

FB-FR-CCP

Frame Based, Functionally Reliable, Congestion Control Protocol

Revision 1.7

Ivan Beschastnikh

August 20, 2005

Contents

| | | |
|-------------------------------|----------------------------------|-----------|
| | 4.10 Sync Packet | 9 |
| | 4.11 Checksums | 9 |
| 1 Protocol Overview | 1 | |
| 2 Connection States | 3 | |
| 3 Reliability Function | 3 | |
| 4 Packet Formats | 4 | |
| 4.1 Generic Header | 5 | |
| 4.2 Request Packet | 5 | |
| 4.3 Response Packet | 6 | |
| 4.4 Data Packet | 6 | |
| 4.5 CC_Ack | 7 | |
| 4.6 Vec_Ack | 8 | |
| 4.7 Sync_Ack | 9 | |
| 4.8 CloseReq Packet | 9 | |
| 4.9 Reset Packet | 9 | |
| | 5 Connection | 10 |
| | 5.1 Set Up | 10 |
| | 5.2 Handshake | 10 |
| | 5.3 Data Transfer | 11 |
| | 5.3.1 Receiver | 11 |
| | 5.3.2 Sender | 13 |
| | 5.4 Termination | 14 |
| | 6 Sync | 15 |
| | 7 Rate Controls | 15 |
| | 7.1 Congestion Control | 15 |
| | 7.2 Flow Control | 15 |
| | 8 Security | 16 |

Introduction

FB-FR-CCP (Frame Based, Functionally Reliable, Congestion Control Protocol) is a connection based, bidirectional transport level protocol that is meant to work on top of IP (Internet Protocol). The protocol is TCP friendly, and is functionally reliable. This document describes the specifications of the protocol. It does not describe an API, nor does give justification as to how or why this protocol should be used or implemented. For the description of the API please refer to [1] and for a nontechnical discussion of implications for this protocol, please refer to [2].

1 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. From the protocol perspective of data exchange, it doesn't matter which endpoint was the initiator or the acceptor of the connection.

Data from the application is not streamed, it is handed down to the protocol in discrete units which are termed *frames*. The protocol breaks down frames into *fragments*, each of which is payloaded onto a single packet for transmission. A frame is the atomic unit of reliability in FB-FR-CCP. At the receiver side, fragments are glued back together into frames. Once a frame is received in full, it may be collected by an application.

There are size limitations for frames and fragments. Maximum frame sizes are exchanged during the protocol handshake and the size of a frame is sent during the initial transmission of the frame. Once the frame size is acknowledged, it is not sent again. Fragment size remains constant for a particular frame, although it may vary between frames. Unless set differently by the application, the fragment size is set to the *MTU* (Maximum Transmission Unit) for the connection path. This document does not describe how the MTU or the more elusive *PMTU* (Path MTU) are to be determined.

Upon detecting that the sender has transferred the whole frame, the receiver evaluates a boolean *reliability function* that takes the state of the frame, its fragments and other information as its arguments. The function outputs a one if the frame should be kept, and its missing fragments are to be retransmitted. A zero on the other hand implies that the transmission of this particular frame should be halted and the frame should be dropped.

This document specifies only one function of reliability although others might make more sense depending on the use of the protocol. Some alternatives which can extend the usefulness of the protocol are described in [3].

If the function evaluates to a one, the receiver sends a negative acknowledgement vector (nak vector) to the sender. This vector will let the sender know which fragments are missing from the frame on the receiver side. The sender will then resend only these packets in the same manner as before, receiving acks from the receiver for congestion control. This process of resending missing fragments specified by the nak vector and then evaluating the reliability function 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. At any given moment, there can be a number of incomplete frames. To determine the sending order, the sender prioritizes the frames based on their age, the older the frame, the higher its priority. At the moment there is no other qualification of priority.

Termination can be initialized by any endpoint at any time, there are two forms. A connection may be reset immediately or it can be delayed until the reset is acknowledged by the other endpoint in order to make it easier to clean up state at the other endpoint.

The rate control for the protocol described in this document is window based. The receiver sends back acknowledgements (acks) within a *ttaw* (time to ack window) which has units of time. These packets carry information relevant to congestion control, such as the number of packets missed since the previous ack. These acks are not used to determine specific lost fragments, they determine the sending rate and help establish flow control at the receiver. The congestion control is tcp friendly although more experimentation and theoretical work is needed to verify this.

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.

2 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 during that particular state. 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

3 Reliability Function

The reliability function is always defined with respect to a frame. The one used in this document is a kind of ratio that depends on the number of fragments missing since the last nak vector (see below).

Before each successive attempt to resend the frame, the receiver must compute the boolean reliability function taking the state history of the frame as arguments to determine whether to send back a nak vector specifying which fragments are to be resent.

For the purpose of this document, the reliability function is defined as follows:

Let F be a frame composed of fragments f_1, \dots, f_N , each of size S . Also, let C be the *coefficient of reliability*, R be the *retransmission count threshold*, and M be the *missing fragments threshold*.

Define the space of negative acknowledgment vectors as the row space of a $0, 1$ matrix V of dimensions (TxN) . Row t of this matrix specifies an acknowledgement vector a_t sent at timestep t where $a_{tj} = 1$ if the negative acknowledgment vector includes f_j , otherwise $a_{tj} = 0$. Note that we have to evaluate our reliability function at timestep 0 (before we sent the first acknowledgement vector). For this reason, we define a_0 as a row of ones. This is intuitive as the sender didn't have any fragments for the frame and would have requested all of them if it could. Thus the function is only valid for $t \geq 1$

We define our reliability function ϕ as a function of timesteps:

$$\begin{aligned} \text{Let } a_t &= (a_{t0}, \dots, a_{tN}) \\ \text{Let } \tilde{t} &\text{ be the smallest } t \text{ such that } \sum_{j=0}^N a_{tj} < M \end{aligned}$$

$$\phi(t) = \begin{cases} t < \tilde{t} & \begin{cases} 0 & \text{if } \frac{\sum_{j=0}^N a_{tj}}{\sum_{j=0}^N a_{(t-1)j}} \leq C \\ 1 & \text{otherwise} \end{cases} \\ t \geq \tilde{t} & \begin{cases} 1 & \text{if } (t - \tilde{t}) < R \\ 0 & \text{otherwise} \end{cases} \end{cases}$$

To summarize the above, the coefficient of reliability C is in the range $[0, 1]$. If the computed ratio of frags received since last nak vector to the number of frags specified in the last nak vector is less than the coefficient, the function evaluates to zero, otherwise the function evaluates to a one. The formula is independent of the fragment size S as it would cancel if we were to weigh the fragments by their sizes (if introduced into both the numerator and the denominator of the ratio). When the number of missing frags decreases however, the evaluation changes to make up for the discretization. Instead, below a threshold M , the function uses a hard retransmissions limit R to determine when to give up on the frame.

In this way, the coefficient penalizes those frames that lose most fragments regardless of the time it takes to transmit each or all of them. A coefficient value of 0 indicates a completely reliable connection and a value of 1 indicates that we never want to resend a fragment to recover from fragment losses. The missing frags threshold tries alleviate the unstabling effect of discretization on the ratio when the number of missing fragments is too small.

The constants C , M , and R may be unique to a frame or a connection. Their values should ultimately be determined by the application.

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 4.4 lists all packet types with the state ids in which Receiving and Sending them makes sense.

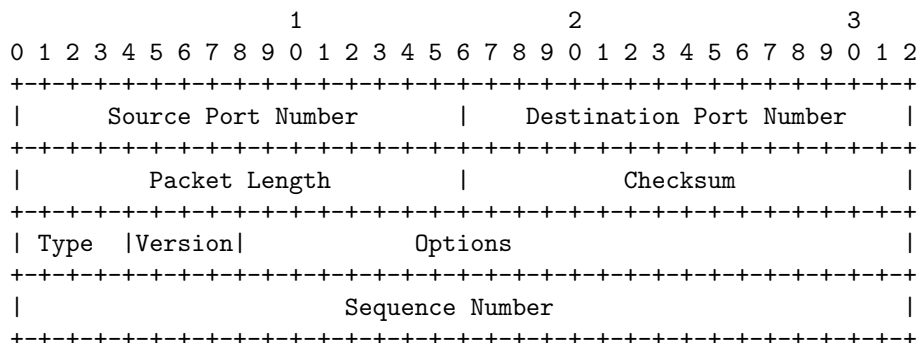
| ID | Type | Receiving States | Sending States | Description |
|----|----------|------------------|----------------|--|
| 0 | Request | 2 | 1 | Sent by C ; initializes a connection with S ; 1/3 Handshake |
| 1 | Response | 3 | 4 | Sent by S ; ack of the Request packet by C ; 2/3 Handshake |
| 2 | Data | 4,5 | 5 | Transmits application data |
| 3 | CCAck | 5 | 5 | Acknowledgement regulating congestion control |
| 4 | VecAck | 5 | 5 | Negative acknowledgement vector specifying missing fragments |
| 5 | DataAck | 5 | 5 | A piggybacked ack packet on top of a data packet |
| 6 | CloseReq | 5 | 5 | A request to disconnect, elicits a Reset Packet |
| 7 | Reset | 3,4,5 | 3,4,5 | A reset of the connection |
| 8 | Sync | 5 | 5 | Resynchronizes a connection after bursts or loss |
| 9 | SyncAck | 5 | 5 | An acknowledgement of the Sync packet |

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 Packet Header

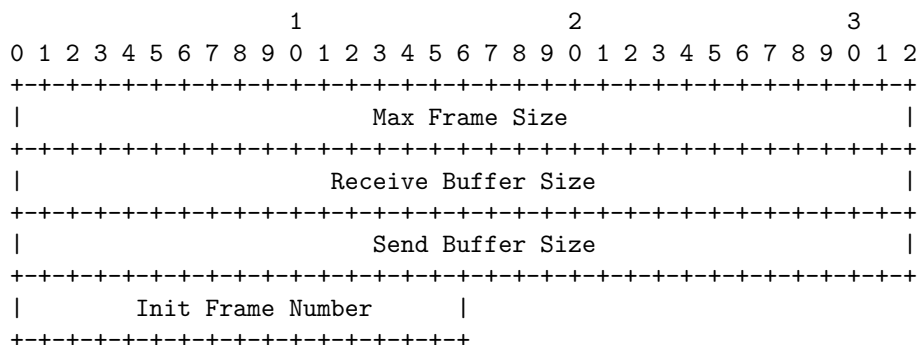


Every packet in FR-FCPP has a generic header to which the formats outlined in the sections below are postfixed. Some of these packets do not carry any data besides the generic header.

With each packet we identify source and destination ports as UDP and TCP protocols. We also send the packet length and a checksum of the entire packet. These initial fields reflect the UDP header exactly. This is done so that FB-FR-CCP can be more easily implemented on top of UDP. These fields will not be mentioned elsewhere in this document and can assume to have UDP behavior.

The generic header also includes a packet type field (see Table 4.4), the protocol version, a space for options and a sequence number. Interpretation of the options and the sequence number fields depend on the type of the packet.

4.2 Request Packet



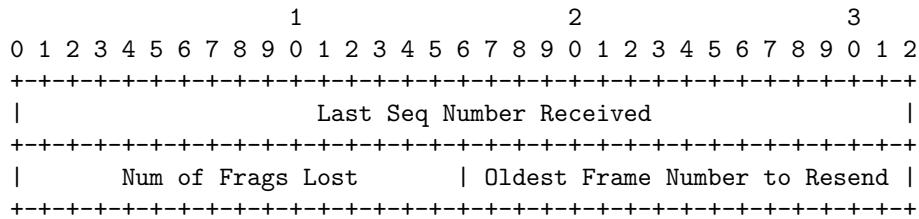
A request packet is sent by the client to initiate a connection with a server. This packet parametrizes the sending endpoint for the connection to follow.

The frame length field is an optional field appearing only when the `OPT_FRAME_LENGTH` option is set in the generic packet header. This field is present in the initial data packets, but once a data packet carrying the frame length has been acknowledged, the frame length field is not sent.

Acknowledgement Packets There are three distinct forms of acknowledgement. The first is congestion control acknowledgements, which we will term `CC_Acks`. The second type of acknowledgement is a negative vector acknowledgement (nak vector ack), which we will term `Vec_Ack`. The last is a synchronization acknowledgement which we term `Sync_Ack`. The three forms play different roles in the protocol. `CC_Acks` are acknowledgements sent regularly for congestion control. `Vec_Acks` are sent after an incomplete frame is transferred, `Sync_Acks` are responses to `Sync` packets that synchronize the endpoints' congestion controls. `DataAcks` are not a distinct form of acknowledgment since they are `Data` packets with prefixed (piggybacked) `CC_Ack` packet data.

| 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 receive needs resent |
| <code>Sync_Ack</code> | Prompted by a <code>Sync</code> packet, used for estimation of <code>RTT</code> by the receiver |

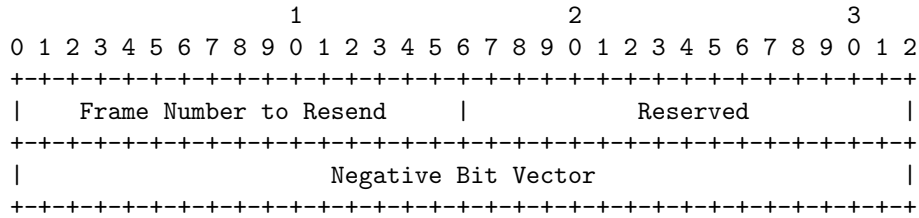
4.5 `CC_Ack`



`CC_Acks` (congestion control acknowledgement) packets must be sent once per `TTAW`. Their role is to signal the other endpoint of possible path congestion. 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.

4.6 Vec_Ack



The Vec_Ack (negative vector acknowledgement) packets 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 to continue receiving the frame. Specifically the receiver needs the missing fragments specified by the negative bit vector field.

This packet may effect the sending priority of the endpoint receiving this packet. If the frame being currently sent is older or the same as the frame number field, the sender updates the state of the frame to reflect the fragments that must be resent and resends them. If the current frame is newer, the sender must stop sending it and begin resending the frame specified with the frame number field. The sender resends only those fragments that are specified by the negative bit Vector field in the Vec_Ack.

4.7 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. The Sync_Ack does not carry a payload. It's size is that of the generic packet header.

4.8 CloseReq Packet

The Close_Req packet is the “nice” way to close a connection as the sender will wait (but eventually timeout) on an acknowledgement packet from the receiver. The Close_Req does not carry a payload. It's size is that of the generic packet header.

4.9 Reset Packet

The Reset packet is a second and a last resort method to close a connection. The sender thereafter drops all future incoming packets. The Rese packet does not carry a payload. It's size is that of the generic packet header.

4.10 Sync Packet

The Sync packet is meant to stabilize the congestion control. It is used in conjunction with the Sync_Ack acknowledgement sent as a reply to the Sync packet and used to estimate the RTT between the endpoints. The Sync packet does not carry a payload. It's size is that of the generic packet header.

4.11 Packet 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.

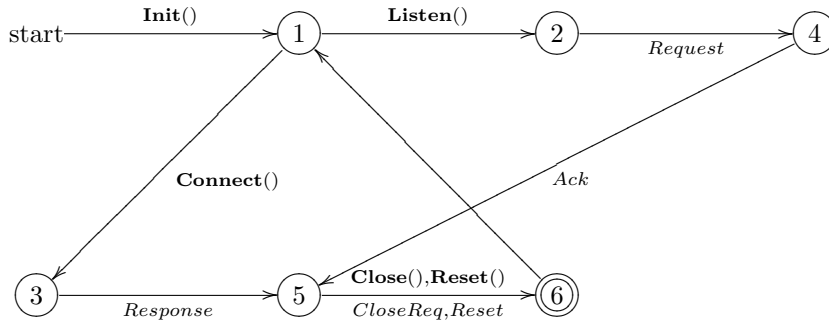


Figure 1: DFA representation of the protocol connection states

Another feature is checksumming. UDP checksums cover the whole datagram plus some IP fields, obviating the need for checksums in a FB-FR-CCP prototype built on top of UDP. Thus the initial prototype uses UDP to check and attach checksum to all incoming and outgoing 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 4.4. 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). The diagram also leaves out the details of the source address and port numbers. These are cached during the handshake and validated for each incoming packet.

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

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

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.

Each frame sent by a sender is associated with an identifier that is initialized and exchanged during the handshake. The initial value for this frame number is generated independently by each endpoint and exchanged during the handshake. The recommended procedure to generate the initial frame number is to 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.

The bidirectional nature of the connection following the handshake 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, to disambiguate, the local `FRAGMENT_SIZE` for a frame as seen by an endhost will be denoted `L_FRAGMENT_SIZE`, likewise the remote attribute becomes `R_FRAGMENT_SIZE`.

5.2 Handshake

Client The client initiates the connection by sending a Request packet. 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 stops retrying and signals an error to the application.

Upon receiving a Request packet, a server may take an indefinite time to respond. The Request and Response pair of packets should not be used to compute the initial `RTTn` estimate at the client endpoint.

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.

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 (relative to a frame), 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 evaluate the reliability function and either keep the frame around or discard it depending on whether the function evaluated to a 0 or a 1 respectively.

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, if the frame is again incomplete after all the fragments have been retransmitted, we repeat the process as necessary either until the reliability function evaluates to a 0 or we received the full frame. Once the frame is complete, we allow the application to pick it up and also let the sender to send us new frames.

5.3.1 Receiver

Designate the oldest frame the receiver is interested in (missing packet for) be `Oldest_Frame`, and the newest frame the receiver has seen so far in the `FRAME_WINDOW` be `Newest_Frame`. Also let the sequence number of the last data packet received be `Last_Seq_Number`.

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 carrying packet increments the sequence number independently of the frame number- this lets us determine how many data carrying packets were sent but lost or delayed between the reception of any two data carrying packets. Note that it is assumed that the packet has passed the checksum validation and invalid attributes are not due to data corruption.

Let D be a data packet with attributes Seq_Number , $Frame_Number$, $Frame_Len$, $Frag_Number$, $Data_Len$. Upon receiving D the receiver validates the attributes with a set of rules, and ignores D if any of them are true.

- Packet does not carry a $Frame_Len$ (controlled with the OPT_FRAME_LENGTH option) but frame $Frame_Number$ doesn't yet have a defined $Frame_Len$.
- $Seq_Number \leq Last_Seq_Number$
- $Frame_Len \neq$ stored frame length for the frame $Frame_Number$ (only if we have a record for this frame already)
- $(Frame_Len > MAX_FRAME_SIZE) \vee (Frame_Len < FRAGMENT_SIZE)$
- $(Frag_Number + 1) \times FRAGMENT_SIZE > Frame_Len$ (note that $Frag_Number$ starts at 0)
- $Data_Len > FRAGMENT_SIZE \vee Data_Len = 0$
- $Frag_Number > \lceil Frame_Len / Frag_Size \rceil$
- $\phi(FrameHistory) = 0$ where ϕ is defined previously as the reliability function.

A New Frame Number (ie. $Oldest_Frame \leq Frame_Number \leq Newest_Frame$)

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

Note that the *min* in the definition of M , in a sense limits our scope of frame vision to the `FRAMES_WINDOW`.

$M = 0$: 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`.

$M > 1$: 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 `Frag_Number * L_FRAGMENT_SIZE`.

Urgency

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

- `Urgency[frame_number]` is incremented by one. Initially `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 `Urgency[frame_number] \geq 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 ($\text{Oldest_Frame} \leq \text{Frame_Number} \leq \text{Newest_Frame}$)

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

- `Urgency[frame_number]` increments by one and if `Urgency[frame_number] \geq 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.

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

- `Urgency[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.

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 `UnAked_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 `UnAked_Queue`.

The `UnAked_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 queuing 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.

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 WINDOW cannot be smaller than one 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.

The `WINDOW` size is actually a minimum of `SBS_R` and current window of the congestion control (ie. $\min(CC, FC)$).

8 Security

The security

References

- [1] "FB-FR-CCP API", To be written
- [2] "Nontechnical Discussion of FB-FR-CCP" To be written
- [3] "Reliability Function Alternatives to FB-FR-CCP" To be written

| 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 b |
| 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