OpenLCB Technical Note	
Stream Transport	
Jan 21, 2026	Preliminary

1 Introduction

This Technical Note contains informative discussion and background for the corresponding “OpenLCB Stream Transport Standard”.

Some wire protocols can support only very short packets/frames, e.g. CAN with a limited header and data payload. OpenLCB needs to be able to move large chunks of data that may not be bound by any size.

A pair of nodes may be able, but are not required, to maintain up to 255 simultaneously unique open/active streams between themselves, as a pair.

2 Annotations to the Standard

This section provides background information on corresponding sections of the Standard document. It is expected that two documents will be read together.

2.1 Introduction

2.2 Intended Use

A stream may be useful when:

- The amount of data to be transferred between two nodes is large.
- The amount of data to be transferred is unbounded.
- A persistent open channel is required.
- Flow control is required.

2.3 References and Context

For more information on format and presentation, see:

- OpenLCB Common Information Technical Note

For more information on MTI values and how they are laid out, see:

- OpenLCB Message Network Standard
- MTI allocation table.

2.4 Definitions

2.4.1 Stream Initiate Request

Stream. Other examples of streaming protocols would be TCP, or a unix pipe. The big difference to TCP is that the OpenLCB streams are uni-directional, whereas TCP connections provide bi-directional data transport. TCP also allows some out-of-band information to be sent during a live connection, although this is a rately used feature. OpenLCB does not allow out-of-band information; any out-of-band information needs to be sent using different protocols. It is expected that for most streaming applications the application level protocol will define a sequence of actions, where some may use direct OpenLCB messages, others may use datagrams, and the stream

The **Source node** and **Destination node** concepts are defined based on the direction of data flow. The source node is the node sending the data, irrespectively of which node requested the data transfer. Let's take the example of the Memory Configuration Protocol Streaming Read command. Here the client (say a PC-based configuration tool) the sends a datagram to the server (say an OpenLCB board, or a train node), requesting a read using streams, but the server will be the Source node and the client will be the Destination node. This also means that the server will send the Stream Initiate Request message to the client. Also the Source Stream ID will be assigned by the server, and the destination Stream ID will be assigned by the client.

Source Stream ID and **Destination Stream ID**, from a formal correctness perspective, are only used to disambiguate between multiple concurrently open streams between two given nodes. A stream is uniquely identified by (Source Node ID, Destination Node ID, Source Stream ID) triple, as well as the (Source Node ID, Destination Node ID, Destination Stream ID) triple. This means that Source Stream ID, by itself, is not required to be unique; neither Destination Stream ID, b itself.

There is a large flexibility on the implementation for how these IDs can be used. A small node may use only one fixed stream ID and not ever allow multiple streams to be open concurrently. A medium-sized node may have a fixed number of slots for open streams, and could use the Stream ID as a direct slot index. A large node could dynamically allocate the Stream objects, and keep an associative array of (opposite end Node ID, stream ID) → object reference. For ease of implementation, the opposite stream ID is always sent along with the streaming messages so that the receiving node can perform a direct lookup of the data associated with the stream and not need to do a search for the matching state/buffer.

2.5 Message Formats

Every stream message type contains one or both of a Source Stream ID and/or Destination Stream ID. An individual stream can be identified by the combination of:

1. Source Stream ID
2. Destination Stream ID
3. Source Node ID
4. Destination Node ID

It is important to note that neither a Source Stream ID nor a Destination Stream ID must be unique on its own, rather, they must be unique when used with a given node pairing.

Technically only one of the Source or Destination Stream ID's, not both, are required to uniquely identify a stream. This fact is exploited in order to maximize the efficiency in a Stream Data Send message by excluding the Source Stream ID and including only the Destination Stream ID.

The reason that both Source and Destination Stream ID's exist is to provide each of the node pairs involved with the connection an opportunity to allocate an identifier optimized to its own internal mapping of stream metadata to identifier.

There is one Stream ID (0xFF) that is reserved which must never be sent and must be ignored upon receipt. This Stream ID reservation exists in order to allow an implementation the opportunity to use this value for the purpose of marking a stream mapping as closed or available-for-use without requiring an additional memory location to hold this information.

2.5.1 Stream Initiate Request

This is an addressed message sent from the stream source node to the stream destination node. It contains a unique Source Stream ID, a suggested Max Buffer Size (in bytes), reserved flag bits, and optionally a Stream Content Type.

The suggested Max Buffer Size chosen by the stream source is somewhat arbitrary, but could be implementation specific. The resulting final Max Buffer Size of the stream may be smaller but must never be larger than this initially suggested value.

2.5.2 Stream Initiate Reply

This is an addressed message sent from the stream destination node to the stream source node in response to a Stream Initiate Request. It contains a unique Destination Stream ID, a unique Source Stream ID, a final Max Buffer Size (in bytes), and an error code field.

The final Max Buffer Size must be less than or equal to the suggested Max Buffer Size in the Stream Initiate Request being replied to.

2.5.3 Stream Data Send

This is an addressed message sent from the stream source node to the stream destination node. It contains the Destination Stream ID followed by the payload bytes.

The stream source node must maintain an internal count that is initialized to Max Buffer Size. For every single payload byte sent, the source must decrement this count by one. A single Stream Data Send may contain a payload in number of bytes up to this count value, or may be broken into smaller messages so long as in aggregate, the multiple Stream Data Send messages do not result in the internal count decrementing below zero.

A Stream Data Send message may also contain zero payload bytes. This is considered a flush, and should result in the destination node (and any gateways in between) flushing any internal buffering for efficiency that may be occurring.

2.5.4 Stream Data Proceed

This is an addressed message sent from the stream destination node to the stream source node. The destination node sends this message when it is able to receive an additional Max Buffer Size bytes of data beyond what the stream destination node has already informed the stream source node it can receive. When the stream source node receives this message, it can add Max Buffer Size to its internal count of bytes that it may send with Stream Data Send messages. It is possible, and acceptable, that through this mechanism, the count internal to the stream source node could increment to a value larger than Max Buffer Size.

2.5.5 Stream Data Complete

This is an addressed message sent from the stream source node to the stream destination node for the purpose of gracefully shutting down a stream. This message contains a Source Stream ID, Destination Stream ID, and optionally, a 4-byte value representing the total number of data bytes sent while the stream was open. Once this message is received, the stream source node may free up the previously allocated Source Stream ID, along with any metadata associated with the stream.

2.6 States

A stream either exists or doesn't; those are states. There are also some intermediate states during setup and take down.

2.7 Interactions

Optionally, the destination can request that the source start a transfer. This uses some mechanism not discussed here, e.g. Datagram messages.

The source sends an "Stream Initiate Request (Addressed)" to the destination. It carries a "Max Buffer Size" value (2 bytes), a “Type Included” boolean flag (1 bit in a byte; see below for meaning) and a "Source Stream ID" (1 byte) in the data section. The combination of source, destination, and Source Stream ID must uniquely identify this stream transmission.

If the destination node does not wish to receive the stream, it returns a "Stream Initiate Reply (Addressed)" marked “reject”, with a "Max Buffer Size" value (2 bytes) of zero, the “Type Included” boolean flag (1 bit in a flag byte), "Source Stream ID" (1 byte) and "Destination Stream ID" (1 byte). The message includes flags set to represent exactly one of several conditions:

“Permanent error” - This node does not process streams, will not accept a stream from this source, or for some other reason will not ever be able to accept this stream. The Stream Initiate Request should not be retransmitted. Optionally, the node can mark the reply with one or more of several conditions:

“Information Logged” - the node supports the logging protocol and information was logged for later retrieval.

“Invalid Stream Request” - something made the stream request improper, such as longer than the max permitted length. Proper requests might be acceptable.

“Source not permitted” - streams from this source will never be accepted

“Streams not accepted” - this node will not accept stream requests under any circumstance. A node can also reject the interaction instead of sending Stream Initiate Reply with this code.

“Buffer shortage, resend” - The node isn't able to receive the stream because of a shortage of buffers. The sending node should resend at its convenience.

“Stream rejected, out of order” - Should not happen, but an internal inconsistency was found in the CAN frames making up a stream. Sender can try again if desired.

If the destination node wishes to receive the stream, it returns a "Stream Initiate Reply (Addressed)" marked “accept”, with a "Max Buffer Size" value (2 bytes), the “Type Included” boolean flag (1 bit in a byte), "Source Stream ID" (1 byte) and "Destination Stream ID" (1 byte). The "Max Buffer Size" is less than or equal to the value in the Initiate Stream Request, and is the negotiated buffer size for this transfer. If it's zero, the request to start the stream has been rejected, and the exchange is over. The source can try again later.The Source Stream ID is the same as the value in the Initiate Stream Request, and is returned for identification and the convenience of the source. The destination doesn't do anything with it except return it. The source can use it to match up multiple operations, as a way of identifying buffers, or for any other purpose.The Destination Stream ID is used to tag the data sent to the destination. It has no meaning to the source. The destination can, but need not, use it to associate the stream data with a particular buffer or usage. Multiple simultaneous streams can use the same Destination Stream ID value.

The source starts sending bytes using Stream Data Send (Addressed) messages, each carrying up to the message size limit on the particular wire protocol and the Source and Destination Stream ID. After sending exactly Max Buffer Size bytes in one or more messages, the source pauses.

If the “Type Included” flag was true during initialization, the first 6 bytes of the data are a unique data-type indicator. These are allocated the same way that Node IDs, etc, are allocated, but from a separate name space. If that “Type Included” flag is false, some higher-level protocol must identify the data-type of the stream data. The idea is that a stream is a lot of data; there's not much use for one otherwise because of the setup overhead (code and time). So six bytes for a stream type identifier isn't a large cost (unlike e.g. a datagram, where it would be a 10% overhead). A UID as a Stream type ID has the advantage that it's not ever going to collide, so people developing protocols don't have to coordinate it (e.g. useful for future expansion, where a protocol can be locally developed and then deployed more widely). And it still fits in the 1st CAN Frame of the stream, which simplifies what nodes have to do to figure out what type of (unsolicited) data this is. On the other hand, streams have a stream ID, so that a protocol can use some other mechanism (messages, datagrams, or even other streams) to pass the information about what a specific new stream means. That means the type info at the start of the stream isn't as necessary as in e.g. datagrams. (We really wanted to avoid situations like "The next datagram you receive carries the data for this request", because "next" is a hard concept to ensure in code that's doing several things independently)

On CAN, the Stream Data Send message does not carry the Destination Stream ID field. Messages to CAN get split into frames and forwarded without the field. Messages from CAN have to have it inserted by the gateway as part of the translation process. The gateway can do this as part of the (optional) buffer accumulation from frames into larger transfers.

Upon receiving Max Buffer Size bytes via one or more messages, the destination sends a Stream Data Proceed (Addressed) message to the source, carrying the Source Stream ID and Destination Stream ID. This tells the source that there is enough buffer space available that another Max Buffer Size bytes can be sent. The destination can also set flag bits (error codes) in this message to indicate that no more data should be sent and the transmission should be terminated early.

The destination may, but is not required to, send a Stream Data Proceed (Addressed) message to the source before receiving the full count of Max Buffer Size bytes. In that case, the source does not stop transmission after sending Max Buffer Size bytes, but continues for an additional Max Buffer Size bytes of transmission.

When the last data has been sent, the source sends a Stream Data Complete message carrying the Source Stream ID and Destination Stream ID to indicate that all data has been sent. Optionally, this can carry a four-byte length of the stream data transferred for checking. A zero value, if present, means the length is unknown. When this message is received by the destination, the transfer is complete.

3 Background Information

3.1 Stream Content Type Identification

Stream senders can, but don't have to, identify the content type (format, meaning, etc) of the data stream at the front. As part of their private protocol, they can use whatever structure they'd like for the data.

General content types can be defined using unique identifiers. The “Type Included” flag identifies that the first eight bytes of stream data are to be interpreted as stream content type information. These can be allocated via the same mechanisms as EventIDs. Well-known ones will be published by OpenLCB. These identifiers are recorded in a separate worksheet.

3.2 Buffer Size

The buffer size can be up to 2^{16}-1 bytes = 64KB-1. This is driven by the size of the initiate messages, which we want to keep in a single CAN frame. Although some transport mechanisms might be able to profitably use larger buffers, it seems unlikely that a layout control bus will need to have a higher message efficiency or lower latency than this size will allow. Should that turn out to be necessary in the future, an alternate form for the stream initialization messages can be defined for those large-message nodes. (It's unlikely they're running over CAN)

3.3 CAN Discussion

On a CAN segment, the data limit is 8 bytes. This requires use of as little control information as possible, so that as many of those eight bytes can be used for data as possible.

OpenLCB/S9.6 CAN frames can hold the source and destination aliases in the header. A dedicated “stream data transfer” format can also be coded entirely in the header. This then allows 8 bytes of data per transfer if the Source Stream ID and Destination Stream ID need not be carried. By specifying that only one stream can be transferred on a CAN link between any two nodes at a time, no Stream ID would be needed. Although initially considered, this was thought to be too inflexible. Instead, we chose a format where the Destination Stream ID is carried, reducing the payload to seven bytes per frame. This requires that the Destination Stream ID be unique among source-destination node pairs, without relying on the Source Stream ID to be part of the unique stream identification. Since a node knows that it's originating the stream, this can be ensured in advance for point-to-point connections. Gateways will need to track the Source Stream ID to restore it to the full format. Since they are tracking the stream in any case for buffering purposes, this is thought to be not a large problem.

3.4 Implementation Notes

The "Stream Data Proceed" from the destination is clearance to send another buffer-size-worth of data. To achieve better performance, the destination can send it before receiving the entire buffer-size-worth of data, as soon as it has room to receive what's already been OK'd plus one more buffer size. For example, a destination with a 4kB buffer could reply with Max Buffer Size of 2K, followed by an immediate Stream Data Proceed, to do single overlap of the transfer.

Intermediate nodes need to be able to handle transfers, and therefore need permission to lower the Max Buffer Size on the outbound Stream Initiate Request message. The length in the returned Stream Initiate Reply can't be changed, as the destination need to know when to send clearance for another buffer's worth of data.

A CAN ↔ Ethernet bridge might receive several kilobytes of data from the Ethernet side at a time. It's then responsible for breaking that up into CAN frames and forwarding it. It can reduce the buffer size of transfers if need be to ensure there's a place to store the data while this is done.

Table of Contents