Authors:
Andrey Krokhin (Affinegy)
Way Vadhanasin (Microsoft)
Daniel Mihai (Microsoft)

Version: 2
Revised: July 6, 2016

ASACORE-2946 Notes and Proposals

Most of the changes required to implement ASACORE-2946 and related tasks need to be done to the RemoteEndpoint class. The tasks are discussed in detail below, but due to code interdependence, they might have to be combined or further split into subtasks.

Memory Usage

Currently, there is no way to get the total memory used by all nodes connected to a single Routing Node. Each instance of RemoteEndpoint (which can be either a Routing Node or a Leaf Node) has its own message queue (txQueue) and threads queue (txWaitQueue). A static class variable, such as totalMsgCount, can be added to the RemoteEndpoint class, which should be incremented for every message received. Memory usage is comprised of two parts: stack memory, which can be calculated by totalMsgCount * sizeof(Message), and heap memory, which is consumed by pointers to other objects inside the Message class. In particular, `msgBuf`, `refMsgArgs`, and `handles` are dynamically allocated (there may be others). One possible way to track dynamic memory usage is to add it in constructors/destructors of these objects, similar to

Constructor

```
…
if (s_trackMemoryUsage) {
 QCC_VERIFY( AddAndFetch(&s_totalMemoryUsed, sizeof(*this) ) > 0 );
    }
}
```

Destructor

```
…
if (s_trackMemoryUsage) {
 QCC_VERIFY( SubtractAndFetch(&s_totalMemoryUsed, sizeof(*this) ) > 0
);
    }
}
```

The objective would be to enforce memory limits (which could be infinite for no limit). Each RemoteEndpoint object would be able to return its memory usage, and take action (such as disconnecting from the Router Node) if the limit is exceeded. Optionally, the Router Node could stop accepting further connections from Leaf Nodes if its own limit is exceeded. We could also track global memory usage for all instances of RemoteEndpoint: this might be implemented as a static class variable.

Refactoring PushLeafNode() and PushRouterNode()

The RemoteEnpoint class processes messages from both leaf nodes and routing nodes. The code in PushLeafNode() and PushRouterNode() is largely duplicate, and can be refactored into a common function.

Message queues

Currently, there is inconsistent treatment of messages in routing nodes and leaf nodes. PushMessageLeaf() checks if MAX_TX_QUEUE_SIZE, hardcoded as 1, is exceeded, before deciding to add the message to txQueue. PushMessageRouter() checks whether the message is a “control message” and if it is, whether maxControlMessages is exceeded, before deciding to add the message to txQueue. If the message is not a “control message”, it checks whether MAX_DATA_MESSAGES, also hardcoded as 1, is exceeded, before deciding to add the message to txQueue.

The current code in PushMessageLeaf() and PushMessageRouter() is unnecessarily complicated. There should be a single message queue and a single size limit for that queue. Control and data messages should be added to the same queue and treated in the same way.

The mechanism for removing messages from queue should also be reworked. Currently, messages are only removed if (1) they have been successfully delivered, (2) their TTL has expired, in PushLeafNode() and PushRouterNode():

```
/* Remove a queue entry whose TTLs is expired.
 * Only threads that are the head of the txWaitqueue will purge this deque
 * and enqueue new messages to the txQueue.
 * This is to ensure that the original order of calling of PushMessage
 * is preserved.
 */
         uint32_t maxWait = Event::WAIT_FOREVER;
         if (internal->txWaitQueue.back() == thread) {
             deque<Message>::iterator it = internal->txQueue.begin();
             while (it != internal->txQueue.end()) {
                 uint32_t expMs;
                 if ((*it)->IsExpired(&expMs)) {
                     internal->txQueue.erase(it);
                     break;
                 } else {
                     ++it;
```

```
                    if (maxWait == Event::WAIT_FOREVER) {
                        maxWait = expMs;
                    } else {
                        maxWait = (std::min)(maxWait, expMs);
                    }
                }
            }
```

When a new message gets added to the queue past a certain queue size, or when RN's global memory usage becomes close to the limit, the current consensus is that we should disconnect the endpoint, and destroy its queue. We could also implement a mechanism for deleting messages from queue (and logging an error) if delivery was attempted but failed after a certain time.

Timeouts

There are two types of cases in which an endpoint is closed: a timeout has been exceeded (in Read/WriteCallback() functions), and exceeding maxControlMessages (in PushMessageRouter()). Furthermore, these limits do not appear to be independent, as `maxControlMessages = sendTimeout * MAX_CONTROL_MSGS_PER_SECOND`.

To simplify code, the code branch that executes a node disconnect due to timeout could be removed, and maxControlMessages could be replaced with maxMessages and maxMemory (see "Memory Usage").

Sender vs Receiver

One of the disconnect conditions described happens when maxControlMessages is exceeded in a particular branch of PushMessageRouter():

```
    if (IsControlMessage(msg)) {
        if (internal->numControlMessages < internal->maxControlMessages) {
            internal->txQueue.push_front(msg);
            internal->numControlMessages++;
            if (wasEmpty) {
                internal-
>bus.GetInternal().GetIODispatch().EnableWriteCallbackNow(internal->stream);
            }
        } else {
            QCC_LogError(ER_BUS_ENDPOINT_CLOSING, ("Endpoint Tx failed (%s)",
GetUniqueName().c_str()));
            internal->stream->Abort();
            internal->bus.GetInternal().GetIODispatch().StopStream(internal-
>stream);
            SetState(Internal::EXIT_WAIT);
            Invalidate();
            status = ER_BUS_ENDPOINT_CLOSING;
        }
```

Here, the routing node (receiver) gets disconnected, even though the leaf node (sender) was likely responsible for message overflow. In PushMessageLeaf(), there is no similar disconnect mechanism.

We need to keep track of “noisy apps” that generate excessive message volume, and disconnect them instead of the routing node. Different metrics or limits could be applied here: the absolute number of messages in txQueue, whether a node uses more memory than other nodes (as explained in “Memory Usage”), or the frequency of messages (this can be implemented as a function that goes through each message in txQueue, calculates its timestamp difference with the next message, and computes the average).

There also needs to be a mechanism for making the Routing Node disconnect a Leaf Node that violates one of the limits (memory, queue size, message frequency, etc.). If a new message arrives to the Routing Node, we could do one of:

- If that Message arrived from Leaf Node, the Routing Node should be able to insert this Message into a new list attached to Leaf Node, or

- The Message object could own a reference to the Leaf Node

Currently, we do not have a way to force a disconnect of a leaf node from the routing node.