Backpressure model for Storm 2.0

STORM-2306

Roshan Naik
Dec 6th 2017

This new Backpressure model is designed with the following in mind:

- **Tightly Integrated:** Into the new messaging subsystem.
- **Bubble Up:** The back pressure should bubble up from downstream components to their upstream components. We do not want to apply the brakes immediately on all topology components as soon as a single bolt instance gets backlogged by setting a global 'throttle on/off' flag.
- **Work with or without ACKing enabled:**
- **No external dependencies:** On Zookeeper or other external services.
- **Light weight**
- **Prevent Deadlocked cycles:** Sometimes, it is possible that some components waiting on each other form a cycle and become deadlocked. The backpressure model and messaging subsystem are designed to prevent the formation of deadlocked cycles.

The fundamental role of the BackPressure model is to communicate backpressure from backlogged receivers to the senders so that the senders can be throttled. A sender's attempt to send messages to a backlogged task must be failed and it is required to retry until the situation changes. The retry semantics are the same whether the sending and receiving tasks are colocated in the same worker or spread out on different workers. The retry logic employs a configurable BackPressureWaitStrategy, which puts the tasks to sleep in between retry attempts to conserve CPU.

Below is a description of how the Backpressure situation is communicated and handed within a worker and across workers:

1. Backpressure within a worker:

The simplest form of Back pressure is between two (or more) communicating components running within the same worker. This situation is handled easily by using bounded receiveQs and eliminating the overflow list that currently found in the DisruptorQ's batcher. The sender must retry as long as the destination recvQ is full. This effectively throttles the upstream spout/bolt and makes the back pressure model simple and lightweight.

When there is backpressure from downstream component within a worker, there is a possibility of formation of deadlock cycles (when ACKing is enabled). As described in section

3.6 of the Messaging Redesign Document, the pendingEmitsQ prevents such deadlocks within a worker process.

2. Backpressure between workers:

The Netty communication between Storm workers is asynchronous for performance reasons. Messages are dispatched to other connected workers in a "fire and forget" fashion. The downstream worker also asynchronously informs the upstream workers of any changes in its back-pressure situation to temporarily pause delivery of additional messages to its backlogged task. This introduces a small time lag (approx ~1 or 2ms for two workers on same host) before the upstream worker receives the back-pressure notification and is able to stop sending additional messages. This is referred to as the **halt-lag.**

a) Throttling upstream tasks: Once the upstream worker receives a BackPressure status indicating that one or more tasks in the downstream worker are backlogged, it stops sending additional messages to such tasks until it receives a new BackPressure notification indicating a change in situation. It is important to note that the upstream worker can continue sending messages to other tasks that are not backlogged. Any attempts by an upstream task to send messages to a backlogged remote task will fail and it will have to keep retrying till the backpressure situation changes. In this fashion the backpressure situation is cascaded between tasks running on different workers.

b) The OverflowQ: During the brief halt-lag, the downstream worker continues to accept any additional messages for the such task and stashes in them into an *overflowQ* (which is integrated into the task's JCQueue). The overflowQ inside each recvQ is only used for storing overflow messages received from other workers. It is not used for communication between tasks within the same worker.

The overflowQ has two important functions. First, it allows us to retain asynchronous communication between workers by temporarily holding on to messages that cannot be added to a tasks's recvQ. In synchronous communication mode, we would not need this, as the receiver can refuse to accept the current batch of messages by sending back an error in the synchronous response.

Secondly, it helps prevent deadlocked wait cycles in multi-worker setups. By providing a additional space for incoming messages, it unblocks the receiving worker so that it can continue accept messages for tasks that are not backlogged. If the receiving worker was not able to accept messages to tasks that are not backlogged, all remote tasks sending messages to any task in this blocked worker would have to wait on that backlogged task. This could form cycles in directed-wait-graph. And these cycles in wait-graphs lead to deadlocks.

c) Preventing Out of Memory: Due to the fire-and-forget asynchronous netty communication, there is a minor possibility that the BackPressure status gets dropped in transit and does not reach the upstream worker. This can cause the upstream worker to bombard the backlogged task's unbounded overflowQ and potentially crash it's worker with an OutOfMemoryException. To prevent this, each time when the overflowQ size increases by N elements, the current BackPressure status is resent to upstream workers as a reminder. A backpresssure status is also sent every time a new connection is accepted from another worker.

The chances of unchecked growth of overflowQ leading to an out of memory situation is quite slim but for the rare possibilities, the configuration *topology.executor.overflow.limit* allows placing an upper limit on how large the overflowQ can grow. When this limit is reached, the worker starts dropping messages destined to such a task till the overflowQ size shrinks below the limit.

d) Ordering Guarantees: To ensure correct ordering semantics, as long as the overflowQ of the backlogged task is not empty, the worker continues to add new messages to the overflowQ. Once the task has caught up and fully drained its overflowQ, the worker can resume inserting messages into the primary recvQ.

Tasks alternate between consuming tuples from their main recvQ and their overflowQ. Both Qs are processed in order, but there are no ordering guarantees between the two queues. The guarantee that a task will process msgs strictly in the order it was emitted by any given upstream task, but does not guarantee any ordering between msgs from different upstream tasks.

e) Detecting and Communicating Change in BackPressure:
A timer thread (called backpressure-check-timer) in each worker periodically monitors local backlogged tasks to see if their overflowQs have been drained. If so, it sends a BackPressure status update to connected workers indicating the change, so that they can resume sending messages to them.