On OpenLCB CAN Timing and Buffering

Bob Jacobsen

May 14, 2024

1 Introduction

Some of the OpenLCB protocols depend on a request-reply sequence. For example, a SNIP request[1] received by a node will result in that node sending back multiple CAN frames of SNIP reply information to the initiating node.

To do that, the node has to have buffers that record the state of the in-progress interactions. If the node is processing multiple requests at the same time, multiple buffers will be needed.

In this note, we consider the specific effects of CAN transmission and inter-frame timing on the buffer requirements for OpenLCB CAN nodes. We discuss how current OpenLCB networks behave given existing implementations. We provide recommendations for changes to Standard and Technical Note language, and for the definition of the new Events With Payload protocol.

2 Expected Sequences

There are many ways to organize the use of memory in an OpenLCB implementation, but all of them require some space to store state when a request is received. When e.g. a SNIP request is received, at a minimum the receiving node needs a buffer in which to remember that a SNIP reply should be sent and who to send it to. Depending on the implementation, more information might be buffered.

2.1 SNIP Protocol

It would appear from Figure 1 and Figure 2 that only one SNIP receive/reply buffer is needed: Because the SNIP reply has higher CAN priority than the SNIP request, a second

[1]We're going to use common OpenLCB terminology without defining each term here. For unfamiliar terms, see the OpenLCB Glossary and the OpenLCB Standards and Technical Notes.

request during the first reply is delayed until after that reply is sent. But note there's a stretch of idle CAN bus before the reply starts to send. One or more additional requests can be received during that interval, depending on how long it is. See Figure 3. That delay between request and response has been observed to be from about 0.4 milliseconds to 20 milliseconds on various implementations.[2] Although it's perhaps unlikely that multiple requests will arrive during this short time, should that happen a node with only a small number of fixed buffers may be overwhelmed.

Some nodes have been observed to reply with an Optional Interaction Rejected (OIR) message when they run out of buffers. This is a Standard compliant[3] way of handling a shortage of buffers. The OIR message contains the SNIP MTI, so it provides all the information needed to retransmit the SNIP request.

Many implementations will not retry the interactions when the temporary Optional Interaction Rejected message is received. It's strongly recommended to design nodes so that they can take part in multiple overlapping SNIP interactions without running out of buffers in the first place.

SNIP request A→	SNIP request A	
	idle	
	SNIP reply A1	← SNIP reply A1
	SNIP reply A2	← SNIP reply A2
	SNIP reply A3	← SNIP reply A3
	SNIP reply A4	← SNIP reply A4
	SNIP reply A5	← SNIP reply A5
	SNIP reply A6	← SNIP reply A6

Figure 1: Overview of a single SNIP interaction. The right and left columns show frames attempting to be sent by separate nodes. The center column shows what frame, if any, has succeeded in arbitration and appeared on the CAN bus.

[2]See the "Observations About OpenLCB CAN Timing" note.

[3]See section 3.5.1 Reject Address Optional interaction of the Message Network Standard and associated Technical Note section. Note that SNIP is an optional protocol defined by the separate Simple Node Identification Protocol Standard.

SNIP request A→	SNIP request A	
	idle	
	SNIP reply A1	← SNIP reply A1
SNIP request B→	SNIP reply A2	← SNIP reply A2
	SNIP reply A3	← SNIP reply A3
	SNIP reply A4	← SNIP reply A4
	SNIP reply A5	← SNIP reply A5
	SNIP reply A6	← SNIP reply A6
	SNIP request B	

Figure 2: Second SNIP interaction requested while a reply is still in process. Because the reply is higher priority than the request, the request doesn't successfully arbitrate and appear on the bus until the reply is completed.

SNIP request A→	SNIP request A	
SNIP request B→	SNIP request B	
SNIP request C→	SNIP request C	
SNIP request D→	SNIP reply A1	← SNIP reply A1
	SNIP reply A2	← SNIP reply A2
	SNIP reply A3	← SNIP reply A3
	SNIP reply A4	← SNIP reply A4
	SNIP reply A5	← SNIP reply A5
	SNIP reply A6	← SNIP reply A6

Figure 3: Additional SNIP requests arriving before the SNIP reply starts to be sent. In this case, three requests fit within that time, while a fourth one has to wait for later arbitration because the reply has started. This sequence would normally be followed by replies to the 2nd and 3rd requests, then the 4th request and its reply.

SNIP request A→	SNIP request A	
SNIP request B→	SNIP request B	
	idle	
	OIR to request B	← OIR to request B
	SNIP reply A1	← SNIP reply A1
	SNIP reply A2	← SNIP reply A2
	SNIP reply A3	← SNIP reply A3
	SNIP reply A4	← SNIP reply A4
	SNIP reply A5	← SNIP reply A5
	SNIP reply A6	← SNIP reply A6

Figure 4: Second SNIP interaction requested and rejected. The receiving node sends an Optional Interaction Rejected (OIR) message with a temporary-error code in reply to the SNIP request B instead of sending a SNIP reply. The requesting node may or may not retry the second SNIP request later.

2.2 PIP Protocol

The PIP reply is a single frame. Figure 5 shows the basic operation.

The frames of the PIP protocol have a higher priority than the SNIP protocol frames. Should a PIP request be made during a SNIP reply, it will successfully arbitrate and appear on the CAN bus immediately. See Figure 6 for an example of this. It's up to the receiving node's implementation as to when the PIP reply will be sent in this case. Several approaches have been seen:

1. As soon as possible. Generally, since there's a SNIP reply frame (e.g. A4 in the Figure) already loaded for transmission, the PIP reply will be sent right after that one. See Figure 6.
2. At the end of the current SNIP reply transmission, before any other SNIP replies that may be queued up. See Figure 7.
3. After all queued SNIP replies.

The first of these will require fewer PIP buffers in the (rare) event that additional PIP request(s) arrive before the PIP reply is sent.

PIP request →	PIP request	
	idle	
	PIP reply	← PIP reply

Figure 5: Basic PIP interaction.

It's possible that multiple PIP requests can arrive before the first PIP response has been sent. This is similar to the multiple-SNIP requests example shown in Figure 3. PIP requests are mandatory messages and nodes are expected to handle this as a normal condition.

Technically, the node can send an Terminate Due to Error (TDE) message with a temporary error code[4] to prompt the requesting node to resend the request. Like the OIR response to a SNIP message, the MTI in the TDE message provides enough information for the transmitting node to retry the PIP request without any additional state information. But relying on this retransmission of the request is not recommended.

[4]See section 3.5.3 Reject Addressed Standard Interaction Due to Error and section 3.5.5 Error codes of the Message Network Standard and associated sections in the Technical Note.

SNIP request A →	SNIP request A	
	idle	
	SNIP reply A1	← SNIP reply A1
	SNIP reply A2	← SNIP reply A2
	SNIP reply A3	← SNIP reply A3
PIP request →	PIP request	← SNIP reply A4
	SNIP reply A4	
	PIP reply	← PIP reply
	SNIP reply A5	← SNIP reply A5
	SNIP reply A6	← SNIP reply A6

Figure 6: A PIP interaction requested while a SNIP reply is still in process. Because the PIP request is higher priority than the SNIP reply, the request successfully arbitrates and appears on the bus before the reply is completed. When the node replies to the PIP request, e.g. right away or after a delay, is up to the node's implementation. Here the node is trying to reply to the PIP request as soon as possible.

SNIP request A →	SNIP request A	
	idle	
	SNIP reply A1	← SNIP reply A1
	SNIP reply A2	← SNIP reply A2
	SNIP reply A3	← SNIP reply A3
PIP request →	PIP request	← SNIP reply A4
	SNIP reply A4	
	SNIP reply A5	← SNIP reply A5
	SNIP reply A6	← SNIP reply A6
	PIP response	← PIP reply

Figure 7: A PIP interaction requested while a SNIP reply is still in process. Because the PIP request is higher priority than the SNIP reply, the request successfully arbitrates and appears on the bus before the reply is completed. When the node replies to the PIP request, e.g. right away or after a delay, is up to the node's implementation. Here, we show the PIP reply being sent after the SNIP reply is complete.

2.3 Datagram Protocol

Similar considerations apply to datagram interactions. Datagrams are the lowest priority frames[5] so they can be interrupted by the frames of higher priority interactions. In addition, datagram-using protocols often take a stimulus-response form, where receiving a command datagram causes a response datagram to be returned. For example, a Read datagram received by a node will cause that node to send a Read Reply datagram. See Figure 8 for a typical sequence. When a node starts this sequence, it should make sure that it has buffer space available to handle the entire interaction including both the Datagram Received OK message and the reply datagram.

Command datagram →	Command datagram	
	idle	
	Datagram Received OK	← Datagram Received OK
	idle	
	Reply Datagram first	← Reply Datagram first
	Reply Datagram middle	← Reply Datagram middle
	Reply Datagram last	← Reply Datagram last
	idle	
Datagram Received OK →	Datagram Received OK	

Figure 8: A typical datagram exchange sequence. The node receives a command datagram and acknowledges it promptly. It then sends the requested data in a reply datagram, which the originating node then acknowledges.

There are a number of places where another request datagram could be received before the sequence is finished.

A datagram transmission may be interrupted by transmission of another datagram. This is because the priority of the first frame of a multi-frame datagram was defined as higher than the priority of the middle frame, which in turn was defined as higher than the last frame.[6] See Figure 9 for an example of this. The two datagrams become fully mixed, with both sending all their middle frames before any final frames can win arbitration and be put on the CAN bus.

Should there not be enough buffering space when receiving overlapping datagrams, the

[5]Theoretically, stream frames are lower, but they are not yet seen in the wild, and in any case don't change how datagrams arbitrate.

[6]This is now generally recognized as a mistake, as it allows the interruption of one datagram by another. At the time, it was defined this way to ease the work of gateways and routers that reorder higher priority traffic ahead of lower priority traffic.

Datagram B first →	Datagram B first	
Datagram B middle →	Datagram A first	← Datagram A first
	Datagram A middle	← Datagram A middle
	Datagram B middle	← Datagram A last
Datagram B last →	Datagram A last	
	Datagram B last	

Figure 9: A datagram transmission being interrupted by another datagram. The priority of a first datagram frame is higher than a middle or last frame. Note that, for identical frames, datagrams from A will have priority over B due to including the address in the arbitration.

Datagram Transport Standard provides a mechanism[7] for rejecting a datagram and requesting later retransmission, but note that the retransmission is optional. Not all nodes will implement retransmission. It's strongly recommended that nodes receiving datagrams configure their buffering so that this is not necessary.

2.4 Event Transport Protocol

PCER messages arrive asynchronously, and nodes must be able to keep up with them at the full line rate because large numbers of them can arrive consecutively.

The Identify Consumers, Identify Producers, and Identify Events messages are requests that can generate a large number of Consumer Identified and Producer Identified messages in reply. Note that these replies are global messages, not addressed to the particular node that made the request. If multiple requests are made before the reply is sent, only one reply needs to be sent. See the Event Transport Technical Note section 2.6:

> "Nothing prevents combining replies to multiple requests. This allows simplified implementations, for example setting a (per-event) bit to indicate that a reply can be sent when time/priority is available."

In other words, this buffering needs only scale with the number of events that the node consumes or produces, not with the number of requests being processed.

For example, an implementation could associate a single "Send Producer Identified message" with each produced event, and similar for consumed events. Those bits would be set when a request to verify producers, consumers or events was received, and reset when the appropriate message was sent on a time-available basis. No additional buffering would be needed.

[7]See section 4.3 Datagram Rejected and section 6.2 Rejected Transmission of the Datagram Transport Standard and associated sections of the Technical Note.

2.5 Traction Protocol

The interesting case for the Traction Protocol is sending e.g. Set Speed/Direction instructions from one or more throttle nodes to a train node. These use several bytes of payload to define the instruction in addition to the Traction Control Command MTI. The protocol defines these instructions as not receiving any reply message under normal conditions. They are sent blindly, relying on the guaranteed-delivery nature of an OpenLCB network.

If a node runs out of buffer space to receive a Set Speed/Direction or similar message, the node could send an OIR message as discussed in section 2.1 above. However, the OIR message does <u>not</u> contain enough state information for the transmitting node to resend the message without additional stored state: The OIR only contains the MTI, not the content bytes that identify whether this is a Set Speed/Direction instruction vs. a Set Function instruction vs a Emergency Stop instruction, etc.

Further, it is difficult for a throttle node to retain enough information to resend on receipt of an OIR. How long should this information be retained? Which traction message was rejected? In normal operation, there are no replies from the train node to queue the throttle node's buffer management.

A node implementing the train side of the Traction Protocol must be able to keep up with traction protocol messages at full speed.

3 Effect of Inter-Frame Delays

When a node does not send the frames of a reply immediately sequentially, without any additional time between reply frames, there are even more possibilities for requiring additional buffers. If there is a delay of even 4 microseconds, a CAN bit time, a lower priority frame from another node can successfully arbitrate and be transmitted before the next frame of the reply

For example, the hold-off of a second SNIP request seen in Figure 2 will no longer be effective. See Figure 10.

SNIP request A→	SNIP request A	
	idle	
	SNIP reply A1	← SNIP reply A1
	short idle	
SNIP request B→	SNIP request B	
	SNIP reply A2	← SNIP reply A2
	short idle	
	SNIP reply A3	← SNIP reply A3
	short idle	
	SNIP reply A4	← SNIP reply A4
	short idle	
	SNIP reply A5	← SNIP reply A5
	short idle	
	SNIP reply A6	← SNIP reply A6

Figure 10: Second SNIP interaction requested while a reply is still in process with short interframe delays in the reply. The second SNIP request, even though lower priority than the reply frames, can successfully arbitrate into an inter-frame gap and be sent.

This lengthens the window in which overlapping requests can require additional buffers from the time between the request and the start of the reply to the time between the request and the end of the reply. SNIP replies can contain ten or more frames, so this can be a significant extension of the window to receive additional requests. A node's buffer implementation needs to take this into account.

4 Current Status

OpenLCB is designed relying on guaranteed delivery of messages. The protocols keep their buffering needs under control through message and reply priorities that in turn rely on rapid responses with no inter-frame times.

Given that many existing nodes delay their responses and have non-zero inter-frame times, how can this be working now?

Current nodes can be conceptually categorized in a couple of ways. Note that this is just a way of thinking about how nodes are implemented in practice, and doesn't have any particular grounding in the OpenLCB Standards.

First, by their message buffering model:

Nodes With Buffer Pools These are nodes that are based on general purpose processors with lots of CPU and memory. They use a large pool of general purpose memory that lets them buffer large amounts of send and receive traffic. This means they will rarely run out of buffer space. They tend to have longer delays between request and response because of their buffering structure and/or because they're attached to the CAN network via e.g. USB adapters.

Nodes With Dedicated Buffers These typically have smaller processors, typically microcontrollers. They have limited memory, so have to carefully allocate individual buffers to specific purposes. Because they typically run closer to the hardware, these nodes can have shorter times between request and response.

Second, by how they use the OpenLCB protocols:

Control nodes These have human interfaces or independent logic that drives sending requests to other nodes. These can e.g. initiate PIP and SNIP requests to other nodes, send datagrams to other nodes to do memory read or write operations, request event producer and consumer status, etc.

Leaf nodes These are nodes that initiate global messages at startup and under the Event Transfer protocol, but otherwise only respond to messages from other nodes. These nodes do not initiate any interactions that use addressed messages.[8]

The two ways to categorize nodes tend to be correlated, with control nodes often being implemented in the buffer pool style and vice versa. That correlation is not exact, but there are no known control nodes implemented with dedicated buffers.

Control nodes may do startup sequences such as:

- Issuing a Verify Nodes Global message to get a list of the nodes on the network

[8]Implementing the Simple protocol subset is not quite the same thing as a being a leaf node.

	Dedicated Buffers	Buffer Pools
Control nodes		CS-105 JMRI LCCtools
Leaf nodes	Arduino	LRT Fast Clock

Figure 11: A rough categorization of node types with examples.

- For each node, issuing a PIP request to that node to see what protocols it supports
- When applicable, issuing a SNIP request to each node to get its ID information.

That sequence by itself won't cause an issue with buffering in the leaf nodes. Although the control node may be directing overlapping requests to the CAN network, they're going to separate nodes. Each individual leaf node is getting just one request at a time.

Problems can arise when multiple control nodes run this startup sequence at the same time. Even if they were not precisely synchronized at the start, load-related frame synchronization[9] due to lots of Producer Identified and Consumer Identified messages can result in the lower-priority PIP and SNIP messages being placed on the CAN bus at similar times.

Control nodes may also communicate with leaf nodes to configure them. This involves steps like:

- Use datagrams implementing the Memory Configuration Protocol to retrieve the CDI contents via a series of Memory Read datagrams, which return Memory Read Reply datagrams. See Figure 8 for an example of this.
- Use the Memory Configuration Protocol to read the configuration memory contents.
- Use the Memory Configuration Protocol to write new configuration memory contents.

The first two of these involve sending small (one or two frame) datagram messages to the leaf node, and receiving longer (up to nine frames) messages back from the leaf node. The third involves sending datagrams of up to nine frames to the leaf node and getting back small datagrams.

Nothing positively prevents overlapping configuration operations with a single leaf node.[10] Never-the-less, it's uncommon to do simultaneous overlapping configuration operations, as that's generally viewed as a bad practice.

[9]See Section 1.2 "CAN collisions in loaded networks" of the OpenLCB CAN Frame Transfer TN for additional discussion of this.

[10]There is a locking interaction defined in the Memory Configuration protocol, but in practice nodes do not implement that interaction.

If overlapping memory configuration protocol datagrams are sent to a single dedicated-buffer node resulting in a buffer overflow, there is enough information in the 1st frame of the datagram to allow the node to reply with a temporary error[11] and ignore the rest of the datagram. The temporary error should prompt the originating node to try again.[12]

The net effect of all this is that nodes are generally able to handle the traffic they must process, with only small (nut non-zero) chances of buffer overflow. Still, as OpenLCB networks grow and traffic increases, the chances of problems due to imperfect implementations grows.

5 General Recommendations

To reduce the required buffering, all nodes should:

1. Send replies as soon as possible after receipt of the request.
2. Send the frames of a single reply message consecutively, with no intervening idle time on the CAN bus.
3. When there are multiple reply messages pending, send their frames consecutively, with no intervening idle time on the CAN bus.
4. Handle buffer short-falls by issuing temporary error messages.

Control nodes should additionally:

1. Properly handle temporary errors by retrying requests.
2. Limit requests to an individual node to just one at a time.

5.1 Standard and TN Language Recommendations

As of this writing, Message Standard 7.3.7 (current draft) says:

> "CAN implementations shall send the frames of a message together to reduce buffering. Higher-priority messages may be sent in the middle of a lower-priority message, including when the higher priority message itself is fragmented into multiple frames." [13]

There's been a request for clarification as to what "send the frames of the message together" should require: Does it mean consecutively on the CAN bus with no intervening inter-frame idle time? Or some less stringent requirement?

[11] See Datagram Transport Standard section 6.2
[12] Another thing which not all nodes implement.
[13] The last phrase of the second sentence has been added in the most recent draft.

There already are multiple implementations that send the frames of a message with inter-frame idle times. Some of those are quite popular.

With the exception of Events With Payload (EWP, see below), inter-frame idle times only cause buffering trouble for the node who's transmitting with inter-frame idle time. See e.g. the discussion of the SNIP examples above.

In that case,

- We can make a strong recommendation that node implementations send the frames of a message consecutively, with no inter-frame idle time. Associated with that should be a statement that the design of the node's buffering must take into account the possibility of extra buffering required by any inter-frame idle time.
- We can require that the frames of a message be sent consecutively on the line, with no frames of equal or lower priority being interspersed.[14]

Proposed language for the Message Standard is then:

> "CAN implementations shall send the frames of a message consecutively on the CAN bus, with no frames of equal or lower priority being interspersed. Higher-priority messages may be sent in the middle of a lower-priority message, including when the higher priority message itself is fragmented into multiple frames."

paired with proposed TN language:

> "The requirement that frames of a message be sent consecutively is meant to reduce and simplify buffering requirements. Note that this language does not require that the CAN frames be sent with no inter-frame idle time. It's strongly recommended that the frames be sent with no inter-frame idle time, because that can reduce and simplify buffering requirements even further, but that is not required by this Standard."

See the following section for discussion of a stronger requirement for EWP messages.

6 Implications for Events With Payload

Events with payload (EWP) are a way of globally announcing more than just the eight bytes of a PCER message.[15]

[14]This is believed to be the original meaning of the Standard's text. Certainly, several implementors have interpreted it that way.

[15]As of the time of writing this, events with payload are still in proposal form.

The frames of an EWP message are defined to have the first frame's priority higher than the middle frames', which in turn have higher priority than the last frame. With this set of priorities, each EWP message will complete on CAN before the next starts so long as the frames are sent consecutively, without an inter-frame idle time.

The first frame of a EWP message carries the same information as a regular PCER message. A node can skip an unwanted EWP's payload without having to buffer the whole thing by ignoring the frames marked middle and last.

The question is whether to require transmission without inter-frame idle time to make that work properly.

How much trouble will be caused if that's not required?

6.1 Modeling Mean Time Between EWP Collisions

How big of a problem is collisions between EWP messages if inter-frame gaps are permitted?

We can create a simple model for how often they happen. We assume that each message is independently emitted[16] at a random, evenly distributed time with average rate of R(Hz). The messages carry B bytes of payload.[17] The inter-frame gap times are short, but non-zero.

There are then $B/8$ frames in the message, so there are $B/8-1$ inter-frame intervals where a collision can occur. There are approximately 1000 full-size frames per second on the CAN bus. The time for each message where collisions can occur is then approximately $\tau = (B/8-1)/1000$ (seconds).

The assumption of a random, evenly distributed nature of the messages leads to a Poisson distribution with parameter

$$\lambda = R\tau = R(B-8)/8000$$

The probability of one or more messages during the time T of a particular prior message is then

[16] This is certainly a questionable assumption. If the EWP messages are e.g. RFID reports, a train doesn't elicit those randomly, but rather at particular intervals as it travels around the layout. That would lead to sub-Poissonian statistics and fewer collisions than we model here. On a big layout with multiple trains running and various sources of events, the kind of layout likely to lead to collisions, the separate trains and sources are somewhat independent, leading to our assumption here.

[17] The EWP proposal does not currently limit the maximum size of an EWP message. This should perhaps be revisited. A specified maximum size will simplify arranging for sufficient buffering in general-purpose nodes.

$\downarrow B\backslash R \rightarrow$	0.1	0.2	0.5	1
16	100,005	25,003	4,001	1,001
32	33,338	8,336	1,334	334
64	14,291	3,574	572	143
128	6,672	1,669	268	67
256	3,231	809	130	33

Figure 12: Mean time for a EWP collision in seconds. The horizontal axis is message rate in Hertz; the vertical axis is number of bytes in the EWP messages. The value of λ ranges from 0.0001 at the upper left to 0.0310 at the lower right.

$$P(\text{collision}) = 1 - \frac{\mathrm{e}^{-\lambda}\lambda^0}{0!} = 1 - \mathrm{e}^{-\lambda}$$

Since there are R messages per second, the collision rate κ per second is then

$$\kappa = RP = R(1 - e^{-\lambda})$$

And the mean time to a collision τ is then the inverse:

$$T = \frac{1}{\kappa} = \frac{1}{R(1 - e^{-\lambda})}$$

You can see this calculated for rates from 0.1 to 1 Hz and message lengths of 16 to 256 bytes in Figure 12. It ranges from about 30 hours for 16 byte messages averaging 10 seconds apart, to about 30 seconds for 256 byte messages at 1 Hz.

As an aside, we can make a simpler formula by noting that λ is small, so we can replace $e^{-\lambda}$ with $1 - \lambda$. We can then reduce the calculation to

$$T = \frac{1}{R\lambda} = \frac{1}{R^2(B-8)/8000} = \frac{8000}{R^2(B-8)}$$

You can see the inverse dependence on R^2 and $B - 8$ in the table.

Note that the most common collision may be just one EWP message arriving during another, but larger overlaps are possible. In the model explored here, double collisions, with two additional EWP messages arriving during a first message, are a factor of λ less likely.

In the worst case cell of the table, with 1Hz of 256-byte EWP messages, λ is 3%, implying a double collision approximately every 1,000 seconds. Collisions with three messages overlapping the first message, a total of four, are another factor of λ down, at 0.1%. They could be expected to happen every ten hours or so in that worst-case condition. On the other hand, there might be reasons for a railroad to produce multiple messages almost simultaneously due to some specific event. So it seems reasonable to be concerned about overlaps of multiple messages, up to perhaps five or six messages.

6.2 Handling Overlapping EWP Messages in the Standard

The Events With Payload protocol is meant to be a general mechanism. We can't know all the uses, hence typical B and R values, that will appear over time. There's already a proposal[18] that would use EWP messages of 32 - 64 bytes and up. A large layout could send these every few seconds, leading to a mean time to collision that's shorter than an operating session.

EWP messages are global, with no general way to request a retransmission. Nodes must be able to receive and process them at the full rate with which they can arrive.

For example, a router which is converting back-and-forth between CAN and TCP connections[19] has to deal with any size and rate of EWP messages that might arrive. That router has no way to throttle those messages by its choice of operations, unlike control nodes which can decide when to make requests. Nor does it know in advance the payload size of those EWP messages unless some upper limit has been defined by the EWP payload itself.

If interframe-gaps are permitted, collisions can happen and would need to be robustly handled. Losing colliding messages due to buffering issues is unacceptable.

To ensure that no messages are lost, the first step is to prevent colliding EWP messages from overlapping on the CAN bus through the use of the CAN priority mechanism. This requires language in the Standard of the form:

> "The CAN frames making up a single Producer/Consumer Event Report message with payload shall be sent together, with no inter-frame delays or intervening frames of other messages."

Even if the Standard requires that the frames of an EWP message be sent without interframe idle time, it seems likely that some idle-time gaps will happen anyway due to less-than-rigorous implementations. This argues for text in the Standard of the "send tight, receive loose" type:

[18] See the draft Location Services Working Note.

[19] The general case of TCP buffering and rate limiting is beyond the scope of this note.

> "Nodes that consume Producer/Consumer Event Report messages with payload shall be prepared to receive and process Producer/Consumer Event Report messages that have overlapped."

Some nodes only pay attention to the event ID at the start of the EWP message, basically treating it as a plain PCER message. Overlapping messages needed not be a concern to them. They can be written to skip the payload without buffering it by ignoring middle and final EWP frames. There should be a recommendation in the TN of the form:

> "If a node handles PCER messages but doesn't use the event payload, it can skip the payload without buffering it by ignoring middle and final EWP frames."

For nodes that have to process the full EWP contents, there should be some TN discussion of how many overlapping messages might need to be handled. For example, the designer of the router mentioned above would like to see some discussion of what to expect. Some nodes might only process EWP messages due to some particular protocol which provides context for the EWP message's length and rate. Absent that, only general statements can be made such as:

> "Implementors should assume that EWP messages may overlap. Although the most probable collision case is that an EWP message will only overlap with one other EWP message, this cannot be assumed in general. Best practice is to allow for variable buffering up to the available node memory."

Note that this would be more concrete if the EWP protocol itself put a maximum length on the message payload so that buffer calculations can be based on that as a worst case. We therefore recommend that the EWP payload be limited to 256 bytes, including the original Event ID.

7 Summary

In practice, most current OpenLCB installations are unlikely to have buffering problems in normal operation. Many of them have only one or even zero control nodes driving interactions, resulting in limited opportunities for message buffering to be an issue.

Still, there's reason to be concerned. There are already OpenLCB installations with multiple control nodes that can be creating buffering issues. There will be more and more of these as OpenLCB installations grow. Throttles and phone apps are appearing that can act as control nodes. Nodes using the Event With Payload (EWP) protocol will roll out, creating a new category of issues.

As the EWP capability is now being introduced, this is a good time to both define it to reduce buffering as much as possible. At the same time the requirements in the existing

Standards and TNs should be strengthened as much as realistically possible. To that end, there are specific recommendations in sections 5.1 and 6.2 above.