Storm 2.0 Messaging System Redesign

[STORM-2306]

Roshan Naik
Jul 24th 2017

1. Current Messaging Architecture

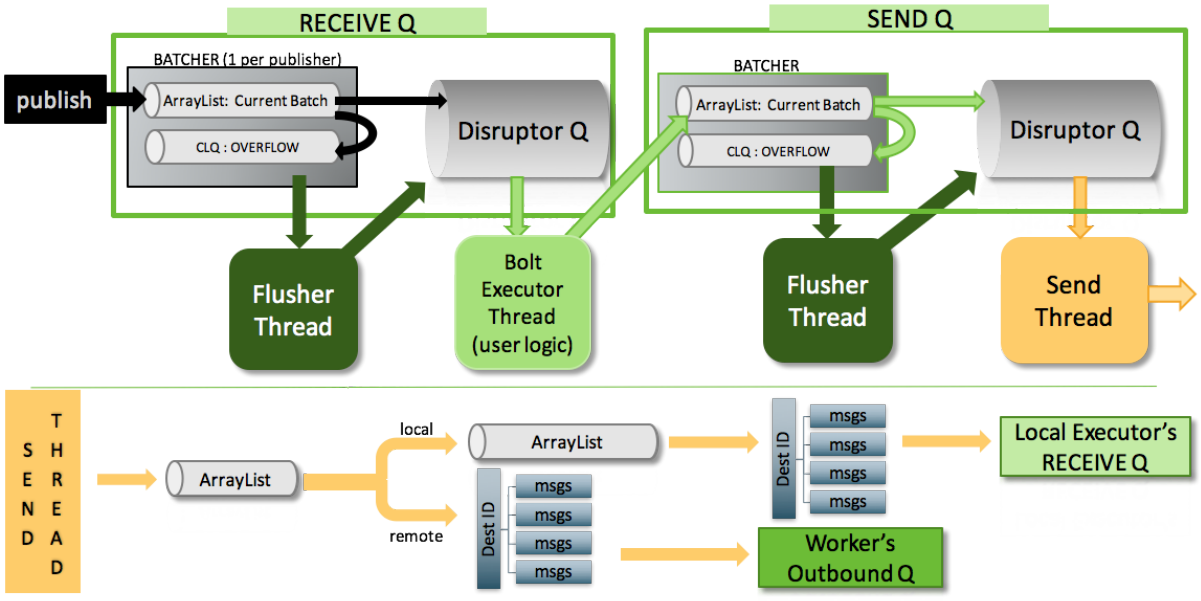

Fig 1. Current Architecture

2. Revised Messaging Architecture

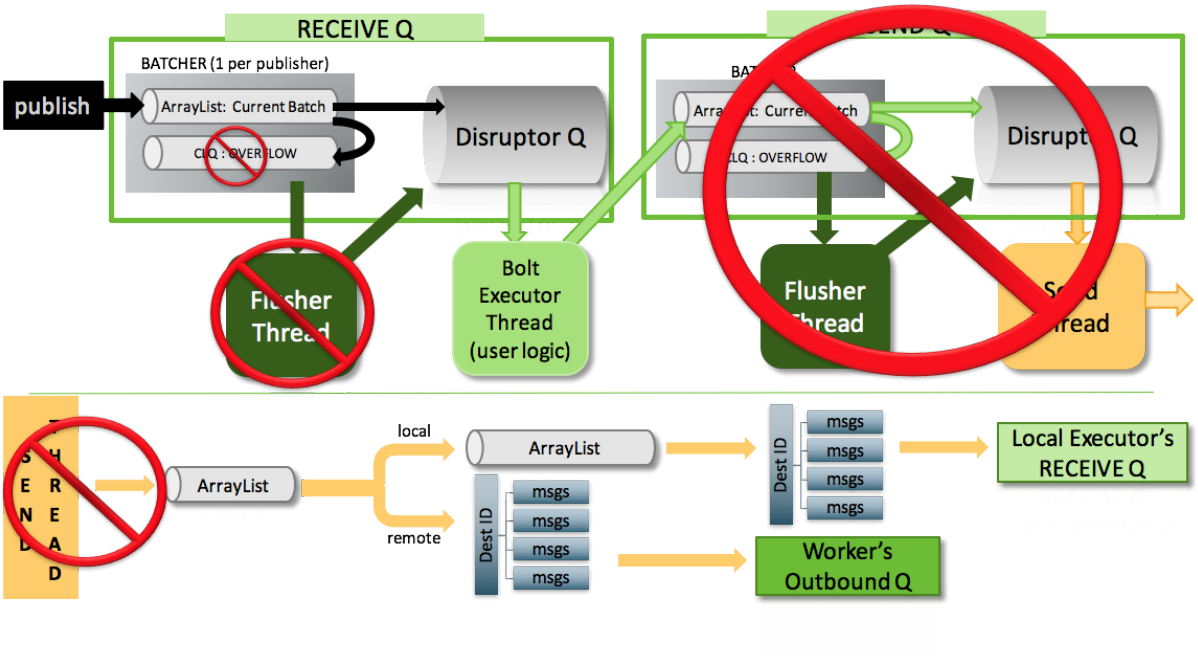

Fig 2. Changes to Current Architecture

We eliminate the SendQ, Send thread, Flusher threads and overflow queue. This simplifies the design significantly.

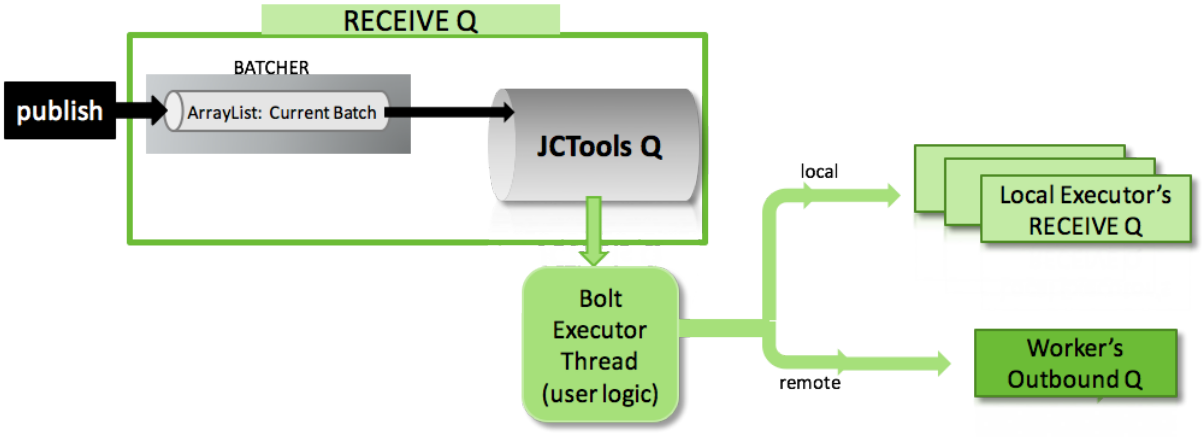

Fig 3. New Architecture

3. Major Changes

Some major changes that may not evident from the diagram:

3.1) JCQueue.java replaces Disruptor.java:

Storm has a class called Disruptor that is a wrapper around the ring buffer provided by the LMAX Disruptor library. This class is not to be confused with another class also called Disruptor that is also part of the LMAX Disruptor library. This class has been replaced with a class called JCQueue. It uses the messaging queues from JCTools instead of Disruptor. The JCQueue will internally instantiate either the MPSCArrayQueue for the multi producer case or the SPSCArrayQueue for the single producer case, based on the constructor arguments.

3.2) One thread per executor:

The existing architecture requires 4 threads per spout/bolt instance. 1 executor thd, 1 send thd, 2 flusher threads. The new design needs only 1 executor thread.

3.3) Elimination of Two Level batching:

In the existing architecture, the individual elements in 'currentBatch' list are actually ArrayLists<Tuple> and not Tuple as one might expect. Consequently each element in the DisruptorQ actually contains ArrayList<Tuple>. The publishing upstream bolt/spout output collector's emit() call groups the outgoing tuples into an ArrayList before writing. The batcher then waits for the configured `topology.disruptor.batch.size` items (each an ArrayList) before draining them into the Disruptor Q. So there is two level batching that is basically occurring in the publish path. The batch size used for both the first level of batching in the upstream spout/bolt and the size of currentBatch's size is the same (`topology.disruptor.batch.size`). It has a default value of 100, making the effective batch size = 100*100 = 10k.

The new design eliminates the first level of batching and retains just the current batch. Consequently the JCToolsQ will now contain Tuple objects as individual elements.
The configuration `topology.producer.batch.size` Is used to control this batch size.

3.4) What about Flushing / Flusher Threads ?

When there is batching going on, we also need a way to flush the batch out periodically to ensure tuples don't get stay there for long time when there is now new incoming data. The existing design use one Flusher thread per DisruptorQ for this. One flusher each for the sendQ and recvQ. So two flusher threads per spout/bolt executor.

The new design eliminates the flusher thread and introduces a 'FlushTuple'. When a spout/bolt executor receives this tuple it flushes any buffered output. There is a single global timer thread in the worker process that periodically sends a FlushTuple directly to all the recvQs of local executors. The configuration `topology.batch.flush.interval.millis` controls how frequently this tuple is generated.

When a executor's recvQ is full, it is an indication that either that there is backpressure situation going on or there sufficient incoming traffic. So if the recvQ of any executor is full and unable to accept the FlushTuple at any given time, the timer thread will skip over that executor and try sending it again at the next interval.

3.5) Lock Free critical path

All locks and lock based data structures in the critical messaging path have been eliminated. However, If debug logs are enabled there could be some locks kicking in as the LOGger appears to be using locks internally.

Some key ones that are gone:

- ArrayBlockingQueue in FlusherPool's `ThreadPoolExecutor`. Due to elimination of the Flusher thread, we dont need a substitute for this.
- The flushLock (of type ReentrantLock) used by the ThreadLocalJustInserter which kicks in when batching is disabled (i.e batchSz=1). This lock was used to scynhronize between the flusher thread
- The ConcurrentHashMap (`_batchers`) used to manage a per producer thread BatchingInsergter (or non batching inserter) by the Disruptor. In the new design, for the non-batching case, we use a single shared thread safe DirectInserter object among all producer threads. For the batching case, we store the batcher instance in the Thread Local Storage of the producer thread.

3.6) What about Overflow List ?

The existing design maintains an overflow list in the batcher. This is an unbounded linked list. If the DisruptorQ is full and it is time to flush() the current batch (because it's full or batch timeout has kicked in) into the DisruptorQ, then the currentBatch is drained in to the overflow.

If topology.max.spout.pending and back pressure (BP) are not enabled, this overflow will grow in an unbounded manner and eventually crash the worker process with an OutOfMemoryException. The max.spout.pending only works in the ACKing enabled case. For the Acking mode, max.spout.pending provides a cheaper alternative to the current BP model to avoid the out-of-memory situation.

In next section we will see how eliminating the notion of an unbounded overflow automatically gives us a solution to this out-of-memory problem, and a very cheap back-pressure model that works in both ACKing and nonAcking modes.

[Updated Dec 6th 2017] The overflow list has been replaced with a 'pendingEmitsQ' that has different semantics. Its purpose is to eliminate the possibility of deadlocked cycles in the presence of an ACKer bolt. The simplest cycle in the topology graph is of the form: Spout->ACKer->Spout. Such cycles can get deadlocked if the spout's emit is blocked on the ACKer bolt's full queue and consequently unable drain its recvQ which then gets filled with ACK tuples from the ACKer bolt. Consequently the ACKer and Spout are blocked on each other and no component is able to emit, the topology enters a deadlocked state. A bit more complex graph is Spout -> Bolt -> ACKer -> Spout.

Pending Emits Q: The SpoutOutputCollectorImpl.emit method (or BoltOutputCollectorImpl.emit) will only emit to new 'pendingEmitsQ' the first time it detects that the destination recvQ is full. The pendingEmitsQ is hosted by SpoutExecutor and is not part of JCQueue. The SpoutExecutor is able to detect if it's pendingEmitsQ is empty or not. After invoking spout.nextTuple(), if it notices that the pendingEmitsQ is not empty, the SpoutExecutor will not call spout.nextTuple() anymore until the pendingEmitsQ is fully drained. At this point, the SpoutExecutor alternates between trying to drain the overflow (in a non blocking manner) and processing incoming messages like ACKs and metrics. Since the spout is now able to drain its receive queue even if it is unable to write to an outbound queue, deadlock formation is prevented.

This Q holds only the *overflow* emits that occurred within a single nextTuple() invocation. By not invoking nextTuple() until pendingEmitsQ is drained, we prevent it from growing in an unbounded manner and lead to an OutOfMemory situation. If we intend to support the specialized use case of Spouts that want to spin up background threads and perform emits()

outside of a nextTuple() call, a special form of emit() API would be required to ensure that overflow does not grow in a unbounded fashion.

3.7) Back Pressure: [updated on Dec 6th, 2017]
This section has been moved to a separate document:

3.8) Garbage Collection (GC) Pressure:
The best garbage is the one that is never produced. Good garbage is the one that is recycled. Reducing allocations and reusing objects when possible is key to reducing GC pressure. Fewer memory allocations requests means lower latencies and better throughput. Additionally fewer objects need to be collected during GC cycle, so faster GC and less frequent GC. both of which help narrow the gap between max and avg latency.

Much work has gone to identify the allocations occurring in critical path. Allocating one object in the critical emit/consume path between two executors causes millions/thousands of allocations per second for every executor->executor edge in the worker.

Example: A common problematic pattern in the critical path that has been eliminated is iterator based loops like this:

```
for( String str : listOfStrs ) { … }
```

These have been replaced with index based loops:

```
for( int i=0; i < listOfStrs.size();  ) { … }
```

Iterator based loops require allocation of iterator object unlike indexed loops. In the critical messaging path such allocations are undesirable.

3.9) Optimizations in Outbound Path from Spout/Bolt to Worker Transfer Queue:
Tuples that are destined to the executor in a different process are sent to the WorkerTransfer Queue. The Worker Transfer thread drains this queue and send the tuples to their appropriate destination via netty.
Following changes have been done in this path from the spout/bolt to the WorkerTransfer Queue.

- **Eliminate two level batching.** Here too there was a case of two level batching similar to the one discussed previously. The emitting bolt would group remote bound messages into a hashmap bucketed by destination. This entire map would then be inserted into the TransferQueue as one element. These hashmaps would get buffered in the outbound ‘currentBatch’ before getting actually getting flushed into

the TransferQueue. This has two level batching has been flattened out. This also simplifies the

- **Kryo Serialization** : for outbound tuples has been moved out of Bolt/Spout emit() path. The serialization work is now handled by the Worker Transfer thread.

3.10) Reduce polling on the receive Queue in the Spout executor:

When ACKing is disabled, the spout rarely receives any incoming messages. So checking the recvQ for new messages after every nextTuple() slows things down. We can now control how often the spout should skip polling the recvQ using `topology.spout.recvq.skips` (defaults to 3). This means it will invoke nextTuple() three time before check the recvQ.

4. Things deliberately left out:

Below areas with performance issues were discovered, but in the interest of time I decided to exclude them from this work.

- **Outbound Path:** The path from the WorkerTransfer thread to the other worker process has not been tweaked and there exists much opportunity to tune this. For example, messaging within the Worker process is lock-free but there exists locks in the outbound path. There maybe room to potentially optimize Serialization as well.
- **ACKing path:** has a severe throughput bottleneck which appears to be purely an implementation issue and not a fundamental design issue. This needs to be addressed urgently. I can't imagine why this should have more than 20% degradation in throughput, at worst.
- **RateTracker:** There appears to be some bottleneck in RateTracker used for Metrics.
- **LoadAware messaging**: causes approx 20% degradation in throughput.
- **OutputCollectors:** Both `SpoutOutputCollectorImpl.sendSpoutMsg()` as well as `BoltOutputCollectorImpl.boltEmit()` have throughput bottlenecks. One of the issues was some runtime checks (to count fields in the tuples) being performed in the TupleImpl constructor. I have disabled this check in favor allowing the bolt logic to raise exceptions if they don't find the fields they are looking for. Another bottleneck in the emit path exists in `ExecutorTransfer.transfer()` call which happens immediately after instantiating `TupleImpl` in both the output collector impl code paths.