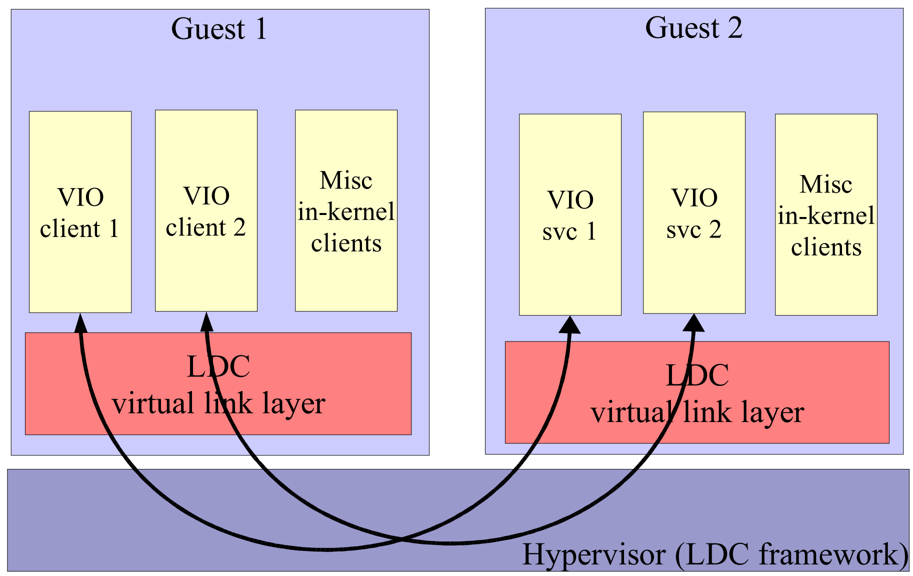
4.3 LDC virtual link layer

Logical domain channels provide a virtual link layer abstraction that are designed as point-to-point communication channels between logical domains or between a logical domain and an external entity such as a service processor or the Hypervisor itself. Logical domain channels provide an encapsulation protocol onto which higher level transport can be built such as TCP/IP and PPP.



Within a LDom a LDC is instantiated as a single endpoint (unless the LDC loops back to the same LDom). The identity of the owner of the other endpoint is opaque to the LDom - this enables LDCs to be re-connected to other endpoints at will. Conventional attestation protocols may be layered on top of the basic LDC mechanism if the identity of the owner of the other end of a LDC is required. Such attestation is beyond the scope for this document.

Logical Domain Channels (LDC) provide two ways of transferring data between endpoints. A simple packet based transfer mechanism where data is sent in 64-byte packets. The second approach allows clients to export regions of their memory address space with clients at the other end of specified LDC connections. The importing clients can then access the remote memory region by either mapping it into its address space, use an Hypervisor API call to copy data to/from exported memory, or program an IOMMU (or MMU) to directly read/write the memory.

4.3.1 Communication overview

Data transferred between domains can be encapsulated into LDC packets or transferred directly from one domain's memory to another using the Hypervisor shared memory communication support. The link layer protocol defined here provides clients the ability to choose either mechanism for data transfer. The link layer will fragment and reassemble messages as part of the transfer. It will insert additional header information as part of each packet to indicate the start and end of a fragmented data transfer. The LDC link layer uses network byte ordering to transfer all data. The actual details of the transfer protocol itself will be invisible to the clients.

- **Packet Based Transfer**

  Data can be transferred out of a virtual machine by encapsulating it into LDC packets or transferring it directly from one domain's memory to another using the Hypervisor shared memory communication support. The link layer protocol will provide client drivers the ability to choose either mechanism for data transfer.

  In the case of the packet based mechanism, the link layer protocol will fragment and reassemble messages as part of the transfer. It will insert additional header information as part of each packet to indicate the start and end of a fragmented data transfer. The actual details of the transfer itself will be invisible to the client driver. It is recommended that this approach be used only for short messages.

- **Shared Memory Access**

  The shared memory access mechanism allows a client driver to make sections of its memory visible to other domains. This support is build on top of the underlying Hypervisor infrastructure for setting up memory map tables to share memory segments. Client drivers will use the interface to obtain a cookie associated with the memory they want to expose. The client can then send the cookie to a client driver in a remote domain using the packet based transfer. The receiving client can then request its LDC framework to consume the cookie and map the remote domain's memory into its address space. Once the mapping is completed, clients can read, write these shared memory regions and also setup DMA operations to directly transfer data into or out of domain buffers.

  A slight modification to the direct memory map is the copy option, where the data is copied in to or out of the buffers that have been exposed by a virtual device client or server via a Hypervisor API. In this approach, when a virtual device wants to send data, either the device client or server will first copy the data from the exporter's memory to a local memory buffer.

  Both methods of data transfer is provided because all virtual machine client may not allow shared memory communication either due to technology limitations or security concerns.

- **Protocol modes**

  Clients of the LDC mechanism can either be clients that implement sophisticated transport layer like capabilities i.e. virtual ethernet with a TCP/IP stack, or a simple client with no special transport capability like the FMA daemon or a virtual console device. These clients have different reliability requirements on the underlying virtual link layer protocol. The virtual link layer protocol will meet the requirements of either type of client by implementing three different types of data transfer protocol.

  - *Raw mode*

    The raw virtual link layer protocol protocol does not add any overhead by appending any headers and sends only 64-byte packets at a time. It has no support for session management, message fragmentation and re-assembly, or retransmissions. It provides a very thin layer over the Hypervisor interface and mostly passes through read and write requests to the Hypervisor.

  - *Unreliable mode*

The unreliable link layer protocol will implement a communication mechanism that will include support of connection establishment via a simple handshake protocol. It will also implement support for negotiating a session and detecting session termination. It will only implement support to detect either lost or out-of-order packets, and not reassemble out of order packets and only stitch together packets received in order. The unreliable mode also supports fragmentation and reassembly of LDC datagrams. Clients of this link layer mechanism will need to implement their own error detection mechanism and do the required retransmission.

- *Reliable mode*

  The reliable link layer protocol implements all the support encompassed within the unreliable link layer protocol. In addition, it implements support for streaming buffers, detecting out-of-order packets and packet loss and acknowledges received packets. The primary distinction of reliable mode is to provide an error detection capability via packet ACKs and NACKs.

4.3.2 Packet formats

The Hypervisor LDC framework provides the capability to deliver 64-byte packets between peer channel endpoints. It does not impose any predefined format for each word in the 64-byte packet. Depending on whether the clients want to use a raw, reliable or unreliable link mode, the link will utilize different formats for each LDC packet. In the case of the reliable link each packet will consist of a 16-byte header, and 48-bytes of data payload. The unreliable link will have a smaller 8-byte header, and contains 56-bytes of data payload. The raw link will utilize the complete 64-bytes for the data payload. The high-level format of the raw, unreliable and reliable packet is shown below:

Raw Datagram Packet:

```
             6
             3                                                 0
            +---------------------------------------------------+
 word 0-7:  |                   data payload                    |
            +---------------------------------------------------+
```

Unreliable Datagram Packet:

```
             6                    3 3   2 2    1 1
             3                    2 1   4 3    6 5     8 7    0
            +----------------------+-----+------+-------+------+
 word 0:    |         seqid        | env | ctrl | stype | type |
            +----------------------+-----+------+-------+------+
 word 1-7:  |                   data payload                    |
            +---------------------------------------------------+
```

Reliable Datagram Packet:

```
             6                    3 3   2 2    1 1
             3                    2 1   4 3    6 5     8 7    0
            +----------------------+-----+------+-------+------+
 word 0:    |         seqid        | env | ctrl | stype | type |
            +----------------------+-----+------+-------+------+
 word 1:    |          ackid          |           (reserved)    |
            +---------------------------------------------------+
 word 2-7:  |                   data payload                    |
            +---------------------------------------------------+
```

Description:

- Packet Type (Word 0, Bits 0-7): Each packet sent from one LDC endpoint to another can consist of either control, data or error information or a combination there-of. The appropriate '*type*' field bit(s) are set to indicate packet contents.

LDC_CTRL	0x01
LDC_DATA	0x02
LDC_ERR	0x10

- Packet Sub-Type (Word 0, Bits 8-15): The *stype* field contains values INFO, ACK or NACK and defines the type of data, control or error message. The combination of the *type* and *stype* fields define the nature of the message.

LDC_INFO	0x01
LDC_ACK	0x02
LDC_NACK	0x04

- Control Info (Word 0, Bits 16-23): The *ctrl* field contains either basic control information and/or error information. The control info values currently supported are listed below:

  Basic Control Values :

LDC_VERS	0x01	Link Version
LDC_RTS	0x02	Request to Send
LDC_RTR	0x03	Ready To Receive
LDC_RDX	0x04	Ready for data exchange

- Packet Envelope (Word 0, Bits 24-31): The *env* field, depending on the packet type, contains either control or data related information. If the packet contains a control info of type RTS or RTR, the envelope contains protocol mode and will have one of the following values:

LDC_MODE_RAW	0x0	Raw Mode
LDC_MODE_UNRELIABLE	0x1	Unreliable Mode
(RESERVED)	0x2	
LDC_MODE_RELIABLE	0x32	Reliable Mode

  When using RAW mode, since there is no handshake as part of the protocol, the RAW mode value specified above is never exchanged as part of the packet envelope. It is only specified here for completeness.

  In the case of packets containing data, the envelope contains the number of bytes in the current packet. It also contains information pertaining to fragmented transfers. The format of the envelope for a data packet is shown below:

```
      3       3     2                           2
      1       0     9                           4
   +--------------+---------------------------+
   | stop | start |          pkt_size          |
   +--------------+---------------------------+
```

When a message is fragmented, the first fragment has the *start* bit in the envelope field, set to 1. The last fragment has the *stop* bit set to 1. Intermediate fragments between a start and stop packet have neither bit set. In the case of a single packet transfer (less than the max payload), both start and stop bits in the envelope are set to 1.~~*Session ID (Word 0, Bits 32-63): Each channel endpoint generates a unique session identification (sid) that the other side has to use in all future data exchange. At the start of initial handshake process each side sends its session ID to the other side. Following a successful handshake each side maintains its session ID and the session ID it received from its peer. During data transfer, a sender will send with the packet the receiver's 'sid', and the receiver will validate this against the session ID it is maintaining. When the session is torn down, and a new session is established, new SIDs are exchanged by both ends as part of the negotiation. Transmitting incorrect session IDs during regular data transfer results in the channel getting reset.*~~

- ~~*Implementation Note: In order to generate a unique session ID, it is recommended that the link uses 32-bits from the CPU tick register as the session ID.*~~
-
- Sequence ID (Word 0, Bits 32-63): The seqID field is populated with an unique sequential number for every packet sent from one endpoint to another. This is used by the receiver to detect and enforce packet ordering, and acknowledging received packets.

The AckID field~~s~~ below ~~= SequenceID, and AckID are~~is only used for the reliable link implementation*Implementation Note: In order to generate a unique session ID, it is recommended that the link uses 32-bits from the CPU tick register as the session ID*.

- ~~Sequence ID (Word 1, Bits 0-31): The *seqID* field is populated with a unique sequential number for every packet sent from one endpoint to another. This is used by the receiver to detect and enforce packet ordering, and acknowledging received packets..~~
- Acknowledgment ID (Word 1, Bits 31-63): An endpoint can acknowledge packets it has received by sending an ACK back to its peer. The 'ackid' field contains the sequence ID of the last packet received in correct order by an endpoint. The peer may send~~ a ~~ separates messages to ACK received packets or embed~~ded~~ acknowledgments in data packets.

4.3.3 Communication protocol

The link layer implements a thin connection establishment, tear down and data transfer protocol on top of the Hypervisor infrastructure. When clients opens a channel for communication, the link allocates memory for transmit and receive queues and registers these with the Hypervisor. Since neither endpoints have any knowledge about a endpoint's capabilities and whether it is ready to receive data , a simple handshaking protocol is needed to prior to starting the data transfer. This also ensures that clients can start and terminate their sessions independent of each other, and reestablish a connection when necessary.

Implementation Note: In the case of a reliable connection, the link should buffer outgoing messages for retransmission purposes. It will mark packets in the transmit queue as completed when it receives ACKs. In the event of a packet loss / timeout, this allow the link to retransmit pkts.

- **Session establishment**
  - After setting up the Tx and Rx queues, either endpoint will initiate a version negotiation by sending a LDC_VERS message, with the version number it supports in the second word of the message. The link will use a simple count down algorithm so that both sides use to agree on a mutual version. If the peer endpoint agrees with the same version or the same major but a lower minor version, it will respond back with an ACK (same msg with the ACK bit set). If it does not support the version, it will respond with an error message NACK and also set the version field to the next lower version version it supports. If it does not support a lower version, it will set the version fields to zero. The sender can then re-send another VERS request with the received lower version or a new even lower version. This will continue on until either the endpoint initiating the VERSION handshake exhausts all the version it supports or the peer accepts a version or responds with an NACK message with version set to zero.

```
6                       3 3   2 2    1 1
3                       2 1   4 3    6 5    8 7    0
+-----------------------+-----+------+------+------+
|                       |  -  | VERS | INFO | CTRL |
+-----------------------+-----+------+------+------+
|                       |   major    |    minor    |
+-----------------------+------------+-------------+
```

  - Following the version negotiation, either endpoints will negotiate a 3-way handshake. As part of this handshake, the endpoints will exchange initial sequence IDs ~~and session IDs~~ for the session.

```
6                        3 3    2 2     1 1
3                        2 1    4 3     6 5     8 7     0
+------------------------+------+-------+-------+-------+
|          SIDseqID               |  -   | RTS/  | INFO  |
CTRL |
|                        |      | RTR   |       |       |
+------------------------+------+-------+-------+-------+
|                        |         init_seqID           |
+                        +                              +

6                        3 3    2 2     1 1
3                        2 1    4 3     6 5     8 7     0
+------------------------+------+-------+-------+-------+
|      peer_SID          peer_init_seqID + 1      |  -   |
RDX  | INFO | CTRL |
+------------------------+------+-------+-------+-------+
|                        |    peer_init_seqID + 1       |
+                        +                              +
```

- The sending link endpoint aka endpoint_A will initiate an handshake with the other side i.e. endpoint_B by sending an LDC_RTS message that contains ~~a unique Session ID,~~ the initial seqID (if reliable), and the mode it would like to use for communication.
- If endpoint_B has setup a receive queue, it will either:
  - respond back with a LDC_RTR message, that contains its initial seqID ~~(if reliable) and session ID~~ and the matching link mode message.
  - endpoint_A will then respond back with a LDC_RDX message. This will mark the channel status as UP and data transfer can now commence.
- If endpoint_B has not setup a receive queue, the hypervisor send (*hv_tx_set_qtail*) operation will fail.

```
                    Endpoint A                      Endpoint B
                        |                               |
                        |                               |
                        |     CTRL/INFO/VERS(ver_3.x)   |
                        + ----------------------------> +
                        |     CTRL/NACK/VERS(ver_2.x)   |
         version        + <---------------------------- +
       negotiation      |     CTRL/INFO/VERS(ver_1.x)   |
                        + ----------------------------> +
                        |     CTRL/ACK/VERS(ver_1.x)    |
                        + <---------------------------- +
                        .                               .
                        .          CTRL/INFO/RTS        .
                        |      (sid_A, seqid_A, mode)   |
                        + ----------------------------> +
                        .                               .
                        |          CTRL/INFO/RTR        |
        handshake       |      (sid_B, seqid_B, mode)   |
                        + <---------------------------- +
                        |                               |
                        .          CTRL/INFO/RDX        .
                        |     (sid_A, seqid_A+1, mode)  |
                        + ----------------------------> +
                        |                               |
                    data xmit                      data xmit
                        |                               |
```

Following a successful handshake, both sides can start transmitting data. ~~All future exchanges should contain the negotiated peer's session ID in the 'SID' field of the packet.~~

- **Session termination**

  A session between two endpoints can be torn down either due to a packet error, repeated packet loss, too many retransmissions or at the request of a client. A session is normally terminated by either un-configuring or reconfiguring the receive queue. On receiving a CHANNEL_DOWN or CHANNEL_RESET notification from the Hypervisor the receiver will reset its internal state from which a version negotiation and handshake will need to occur prior to fresh data transmission.

- **Session status notification**

  A session is established when either endpoints initiate a handshake or is terminated following an Rx queue un-configuration or reconfiguration. Following either events, the link can notify its client about a change in session state via the callback registered by the client.

- **Data transfer**

Packet format:

When sending data to its peer, depending on the size, a link will either send the data in one packet or fragment the data into multiple packets. The *type* field in the msg pkt will be set to DATA for all packet based transfers. The *stype* field will be of value INFO and the *envelope* field will contain the number of bytes being sent in each packet. The *start* and *stop* bits are used to indicate the start and end of a fragmented transfer. The first packet in the transfer will have the *start* bit set to 1. Subsequent packets have neither the *start* nor *stop* bit set. The last packet sent as part of a fragmented transfer will have have the *stop* bit set to 1. If the data is transmitted in a single packet, both the *start* and *stop* bit will be set to 1.

Streaming support:

The Reliable mode also implements support for streaming data transfers. It does this by breaking each message into MTU size blocks, specified by the client at the time of channel initialization. During send (ldc_write), each message is first broken up into MTU size blocks before being transmitted using the packet transfer approach discussed above. On the receiving end, the link layer passes data back to client in MTU size blocks without any reassembly. Using streaming eliminates the need to allocate very large Tx and Rx queues in the link layer as very large messages can be transferred in MTU size chunks.

Message ACKs:

Message ACKs are used in the case of reliable link mode to indicate data transfer progress. A client can only queue a fixed number of packets, after which it will have to wait for an ACK from the receiver before it can send more packets. The receiver will periodically respond back with a DATA/ACK control message, and the '*ackid*' field will contain the sequence ID of the last packet it received in correct order. Since the packet control field bits for an ACK message do not overlap with those of a regular data packet, a endpoint can send an ACK message embedded in a data packet.

Transmit queues and retransmissions:

In the case of a reliable link, the link will retransmit the packets in the event of a data loss. For each message sent by a client, the link will maintain it in a list of message segments. Each segment corresponds to one more fragments i.e. packets in the transmit queue. It will store the seqID corresponding to first fragment with the segment. It will initiate a send by storing the fragmented packets in the transmit queue. At the same time it will start a timer for the message. If a ACK for the packets are not received before the timer expires, the sender will retransmit the message with the same set of start of end seqIDs. If an duplicate ACK is received, it will discard it.

The sender will also maintain a head and tail pointer to keep track of the packets that have been transmitted and the ones that have been ACKed. In the event of a timeout, the sender will retransmit packets by copying over the packets into queue locations starting at tail location. All packets in the queue will purged when a session is torn down and/or established.

There are multiple retransmit scenarios and these are handled in the following manner:

- Packet loss

This is the simplest of all cases. In the event of packet loss, the receiver will discard all future packets until it receives a packet in correct sequence. The sender will initiate retransmission on timeout.

- Premature timeout / Delayed ACKs

  There are cases when the receiver is backed up and does not respond to the sender in a timely fashion. This will cause the sender to timeout prematurely and retransmit the segment's packets to the receiver. It might either during the retransmission or subsequently receive ACKs for the first transfer. When it receives the ACK, it can mark the message segment as successfully sent. It will then ignore any duplicate ACKs received as a result of the retransmission. Similarly, the receiver will discard packets associated with the retransmission (same seqID range), if it had previously received the message successfully. Even if the receiver discards incoming messages as duplicates, it will need to ACK the messages as earlier ACKs could have been lost.

- Lost ACKs

  In the event, the message was sent successfully, but the ACK was lost, the sender will eventually timeout and retransmit the segment packets. Since receiver already received the message, it will discard the message but still send an ACK. If there is an error during retransmission, the receiver will discard the packets as before.

Link errors:

Either during the initial handshake or during the course of data transmission, either endpoints can detect an error and take the corresponding action. The errors currently detected and handled within the link are listed below:

- Packet error

  During data transmission, packets can either get dropped or gets sent out of order. When the receiver detects a packet that is out of order, it will purge all pending packets in its transmit queue, until it finds a packet with the correct sequence. The unreliable link does not support retransmissions, and packets are dropped on error. Transmit sequence errors are detected via invalid start/stop bits in pkts.

  In the case of reliable link mode, packet loss is detected using seqID. It will send an ACK for the last packet that was received in correct order. This allows the sender to determine what seqID to start the retransmission from. Since there might be packets in flight (pkts between the ACKd pkt and the current TX tail ptr), the receiver will have to continue dropping all future packets until it receives a packet with the seqID that corresponds to the lost packet. The sender will eventually timeout and recopy lost or unacknowledged packets starting from the current tail location and initiate the retransmission of packets starting with the lost packet.

Link interrupt handler:

Links that are capable of handling interrupts can register an interrupt handler for each LDC channel with a target CPU to which the interrupt should be delivered. The link should allocate the CPU to channels in a round-robin manner. When a channel has pending data in its LDC queue, the Hypervisor will send a dev_mondo interrupt to the link. The link will either process the packet in the queue (if it is a control packet), or invoke the client's callback (if it is a data packet) to let it know that there is pending data.