ProntoQueue

# Introduction

A ProntoQueue is a mechanism used 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 ProntoQueue in shared memory.

## Clients

ProntoQueue play one of two roles.

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

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

# Client API

The ProntoQueue API that 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 ProntoQueue.

Start with a Connection

* Create a ConsumerWaitStrategy which specifies how Consumers wait for messages
* Create 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 ProntoQueue does not do any dynamic memory allocation. Thus it must be configured properly at initialization time to achieve the best performance.
* Create a Connection object.
* Either:
  + Create a new ProntoQueue using the CreationParameters, or
    - For an in-process ProntoQueue call createLocal()
    - For a shared memory ProntoQueue call createOrOpenShared()
  + Attach to an existing ProntoQueue that was created by another process in Shared Memory.
    - Call either openExistingShared() or createOrOpenShared()
    - Note that CreationParameters are not needed for the openExistingShared() method.

This Connection object may now be used to create a Producer, a Consumer (or both).

## Add a Consumer

* Create a Consumer passing the Connection as an argument to the constructor.
* Create a Message and use the Connection to initialize it.
  + 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 ProntoQueue 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 ProntoQueue was created.
      * getNext() may also return false if the ProntoQueue is shutting down.
* Use methods on the Method 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.

Add One or More Producers

* Create a Producer passing the Connection as an argument to the constructor.
* Create a Message and use the Connection to initialize it.
  + The connection may be passed to the Message constructor, or an uncommitted message may be created then later initialized by calling Connection::allocate(Message &)
  + 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.
* 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 ProntoQueue 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.

A typical example:

///

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.

///

Users of ProntoQueues may stop reading here. Developers and maintainers (and curious users) may continue

to:

///

Theory of Operation

///

ProntoQueues vs RingBuffers

///

A ProntoQueue is very similar in purpose and implementation to a RingBuffer. Since a RingBuffer is a

more familiar concept to most programmers, it is worth providing the reasoning behind using the

ProntoQueue abstraction rather than a Ring Buffer.

///

Notice that neither a RingBuffer nor A ProntoQueue is a direct match for the way computer memory is

actually organized. Computer memory is not organized in circles, the way a RingBuffer presents it. Computer

memory is accessed linearly the same way a vector is, but it is not infinite.

///

As it turns out a RingBuffer an awkward abstraction to implement because the boundaries of the actual memory

block used to hold the entries do not have convenient representations in ring-buffeRese. Operations such as

comparing two positions within the buffeR are more difficult when the buffeR boundary may, or may not appear

between the two positions.

Copying data into or out of a RingBuffer can get complex -- particularly when

the external source or destination is not itself contiguous. (For example when copying data directly from one

RingBuffer to another.) None of these issues are show-stoppers, they just make the implementers life more

difficult, and if they are not handled well, they may lead to performance issues.

///

It turns out that simulating infinity is easier than simulating circularity.

///

For all practical purposes, the indexes into the ProntoQueue can be treated as unbounded[note below] integers.

They can compared incremented, etc. freely. It is only when an index needs to be resolved into the actual

address of an entry that the mapping operation need to be applied. This operation can be localized and hidden

so that most of the time it can be ignored. (The copying data issue is handled by using referencs to fixed size

memory areas.)

///

Note: unbounded means if you use a 64 bit unsigned integer for an index and publish an entry every nanosecond, you

won't run out of indexes for several hundred years.

///

Rather than worrying about boundaries every "N" entries the way a Ring Buffer does, the ProntoQueue must be aware

that only the last "N" entries are visible at any time. As this visible window moves forward in the vector, previous

entries in the vector become invisible forever. As long as the producers are careful not to get more than N entries

ahead of the consumer, this is not a problem. Using unbounded indexes makes it easy for the producers to enforce this

limit.

Offsets vs Addresses

///

In this implementation of A ProntoQueue 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 A ProntoQueue 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 suport is incomplete and there are a few issues

to resolve before it goes live.

///

Messages: Memory moves vs Pointer moves.

///

A message for the point of this discussion is a handle to a block of memory.

///

In spite of advances in CPU design over the years, the memcpy function still shows up regularly in profiler reports.

This implementation of A ProntoQueue 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 ProntoQueue. Thus there is a pool of memory in the ProntoQueue 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 ProntoQueue 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.) 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 ProntoQueue can be mapped

into a corresponding Java implementation, this may alleviate that problem.

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