): reliable in-order exactly-once delivery
- Multiplexes multiple independently flow-controlled streams into one connection
- Support for delivering unreliable datagrams
- Explicitly does not implement cryptography or packet delivery, such responsibilities are delegated to a lower transport in a modular fashion
- Support for multipath, including simultaneously using multiple paths and automatic failover
- Support for both "push mode" (where kinesin-rdt pushes packets to the lower transport, employing congestion control as necessary) and "pull mode" (where the lower transport polls kinesin-rdt for packets; kinesin-rdt disables congestion control in this mode, as it is expected that the lower transport provide such functionality)
    - Allows running on many existing protocols, including those based on both TCP and UDP
- Open source (CC BY-SA 4.0, MPLv2)
- This document: kinesin-rdt
- Old kinesin-rdt (extremely out-of-date do not use): kinesin-rdt old

Motivation

This section intentionally left blank

Why not…?

- QUIC?
    - QUIC is a rather complex protocol; much of that complexity generally isn't needed
        - kinesin-rdt is notably not significantly simpler
    - QUIC does not provide enough flexibility
        - QUIC depends on a client/server model, which may not hold
        - UDP is required and QUIC cannot run on other transports, such as a routed overlay network (kinesin)
        - No support for running on existing congestion-controlled protocols such as TCP
    - No native support for multipath
- SCTP over DTLS?
    - Initial handshake too long
    - Message-based, not stream-based, which introduces additional overhead when using stream-based protocols on top (the reverse is generally not true)

    - Unreliable datagrams do not appear to be standardized
    - We could do better
- HTTP/2?
    - Runs on TCP and hence cannot do unreliable delivery
        - This is needed if IP is to be a payload, which is the ultimate goal of kinesin
- A bunch of TCP connections over Wireguard?
    - Implementing an entire userspace networking stack is a pretty flawed solution to the problem
    - TCP congestion control doesn't work very well with a large amount of connections (see BitTorrent)
    - We could do better

Some goals

- Flexibility: the protocol should hopefully work practically anywhere
- Modularity: each component of the protocol should be useful by itself and reusable in other projects
- Concurrency: major implementations of QUIC and related protocols generally hold global locks over protocol state, which is sad
- Learning: the dude who wrote this document would like to learn about network protocols by doing one himself

Applications

- General purpose networking that would otherwise benefit from multiplexing, such as SSH or HTTP
- Game networking, where there is a combination of reliable and unreliable network traffic
- Persistent network connections where roaming is desired, such as SSH
- Media streaming, where deadlines exist and unreliability is allowed
- Overlay network traffic (see kinesin)

Disclaimer

Significant parts of kinesin-rdt were plagiarized[1] from QUIC and other protocols. References are provided. kinesin-rdt does not provide significant benefits over existing protocols. No significant effort will be made to persuade the reader to use kinesin-rdt, not by this document nor by the author. kinesin-rdt is not designed to solve your problems and most likely will not do so. The primary author of this document is not a network engineer nor a professional in any related field. There is no good reason for you, the reader, to use (or even consider using) kinesin-rdt; use it at your own peril. This document is a living document, and it may change at any time. kinesin-rdt is not related to the kinesin motor domain except by name. kinesin-rdt is brought to you by Hellomouse.

[1] The word "plagiarism" is not used here in a literal sense.

License

Definitions

- kinesin-rdt: The protocol described by this document
- Packet: sequence of frames
- Frame: one distinct unit of information in the protocol
- Peer: an entity participating in a kinesin-rdt connection
- Endpoint: one of two termination points of a path, owned by a peer
- Path: any route where data can be transmitted from one peer to another, terminated by two endpoints
- Stream: bidirectional channel where an ordered sequence of bytes can be delivered from one peer to another reliably
- Unreliable datagram: unit of data sent by kinesin-rdt with no reliability guarantees
- Lower transport: a transport used by kinesin-rdt to deliver frames to the peer
- Push mode: mode of operation for the lower transport where kinesin-rdt handles congestion control
- Pull mode: mode of operation for the lower transport where the lower transport handles congestion control
- Variable-length integer (varint): a 1, 2, 4, or 8-byte long integer with a max value of 2^62 (see QUIC)
    - Stolen table (https://www.rfc-editor.org/rfc/rfc9000.html#name-variable-length-integer-enc)

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

- Application: software (or something else) which builds on top of kinesin-rdt

Lower transport

- kinesin-rdt is designed with the specific goal of allowing modularity regarding the lower transport
- Only a few requirements are necessary for the lower transport

Requirements

- Must be able to deliver kinesin-rdt packets
- Must be able to associate received packets to a specific path in a connection
    - If the path abstraction is not needed, only one path per connection should be used
- Must ensure packets are delivered as they are sent and have not been tampered with. Cryptography should be used.
- If cryptography is used, replay protection should be employed

Push mode

- Congestion control handled by kinesin-rdt
- Generally to be used if the lower transport uses a protocol providing no guarantees regarding congestion control, such as UDP or IP
- Lower transport can opt in to path MTU discovery, where the protocol will attempt to automatically regulate packet size

Pull mode

- Congestion control handled by the lower transport
- Generally to be used if the lower transport already provides mechanisms for congestion control or flow control, such as TCP and SCTP
- Flow control for individual streams is still handled by the protocol
- Reliable delivery need not be guaranteed, the protocol will automatically recover from lost packets either way

Connections

- TODO: Connection parameters
    - Max unacked streams accepted
    - Stream initial window size
- Connections are symmetric, not client/server
- Connection establishment
    - Lower layer is expected to create the connection
    - No standard handshake defined
    - The protocol wants an initial RTT estimate for the first path
- Connection termination
    - TODO:

Packets

- Any sequence of one or more frames
- Frames are of a known size so packets are just frames concatenated

Frames

- Unit of data for which pretty much everything is based on
- Frame types currently not defined (TODO)

Streams

- Several streams exist per connection and provide multiplexing
- Streams are reliable
    - Similar to TCP
    - Exactly-once, in-order delivery of a sequence of bytes
    - Lost data will be retransmitted (handled by reliability)
- Streams are lightweight
    - Opening and closing streams should have minimal overhead
    - An application should be able to hold a large amount of streams open without any decrease in performance
    - Unlike multiple TCP sockets, opening multiple streams does not incur overhead related to multiple independent instances of congestion control
- Streams are independent
    - Head-of-line blocking (such as if implementing multiplexing over one TCP socket) is not an issue
    - No ordering guarantees are provided between streams

Stream lifecycle

- Streams are created by sending a stream data frame for a given stream id
- Further data segments are sent with stream data frames with increasing offset, lost segments are retransmitted
- End of stream is signaled by sending a "stream finished" frame containing total bytes sent (corresponding to highest byte number + 1)
    - Implementations MUST NOT send data frames containing data past this limit. If this occurs, reset the stream (see below) and send protocol error
    - Streams are allowed to be half-open, similar to TCP
- When "stream finished" frame received, send "stream finished acknowledged" frame only after all bytes up to the given offset have been received
- Stream is considered gracefully closed when "stream finished acknowledged" frames for both directions have been acknowledged by reliability
- Streams may be reset by sending a "stream reset" frame
    - Immediate teardown of the stream is expected on both ends
    - No further data may be sent, all received frames for the stream must be ignored
    - Should generally not be used except in exceptional circumstances (such as invalid stream state or invalid application state)
    - Data loss is allowed to occur in the reset process!
- TODO: state diagram?

Stream identifiers

- Each stream has an identifier
- Stream identifiers are sent as a varint (max value 2^62)
- Identifiers 0 and 1 are reserved for the link control stream
- All other identifiers may be used for application data
- Each peer has their own stream id space
    - Least significant bit of each id used to denote local/remote stream
    - Peers can only originate streams with LSB = 0 (denoting local)
    - Stream IDs are sent from the perspective of the sending peer, on receive, the LSB is flipped (bitwise xor 1) to transform to local perspective before processing
    - Example: peer A sends a data frame for stream id 2 (“local stream id 2”), peer B would respond on that stream by sending data frames with stream id 3 (“remote stream id 2”)
    - Preserves connection symmetry
- Stream identifiers MUST NOT be reused

Flow control

- Each stream is independently flow-controlled
- Receiver communicates available window size to sender by sending the max byte number in the stream which it is willing to receive
- Stream flow control limits must not be exceeded
    - If limits are exceeded, the stream is assumed to have reached an invalid state
    - The problematic stream must be reset and a protocol error message sent

The link control stream

- Stream identifiers 0 and 1 are reserved for the link control stream
- Message format
    - Chunk format
        - Length, 2 bytes
        - Data
    - If length = 0, zero-length message
      If length in [1, 65534], data should be of that length and this chunk is the last chunk of the message
    - If length = 65535, data should be 65534 bytes and expect another chunk immediately after
    - Less efficient than varint length + data but allows streaming
- Why not use frames?
    - Frames are limited in size by path MTU, which may be highly variable
    - The reliability layer does not perform duplicate detection or ordering guarantees
    - Significant amounts of complexity may be avoided by simply reserving a stream for control messages
- Link control stream is assigned the highest stream priority,

- TODO: describe link control stream message types

Reliability

- Loss detection and retransmission is handled per connection, across all streams and paths
- Selective acknowledgements are used (very similar to QUIC)
- Reliability does not directly retransmit packets, rather, it provides loss and acknowledgement notifications to other components
    - Frames will be retransmitted by components if necessary in a new packet containing a new packet number
    - In the case of streams, retransmissions may be of a different length
- Packets consisting of one or more reliable frames are tagged with a reliability header at the beginning of the packet
- The reliability layer does NOT provide duplicate detection or ordering
    - Frames are expected to be idempotent and tolerant of reordering
    - For control messages requiring ordering and/or exactly-once delivery, the link control stream should be used
- Each packet number is only used once
- Packet numbers are assigned in an increasing manner without gaps
- Reliability tag frame fields:
    - ~~Shard identifier (1 byte): identifies reliability shard for current packet~~
    - Packet number (varint): identifier for current packet
    - Lowest relevant packet number (varint): lowest packet number acknowledgement frames should reference
- Acknowledge frame fields:
    - ~~Shard identifier (1 byte)~~
    - Delayed time (varint)
    - Highest acknowledged packet number (varint)
    - Ack list (series of varints)
- Sharding not considered for now
- TODO

Sharded reliability

- Experimental idea: not sure if a good idea
- To allow better concurrency, reliability may be sharded into several instances
- Reliability messages contain shard identifier along with other data
- Connection starts with one instance of reliability, further shards are negotiated over link control

Unreliable datagrams

- Unreliable datagrams will not be retransmitted or have loss/acknowledgement notifications
- Datagrams are sent on channels (represented by varint)
- Channels are arbitrary and the protocol does not define anything else about them other than the fact that they exist
- No fragmentation performed (see unreliable streams)

Congestion control

- Congestion control is performed per-path and estimates reasonable maximum data rates for each path
- It is updated by packet loss and latency notifications from reliability
- ECN may also be used as rate feedback
- Design is modular (similar to TCP), allowing different algorithms to be used
- Congestion control is only used in push mode, where the protocol controls when packets are sent
- Most elements of the protocol are subject to congestion control, but some parts may opt out (such as unreliable datagrams)
- TODO: THIS DOES NOT WORK

Paths

Path migration (multihoming)

Endpoint verification

- Prevent reflected DDoS

Path MTU discovery

- Ensure packet sizes are sane

Multipath

Implementation notes

Epoch-based congestion control

- Note: currently rejected due to numerous issues
- Goal: ensure congestion control can be performed in parallel

    - Ensure a critical part of the send path is not serialized due to excessive synchronization
- Perform congestion control in epochs (distinct time intervals)
- Data: RwLock<CurrentEpoch>
    - target_rate: u64 (target send rate of current epoch)
    - epoch_start: Instant (start of current epoch)
    - epoch_duration: Duration (max length of current epoch)
    - current_epoch: u64 (incremented every time a new epoch is started)
    - bytes_sent: AtomicUsize (bytes sent in current epoch)
    - prev_epochs: VecDeque<PrevEpoch> (stores previous elapsed epochs)
        - id: u64 (same as current_epoch)
        - target_rate: u64
        - epoch_start: Instant
        - epoch_duration: Duration (actual length of epoch)
        - bytes_sent: u64
- Approximate send path
    - Loop indefinitely
        - Acquire read lock
        - If epoch is over (Instant::now() > epoch_start + epoch_duration)
            - Record current_epoch
            - Release read lock and acquire write lock
            - If current_epoch matches previous
                - Create new PrevEpoch with current values and append to prev_epochs
                - Set epoch_start to Instant::now()
                - Set epoch_duration to something (TODO: not sure)
                - Tick congestion control algorithm and set target_rate to estimated bandwidth of path
                - Increment current_epoch
            - Drop lock and continue loop
        - Otherwise return read lock from loop
    - TODO: something about a CAS loop on bytes_sent
        - Calculate available bytes in current epoch with epoch_start, target_rate, and bytes_sent
        - Update bytes_sent as applicable
        - Return something, probably including:
            - Target rate and currently available bytes, so caller can estimate how long to wait, sort-of, but not actually because other threads may mutate stuff
            - Number of bytes currently allocated to the sender
        - Or don’t CAS, use atomic fetch_add/fetch_sub
- If congestion control needs to update rate estimates immediately, push new epoch
- If allocated bytes were not used, check current_epoch, and if it is the same, perform atomic subtract on bytes_sent

Work queueing

- In certain cases it may be desirable to defer work
- Introduce a tick function that is scheduled periodically externally and executes items in the work queue
- Introduce a timer queue that is similar in functionality to the work queue but instead does stuff More Later™
- Example: reliability services
    - One connection instance has multiple reliability instances to allow greater parallelism
    - Work (incoming reliability metadata, ack processing, etc) is queued into a channel
    - Then, a lock on the reliability state is attempted
    - If the lock succeeds, the channel is drained and work processed
    - If another thread is already holding the lock, it is assumed that they are already working on the queue
    - If the incoming work for a specific instance is too long, processing of work may stop so the thread can yield to others. The work should be scheduled.
    - After a thread drains the queue, it should unlock the mutex and check the queue again to prevent possible race conditions
- Implementation ideas: use HashSet with enum describing work type and additional information
- The target is to avoid having threads wait on locks, especially if they are called from async code
    - If try_lock is not an option, timeouts on lock acquisition should be used to avoid long waits
- What if incoming packets overwhelm queues?
    - The Oh No Queue™ (less stupid name: deferred dispatch queue)
    - If incoming packets fail to dispatch due to full queues, they are appended to the Oh No Queue™
    - On receive of any packet, the Oh No Queue™ is checked to ensure it is not full, if it is, the packet is dropped
    - The Oh No Queue™ should never be written to in normal operation, only cases (that I can think of) are (1) sender completely ignored stream-level flow control, (2) sender is spamming control frames, or (3) receiver is half-dead anyways

Replay protection

- Keep array of AtomicUsize instances, where each bit is a valid nonce
- Keep offset/begin/length
- Structure is circular buffer
- Entire thing is protected by a RwLock, flipping bits needs only read lock, shifting window requires write lock
- Moving the window forward is relatively inexpensive

Stream ready state tracking

- If a stream currently has no outbound data queued and new data needs to be sent, send a message (by channel) to the stream controller informing it that the stream is now ready
- If a stream already has outbound data then it is assumed that a previous instance has already notified the stream controller of pending data
- On the stream controller, the outbound path polls the incoming channel for ready messages and stores
- How to preserve consistency
  - The stream outbound state should be protected by a Mutex
  - The stream controller's ready list should be protected by a Mutex
  - Outbound path should read lock the ready list, take *some* entries (TODO: probably define this properly), then unlock
  - If outbound path drains stream, add that stream into a temporary thread-local list
  - When done, acquire write lock on ready list, then acquire lock on outbound state of every stream in the list and ensure it is empty, if so, remove the stream from the ready list
- Stream priority
  - Ready list can be map, stream id -> last known priority
  - Outbound path can check if priority matches, if not, acquire write lock on ready list and update it
- This way, outbound path is *mostly* parallel
  - … which is probably good enough

Stream implementation

- Outbound path
  - Backing buffer with offset
  - RangeSet of queued segments
  - RangeSet of delivered segments
  - Cannot use only one RangeSet as delayed loss notifications may cause significant unnecessary retransmissions
- Inbound path can be list of received and buffer
- Keep stream inbound and outbound state separate; lock separately as well

Reliability

- Outbound path
  - VecDeque for packet numbers and information
  - BTreeMap for (sorted) retransmission timeouts to packet numbers
  - Eager retransmits: while iterating down packet queue, track most recently acked packet of each path. When encountering packet sent down same path as a more recently acked packet, check to see if it was sent more than 2RTT * var (or another configurable criteria) before most recently acked packet, and if so, perform an eager retransmit

    - https://www.rfc-editor.org/rfc/rfc6298.html
    - Some issues with the previous stuff: the min_acked value (minimum packet number for which to send acks) is used to reduce the size of ack packets. Value is computed approximately from the max RTT and is used to allow reordered packets (which may otherwise be dropped) to be acknowledged. Retransmission timeouts map is likely unnecessary. Eager retransmits can still be performed.
    - Possibly use RangeSet to skip updates for certain ranges as an optimization (TODO: this statement was added on 7/29 and I forgot what it meant)
    - Potentially automatically enable/disable eager retransmit depending on how many paths there are
    - Generate new packet numbers by incrementing an atomic (or value behind mutex if no support for 64-bit atomics)
    - Packets may arrive out of order in queue, have "hole" entries
    - Packet numbers may not necessarily have monotonically increasing "sent" timestamps
- Inbound path
    - RangeSet for received packets
- Sharding
    - Sane value for “max reliability instances”, maybe 16 or something
    - Use std::sync::OnceLock<Box<T>> for lazy initialization of instances, keep preallocated array for concurrent access

Miscellaneous

- Inbound path (dispatch etc) is *probably* parallel
- Outbound path is also *probably* parallel
- Outbound path: keep atomic variable is_blocked (or in scheduler), unset on stream write, congestion control unblocked, or reliability unblocked, set when outbound is blocked
- Evaluate use of https://docs.rs/arc-swap
- Rust feature once_cell (OnceLock, previously known as SyncOnceCell) may get renamed

Current issues

- kinesin-rdt when???™
- Reliability: outbound sequence number allocation failure due to packet queue length limits
    - “Approximately” sync packet queue length to an atomic size variable
    - On failure to allocate sequence number, don’t generate new frames
    - If frames have already been generated, call loss handler
        - Implementation idea: handlers are FnOnce, store handlers in Option, on drop, debug assert handlers to be None
- (Accepted) Evaluate new reliability system
    - Separate reliability from streams

    - To avoid lock contention, have several individual reliability controllers
    - Packets containing at least one reliable frame are prefixed with a “reliability control” frame, which contains a reliability number that will be acked
    - If a packet consists entirely of non-reliable frames (acknowledgements, unreliable data frames, etc), a reliability control frame may be omitted
    - Reliability control frames are also used for congestion feedback
- Is packet overhead too high?
    - ethernet frame header: 14 bytes
    - ipv6 header: 40 bytes
    - udp header: 8 bytes
    - path identifier: 4-8 byte varint, maybe?
    - crypto header (20 bytes total)
        - nonce: 4 bytes, with rekeying
        - poly1305 authenticator: 16 bytes
    - kinesin-rdt packet (about 63 bytes overhead)
        - reliability tag frame: up to 18 bytes (shard id, packet number, lowest relevant packet number)
        - reliability acknowledgement: around 26 bytes in a pretty bad case (shard id, highest packet number, ack series)
        - stream data frame: up to 19 bytes header length (stream id, stream offset, data length)
    - On a 1500 byte MTU link, about 10% overhead per packet (rather significant!)
- Multiple possible ways of doing congestion control
    - Use congestion windows with BDP calculations, like practically every other protocol (QUIC, TCP, SCTP, etc)
        - Upside: proven to work over a really long time
        - Upside: potentially more accurate and responsive due to immediate feedback from ACKs
        - Upside: simply subjecting practically everything to congestion control is probably simpler
        - Downside: potentially tight coupling between congestion control and reliability
        - Downside: acknowledgements must be sent for almost all frames (including unreliable frames) to maintain the congestion window, which may be a significant source of protocol overhead
        - Downside: certain applications of unreliable frames (such as tunneled TCP/IP) may already utilize their own congestion control methods, and layering two instances of congestion control may result in issues
            - This is not necessarily a downside
            - Excluding certain frames from congestion control is still possible
    - Estimate current link bandwidth and apply that as a limit to outgoing traffic, adjust rate limit as necessary
        - Upside: potentially less susceptible to latency and jitter

        - Upside: ability to control when ACKs are sent (for example, reliability tags may be applied to a certain ratio of unreliable frames to maintain congestion estimates)
        - Upside: potentially less overhead due to ACKs on unreliable frames
        - Downside: potentially less accurate than congestion windows
        - Downside: current implementation (see Epoch-based congestion control) may result in unpredictable bursting patterns
        - Downside: providing the ability to exclude unreliable frames from congestion control may potentially lead to congestive collapse
        - Downside: loss of packets does not immediately slow transmission rates as it would with windows
    - Some combination of the two?
        - While transmitting unreliable packets, batch several packets under a single packet number according to reliability
            - Window-based congestion control still used
            - ACK of the packet carrying the packet number is handled as if all packets in the batch are acknowledged
            - Less ACKs needed, however, this may not realistically be important given how ACKs are transmitted
            - Extra complexity
- Pull mode outbound packet queueing may result in infinite loops?
    - Use separate queues for frames which want reliable delivery

TODO items

- Write this document lol
- Fix up references to use proper format
- ~~Remove assorted nonsense and other bad jokes~~
- On new stream, insert id into “used streams” segment list to prevent id reuse
- Determine if the sharded reliability model has negative interactions with other components
- Stream scheduling, especially stream priority
- Timers scheduling
- Evaluate issues with running kinesin-rdt over a protocol which implements congestion control
- Determine when acknowledgements should be sent (ACK batching, etc)
- Interaction of sharded reliability with retransmission timeouts
- Determine how to gracefully close down connections
- Path management
- Determine how differing latencies on different paths should be handled
- Evaluate other strategies for eager retransmit/loss detection
- Figure out stream number limits
    - Stream limits may not necessarily need to exist, stream flow control can be used to signal unwillingness to handle more streams

    - Receiving peer can send “stream flow control limit: 0 bytes” to signal stream limit condition
    - Sending peer should avoid opening more streams until condition is resolved (keep count of streams where flow control limit is 0)
    - This unfortunately involves discarding acknowledged stream data, but this isn’t *that* bad as this only happens if stream limits are hit
    - This implies data may not be removed from the buffer until at least one
    - Additional use exists in temporarily reducing initial stream limits
    - If malicious activity is suspected, the connection can be terminated
    - QUIC does stream limits by (1) assuming opening stream *n* means opening all streams less than *n*, and (2) limiting the max stream number
    - SCTP does not limit streams
    - TODO: elaborate on this more, clean it up, evaluate tradeoffs between this and the QUIC implementation
- Define frame types
- 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.
- Consider possibility of stream out-of-band signaling
    - Short (cannot be fragmented), unordered, and reliable data sent at high priority
        - Not flow-controlled, if recipient does not read fast enough, messages are simply lost
    - Intended for use as out-of-band notifications rather than general purpose datagrams (see unreliable datagrams section)
    - Somewhat similar to TCP URG
    - This can be used to implement out-of-order messaging on top of the stream abstraction
        - Send byte offset of the start of each message in the stream out-of-band, segments can then be reassembled out of order
        - If out-of-band message is lost, may lose efficiency but will not lose data
- Stream out-of-order reads and writes
    - See also: out-of-band signaling
    - Allow streams to be written out of order on sender side, with gaps in the stream (subject to flow control limits)
        - Requires additional RangeSet describing valid ranges in outbound buffer and to disallow writing more than once
    - Allow stream segments to be read out as they are received, without needing to wait for all previous segments to arrive
- Stream deadline-based retransmit
    - Provide option to cancel retransmits for all segments below a given offset
    - Can be combined with stream out-of-order mechanism (see above) to prevent retransmitting expired messages
- Consider whether Nagle's algorithm should be utilized on streams and how to do so

- Current idea generates ACKs by pull mode, cwnd-based congestion control requires both push and pull (or deadlock)
- Very early idea: stream recycling
    - DOES NOT WORK IN CURRENT STATE: not safe against certain replay attacks
    - Two separate systems: one to facilitate recycling application streams, one to facilitate closing and recycling the link control stream
    - Not particularly useful especially given the excessive complexity, but can keep stream identifiers low to reduce 4-6 bytes of overhead per stream frame
    - The following steps are performed on the link control stream
    - For recycling application streams:
    - Stage 1: leader election
        - Send message to peer requesting to become "recycle leader"
        - Peer acknowledges or rejects
        - On failure, retry after random interval
        - It's more or less Raft
    - Stage 2: proposal
        - Leader proposes one stream identifier range that it believes are no longer in use
        - Follower accepts if it agrees, or rejects if it still considers any stream to be active
        - Not decided yet how to encode ranges, especially given stream id perspective-based encoding
        - Ensures both peers agree on which streams are free
        - Repeat as necessary
    - Stage 3: commit
        - Leader concludes proposal stage by sending commit message
        - Upon receiving commit message, peer sends commit ack message and may now reuse previously proposed streams
        - Upon receiving commit ack message, leader may now reuse previously proposed streams
        - Leader state dropped upon sending commit message
    - For recycling link control streams (likely even less useful):
        - Peer proposes to close one link control stream
        - Receiver either accepts or rejects proposal
        - Sending peer is hereby leader
        - Upon receiving accept, leader sends stream end frame for proposed stream
        - After stream is fully closed, the leader requests the follower to reopen the stream
        - Follower reopens stream by sending ping message
        - Leadership dropped after new stream is open
- Stream lifecycle, take 3
    - Note: the following terms are used interchangeably: “window size” and “flow control limit”

    - Streams are opened by sending a stream data frame with the new stream id
    - Stream preamble: stream type (stream, packet, unreliable packet), application-defined data (priority etc)
        - Multiple stream types necessitates preamble so receiving peer knows how to deal with it
        - Application-defined preamble may contain stream metadata, such as “stream in reply to” etc
            - Length restricted to 255 bytes (by length field)
            - Presented directly to the application upon stream open event
            - TODO: this is probably not very necessary, the application can just deal with it themself
    - Initial stream window size negotiated shortly after connection (use sane defaults, like 4 kbytes)
    - Initial window can be reneged, peer B can send (for example) window max 0 to pause inbound stream and then discard all received data
        - Peer A will re-send bytes when peer B later announces increased window
        - Initial window size only applies prior to receiving window update from peer
        - Flow control limit can no longer be reneged after receiving first window update
            - Any smaller limit received after a larger limit will be discarded
            - This prevents out-of-order packets from causing problems as well
        - This can be used to implement flow control for total number of streams
            - Locally-initiated streams prior to receiving window limit greater than initial window size are considered “unacked” (TODO: bad name?)
            - Connection parameter “max unacked streams” advertises how many streams the peer is willing to have in the “unacked” state
            - A peer that has reached the “max unacked streams” limit should no longer initiate any more streams until the condition is resolved
            - Any streams opened in excess of this limit may be reset
    - Stream open:
        - Peer A initiates stream, peer B receives
        - Stream preamble sent as first bytes of stream
        - Stream data immediately follows preamble in stream
        - Can send up to initial window size
        - Peer B replies with window size
            - If stream type is unreliable packet, received window size is expected to be max possible varint ($2^{62} - 1$), as flow control should not be used for unreliable streams
        - Stream preamble only sent by peer A when initiating stream, peer B is not expected to send preamble
    - Stream close: same as before
- Connection parameters should probably be negotiated after connection open

    - Lower transport handles connection open and shouldn’t need to deal with connection parameters aside from protocol version
    - Use sane defaults or disable things until negotiation completes
    - Negotiation should happen over the link control stream
- Multipath session resumption
    - For crypto, negotiate session resumption token and key
    - Token will be sent in resume handshake
    - Key is secret value to be mixed into shared secret for crypto on new path (likely by hashing)
    - Authentication procedures can be skipped as successful AEAD decrypt proves both sides have knowledge of previously negotiated resumption key
    - Forward secrecy preserved in new path by doing DH/KEM as part of handshake
- Out-of-order stream state handling
    - Any stream frame aside from stream reset may construct a stream
        - Do not emit “stream open” event to upper layer until full preamble is received
        - Can emit “stream open pre” event or something similar on stream construct
    - Receive stream reset before stream open: immediately discard stream
- Stream initial window reneging does not work
    - Stream limiting still works
    - receiving peer *cannot* send limit 0 and discard data because limit 0 could be followed by larger limit but the first limit (0) could be lost, causing desync as sender never retransmits
    - Could signal stream limit condition by not increasing limit beyond initial limit
    - Note that reducing the initial window limit at runtime may cause peers to disagree on the window limit for a stream
    - Don't allow changing the initial window limit at runtime?
- Mitigate “optimistic ACK” attacks
    - Skip packet numbers every once in a while
- Split varint8 (up to 8 bytes, max 2^62 - 1) and varint4 (up to 4 bytes, max 2^31 - 1)
- Jumbogram support
    - Allow extended packet length field
    - Long stream data frame
- Rate limit stream creation using same mechanism to rate limit stream count: https://blog.cloudflare.com/technical-breakdown-http2-rapid-reset-ddos-attack/
- Reliability improvements
    - Use one global queue, limited by length (somehow?)
    - Use one additional queue per path for ease of implementing more efficient loss recovery
- Packet queueing: outbound deadline mode with codel
    - Maybe also steal ideas from fq_codel for the stream scheduler
- Infinite streams
    - Offsets wrap at 2^62

    - Split the offset space into 16 “chunks” where 5 are valid at any point
    - No replay issues for reliable streams because 2^58 bytes far exceeds the rekey window up to 64 kB frames and 32 bit counter
    - Potential replay issues for unreliable streams in very extremely specific cases

Documentation issues

- If you are reading these docs and you encounter something that you do not understand, is clearly missing, does not make sense, or any other issues, please add them here

References

It's not plagiarism if there's references!

TODO: acceptable citation format

- RFC9000: QUIC, https://www.rfc-editor.org/rfc/rfc9000.html
- RFC9002: QUIC Loss Detection and Congestion Control, https://www.rfc-editor.org/rfc/rfc9002.html
- RFC6298: Computing TCP’s Retransmission Timer, https://www.rfc-editor.org/rfc/rfc6298.html
- RFC793, RFC9293: Transmission Control Protocol, https://www.rfc-editor.org/rfc/rfc9293
- RFC9260: Stream Control Transport Protocol, https://www.rfc-editor.org/rfc/rfc9260
- RFC8899: Datagram Packetization Layer Path MTU Discovery, https://www.rfc-editor.org/rfc/rfc8899.html
- RFC3168: IP Explicit Congestion Notification, https://www.rfc-editor.org/rfc/rfc3168.html
- RFC8684: Multipath TCP, https://www.rfc-editor.org/rfc/rfc8684.html
- RFC8985: RACK-TLP Loss Detection, https://www.rfc-editor.org/rfc/rfc8985.html
- The BBR congestion control algorithm
    - https://github.com/google/bbr
    - https://www.ietf.org/archive/id/draft-cardwell-iccrg-bbr-congestion-control-02.html
    - https://datatracker.ietf.org/meeting/104/materials/slides-104-iccrg-an-update-on-bbr-00
- The CUBIC congestion control algorithm
    - TODO: fix references
    - https://www.cs.princeton.edu/courses/archive/fall16/cos561/papers/Cubic08.pdf
- https://www.rcsb.org/structure/1BG2