

FB-FR-CCP
Frame Based, Functionally Reliable, Congestion Controlled
Protocol
Revision 1.6

Ivan Beschastnikh

June 8, 2005

Abstract

FB-FR-CCP is a Transport Layer Protocol:

Bidirectional
Congestion controlled (TCP friendly)
Frame Based Functional Reliability

1 Introduction

FB-FR-CCP (Frame Based, Functionally Reliable, Congestion Controlled Protocol) is connection based, bidirectional protocol that works on top of IP. The protocol deals in datagrams that hereafter are referred to as frames. One intended design goal of the protocol is to transport data with as little connection tracking overhead as possible. Inspired by DCCP, the protocol is semi-reliable, a notion that requires a more in depth description of how exactly the protocol functions.

2 Protocol Overview

Under FB-FR-CCP, a single connection requires two endpoints. The first is an initially passive endpoint we term as server, this endpoint listens for incoming connections. The second endpoint is initially active as it connects to the server, we term it the client.

The first exchange is a three way handshake. Thereafter the distinction between server and client is lost due to the bidirectional nature of the protocol and we talk about the Sender and Receiver instead, or Endpoint if we don't wish to differentiate. Therefore, from part of the protocol responsible for data exchange, it doesn't matter who was the initiator or the acceptor of the connection.

Data received by an endpoint from the application level is handed down discretely in the form of what we will refer to as frames, it is not streamed. The frames are then broken down into fragments which are then payloaded onto packets and sent off to the Receiver.

There are size limitations on frames and fragments. In particular, fragment size and other parameters particular to an endpoint must be exchanged during the handshake. The fragment size unless specified differently by the application is set to the MTU for the data path.

It is the receiver’s duty to send back acknowledgements (Acks) within a specified time period which we call TTAW (Time to Ack Window). These packets carry information relevant to congestion control (CC), such as the number of packets missed since the previous Ack. These Acks are not used to determine specific lost fragments so as to retransmit them, they only determine the Sender’s sending rate and help establish Flow Control on the Receiver’s side.

Having completed a frame, the Sender may initiate the transfer of another frame. Upon detecting that the Sender has transferred the whole frame, the Receiver computes the ratio of the number of bytes transferred to the total size of the frame and compares this number to the RELIABILITY constant that has been initialized at start up. If the value is below the reliability threshold, the frame is discarded. Otherwise, the Receiver sends an acknowledgement vector to the Sender. The coefficient is in the range [0, 1], where a 0 indicates a completely reliable connection and a 1 indicates that we never want to resend a fragment to recover from fragment losses. Note that in the last case, acks will still be sent back in order to maximize throughput with congestion control, but lost fragments will not be resent.

This will let the Sender know which packets are missing from the frame on the Receiver’s side. The Sender will then proceed to resend only these packets in the same manner as before, receiving acks from the Receiver for CC. This process repeats until the frame is transferred completely, at which point the Receiver will pass the frame to the application level.

The protocol satisfies the invariant that every frame is either delivered in full or dropped completely depending on the comparison of percentage of bytes transferred in the initial attempt to the RELIABILITY constant.

At any given moment, there can be a number of incomplete frames. The sender prioritizes the frames based on their age, the older the frame, the higher its priority. A connection may be terminated immediately or by waiting on an acknowledgement by any one of the endpoints at any time.

The protocol also implements a tcp friendly congestion and flow controls.

The remainder of this document will describe FB-FR-CCP in detail. The organization is structured around connection states, packet types, and congestion and flow controls.

3 Connection States

Table 1 shows the connection states for FB-FR-CCP. The S is the Server and C is the Client. The connection states determine the types of packets that can be sent between the end points at any particular point of time. Their organization guides the connection and logically divides it into relatively independent units that are easier to analyze separately.

State ID	State Name	Description
1	Disconnected	Initial state for S and C
2	Listening	S’s state while listening for incoming connections
3	CConnecting	C’s state while trying to connect to a Listening S
4	SConnecting	S’s state after a reply to a Connecting C
5	Connected	S’s and C’s state when both have received ack of the other’s acceptance of connection
6	Disconnecting	S’s or C’s state while waiting on the other to ack a disconnect

Table 1: State names and descriptions

4 Packet Formats

FB-FR-CCP has various packet types. The type of a packet determines the type of information it carries as well as the state of the sender and sometimes the state of the receiver as seen by the Sender. Some of the packets specify transitions between connection states, others, such as Ack and Sync packets are used to facilitate the data exchange, whereas the data itself is transferred in the Data and DataAck packets. Table 2 lists all packet types with the state ids in which Receiving and Sending them makes sense.

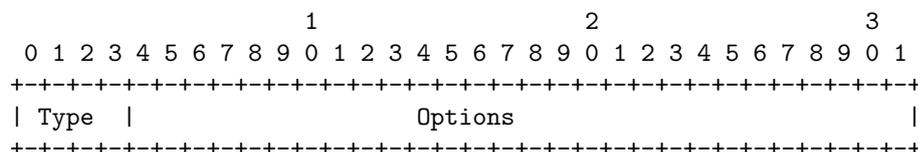
ID	Type	Receiving States	Sending States	Description
0	Request	2	1	Sent by C ; initializes a connection with S ; 1/3 Handshake
1	Response	3	4	Sent by S ; ack of the Request packet by C ; 2/3 Handshake
2	Data	4,5	4,3,5	Transmits application data
3	DataAck	5	4,5	Transmits data with acks
4	Ack	4,5	4,5	Transmits pure acks ; Might include a vector of acks
5	CloseReq	3,4,5	3,4,5	A request to disconnect ; sent by C or S ; elicits a Reset Packet
6	Reset	3,4,5	3,4,5	A reset (kill) of the connection
7	Sync	5	5	Used to resynchronize seq. numbers after bursts or loss

Table 2: Packet types and descriptions

Whenever an endpoint receives a packet that is not in the set specified in the table for the current state, they are to ignore it. Likewise endpoints should not send out packets types in a state other than those specified for each packet type.

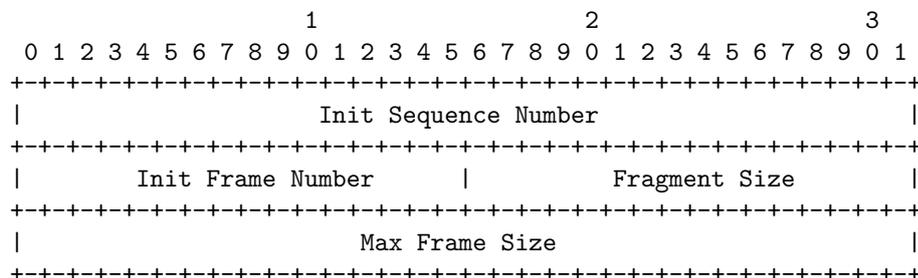
The rest of this section describes the packet types and the information they carry in detail. It might be useful to skip this section and refer back to it when reading the rest of the document.

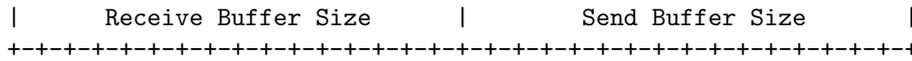
4.1 Generic Header



Every packet in FR-FR-CCP has a generic header to which the formats outlined in the sections below are postfixed. With each packet we identify a packet type (see Table 2), the source and destination ports, and a space for options that may be interpreted differently depending on the type of the packet.

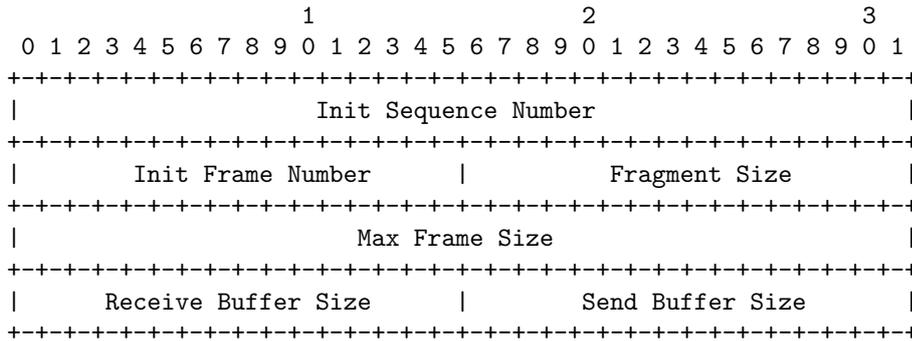
4.2 Request Packet





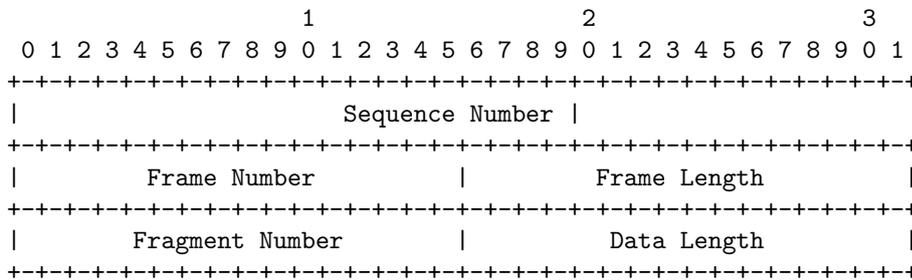
A request packet is sent by the client and initiates connection with the server endpoint. This packet parametrizes the connection to follow. Table summarizes the parameters sent by the client on a Request packet

4.3 Response Packet



Sent by the server endpoint the Response packet is a reply to the Request packet. It must be sent back as soon as possible as the client's initial RTT estimate will come from the pair of Request and Response packets. The Response packet contains the same set of parameters as the Request packet and is identical in format.

4.4 Data Packet



Data packets carry frame fragments. With each Data packet we associate three numbers: sequence number, frame number, and a fragment number.

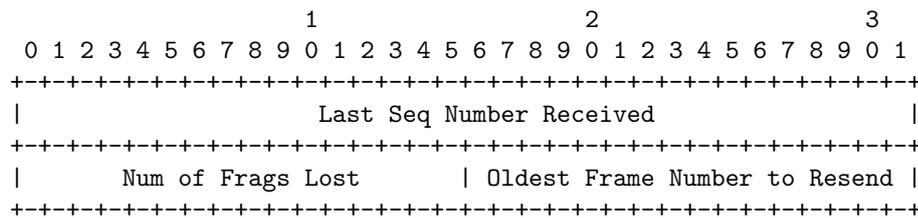
The fragment number cannot exceed `MAX_FRAME_SIZE/FRAGMENT_SIZE` and identifies the fragments location within a frame. Frame numbers and sequence numbers are incremented modulo the maximum number of frames and packets respectively. The sequence numbers are only used to detect losses, and each endpoint maintains its own sequence number independent of the other endpoint, the initial values for these are exchanged with the Request and Response packets.

4.5 Ack and DataAck Packets

There are three distinct forms of acknowledgement. The first is congestion control acknowledgements, which we will term `CC_Ack`. The second type of acknowledgement is a negative vector acknowledgement, which we will term `Vec_Ack`. The last is a synchronization acknowledgement which we term `Sync_Ack`. The three forms play very different roles in the protocol. `CC_Acks` are acknowledgements sent every so often for congestion control. `Vec_Acks` are sent after an incomplete frame is transferred, `Sync_Acks` are responses to `Sync` packets that synchronize the connection. The `DataAcks` are nothing more than `Data` packets with prefixed `CC_Ack` information. All of this is summarized in the following table.

Ack Packet Type	Description
<code>CC_Ack</code>	Sent every <code>TTAW</code> seconds so that the sender may adjust its sending rate
<code>Data_Ack</code>	Carries Data and a <code>CC_Ack</code>
<code>Vec_Ack</code>	Acknowledges a frame and lets the sender know which pieces the receiver needs resent
<code>Sync_Ack</code>	Prompted by a <code>Sync</code> packet, used for estimation of <code>RTT</code> by the receiver

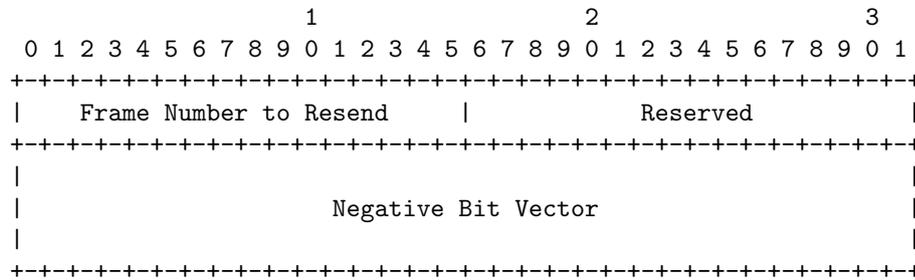
CC_Ack



Congestion control acknowledgements must be sent once per `TTAW`. Their role is to signal the other endpoint of possible congestion in the network. For more see the Congestion Control section.

CC_Ack carries three pieces of information: the sequence number of the most recent data carrying packet, the number of packets this endpoint has estimated as having been lost since the previous transmission of a CC_Ack, and the frame number of the oldest incomplete frame which this endpoint expects to be resent.

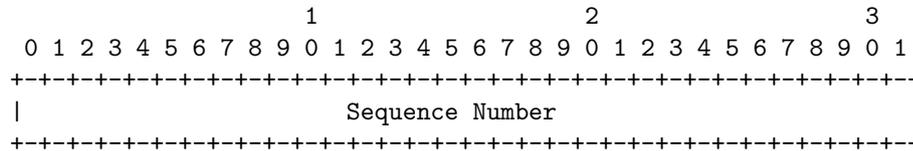
Vec_Ack



The negative vector acknowledgements are a way to tell the sender that the fragments of the frame in question have been transferred at a high enough reliability that the receiver would like the rest of the missing pieces.

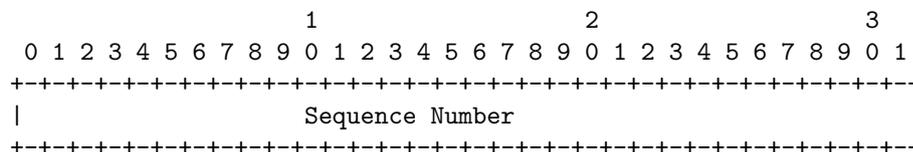
This packet effects the sending priority of the endpoint receiving this packet. If the current frame the Sender is sending is older or the same then it ignores the packet. If the current frame is newer, the Sender must stop sending it and begin resending the frame specified in the Vec_Ack packet by resending only those fragments that are specified by the Negative Bit Vector field in the Vec_Ack.

Sync_Ack



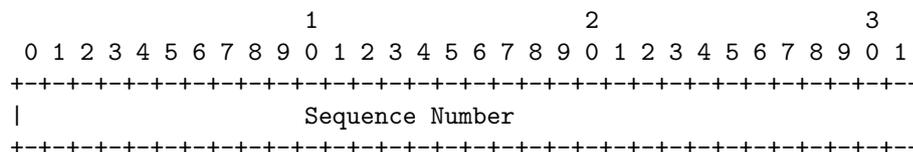
The Sync_Ack packet is sent as a response to the Sync packet and is used to recompute the RTT estimate on the receiver side. In the future it might be extended to buddle a set of other options that might need to change in the middle of a connection.

4.6 CloseReq Packet



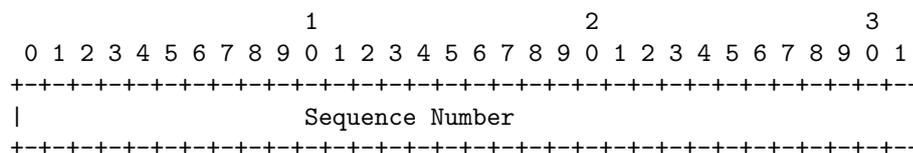
A close req packet is sent with only a sequence number as its payload and is a nice way to close a connection.

4.7 Reset Packet



This packet is sent with only a sequence number as its payload and is a last resort method to close a connection.

4.8 Sync Packet



Sync packets are meant to stabilize the congestion control. Paired with a Sync_Ack acknowledgement, RTT can be estimated.

4.9 Checksums

Because the first prototype of the protocol is built on top of UDP, some features provided by UDP take care of alleviating the problems usually associated with programming directly at the IP level. First and foremost, UDP takes care of removing multiple copies of a packet and has port fields and address fields on every packet.

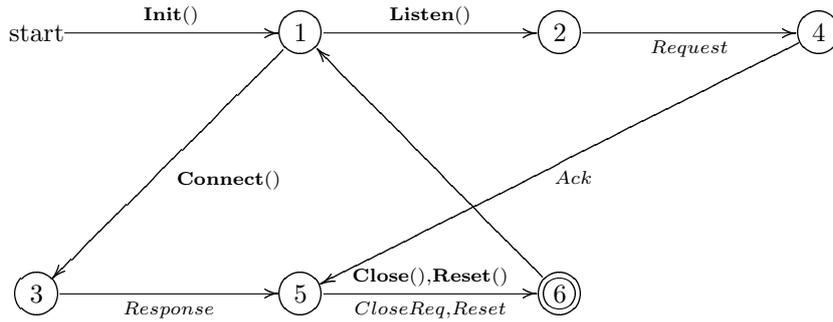


Figure 1: DFA representation of the protocol connection states

The other handy feature that we will rely on is checksums. UDP checksums cover all of the datagram and therefore obviate the need to introduce checksums directly into FB-FR-CCP at this stage, instead we let UDP take care of checking and prepending the checksum of the outgoing and incoming packets.

5 Connection

Figure 1 is a summary of the first two tables in the paper, depicting a deterministic finite state machine where the states are Connection States from Table 1 (identified by their ID from the table), and the transitions (edges) are packet types from Table 2. All the transitions which are missing from the picture are taken to be invalid and are ignored, i.e. they are identity transitions, so for example a Request packet would be ignored if received in a Connected state (ID 5).

Note that the transitions for some of the states depend on factors other than packet types, timeouts might also trigger state transitions for example.

Note that the details of the source network address and port numbers for each packet have been left out below. During the handshake, these are cached and then validated for each incoming packet.

The Following sections will break up the connection into three separate parts- the Set Up, the Handshake, Data Transfer and Termination. Terms and their explanations relevant to each step may be found in the tables relegated to page 10.

5.1 Set Up

Initially both endpoints are in the Disconnected state. To transition into the next state, the protocols at both endpoints must have defined the connection variables summarized in Table 3. Just as with TCP, an endpoint may choose to become a server or a client depending on whether it chooses to listen for a connection or send a Request packet to a listening connection.

5.2 Handshake

Client The client initiates the connection by sending a Request packet. The packet must include the parameters in Table 4. The client then enters CConnecting state. If the server does not reply within a `CONNECTING_RESPONSE_TIMEOUT` seconds threshold, the connection packet is resent. After `CONNECT_ATTEMPTS_THRESHOLD`, the client will stop retrying and signal an error to the application.

Upon receiving a Request packet, a server must reply as soon as it can with a Response packet. This pair of packets (Request and Response) are used to compute the initial RTT (round trip time) estimate by the client.

Server After responding with a Response packet, the server enters the SConnecting state. If the Response packet is lost, the server will wait for the client to retry with the Request packet before sending another Response packet. After `CONNECT_TIMEOUT` second, the server will drop the connection and return to Listening state.

Once the client receives a Response packet from the server, it enters the Connected state. The next packet to come from the client, unless it is a Reset packet, must be an Ack packet and it will place the server into Connected state.

The bidirectional nature of the connection allows us to deal with a “half connection,” and associate with it a Sender and a Receiver both of which may be either of the endhosts.

To disambiguate the attributes of a half connection as seen by an endpoint in Table 3, we prepend “R_” for remote and “L_” for local. As an example, the local `FRAGMENT_SIZE` as seen by an endhost is denoted `L_FRAGMENT_SIZE` and the remote attribute becomes `R_FRAGMENT_SIZE`.

5.3 Data Transfer

Data Transfer under FR-FCPP can be viewed at two different scales. At the fragment or packet level, we are sending fragments of a constant size, not caring about reliability, determined only to send as much as possible under the rate controls.

At the frame level, we are filling up variable sized frames with fragments (all of a constant size except perhaps the last one). After we see that the Sender has moved on to the next frame, we compute the percentage (in bytes) of the frame received (by deducing the number of missed fragments) and either keep the frame around or throw it away depending on the `RELIABILITY_COEF`.

If we keep the frame then we ask the Sender to resend the missing pieces of the frame using a bit vector to represent the missing fragments in the frame and do not let the sender move on to newer frames until this particular one has been completed.

The retransmission of missing fragments happens as before, only now we are determined to receive the full frame and if the frame is incomplete after all the fragments have been retransmitted, we repeat the process as necessary until we received the full frame. Once the frame is complete, we let the application have the frame and allow the Sender to send us new frames.

5.3.1 Receiver

Let the oldest frame the Receiver is interested in be `Oldest_Frame`, the newest frame which we have seen so far in our `FRAME_WINDOW` to be `Newest_Frame`, and let the sequence number of the last data packet received be `Last_Seq_Number`.

Every data carrying packet (a Data or a DataAck packet) has a sequence number, a frame number and a fragment number. Initial sequence and frame numbers are established during the handshake. Fragment numbers are only relevant when talking about a specific frame number. Fragment numbers start at 0 and increment with each data fragment, also each subsequent data packet increments the sequence number independently of the frame number- this lets us determine how many packets were sent or were lost or delayed between the reception of any two data carrying packets. These three attributes determine the actions of the Receiver.

Given a data packet D with attributes `Seq_Number`, `Frame_Number`, `Frame_Len`, `Frag_Number`, `Data_Len`, the Receiver processes D as follows.

First we run the attributes through a set of common validation rules and ignore D if any of the following are true.

- $\text{Seq_Number} \leq \text{Last_Seq_Number}$
- $\text{Frame_Len} \neq$ stored frame length for the frame Frame_Number
(only if we have a record of this frame already)
- $(\text{Frame_Len} > \text{MAX_FRAME_SIZE}) \vee (\text{Frame_Len} < \text{FRAGMENT_SIZE})$
- $(\text{Frag_Number} + 1) \times \text{FRAGMENT_SIZE} > \text{Frame_Len}$
(note that Frag_Number starts at 0)
- $\text{Data_Len} > \text{FRAGMENT_SIZE} \vee \text{Data_Len} = 0$

Note that by this point, the packets have passed the UDP data checksum, therefore a bad attribute is not due to data corruption.

Define the $R(X)$ to be the boolean result of the following inequality:

$$(\text{Bytes Received for a Frame } X) / (\text{Total Number of Bytes in a Frame } X) \geq \text{RELIABILITY_COEF}$$

A New Frame Number ($\text{Oldest_Frame} \leq \text{Frame_Number} \leq \text{Newest_Frame}$)

Let $M = \min(\text{FRAMES_WINDOW} - (\text{Newest_Frame} - \text{Oldest_Frame}), (\text{Frame_Number} - \text{Newest_Frame}))$

$M = 0$: D doesn't fit into our **FRAMES_WINDOW** and we therefore ignore D.

$M > 0$: D has a frame number in the future. This causes the Sender to allocate a buffer of size Frame_Length to accommodate the new frame specified by Frame_Number .

$(M > 1) \wedge (R(0) = \text{true})$: Whole frames were lost or delayed. We send $M - 1$ negative vector acknowledgements filled with 1s to indicate that the Sender received no fragments for any of the $M - 1$ frames between Newest_Frame and Frame_Number . Each of them will carry a Frame_Number attributed of $(\text{Newest_Frame} + k)$ for $k \in \{1, \dots, M - 1\}$.

The data portion of the packet is then placed within the newly allocated buffer for the frame Frame_Number at a location offset by $\text{Frag_Number} * \text{L_FRAGMENT_SIZE}$.

Note that the *min* in the definition of M tells us to discard the frames that are outside of our scope of frame vision controlled by `FRAMES_WINDOW`.

Urgency

For each incomplete frame with $\text{frame_number} \in \{\text{Oldest_Frame}, \dots, \text{Newest_Frame}\}$

- $\text{Urgency}[\text{frame_number}]$ is incremented by one. Initially $\text{Urgency}[i] = 0$ for all i . The value for each frame_number determines when to send another Ack Vector to the Sender urging the Sender to resend the missing fragments of the incomplete frame in which the Receiver is still interested.
- If $\text{Urgency}[\text{frame_number}] \geq \text{URGENCY_THRESHOLD}$ then a `Vec_Ack` packet is sent with the `ACK_VECTOR_OPT` option set to 1, and a compressed bitmap describing the state of the incomplete frame. Each of these packets will carry the attribute `Frame_Number` set to the frame_number .

The `Vec_Acks` tell the Sender that there are incomplete frames to be dealt with, and that they should be attended to first. The missing fragments details are sent so that the Sender will be able to resend them. The Sender must prioritize the sending of fragments based on the age of their frames: the oldest receiving the highest priority.

A Known Frame Number ($\text{Oldest_Frame} \leq \text{Frame_Number} \leq \text{Newest_Frame}$)

$\forall \text{frame_number} \in \{\text{Oldest_Frame}, \dots, \text{Frame_Number}\}$

- $\text{Urgency}[\text{frame_number}]$ increments by one and if $\text{Urgency}[\text{frame_number}] \geq \text{URGENCY_THRESHOLD}$ then we send a `Vec_Ack` back to the Sender urging the resending of the missing fragments for each incomplete frame older than the one we've received.

$\forall \text{frame_number} \in \{\text{Newest_Frame}, \dots, \text{Frame_Number}\}$

- $\text{Urgency}[\text{frame_number}] = 0$

Notes

- `Vec_Ack` for an incomplete frame will be repeated until a fragment destined for this or older frame is received.
- An `URGENCY_THRESHOLD` of 1 will squeeze the frame window to as little as two frames, as an incomplete frame will request its missing fragments immediately after a fragment for a new frame has been received.
- Upon receiving the last fragment for a frame. The Receiver delivers the frame to the application.
- To make sure that the frame is removed from the Sender's buffer space, the frame needs to be implicitly acknowledged by a `CC_Ack` which will specify a frame number attribute that is newer than the frame that has been completed, thus allowing the Sender to free its buffer space of this frame.

5.3.2 Sender

Note that the Sender's set of variables is independent from its other half connection in which this Sender is a Receiver.

Each frame sent by a Sender is associated with an identifier that is initialized and exchanged during the handshake. The initial value of this identifier which from now on we call the frame number is initially generated by each endpoint and exchanged during the handshake. The recommended procedure to generate the initial frame number is use a randomizer modulo $2^{\text{FRAME_ID_LEN}}$, as the purpose of this identifier is two fold: to map packets to frames and to alleviate the danger of spoofing of the connection. The latter is effective because the Receiver is aware of only `FRAMES_WINDOW` frames at a time and drops packets for which the frame number deviates outside of the window. Every successive frame increments the working frame number by one.

Let the newest frame that Sender is sending have the frame number `Last_Frame`.

The Sender maintains two queues for allocated frames. The first is the `Current_Queue`, the second is the `UnAacked_Queue`.

The `Current_Queue` is a priority queue, prioritized by the age of the frames, the older the frame, the higher its priority. The problem of the wrapping of frame numbers is solved by prioritizing in a small frame space limited naturally by queue size constraints.

The `Current_Queue` contains frames that still need some or all of their fragments sent. Each frame in this queue has a fragment bitmap to let the Sending mechanism know exactly which fragments need to be resent. After being resent, the frame is moved to the `UnAacked_Queue`.

The `UnAacked_Queue` contains frames that have been already sent once, but might need some or all of their fragments resent in the future. Thus, depending on a Receiver's Ack pattern, a frame from this queue may move to the `Current_Queue` upon the arrival of a `Vec_Ack`, or it might be deallocated if the Receiver indirectly acknowledges the frame with a `CC_Ack`.

A New Frame Before allowing a new frame to enter the queueing system, the Sender discards the frame if it has an illegal frame length.

Once the frame has been decided as valid, it receives a frame number `Last_Frame + 1`, and it is fragmented into `FRAGMENT_SIZE` sized fragments, except perhaps the last one. Each one of these is enumerated starting from 0. A vector map filled with 0s is associated with the new frame and the frame is then enqueued into the `Current_Queue`.

Sending a Frame

The frame to be sent is taken from the top of the `Current_Queue` and the fragments specified by the associated bitmap are sent in order.

Once all the fragments have been sent, the frame is moved from the `Current_Queue` to the `UnAked_Queue`.

Asynchronous events The Sender must respond to asynchronous Ack packets arriving from the Receiver that have to do with the frames in the queues. The `Vec_Ack` packets trigger a resend of the fragments specified by the bitmap on the packet (once ored with the local copy). `CC_Acks` may trigger a deallocation of the frame due to an implicit positive acknowledgement.

Receiving a `Vec_Ack` Let the attributes of the arrived `Vec_Ack` be `Frame_Number` and `Bit_Vector`.

If frame `Frame_Number` is in the `Current_Queue`, update its vector by bitwise or'ing `Bit_Vector` with the old bitmap.

If frame `Frame_Number` is in the `UnAked_Queue`, move it to the `Current_Queue`.

Receiving a `CC_Ack` Let the `Oldest_Fame_Number_to_Resend` attribute on the `CC_Ack` packet be `Oldest_Frame_Number`.

Deallocate frames from `Current_Queue` and `UnAked_Queue` that are older than the `Oldest_Frame_Number`.

5.4 Termination

Termination can be initiated in two ways by either endpoint. The first way is to send a `CloseReq` packet. This packet will transition the Receiving endpoint into Closing state and will alert the application of the new state. An Ack with a `CLOSING_OPT` option set to one is the only valid acknowledgement of the `CloseReq` packet. After `CLOSING_TIMEOUT` seconds the Sender resets the connection and all packets drops all packets coming from the other endpoint.

A second way to initialize termination is to send a Reset packet. Packets of this type are not acknowledgeable. A Reset packet initiates immediate termination of the connection by both the Receiver of the packet and the Sender.

6 Sync

Sync packets are meant to estimate RTT (round trip time). Upon receiving a Sync packet, an endpoint is required to respond with a SyncAck packet as soon as possible. RTT is used to determine the WINDOW size, these are specific to each endpoint.

7 Rate Controls

Rate controls ensure that the Sender does not overrun the Receiver with its sending rate, and that the protocol is TCP-friendly towards other connections that are coexisting in the network at the time.

Congestion and flow controls control only the rate of Data or DataAck packets, Acks and other packet types are not counted towards the rate. The reason for this is because the overhead is not large enough to make for a substantial contribution- Acks are sent only every TTAW seconds, and are only a small fraction of the full MTU of the connection, even on relatively slow links. Therefore their impact on the friendliness of the connection is minimal. Likewise, Acks are processed on the Receiver's side without being buffered, thus there is no danger of overrunning any buffer.

Note: that the initial implementation is a user level implementation and therefore does not take into account Kernel space buffers which may indeed be overrun but over which the implementation has no control.

7.1 Congestion Control

The CC_Ack acknowledgment packets determine most of the congestion control mechanism.

Both sender and receiver will maintain congestion windows (CW) that keeps a count of the number of packets in flight.

A CC_Ack packet (or a DataAck packet) must be sent by both endpoints at least once every TTAW (Time to Ack Window) seconds. It provides information to the Congestion Control mechanism so that it may determine whether the current CW (congestion window) is too large and should be shrunk or if the network may allow for a higher load.

Let Last_Seq, Num_Lost, and Oldest_Frame_Number be the attributes of the received CC_Ack.

If an endpoint misses an Ack within a TTAW time period, an endpoint halves its sending rate by halving the WINDOW. Note that the rate cannot go below 1 packet.

If an endpoint receives the Ack within a TTAW time period, this indicates that the data link may support more throughput. The endpoint thus increments WINDOW by one.

7.2 Flow Control

Flow Control works alongside Congestion Control to limit the rate of the sender. Flow Control is established by the Receiver and depends on the available buffer size. The two parameters that are prerequisite for this are Sender Buffer Size - SBS, and Receiver Buffer Size - RBS. Since each endpoint maintains a local and a remote copy of both, they will be denoted as L_SBS, L_RBS and R_SBS, R_RBS respectively.

The buffer sizes determine how many packets each endpoint may hold. They are exchanged during the handshake as `MAX_FRAME_SIZE` is known to each endpoint before a connection is established. The client's `RBS` and `SBS` are sent with the Request packet, and the server's `RBS` and `SBS` are part of the Response packet.

8 Security

The security

Name	Units	Description
MAX_FRAME_SIZE	bytes	Maximum fragment size for this connection
FRAGMENT_SIZE	bytes	Fragment size for connection
SEND_BUFFER_SIZE	bytes	Sending buffer size
RECEIVE_BUFFER_SIZE	bytes	Receiving buffer size
RELIABILITY_COEF	float $\in [0,1]$	Controls the discarding of frames after an initial tra
CONNECTING_RESPONSE_TIMEOUT	sec	Timeout for a Response packet
CONNECT_ATTEMPTS_THRESHOLD	int	Number of attempts to connect to a server
FRAMES_WINDOW	int	Maximum number of frames to maintain simultaneously by
URGENCY_THRESHOLD	int	Number of packet to receive for a newer frame before sending a Vec.
FRAME_ID_LEN	bits	Length of the frame number
TTAW	sec	Time to ack window - max seconds to wait before sending

Table 3: Connection Parameters

Parameter	Units	Description
INIT_SEQ	int	Initial sequence number for outbound (data) packets(integer)
INIT_FRAME	int	Initial frame number for outbound frames(integer)
FRAGMENT_SIZE	bytes	Fragment size for local bound packets (in bytes)
MAX_FRAME_SIZE	bytes	Maximum frame size for local bound packets (in bytes)
BUFFER_SIZE	bytes	Buffer size of the local buffer (in frames)

Table 4: Request Packet Parameters