SAR4 : QueueBroker API in JAVA based on Broker/Channel API from SAR1 work

DO NOT START FROM SCRATCH, USE BROKERS AND CHANNELS FROM SAR1 !!!!

CONSIDER THE POSSIBILITY TO SEND/RECEIVE OVER-OVERSIZED MESSAGES !!

Specification

Overview : Queue

A `MessageQueue` instance is a communication channel, a point-to-point stream of bytes geared to handle size-fixed frames of bytes. Full-duplex, each end point can be used to read or write. A connected `MessageQueue` is FIFO and lossless, see Section "Disconnecting" for details about disconnection.

The typical use of queues is by two tasks to establish a full-duplex communication. However, there is no ownership between queues and tasks, any task may read or write in any `MessageQueue` it has the reference to. The particularity to this `MessageQueue` is that it exclusively either sends the full message when the queue is available or raises an exception in any denial case. On the other side, when any task attempts to read a message (a byte array) from a `MessageQueue`, it exclusively either pops the entire first message sent by another task or raises an exception in any denial case. The following rules apply:

- It is entirely thread-safe for the two tasks to read or write at either end point of the queues concurrently.
- Locally, at one end point, two tasks, one reading and the other writing, operating concurrently is safe also.
- However, concurrent read operations or concurrent write operations are not safe on the same `MessageQueue` end point.

A `MessageQueue` is either connected or disconnected. It is created connected and it becomes disconnected when either side requests a disconnect. There is no notion of the end of stream for a connected stream. To mark the end of a stream, the corresponding queue is simply disconnected.

Connecting

A queue is established, in a fully connected state, when a connect matches an accept. When connecting, the given name is the one of the remote `QueueBroker`, the given port is the one of an accept on that remote `QueueBroker`.

There is no precedence between connect and accept, this is a symmetrical rendez-vous: the first operation waits for the second one. Both accept and connect operations are therefore blocking calls, blocking until the rendez-vous happens, both returning a fully connected and usable full-duplex queue.

When connecting, we may want to distinguish between two cases:

1. There is no accept yet
2. There is not such `QueueBroker`

When the named `QueueBroker` does not exist, the connect returns null. Otherwise, the connect waits until there is a matching accept so that a channel can be constructed and returned.

Note: we could consider introducing a timeout here, limiting the wait for the rendezvous to happen.

Sending messages on queues

> Signature: `MessageQueue::send(byte[] bytes, int offset, int length) -> void`

When sending a message, the given byte array contains the bytes to send from the given offset and for the given length. The range [offset,offset+length[must be within the array boundaries, without wrapping around at either ends, otherwise an exception is raised.

This method waits until the queue is available to send the entire message when the queue is neither disconnected or dangling, otherwise it raises an exception of `DisconnectedQueueException`.

Note: a queue is not a byte stream, so we could consider introducing a timeout here, limiting the wait for sending messages to occur.

If this method is currently blocked and the queue becomes disconnected, the method will throw an `DisconnectedQueueException`. Invoking a sending operation on a disconnected queue also throws an `DisconnectedQueueException`.

Receiving messages from queues

> Signature: `MessageQueue::receive() -> byte[]`

This method attempts to pull the first message sent into the queue. It raises an `DisconnectedQueueException` in any denial case. Once reading, the returned byte array contains the bytes read from the received message. The end of stream is the same as being as the queue being disconnected, so the method will also throw an `DisconnectedQueueException`.

However, while reading, the thread won't raise any exception in case the remote queue gets disconnected. It will terminate to pull its entire message and return it to the user. It is impossible to start a receive call to an already disconnected queue, this systematically raises an `DisconnectedQueueException`.

Note: the disconnected exception does not always indicate an error, rarely in fact. The end of stream is an exceptional situation, but it is not an error. Remember that exceptions are not only for errors, but for exceptional situations, hence their

name. The disconnected exception may give some extra information regarding an error if an internal error caused the channel to disconnect.

Disconnecting

A `MessageQueue` can be disconnected at any time, from either side. So this requires an asynchronous protocol to disconnect a `MessageQueue`.

The effect of disconnecting a `MessageQueue` must be specified for both ends, the one that called the method "disconnect" as well as the other end. In the following, we will talk about the local side versus remote side, the local side being the end where the method "disconnect" has been called.

Note: of course, both ends may call the method "disconnect" concurrently and the protocol to disconnect the `MessageQueue` must still work.

Note: since we have not asserted a strict ownership model between tasks and queues, it is possible that a `MessageQueue` be disconnected while some operations are pending locally. These operations must be interrupted, when appropriate, throwing a disconnected exception.

The local rule is simple, once the method "disconnect" has been called on a `MessageQueue`, it is illegal to invoke the methods "read" or "write". Only the method "disconnected" may be called to check the status of the `MessageQueue`. In other words, if the method "disconnected" returns true, the methods "read" and "write" must not be invoked. If they are invoked nevertheless, the invocation will result in an disconnected exception being thrown.

The remote rule is more complex to grasp, that is, when the remote side disconnects a `MessageQueue`, how should that be perceived locally?

The main issue is that there may be still bytes in transit, bytes that the local side must be able to reads. By in transit, we mean bytes that were written by that remote side, before it disconnected the `MessageQueue`, and these bytes have not been read on a local side. Therefore, if we want the local side to be able to read these last bytes, the local side should not be considered disconnected until all these bytes have been read or the `MessageQueue` is locally disconnected.

This means that the local side will only become disconnected when the remote has been disconnected and there are no more in-transit bytes to read. This means that a local `MessageQueue` appears as not yet disconnected although its far side has already been disconnected. This means that we need to specify how should local write operations behave in this half-disconnected state. The simplest is to drop the bytes silently, as if they were written, preserving the local illusion that the `MessageQueue` is still connected.

This behavior may seem counter-intuitive at first, but it is the only one that is consistent and it is in fact the easiest one on developers. First, allowing to read the last bytes in transit is mandatory since it is likely that a communication will

end by writing some bytes and then disconnecting. Something like saying "bye" and then hanging up.

Second, dropping written bytes may seem wrong but it is just leveraging an unavoidable truth: written bytes may be dropped even though queues are FIFO and lossless. Indeed, it is not at all different than if the bytes were written before the other side disconnected a `MessageQueue` without reading all pending bytes. In both cases, the bytes would be dropped.

Design (on Broker/Channel primitive API)

Connecting via QueueBroker

A `QueueBroker` asks its `Broker` to seek for the adequate remote `QueueBroker` to establish a connection between two tasks. The connection is blocking unless the adequate remote `MessageQueue` is returned to the task (when the remote `QueueBroker` accepts the connection) or it is known that the wanted remote broker does not exist.

Note: we could consider introducing a timeout here, limiting the wait for the rendezvous to happen.

Once the connection is establish, each task receives its corresponding `MessageQueue`. A `MessageQueue` contains two `CircularBufferQueue`. One of them is considered as "Input Stream" for one of the tasks and as "Output Stream" for the other task", and vice-versa.

A `CircularBufferQueue` stores entire messages with a fixed size. The `pull` method returns the first entire message put in the buffer and remove all its content from the buffer. This method returns `null` in case the `CircularBufferQueue` is empty. The `push` method attempts to put the given message in the buffer and blocks the writing thread unless the buffer is not full anymore. It is possible to check the fulfillness of a buffer by calling method `full()`:

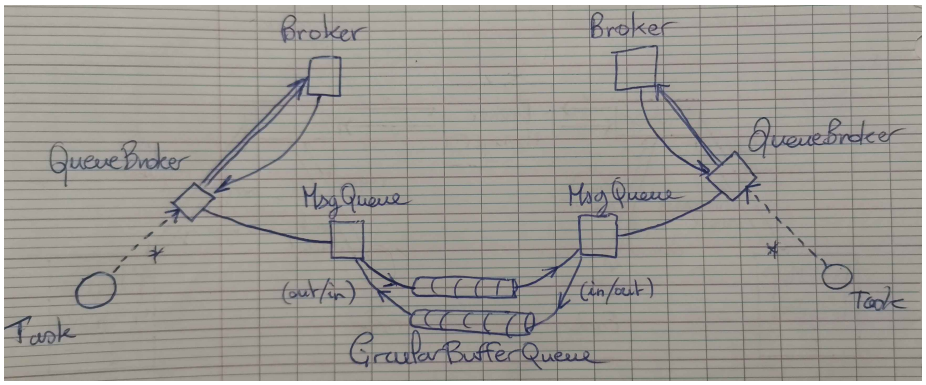

Figure 1: Design

Reading, Writing and Disconnecting.

As mentioned further above, writing or reading can be blocking. When a `MessageQueue` gets disconnected by any of both tasks, all writing threads gets interrupted and throw an `DisconnectQueueException`, whereas all current reading threads finish to read data from the `CircularBufferQueue`. However, it is impossible to read from an already disconnected `MessageQueue`, this raises an `DisconnectQueueException`.