RM multi-thread event processing mechanism

1. Problem statement

We have a large yarn cluster in production, which uses YARN 2.9.2 with Capacity scheduler (multi-thread scheduling) and scales to several thousand nodes. Recently we have observed serious event blocking in RM event dispatcher queue (e.g. average length of events queue >= 50K within 20 minutes).

Table 1 shows the main events in the RMEventDispatcher queue at peek time,
Table 2 shows the average processing time and consuming rate of these events.

Table 1: RMEventDispatcher main pending events

Event type	Number of events	Proportion
RMAppAttemptEvent	93.2k	39.8%
RMAppEvent	46.5k	19.9%
RMNodeStatusEvent	42.7k	18.2%
Other RMNodeEvent	44.3k	18.9%
Other Event	7.3k	3.2%
Total	234k	100%

Table 2 : RMEventDispatcher main event processing time

Event type	Average Event processing time	Events consumed (every second)	Total processing time
RMAppAttemptEvent	5us	9.1k	45.5ms
RMAppEvent	9us	4.7k	42.3ms
RMNodeStatusEvent	176us	4.8k	844.8ms

The above statistics show that:
(1) RMNodeStatusEvent averaging processing time is much more than other events, which takes almost 85% time of 1 second;
(2) Although other event types have more pending events, they take much less processing time than RMNodeStatusEvent.

It means that RMNodeStatusEvent is a time-consuming event, which comes from node heartbeat report. We check the node heartbeat logic and find that node heartbeat report is triggered when any container in a node is completed. It results in an irregular heartbeat behavior and a large number of node heartbeat events. If we change this to regular heartbeat behavior, node heartbeat frequency will decrease, but cluster utilization also goes down. So we decide to enhance the ability of RM event processing.

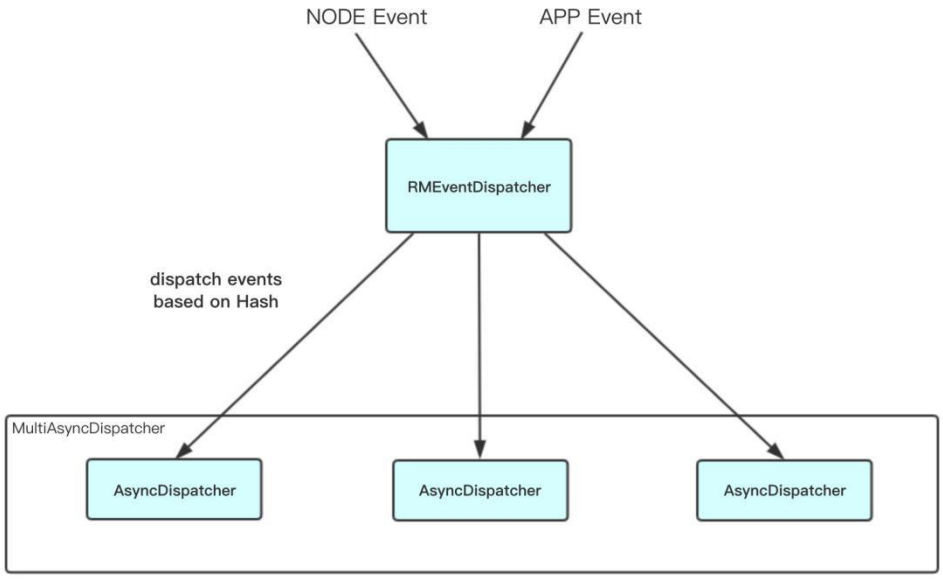
From above analysis, we propose RM multi-thread event-processing mechanism to speed up event processing without breaking state machine.

2. Proposed approach

Instead of using single event dispatcher to process events, we add a multiAsyncDispatcher. It has a configurable number of asyncDispatchers, which handles events like RMEventDispatcher.

See the picture below

- All Events first go into RMEventDispatcher queue. When handling events using specific event handler, we directly dispatch events to multiAsyncDispatcher.
- When multiAsyncDispatcher handle events, it calculates event hash based on event key(e.g. nodeId), then sends event to one asyncDispatcher using mod.



Correctness Consideration

The correctness of the above method is divided into two parts:

1) Sequence of event processing

Firstly, the above algorithm ensures that events of the same state machine are still in the same dispatcher like before. So there is no actual impact on the internal event processing order.

Secondly, consider the order of different state machine events, since they are submitted to RMEventDispatcher unorderly (e.g. nodes heartbeat report), it is also feasible for multiple dispatchers to randomly process different events concurrently.

2) Correctness of event processing

Since every state machine object is protected by inner lock, it is safe to operate concurrently.

Implementation

```
public class MultiAsyncDispatcher {
    private List<AsyncDispatcher> asyncDispatchers;

    public EventHandler getEventHandler(Object key) {
        int index = object.hashCode() % multiAsyncDispatcher.size();
```

```
            return asyncDispatchers.get(index).getEventHandler();
        }

        public void register(Class<? extends Enum> eventType, EventHandler handler) {
            for (AsyncDispatcher asyncDispatcher : asyncDispatchers) {
                asyncDispatcher.register(eventType, handler);
            }
        }
    }

    public class RMNodeImpl implements RMNode, EventHandler<RMNodeEvent> {
        public void handle(RMNodeEvent event) {
            try {
                writeLock.lock();
                ......
            }
            finally {
            writeLock.unlock();
            }
        }
    }
```