High Speed Queue

# Introduction

A High Speed Queue (HSQ) is a mechanism for message passing between components of a system. It is most often used to pass messages between two or more threads in the same process, but it can also be used for intra-process communication by putting the HSQueue in shared memory.

## Clients

HSQueue clients can be either a Publisher or a Consumer.

* Producers create and publish data into the HSQueue.
* Consumers accept and process data from the HSQueue.

Each HSQueue instance supports multiple Producers, but only a single Consumer.

# Client API

The HSQueue API consists of the public interfaces of four primary classes: Message, Connection, Consumer, and Producer. Two additional classes, CreationParameters and ConsumerWaitStrategy are used to initialize the HSQueue.

Start with a Connection

A Connection provides access to a HSQueue. A Connection may be used to create an initialize a new HSQueue, or it may be used to attach to an existing HSQueue (presumably in shared memory.) Once the HSQueue is created or located, the Connection can be used to initialize Producer and Consumer clients.

Step-by-step:

* Construct a Connection object. No construction arguments are required.
* If the Connection will be used to create a new HSQueue:
  + Construct a ConsumerWaitStrategy which specifies how Consumers wait for messages
  + Construct a CreationParameters object containing the ConsumerWaitStrategy and other configuration information such as the number of messages that can be queued, the maximum size of a message, etc.
    - Note that for performance reasons HSQueue does not do any dynamic memory allocation. Thus it must be configured properly at initialization time to achieve the best performance.
  + Call Connection::createLocal() or Connection::createOrOpenShared().
* If the Connection will be used only to locate an existing HSQueue in shared memory
  + Call Connection::openExistingShared();

This Connection object may now be used to create a Consumer and one or more Producers.

## To Add a Consumer

* Construct a Consumer passing the Connection as an argument.
* Construct a Message with no construction arguments.
* Use the Connection::allocate(Message) to initialize the Message.
  + Most consumers will need only one Message which should be created at the time the consumer is being initialized. This Message will be reused for each message received.
* Accept a message from the HSQueue by calling either:
  + - Consumer::tryGetNext(Message &), or
    - Consumer::getNext(Message &)
  + If a message is available, these methods are identical.
  + If no message is available
    - tryGetNext() returns immediately with a false result, whereas
    - getNext() will wait for the message to return using the ConsumerWaitStrategy that was specified when the HSQueue was created.
      * getNext() may also return false if the HSQueue is shutting down.
* Use methods on the Message to access the information from the message.
* When the consumer no longer needs the contents of a message, it can simply reuse the message for the next tryGetNext() or getNext() call.

To Add One or More Producers

* Construct a Producer passing the Connection as the contruction argument.
* Construct a Message with no construction arguments.
  + Most producers will need only one Message object which should be created at the time the producer is being initialized. This Message object will be reused for each message being sent.
* Call Connection::allocate(Message &) to initialize the message.
* Populate the Message with the information to be passed to the consumer.
  + Several methods on the Message make this easy and type-safe. These methods will be described later.
* Publish the message by calling the Producer::publish(Message &); method.
  + When the publish() method returns, the Message will be empty, ready to be populated with the next message.

## Notes

* The producer may take as long as it needs to construct a message directly in the Message. If there are other producers using the same HSQueue they will be unaffected by the time it takes a particular producer to prepare its message for publication. (See the example below.)
* Clients lose access to information in a message once they use the Message object in a publish() or getNext() call. In particular pointers to the data contained in a message are invalided when the Message object is reused.
* When a client is ready to exit, it simply lets the Message and Producer or Consumer objects go out of scope (or otherwise be deleted.) This cleans up the resources used by the client.
  + The Connection will still be ready to service additional clients.
* The Connection object MUST live longer than all clients and Messages that use it.
  + This requirement is not currently enforced.

## Populating a Message Before Publishing it

To be done

## Access Data From a Received Message.

To Be done

# A typical example

A WORK IN PROGRESS.

A publisher that is accepting and publishing incoming multicast (UDP) messages can issue the socket read request using the pointer returned by Message::get() and the message size returned by Message::getCapacity().

When the socket read completes the publisher should use the Message::setUsed() method to record how many bytes of data were received.

Because UDP message boundaries match transmission packet boundaries, the message is now ready to publish.

After the publish() call the Producer has an empty message that it can use for the next socket read request.

When the consumer's call to getNext() returns the Message::get() and Message::getUsed() methods will allow the consumer to access the memory area via its own Message object. This is exactly the same memory area that was used to read the packet from the socket.

# Theory of Operation

## Offsets vs Addresses

In this implementation of An HSQueue addressing is always done via offsets to a base address rather than actual addresses -- up to the point where final resolution into a physical address is needed. This has an added benefit of letting different clients in different processes work effectively with An HSQueue contained in shared memory -- even if the shared memory is mapped into different virtual memory addresses in different processes.

Caveat: At the time this documentation is being written the Shared Memory support is incomplete and there are a few serious issues to resolve before it goes live.

Memory moves vs Pointer moves.

In spite of advances in CPU design over the years, the memcpy function still shows up regularly in profiler reports. This implementation of An HSQueue attempts to avoid memory copies by using pointers and offsets encapsulated in Messages. A Message is a handle to a section of memory. Two messages can be swapped with each other so that message A now points to the memory formerly held by message B, and vice versa. This is a cheap operation compared to the cost of moving the actual data.

To make this work well, all messages should be the same size, and they work best if they are all allocated from the same pool of memory. In particular all the messages used in shared memory must be visible to all of the clients of the HSQueue. Thus there is a pool of memory in the HSQueue itself that can be used to populate Messages.

Cache line alignment and related performance issues.

For performance reasons many of the objects contained in the HSQueue are aligned on cache line boundaries. In general the goal is to have a single writer for each cache line (although multiple readers are allowed.)

This cache line alignment is a tradeoff between memory usage an speed. For this implementation, speed trumps memory size.

On the other hand, some high performance message passing systems pay close attention to the effect of cache-line prefetch. In my tests, this effect is minor or disappears completely in a practical implementation (at least in C++), so it was ignored.

Java developers of high performance systems report encountering issues related to having too many object references which must be visited by the garbage collector. If the offset approach used in the HSQueue can be mapped into a corresponding Java implementation, this may alleviate that problem.