Kafka Controller Redesign

Motivation

Summary of existing controller

Current Kafka controller is a multi-threaded controller that emulates a state machine. It works in the following way.

Maintained state:

1. Replicas of partitions on each machine.
2. Leaders of partitions.

State change input source:

- Listeners Registered to Zookeeper.
    1. AddPartitionsListener
    2. BrokerChangeListener
    3. DeleteTopicListener
    4. PartitionReassignedListener(Admin)
    5. PreferredReplicaElectionListener(Admin)
    6. ReassignedPartitionsIsrChangeListener
    7. TopicChangeListener
- Channels to brokers (controlled shutdown)
- Internal scheduled tasks (preferred leader election)

State change execution:

- Listener threads, KafkaApi thread and internal scheduler thread makes state change concurrently.

State change propagation model:

- P2P blocking channel from controller to each broker.
- Dedicated message queue for each controller-to-broker connection.
- No synchronization on message sent to different brokers.
- No callback for sending messages except topic deletion.

Fail Over/Back model:

- Zookeeper based leader election
- Zookeeper as persistent state store for fault tolerance.

Problems of existing controller

1. State change are executed by different listeners concurrently. Hence complicated synchronization is needed which is error prone and difficult to debug.
2. State propagation is not synchronized. Brokers might be in different state for undetermined time. This leads to unnecessary extra data loss.
3. During controlled shutdown process, two connections are used for controller to broker communication. This makes state change propagation and controlled shutdown approval out of order.
4. Some of the state changes are complicated because the ReplicaStateMachine and PartitionStateMachine are separate but the state changes themselves are not. So the controller has to maintain some sort of ordering between the state changes executed by the individual state machines. In the new design, we will look into folding them into a single one.
5. Many state changes are executed for one partition after another. It would be much more efficient to do it in one request.
6. Some long running operations need to support cancellation. For example, we want to support cancellation of partition reassignment for those partitions whose reassignment hasn't started yet. The new controller should be able to abort/cancel the long running process.

New controller design

Outline

We will keep maintained state and fail over/back model unchanged.

We are going to change the state change propagation model, state change execution and output of the state change input source. More specifically:

1. Abstract the output of each state change input source to an **event**.
2. Have **a single execution thread** to **serially** process events one at a time.
3. The zk listeners are responsible for only context updating but not event execution.

4. Use o.a.k.clients.NetworClient + callback for state change propagation.

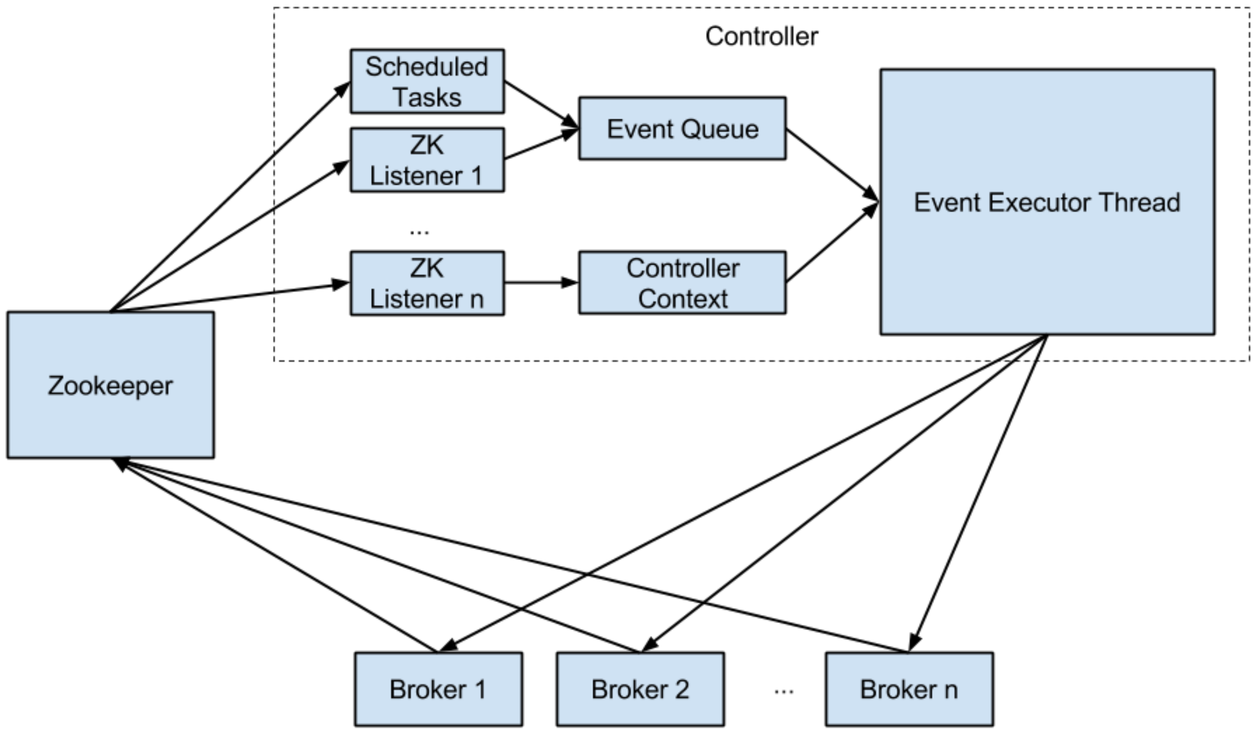
We would also like to

1. Modify KafkaServer to use new NetworkClient and prioritize the controller-to-broker traffic.
2. Change the reads/writes to Zookeeper to maybe use multi-operation or async operation.

Related tickets

KAFKA-2139, KAFKA-2029, KAFKA-1305 (and definitely some other tickets... Appreciate it if people can add it here.)

Architecture



- The Controller Context contains two kinds of information: **cluster reachability** and **Topic State**(Partition, Replica, Leaders, etc)
- Two types of ZK listeners:
  - Responsible of updating **cluster reachability** by listening to broker path in zookeeper.
  - Responsible for create **events** and add them to Event Queue.
- A controlled shutdown event will be generated when receive controlled shutdown from broker.
  - The controlled shutdown process will be changed to make state change and controlled shutdown approval occur in order. (This might involve broker side change as well)
- Scheduled tasks (e.g. preferred leader election) will
- On controller starting up or resignation, a ControllerStartUp/ControllerResignation event will be generated.
- Event Executor Thread:
  - Change **Topic State** in Controller Context
  - Propagate the new state to each broker using o.a.k.clients.NetworkClient in non-blocking way.
- Broker will only trigger Zookeeper data change when:
  1. Broker is down (or long GC)
  2. New topic automatically created

Event Types and Handling Process

Event Types

There will be eight types of events in the controller, which are defined as below. Each listener will generate one type of event, controlled shutdown is the eighth event type.

```
object ControllerEventType extends Enumeration {
  type ControllerEventType = Value
  val AddPartition, TopicChange, DeleteTopic, BrokerChange, PreferredReplicaElection, PartitionReassigned,
      ReassignedPartitionIsrChange, ControlledShutdown = Value
}
```

KafkaControllerEvent

A generic controller event class will be defined:

```
abstract class KafkaControllerEvent(eventType: ControllerEventType) {
  // A set that tracks the responses from brokers
  val unAckedNode = new mutable.HashSet[Int]
  val eventDone = new CountDownLatch(1)

  def makeStatesChange(currentState: PartitionStateMachine): Map[Int, ClientRequest]

  def controllerRequestCallback(response: ClientResponse) {
    handleBrokerResponse(response)
    unAckedNode.remove(response.request().request().destination())
    if (unAckedNode.isEmpty)
      eventDone.countDown()
  }

  def handleBrokerResponse(response: ClientResponse)
}
```

AddPartitionEvent

```
class AddPartitionEvent extends KafkaControllerEvent(ControllerEventType.AddPartition) {
        override def makeStatesChange(): Map[Int, ClientRequest] = {
                // make state change and generate requests to each broker
        }

        override def handleBrokerResponse(response: ClientResponse) {
                // If necessary, do something when response is received
          }
}
```

TopicChangeEvent

```
class TopicChangeEvent extends KafkaControllerEvent(ControllerEventType.TopicChange) {
        override def makeStatesChange(): Map[Int, ClientRequest] = {
                // make state change and generate requests to each broker
        }

        override def handleBrokerResponse(response: ClientResponse) {
                // If necessary, do something when response is received
          }
}
```

DeleteTopicEvent

```
class DeleteTopicEvent extends KafkaControllerEvent(ControllerEventType.DeleteTopic) {
        override def makeStatesChange(): Map[Int, ClientRequest] = {
                // make state change and generate requests to each broker
        }

        override def handleBrokerResponse(response: ClientResponse) {
                // If necessary, do something when response is received
          }
}
```

BrokerChangeEvent

```
class BrokerChangeEvent extends KafkaControllerEvent(ControllerEventType.BrokerChange) {
        override def makeStatesChange(): Map[Int, ClientRequest] = {
                // make state change and generate requests to each broker
        }

        override def handleBrokerResponse(response: ClientResponse) {
                // If necessary, do something when response is received
          }
}
```

PreferredReplicaElectionEvent

```
class PreferredReplicaElectionEvent extends KafkaControllerEvent(ControllerEventType.PreferredReplicaElection) {
        override def makeStatesChange(): Map[Int, ClientRequest] = {
                // make state change and generate requests to each broker
        }

        override def handleBrokerResponse(response: ClientResponse) {
                // If necessary, do something when response is received
          }
}
```

PartitionReassignedEvent

```
class PartitionReassignedEvent extends KafkaControllerEvent(ControllerEventType.PartitionReassigned) {
        override def makeStatesChange(): Map[Int, ClientRequest] = {
                // make state change and generate requests to each broker
        }

        override def handleBrokerResponse(response: ClientResponse) {
                // If necessary, do something when response is received
          }
}
```

ReassignedPartitionIsrChangeEvent

```
class ReassignedPartitionIsrChangeEvent extends KafkaControllerEvent(ControllerEventType.
ReassignedPartitionIsrChange) {
        override def makeStatesChange(): Map[Int, ClientRequest] = {
                // make state change and generate requests to each broker
        }

        override def handleBrokerResponse(response: ClientResponse) {
                // If necessary, do something when response is received
          }
}
```

ControlledShutdown

```
class ControlledShutdownEvent extends KafkaControllerEvent(ControllerEventType.ControlledShutdown) {
        override def makeStatesChange(): Map[Int, ClientRequest] = {
                // make state change and generate requests to each broker
        }

        override def handleBrokerResponse(response: ClientResponse) {
                // If necessary, do something when response is received
          }
}
```

Event Handling Process

The general event handling process would be something like this:

```
while(!shutdown){
        Event event = eventQueue.pollFirst()
        // Make state change
        try {
                val brokerRequests = event.makeStateChange(partitionStateMachine)
                   brokerRequests.map { case (broker, request) =>
                    networkClient.send(request)
                event.unAckedNode.add(broker)
            }
                while (!event.unAckedNode.isEmpty) {
                        try {
                                networkClient.poll(timeout)
                        } catch {
                                case KafkaApiException =>
                                        // Do something
                                case Exception =>
                                        // Error handling
                        }
                        checkNodeLivenessAndIgnoreDeadNode()
                }
        } catch {
                case StateChangeException =>
                        // handle illegal state change
        }
}
```

Long Running Process Handling

Currently replica reassigment is a really long running process. In new controller, we want to support abort/cancellation of the long running process when some event that affects the partition reassignment occurs after the long running process starts. In new controller, we will let later events **overrides** the state change of previous events based on the principle of **avoiding under replicated partition (URP)**.

For example, partition reassignment consists of several events:

1 PartitionReassignedEvent + N * ReassignedPartitionIsrChangeEvent.

Time	Broker 0	Broker 1	Broker 2	Event
0 - Initial State	{t1p1, replica:{0,1}, leader:0, isr:{0,1}}	{t1p1, replica:{0,1}, leader:0, isr:{0,1}}		
1 - Reassign t1p1 to broker 1 and broker 2				PartitionReassignedEvent
	{t1p1, replica:{0,1,2}, leader:1, isr:{0,1}}	{t1p1, replica:{0,1,2}, leader:1, isr:{0,1}}	{t1p1, replica:{0,1,2}, leader:1, isr:{0,1}}	
2.A - Broker 2 down (Scenario 1)				BrokerChangeEvent
	{t1p1, replica:{0,1,2}, leader:0, isr:{0,1}}	{t1p1, replica:{0,1,2}, leader:0, isr:{0,1}}		
2.B - Broker 1 down (Scenario 2)	{t1p1, replica:{0,1,2}, leader:0, isr:{0,1}}			
3 - Another partition reassignment				PartitionReassignedEvent

In scenario 2.A, we may choose to remove broker 2 from replica list or we can just stop listening to ISR change and let a later partition reassignment take care of this.

In scenario 2.B, we are better off to keep broker 2 in the replica list, and not remove broker 0 from isr after broker 2 enters isr so we do not expose to under replicated partition.

In step 3, when another partition reassignment comes, if the reassigned topic partition in step 1 and step 3 are the same, the second partition assignment will overrides the first one. It should:

1. clean up the listeners of the previous partition reassignment on the **overridden** partitions,
2. start the normal partition reassignment process.

Discussion Required

1. As stated in KAFKA-2029, current state change propagation has an issue that in an unbalanced cluster. State change on a heavily loaded broker will be much slower than a lightly loaded broker. This is because of the following two reasons:
    - Controller traffic is not prioritized on broker side. So controller messages needs to wait until some clients requests are handled which takes much longer on a heavily loaded broker.
    - Heavily loaded broker needs to take state change for more partitions while same amount state changes are distributed among several brokers as followers.

   Batching state change into a single request and prioritize the controller traffic on broker will solve the problem. But I wonder is there any specific reason we did not batch the state change in current controller?
2. Ideally, we should try to let the brokers in consistent state if possible. That indicates that we should put synchronization barriers between two events. Which means we do not execute the next event until:
    a. Callback has been fired on each broker, or
    b. A broker is down and we will just skip waiting for its callback and move on.

   Does it make sense to do so? Implementation wise, it means that if a broker is down, we will stop sending message to all brokers until we know it's down. (And this leads to question 3)
3. We actually have two ways to determine if a broker is alive or not: from Zookeeper or from NetworkClient. Currently we take zookeeper as source of truth, but when we send messages, NetworkClient connectivity is the one actually matters. Because Zookeeper timeout could take a while, so what should we do if Zookeeper says a broker is alive but NetworkClient shows it is disconnected. Should we block the current event processing? If there is a long GC, is it possible that NetworkClient says broker is connected but Zookeeper says it's dead? Back to question 2, it is about when we consider "a broker is down".