Packetized, Optionally-Reliable, Orthogonal UDP Streams (POROUS)

David Petrizze

Revision 0.2 / November 2021

1 Introduction

POROUS is a realtime networking protocol and Rust library, developed primarily for use in online, multiplayer games. It implements a connection-based layer over UDP that provides ordered delivery of optionally-reliable packets on a number of virtual channels, ensuring effective use of the network through channel prioritization and congestion control.

This document serves to present the architecture and particular algorithms used by POROUS, which is heavily inspired by the library ENet. It is by no means a full protocol specification, but instead serves to document the system as a whole, and may be applicable to other domains as well. As the protocol is still somewhat experimental, various details presented here are subject to change, although the overarching interface and process should remain stable.

2 Architecture Overview

Fundamentally, POROUS is a unicast, full-duplex, peer-to-peer protocol for packet exchange between two hosts. An application can send and receive packetized data, specifying whether each packet must be resent until delivered (reliable) or be sent once and forgotten (unreliable). By sending packets unreliably, an application can send time-sensitive data without the overhead of resends, and by sending packets reliably, the application can rely on efficient, ordered delivery of data. By carefully placing packets on one of several independent channels, robust, realtime transfer is made possible in the presence of network congestion.

To create a POROUS connection, a host object is created and configured according to the application's needs. The host object manages connections to/from remote hosts, each of which send and receive frames (UDP datagrams) on the host object's UDP socket. Each connection has an application-specified number of virtual channels, which represent independent streams of ordered packets. The application can thus connect/disconnect from remote applications,

and send/receive packets on a per-peer basis, specifying who a packet is sent to, and on which channel a packet is sent.

2.1 Host Objects

In a manner similar to TCP, either host maintains a list of active connections (peers) with which it may exchange information. A host may accept new connections by creating a peer object upon receipt of a connection request, and similarly, create a peer object in an attempt to initiate a new connection. The specific connection procedure used by a peer (see Section 2.3) makes this a straightforward process which does not require a peer to have special states for handshaking as a client or server.

Prior to connecting, the host object is configured with any desired connection parameters. When a peer connects, or when an outbound connection request is made, connection-critical parameters are sent as part of the initial handshake. These parameters are then validated against the remote application's before the connection is established; incompatible parameters will terminate a connection.

To integrate well with a realtime update loop, and to provide time to receive out-of-order packets, POROUS is designed to operate on a periodic, step-by-step basis. Each step, a host object processes all frames received on its (non-blocking) UDP socket, followed by sending any pending outbound frames. In order to perform bandwidth estimation and timeouts effectively, it is assumed that the interval between steps is 1) relatively uniform, and 2) small compared to the connection's round-trip time (e.g. one step per 15ms game update for a 75ms round-trip time).

2.2 Peer Objects

A peer object represents a connection to a remote host, communicating directly with a peer object on the opposite end. A peer may exist in one of 5 states: connecting, connected, disconnecting, disconnected, and zombie. All peers begin in the connecting state, and enter the connected state once a connection has been established, at which point user packets may be transferred. When a peer wishes to disconnect, it enters the disconnecting state and eventually proceeds to the disconnected state. After a final timeout, a peer in the disconnected state enters the zombie state, and is deleted by the host. The possible states of a peer object are shown in Figure 1.

If a connection handshake fails, or the remote peer requests a disconnect, the peer enters the disconnected state. Similarly, if no frames are received for some amount of time, the peer enters the zombie state. The disconnected state exists to mitigate the side-effects of duplicated frames; a peer in the disconnected state never re-initializes the connection, and serves to ignore inbound traffic with the same address/port. Once the connection has timed out, the zombie state indicates a peer is ready to be deleted.

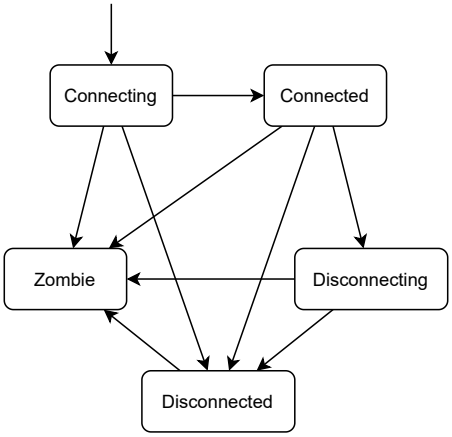

Figure 1: The states of a peer object and possible transitions are illustrated.

2.3 Connection Procedure

While in the connecting state, a peer repeatedly sends connection requests at some interval until a connection acknowledgement has been received. The peer will enter the connected state once both an acknowledgement, and a compatible connection request have each been received. In addition, to properly handle dropped acknowledgements, a peer which is already in the connected state simply acknowledges any further connection requests. Figure 2 illustrates examples of this handshake procedure.

Each connection request contains information about various connection details, such as the protocol version, the number of channels to use, and requested bandwidth limits. If an incompatible connection request is received, the peer enters the disconnected state, terminating the would-be connection without notice. Alternatively, the peer may enter the zombie state immediately so as to minimize the number of peers allocated during a denial-of-service attack.[1]

In addition to connection parameters, a random ID is exchanged as a part of the connection handshake. If a connection acknowledgement is received for an ID which does not match the current request ID, the connection is terminated as stated previously. This random ID improves robustness to duplicated connection requests in a manner very similar to TCP, and is also used to initialize the transfer/receive window described in Section 3.1.1.

In contrast to TCP's 3-way handshake, the 4-way handshake described here both eases peer state management, and simplifies frame specifications. Further, because a 4-way handshake permits peers to connect simultaneously, the connection of applications behind routers performing network address translation (NAT) is made possible without explicit port forwarding rules. Also known as

[1]The specifics of denial-of-service protection at the protocol level are to be determined. However, a proof-of-work mechanism may be of further use here.

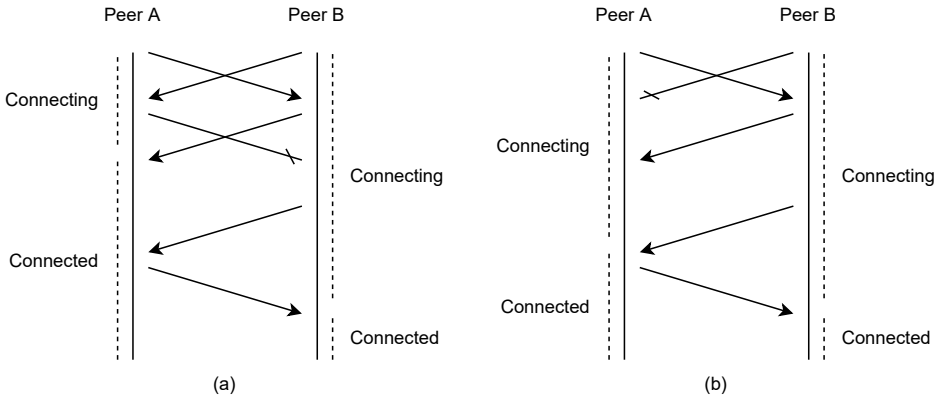

Figure 2: In (a), Peer A's connection acknowledgement is dropped, causing Peer B to later resend its request. Because acknowledgements are sent while in the connected state, Peer B remains able to connect. In (b), Peer B's connection request is dropped, but the handshake continues. Note that in (a), Peer A receives a request followed by an acknowledgement, but in (b), it receives an acknowledgement followed by a request.

UDP hole punching, this is an appealing feature for non-technical users.

2.4 Data Exchange

Once a peer is in the connected state, user packets may be sent. To accommodate a variety of use cases, packets are sent and received per-channel according to one of three modes: *reliable*, *unreliable*, or *passive*. Reliable packets *will* be delivered, unreliable packets *may* be delivered, and passive packets are resent until they arrive, but the receiving channel need not stall for their delivery[2].

All packets received on a given channel will be delivered to the application in the order they were sent. If a reliable packet has been dropped, the channel will cease to deliver packets so as to ensure they are delivered in the correct order. Because channels are sequenced independently, such a stall for one channel does not halt the delivery of packets on another —this is a desirable quality for logically-distinct, realtime data streams. Section 3 details the process by which user packets are sent and received by a peer.

2.5 Disconnection Procedure

To disconnect, a peer enters the disconnecting state and sends disconnection requests until an acknowledgement is received, at which point the peer enters the disconnected state. If a disconnection request is received, a peer in any state other than disconnected or zombie enters the disconnected state (and

[2]Skipping packets that will be resent until acknowledged is somewhat inefficient. However, resends may prove invaluable for time-sensitive packets that arrive in many fragments.

acknowledges the request), thereby terminating the connection. In addition, to properly handle dropped acknowledgements, a peer in the disconnected state will acknowledge any further disconnection requests. This procedure is illustrated in Figure 3.

To ensure peer objects expire after disconnecting, a peer in the disconnecting state always enters the disconnected state after some timeout. Similarly, a peer in the disconnected state always enters the zombie state after a timeout, regardless of any frames it may have received in the meantime. This prevents peer objects from being kept alive by misbehaving hosts, while effectively ignoring benign duplicated traffic.

Either peer may disconnect at any time, but for a connected peer, an effort can be made to ensure any pending packets have been sent prior to disconnecting. In this case, if a disconnection request is received while such pending packets are being sent, it is assumed that the remote peer is not interested in them, and the peer simply disconnects as it would otherwise.

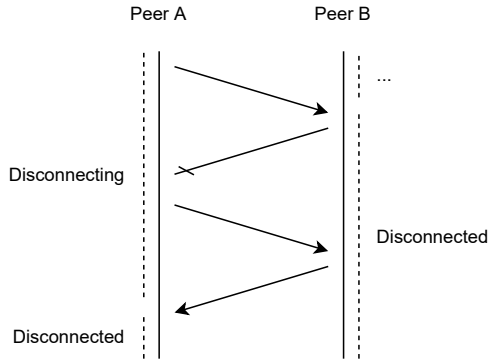

Figure 3: An example disconnection is illustrated. Though the initial acknowledgement has been dropped, further disconnection requests are acknowledged once Peer B is in the disconnected state.

3 Packet Transmission

The process by which user packets are exchanged between peers is divided into two layers: the transfer layer, and the sequencing layer. The transfer layer, a low-level communications layer, provides congestion-controlled and prioritized message delivery between two peers, optionally resending certain messages until they have been acknowledged. The sequencing layer then employs a queuing/dequeuing scheme over the transfer layer so as to provide ordered and deduplicated delivery of packets on independent channels.

The sequencing layer of each peer is comprised of multiple channel objects, the number of which is negotiated before a connection is established. Messages sent by one channel object are delivered by the transfer layer to the channel

object on the opposite end, forming a virtual, full-duplex connection over which which user packets may be exchanged. The overall packet transmission process is illustrated in Figure 4, and the transfer and sequencing layers are described in the following subsections.

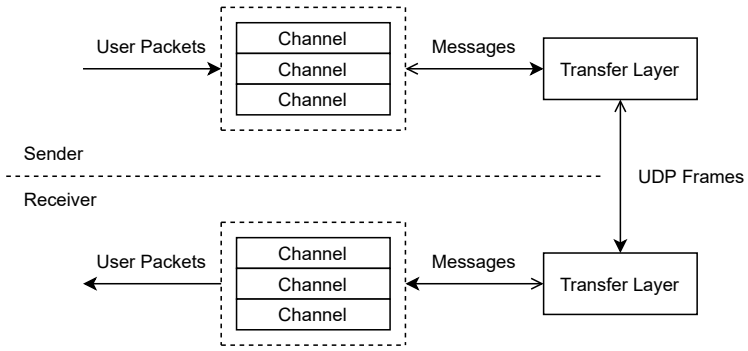

Figure 4: User packets are encoded into messages by the sender's channel objects, and sent via the transfer layer. Received messages are then decoded by the receiver's channel objects, and packets are delivered to the receiving application. Here, filled arrows represent packet data; thin arrows represent acknowledgements and other signaling.

3.1 Transfer Layer

The transfer layer delivers messages according to reliability (reliable/unreliable) and channel priority (low/high). Reliable messages are ensured to be delivered at least once, but no effort is made to sort or to de-duplicate any received messages. However, the transfer layer does ensure that all messages are sent subject to a transfer window, and TCP-like[3] congestion control.

When a message is to be sent, it is placed in a send queue according to its priority. When the application wishes to flush pending data, messages are taken from the front of the send queue and aggregated into data frames, ensuring that no frame's size exceeds a maximum transmission unit (MTU). Assembled frames are then assigned an incrementing sequence ID, sent to the remote peer, and placed on a local transfer queue, which tracks data currently in-transit. The frame assembly process is illustrated in Figure 5.

Once a valid data frame has been received, all of its contained messages are delivered to the corresponding channel, and the sequence ID of the frame is added to an acknowledgement queue. Sequence IDs in the acknowledgement queue are aggregated into data acknowledgement frames, and sent to the remote peer prior to any pending data frames. Once an acknowledgement for a frame has been received, that frame is removed from the transfer queue.

[3]POROUS utilizes a slow start + congestion avoidance scheme based on TCP. However, POROUS is more aggressive in terms of resends and timeouts.

To help ensure that received frames are a part of the current connection (and not, e.g. a delayed, duplicated frame from a previous connection with the same address/port), frames are restricted by explicit transfer and receive windows. That is, a sender only sends frames provided they have a sequence ID within its current transfer window, and likewise, a receiver ignores any received frames with sequence IDs not present in its receive window. Each peer has a transfer window synchronized with the opposite peer's receive window.

Next, to ensure fair use of the network, frames are sent and resent subject to a dynamic congestion window, measured from the front of the transfer queue. Frames are only added to the transfer queue provided they will not exceed this window, and similarly, outstanding frames in the queue are only resent provided that they exist within this window. Congestion control is then performed by adjusting the size of this window in an additive-increase, multiplicative-decrease (AIMD) fashion, thereby regulating the total amount of in-transit data.

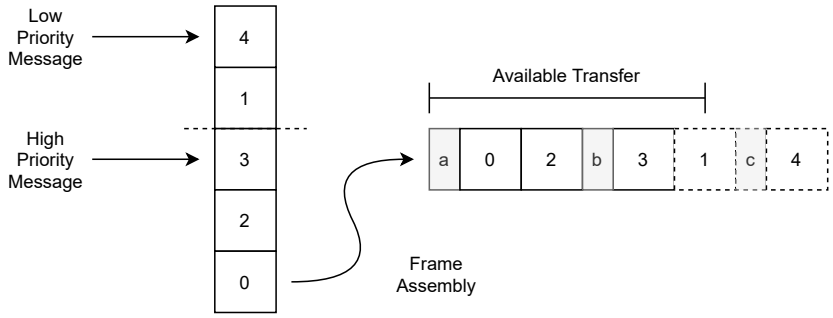

Figure 5: Messages numbered 0–4 are enqueued to be sent reliably. Messages 0, 2, and 3 are marked high priority; they are assembled into frames a and b. Because messages 1 and 4 are low priority, and because the available space in the congestion window is limited, frame b is only partially filled, and neither message is sent for now.

3.1.1 Transfer Window

A peer's transfer window, P_t, is represented by the following modulo range:

$$P_t = [k_b, k_b + N_P) \pmod{X}, \tag{1}$$

where k_b is a value incremented so as to represent the oldest sequence ID in the transfer queue (or the next sequence ID to be sent), N_P is the transfer window size, and X is the wrap-around value for frame sequence IDs. Next, the peer's receive window, P_r, is given by:

$$P_r = [i_b - N_P, i_b + N_P) \pmod{X}, \tag{2}$$

where i_b is a value set to one past the newest received sequence ID. Specifically, when a frame is received with a sequence ID k satisfying:

$$k \in [i_b, i_b + N_P) \pmod{X}, \tag{3}$$

the receiver sets $i_b \leftarrow k + 1 \mod X$.

From here, it can be seen that so long as the sender only sends frames within its transfer window, the receiver will follow, and because its receive window is twice as large, all frames in the transfer window may be received independently of delivery order. An example of this windowing technique for $N_P = 4$ is illustrated in Figure 6. POROUS tentatively uses $N_P = 65536$, and a 4-byte sequence ID (i.e. $X = 2^{32}$).

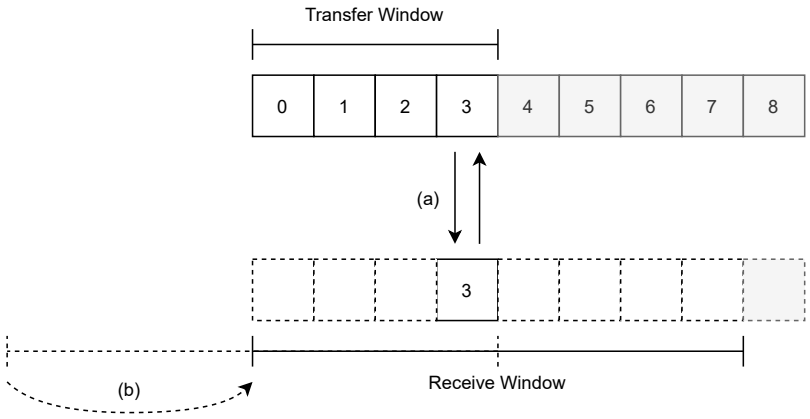

Figure 6: An example transfer for $N_P = 4$ is illustrated, where initially, $k_b = i_b = 0$. After frame 3 is received (a), the receiver sets $i_b \leftarrow 4$, advancing its receive window (b). At this point, the receiver may still receive frames 0–2, and once they are removed from the transfer queue, the sender will advance its transfer window to $k_b = 4$.

3.1.2 Transfer Queue Operation

Once sent, all frames are added to the transfer queue, the total size of which (the flight size) is tracked in bytes. An acknowledgement is expected for all frames, and once received, the corresponding frame is removed from the transfer queue. Each frame added to the transfer queue is marked reliable if it contains any reliable messages, and otherwise, the frame is marked unreliable.

If a reliable frame is determined to have been dropped (see Section 3.1.5), any unreliable messages are removed, and the frame is resent. If an unreliable is considered dropped, it is either a) removed from the transfer queue, or b) converted into an empty, reliable frame (a sentinel frame) which is subsequently sent. In any case, the flight size is updated accordingly, thereby freeing up space in the congestion window.

A dropped, unreliable frame is converted into a sentinel frame if its sequence ID, k, satisfies:

$$k \bmod S = S - 1, \tag{4}$$

where S, the sentinel frame spacing, is an integer divisor of both N_P and X (defined in the previous section). If (4) is not satisfied, the frame is simply

removed from the transfer queue. This sentinel-replacement process ensures a minimum number of acknowledgements per transfer window, and prevents a full transfer window from stalling in the event of heavy packet loss. POROUS uses $S = N_P/2$.

3.1.3 Round-Trip Time Estimation

Ping frames are sent periodically by the transfer layer to measure the connection round-trip time (RTT), at a maximum rate of once every RTT. Upon receipt of a ping frame with some sequence ID, a peer sends a ping acknowledgement frame with the same sequence ID, thereby providing a regular timing mechanism.

The connection will be terminated if no frames are received for a given interval. Thus, in addition to measuring RTT, regular ping requests serve as a keepalive mechanism that prevents the connection from timing out in the event that no packets are sent by the application.[4] In addition, pings are sent separately from the transfer queue, and hence the bandwidth usage of pings (≈ 30 bytes/RTT) is assumed to be negligible.

Each time a ping is acknowledged, the average connection RTT, $T_{rtt,avg}$, is estimated according to RFC 6298 as follows:

$$T_{rtt,avg} \leftarrow 0.875 T_{rtt,avg} + 0.125 T_{ping}, \tag{5}$$

where T_{ping} is the elapsed time of the ping, and $T_{rtt,avg}$ is initialized to 200ms. In addition, the variance of the RTT, $T_{rtt,var}$ is estimated by:

$$T_{rtt,var} \leftarrow 0.75 T_{rtt,var} + 0.25 \left| T_{rtt,avg} - T_{ping} \right|, \tag{6}$$

where $T_{rtt,var}$ is initialized to $T_{rtt,avg}/2$, and this computation is made directly prior to the computation of $T_{rtt,avg}$.

Once these quantities have been computed, a resend timeout value, T_{rto}, is set according to:

$$T_{rto} \leftarrow T_{rtt,avg} + \max\left\{T_{step,avg},\ 4T_{rtt,var}\right\}, \tag{7}$$

where $T_{step,avg}$ is an estimate of the average time between steps. This is identical to the RTO computation presented in RFC 6298, without the minimum value of 1s, and with $T_{step,avg}$ taking the place of the clock granularity, G.

3.1.4 Congestion Window

The congestion window is updated through a standard slow start and congestion avoidance scheme, loosely based on CCID 2 of the DCCP protocol (RFC 4341). Specifically, the method used by POROUS has been adapted to count bytes, rather than frames, in the transfer queue. An example of this process is shown in Figure 7.

[4] It would be possible to measure RTT from data frames and their acknowledgements alone, although this would still require a separate keepalive mechanism. Nevertheless, data-based RTT estimation may be selected in the future.

First, the congestion window is said to be in slow start if:

$$W_c \leq W_s, \tag{8}$$

and is said to be in congestion avoidance otherwise. Here, W_c is the size of the congestion window in bytes, and W_s is the slow start threshold. (These correspond to the `cwnd` and `ssthresh` values used by TCP, respectively.)

Each time a frame is acknowledged in slow start, the congestion window is increased according to:

$$W_c \leftarrow \min\left\{W_c + \text{MTU},\ W_{r,max}\right\}, \tag{9}$$

where $W_{r,max}$ is the maximum congestion window size, defined as:

$$W_{r,max} = BT_{rtt,avg}. \tag{10}$$

Here, B is the negotiated maximum transfer bandwidth in bytes/s, and $T_{rtt,avg}$ is the average RTT computed in (5). The same operation is performed when the window is in congestion avoidance, but only once for each window of data acknowledged without encountering dropped packets.

If any frames in the transfer queue are considered dropped, the sender updates:

$$W_c \leftarrow \max\left\{W_c/2,\ \text{MTU}\right\}, \text{ followed by} \tag{11}$$

$$W_s \leftarrow \max\left\{W_c,\ 2 \cdot \text{MTU}\right\}, \tag{12}$$

thereby backing off just as TCP would. The sender then sets a timer for $T_{rtt,avg}$ seconds, and will not back off again until this timer has expired. This treats closely spaced drops as a single congestion event, and prevents the congestion window from shrinking excessively during brief periods of packet loss.

3.1.5 Frame Resends and Timeouts

Each frame in the transfer queue maintains a timestamp indicating the last time it was sent (or resent). If a frame is not acknowledged within T_{rto} seconds after being sent, it is considered dropped, and the frame is either resent or removed as described in Section 3.1.2.

Next, if the amount of time between any two frame drops exceeds some reset interval, and no acknowledgements have been received in the meantime, the congestion window will be reset. This reset timeout, T_{rst}, is given by:

$$T_{rst} = \rho T_{rto}, \tag{13}$$

where $\rho \geq 1$ is an arbitrary constant. When this timeout is reached, the sender sets:

$$W_s \leftarrow \max\left\{W_c/2,\ 2 \cdot \text{MTU}\right\}, \text{ followed by} \tag{14}$$

$$W_c \leftarrow \text{MTU}, \tag{15}$$

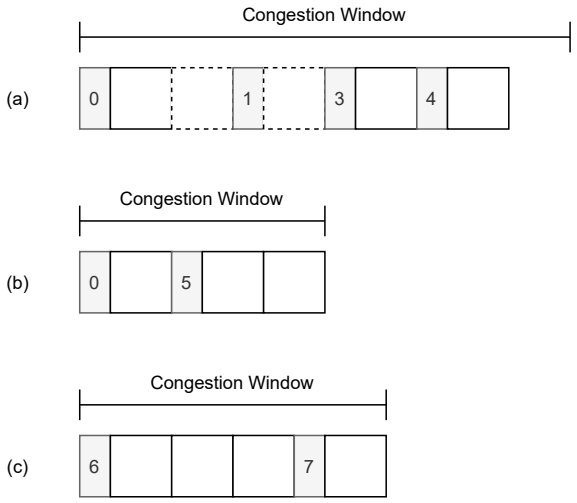

Figure 7: Operation of the transfer queue and congestion window is illustrated. In (a), frames 3 and 4 are acknowledged, and as a result, frames 0 and 1 are considered dropped, their unreliable portions are removed, and the congestion window is halved according to (11). Next, in (b), frame 5 is sent (filling the remaining space in the congestion window), and the reliable portion of frame 0 is resent. Last, in (c), frames 0 and 5 are both acknowledged, causing the congestion window to increase according to (9).

so as to back off effectively in the face of high packet loss.

During a period of significant congestion, it can be seen that the choice of ρ influences how quickly the congestion window will be reset, and also the number of frames that will be resent in the meantime. POROUS tentatively sets $\rho = 8$ in order to reasonably balance resend aggression and congestion backoff.

3.2 Sequencing Layer

The sequencing layer is comprised of channel objects, one for each channel of communication. Each channel object independently delivers user packets to the channel object on the opposite end of the connection, doing so by sending messages over the transfer layer. Channel objects ensure that for any given channel, user packets will be delivered in-order according to one of the three modes described previously: reliable, unreliable, or passive.

When a packet is sent over a channel, the packet is assigned an incrementing sequence ID, and sent to the receiving channel object via one or more *fragment messages*. Received fragments are then added to the receiving channel object's receive queue, which contains packet entries sorted by sequence ID. Once all fragments for a packet have been received, the packet may be reassembled and delivered to the receiving application, provided any previous reliable packets have already been delivered.

To ensure no frame exceeds the MTU, packets are fragmented according to the overhead of frame encoding in the transfer layer, i.e.: `fragment_size` = `MTU` - `frame_header_size` - `message_header_size`. Fragments of reliable and passive packets are sent reliably over the transfer layer, and unsurprisingly, the fragments of unreliable packets are sent unreliably. All messages sent by a channel are sent with high or low priority according to the channel's own priority, as this ensures that no packets are skipped due to having been reordered by the transfer layer.

Next, to differentiate new packets from old, channel objects each maintain a transfer/receive window similar to those used by the transfer layer. Each time a receiving channel object consumes a packet from its receive queue, it advances its receive window, periodically prompting the sender to advance its transfer window via *window acknowledgement messages*. To keep this process stall-free, special, *sentinel packet messages* are sent alongside the fragments of critical unreliable packets, allowing the receiver to skip dropped packets and advance its receive window in all circumstances.

Last, in order to ensure that reliable packets are not skipped, each packet is marked with an optional *dependency*: a packet which must be delivered prior to itself. As packets are sent over a channel, they are marked dependent on the previously sent reliable packet, and such dependency information is included with all packet messages. This allows the receiver to stop and wait for the receipt of reliable packets, should messages be dropped or received out of order.

3.2.1 Channel Transfer Window

A channel object's transfer window, Q_t, is described by the following modulo range:

$$Q_t = [n_b, n_b + N_Q) \pmod{Y}, \tag{16}$$

where n_b is a value incremented in response to received window acknowledgements, N_Q is the size of the transfer window, and Y is the wrap-around value for packet sequence IDs. A channel object's receive window, Q_r, is similarly defined by:

$$Q_r = [m_b, m_b + N_Q) \pmod{Y}, \tag{17}$$

where m_b is a value incremented so as to represent the next expected packet from the sender.

A channel object's receive window is advanced each time a packet is consumed from its receive queue. That is, whenever a packet with sequence ID c is consumed, any previous, incomplete queue entries between m_b and c are deleted, and the channel object sets $m_b \leftarrow c + 1 \bmod Y$, thereby ensuring only future packets are considered.

Once c has been consumed, if m_b is advanced past a sequence id m satisfying:

$$m \bmod M = M - 1, \tag{18}$$

a window acknowledgement containing sequence id m is sent reliably to the sender, indicating that the receiver's window now begins at $m+1 \bmod Y$. Here,

M is known as the window acknowledgement spacing, and it is required that M be an integer divisor of both N_Q and Y.

Once the window acknowledgement has been received, the sender advances its transfer window by setting $n_b \leftarrow m+1 \bmod Y$. To properly ignore old messages, any window acknowledgement for a sequence ID m that is not contained by the current transfer window is ignored. Similarly, if the receiver would send multiple window acknowledgements, only the most recent window acknowledgement must be sent.

It can be seen that this periodic-advancement scheme effectively discerns new packets from old, and maintains synchronization regardless of underlying delivery order. Further, it does so with an adjustable (and minimal) signaling overhead. POROUS tentatively uses $N_Q = 65536$, sets $M = N_Q/8$, and uses a 3-byte packet sequence ID, i.e. $Y = 2^{24}$. This technique is demonstrated by Figure 8.

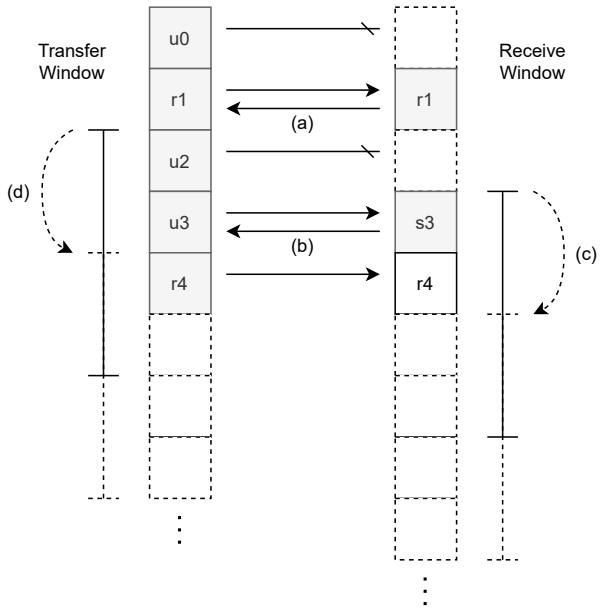

Figure 8: An example transfer is illustrated for $N_Q = 4$ and $M = 2$. Before the current state, packets 0–3 were sent, but all unreliable messages were dropped. After consuming r1, the receiver sent a window acknowledgement (a), advancing the transfer window to $n_b = 2$. Now that r4 has been received, the receiver will consume it and send a window acknowledgement (b) for sequence ID 3. (Because the sentinel entry s3 was not consumed, no window acknowledgement was sent initially.) Afterward, the receive window will advance to $m_b = 5$ (c), and the transfer window will advance to $n_b = 4$ (d).

3.2.2 Sentinel Packets

For any unreliable packet sent with a sequence ID i satisfying (18), an additional sentinel packet message is sent reliably, containing the same sequence ID and dependency as the original packet. When such a sentinel message has been received for a given packet, but that packet is not complete, the packet may be consumed (delivering nothing to the application) under the condition that the newest packet in the receive queue is newer by at least $N_Q - M$ sequence IDs.

This condition is rationalized as follows. If an entire transfer window of unreliable packets has been sent, and all unreliable messages have been dropped, a minimum of one sentinel packet message for every M sequence IDs will be received. Thus, N_Q/M sentinel entries will eventually exist in the receive queue, as shown in Figure 9. So, the first sentinel entry will be consumed, the receive window will advance, and a window acknowledgement will be delivered, thereby advancing the transfer window in the worst-case scenario.

In addition to preventing a stall, this sentinel consumption mechanism allows unreliable packets to be delivered for some time after their corresponding sentinel has arrived (up to $N_Q - M$ sequence IDs). Assuming the transfer window is not filled prohibitively quickly, it can be seen that any receipt bias caused by sentinel packets is mitigated for $N_Q \gg M$.

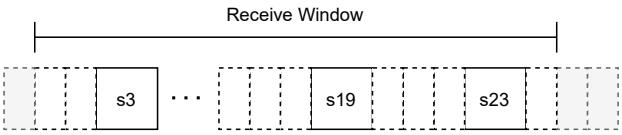

Figure 9: A worst-case receive queue state for dropped unreliable packets is illustrated for $N_Q = 24$, $M = 4$, and $m_b = 1$.

3.2.3 Packet Dependencies

Packet dependencies are indicated by the distance in sequence IDs to the prior packet, and zero if the packet has no dependency. Because the transfer window already enforces a partial ordering of packets, the dependency value is also set to zero if it would be greater than or equal to N_Q. This allows dependency identifiers to be encoded with fewer bytes, and further handles sequence ID wrap-around in an elegant manner. An example illustrating packet dependencies is shown in Figure 10.

When the application requests received data, the receiver iterates the channel's receive queue from front to back, and delivers any complete packets it encounters. To avoid skipping reliable packets, if any packet entry is found with a dependency that has not yet been delivered, that packet is not delivered, and no further entries are considered. Thus, the queue will stall to ensure reliable packets are always delivered prior to any subsequent packets, for any underlying message delivery order. By contrast, if a packet marked passive or unreliable

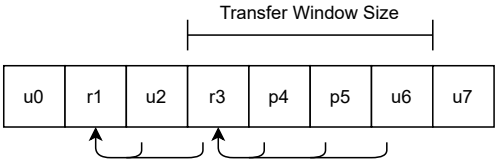

Figure 10: Packet dependencies are illustrated for $N_Q = 4$. Reliable, unreliable, and passive packets are denoted by r, u, and p, respectively. The transfer window ensures u7 will be delivered after r3, and as a result, r3 is not an explicit dependency of u7.

has not yet arrived, the receiver will readily skip it in the event that newer, complete packets exist in the queue. An example of this process is illustrated in Figure 11.

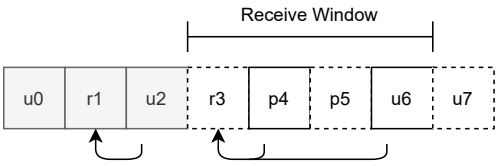

Figure 11: A receive queue and known dependencies are shown for $N_Q = 4$ and $m_b = 3$. Packets p4 and u6 have been received, both of which depend on r3. Once r3 has been delivered to the application, p4 and u6 may also be delivered, although p5 may be skipped if it arrives after r3.