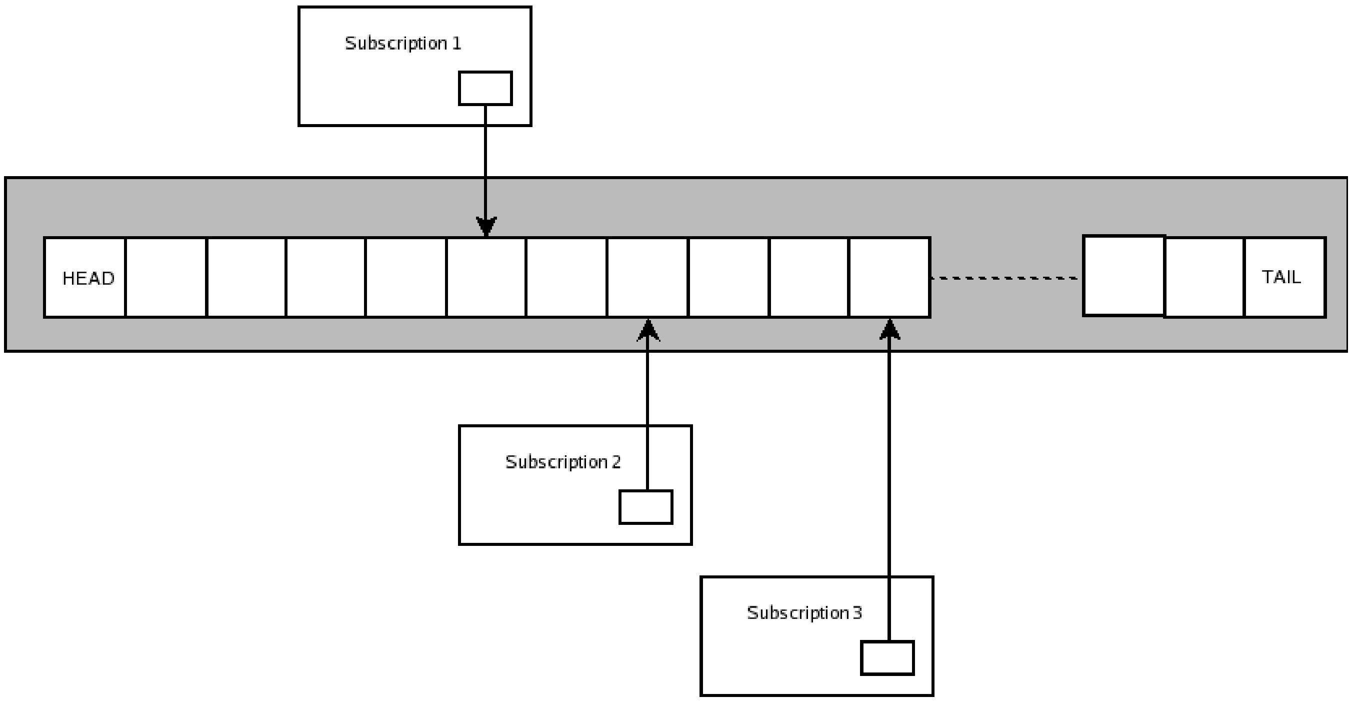
Qpid Design - Queue Implementation

Strict Ordering

The fundamental principal of the Queuing model is that the queue provides a strict order on the messages being enqueued. Furthermore that order is maintained through the lifetime of the entries on the queue: thus if a message is returned (e.g. the prefetched messages being released upon the consumer closing) the order of that message with respect to other messages on the queue is maintained.

The strict ordering is enforced by the use of a queue data-structure. In order for this to be performant, the data structure uses a lockless thread-safe designed based around the same algorithm used in the java.util.concurrent.ConcurrentLinkedList (more precisely it is based on the public domain implementation in the backport util concurrent project). See the section on Concurrent List implementations for more details.

Each subscription keeps a "pointer" into the list denoting the point at which that particular subscription has reached. A particular subscription will only deliver a message if it is the next AVAILABLE entry on the queue after the pointer which it maintains which matches any selection criteria the subscription may have.



Thread safety is maintained by using the thread-safe atomic compare-and-swap operations for maintaining queue entry state (as described above) and also for updating the pointer on the subscription. The queue is written so that many threads may be simultaneously attempting to perform deliveries simultaneously on the same messages and/or subscriptions.

Enqueing

When a message is enqueued (using the enqueue() method on the AMQQueue implementation) it is first added to the tail of the list. Then the code iterates over the subscriptions (starting at the last subscription the queue was known to have delivered for reasons of fairness). For each subscription found it attempts delivery (details describe below). If the message cannot be delivered to any subscription then the "immediate" flag on the message is inspected. If the message required immediate delivery then the message is immediately dequeued, otherwise an asynchronous job is created to attempt delivery at a later point.

(Note there is a "shortcut" path for queues which have an exclusive subscriber. In this case we know there is one and only one subscriber and so we can go directly to trying to deliver to it without worrying about iterators, etc.)

Potential Issue: Looking at the code which performs the check of the immediate flag I believe there is a race condition:

```
        if (entry.immediateAndNotDelivered())
        {
            dequeue(storeContext, entry);
            entry.dispose(storeContext);
        }
```

This does not look to be safe in the case where there is a simultaneous execution of an asynchronous deliver which may acquire the message between the check of immediateAndDelivered and dequeue. Instead of calling dequeue directly we should instead do a safe compare-and-swap test to make sure the entry state is "AVAILABLE" before setting it to DEQUEUED. The implementation of this should probably look much like the implementation of entry.dequeue except for the different expected starting state.
Immediate Delivery

For each subscription to the queue, we call the following code:

```
    private void deliverToSubscription(final Subscription sub, final QueueEntry entry)
            throws AMQException
    {

        sub.getSendLock();
        try
        {
            if (subscriptionReadyAndHasInterest(sub, entry)
                && !sub.isSuspended())
            {
                if (!sub.wouldSuspend(entry))
                {
                    if (!sub.isBrowser() && !entry.acquire(sub))
                    {
                        // restore credit here that would have been taken away by wouldSuspend since we didn't
manage
                        // to acquire the entry for this subscription
                        sub.restoreCredit(entry);
                    }
                    else
                    {

                        deliverMessage(sub, entry);

                    }
                }
            }
        }
        finally
        {
            sub.releaseSendLock();
        }
    }
```

This code first takes a lock on the subscriber (this prevents it being removed while we are carrying out this operation). It then tests if the given subscription can take this message at the moment (see below for more details). It then further tests if there is enough flow control credit to send this message to this subscription. If there is credit (and the subscription is not a "browser" then is attempts to acquire the entry (entry.acquire(sub)). If the acquisition is successful (or if the subscription is a browser and thus does not need to acquire the entry) then the entry is delivered to the subscription, else the credit that would have been used by the message if sent is restored.

The most interesting method called in the above is subscriptionReadyAndHasInterest(sub, entry):

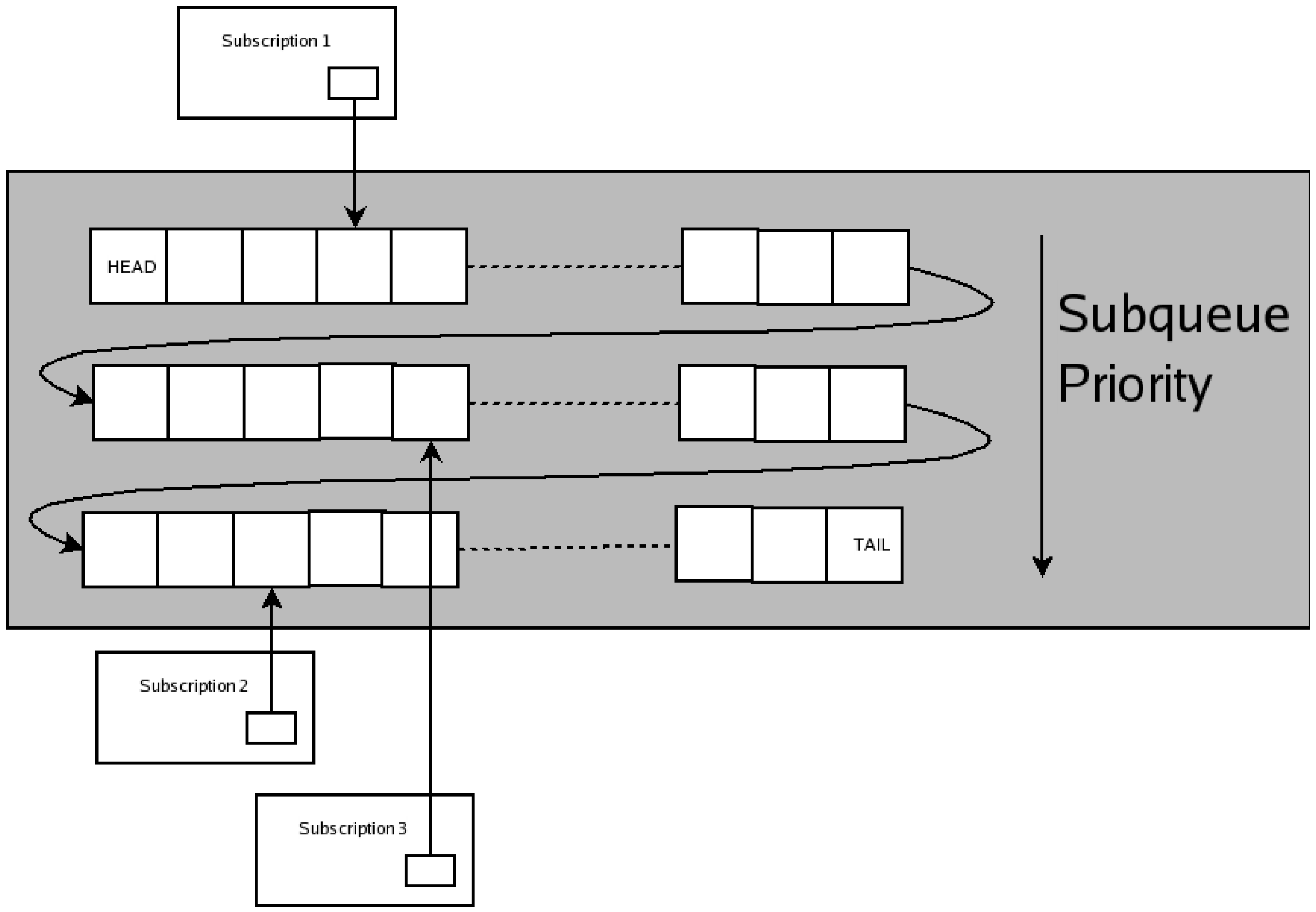
```
    private boolean subscriptionReadyAndHasInterest(final Subscription sub, final QueueEntry entry)
    {
        // We need to move this subscription on, past entries which are already acquired, or deleted or ones it
has no
        // interest in.
        QueueEntry node = sub.getLastSeenEntry();
        while (node != null && (node.isAcquired() || node.isDeleted() || !sub.hasInterest(node)))
        {

            QueueEntry newNode = _entries.next(node);
            if (newNode != null)
            {
                sub.setLastSeenEntry(node, newNode);
                node = sub.getLastSeenEntry();
            }
            else
            {
                node = null;
                break;
            }

        }

        if (node == entry)
        {
            // If the first entry that subscription can process is the one we are trying to deliver to it, then
we are
            // good
            return true;
        }
        else
        {
            return false;
        }

    }
```

Here we see how the subscription is inspected to see where its pointer into the queue (the last seen entry) is in respect to the entry we are trying to deliver. We start from the subscription's current lastSeenEntry and work our way down the list passing over entries which are already acquired by other subscriptions, deleted, or which this subscription has no interest in (e.g. because the node does not meet the subscription's selection criteria); all the while we can update the lastSeenEntry to take it past the entries this subscription has now inspected. Performing this iteration we will eventually arrive at the next entry the subscription is interested in (or just fall off the end of the list). At this point either the next entry that the subscription is interested in is the entry we wish to deliver (success!) or not.

Priority Queues

The fundamental difference between Priority Queues and other Queues is that the strict ordering on the queue is not purely FIFO. Instead the ordering is a combination of FIFO and the priority assigned to the message. To provide strict priority ordering (where a message of higher priority will always be delivered in preference to a message of lower priority) we can implement a priority queue as an ordered list of standard sub-queues with the ordering between them defined such the tail of the highest priority sub-queue is followed by the head of the sub-queue of the next highest priority.

By defining the standard queue implementation such that the methods which determine the ordering between the nodes can be overridden, the implementation of such a strict priority queue is almost trivial.



The interface QueueEntryList provides an extension point for adding a new queue implementation type:

```
public interface QueueEntryList
{
    AMQQueue getQueue();

    QueueEntry add(AMQMessage message);

    QueueEntry next(QueueEntry node);

    QueueEntryIterator iterator();

    QueueEntry getHead();
}
```

The class PriorityQueueList provides the concrete implementation of a strict priority queue as defined above. The constructor takes an argument defining how many priority levels are to be provided.

When a message is added to the list by calling the add() method, the class first works out which sub-queue to add the message to. This is determined by an algorithm identical to that defined in the AMQP0-10 specification and compliant with the JMS requirements. The message is added to the tail of the appropriate sub-queue.

The next() method returns the QueueEntry which logically follows the QueueEntry provided by the caller. First we can simply look at the sub-queue in which the passed QueueEntry is actually in. If there is a subsequent entry in that sub-queue then we use that. If there is no subsequent entry in the sub-queue then we must find the next highest priorty subqueue and take the head of that (repeating until we find a subqueue which is non-empty).

The getHead() method iterates over the subqueues to find the highest priority sibqueue which is non-empty and then returns the head of that subqueue.

The iterator() method returns an iterator that respects the ordering defined above.
The only other difference between a PriortyQueue and the standard queue is that new messages arriving may be logically "before" messages that have arrived previously (i.e. a high priority message is always logically prior to a low priority message in the queue). This means that on arrival of a message into the queue all subscriptions need to be inspected to make sure their pointer is not "ahead" of the new arrival.

Thus the entire implementation of AMQPriorityQueue is as follows:

```
public class AMQPriorityQueue extends SimpleAMQQueue
{
    protected AMQPriorityQueue(final AMQShortString name,
                               final boolean durable,
                               final AMQShortString owner,
                               final boolean autoDelete,
                               final VirtualHost virtualHost,
                               int priorities)
            throws AMQException
    {
        super(name, durable, owner, autoDelete, virtualHost, new PriorityQueueList.Factory(priorities));
    }

    public int getPriorities()
    {
        return ((PriorityQueueList) _entries).getPriorities();
    }

    @Override
    protected void checkSubscriptionsNotAheadOfDelivery(final QueueEntry entry)
    {
        // check that all subscriptions are not in advance of the entry
        SubscriptionList.SubscriptionNodeIterator subIter = _subscriptionList.iterator();
        while(subIter.advance() && !entry.isAcquired())
        {
            final Subscription subscription = subIter.getNode().getSubscription();
            QueueEntry subnode = subscription.getLastSeenEntry();
            while(subnode != null && entry.compareTo(subnode) < 0 && !entry.isAcquired())
            {
                if(subscription.setLastSeenEntry(subnode,entry))
                {
                    break;
                }
                else
                {
                    subnode = subscription.getLastSeenEntry();
                }
            }

        }
    }

}
```

The constructor merely ensures passes up the machinery to ensure a PriorityQueueList (as described above) is used for the underlying queueing model. The getPriorities() method is overridden by delegating to the PriorityQueueList and then the algorithm for updating the subscriptions' pointers into the queue is implemented in checkSubscriptionsNotAheadOfDelivery. Thread-safe compare-and-swap operations are used to update the pointer in-case other threads are also trying to move it; and the loop terminates early if the new QueueEntry has already been acquired.