Kafka Controller Redesign

Purpose

A kafka cluster has one active controller. All brokers are capable of picking up the responsibility. The controller's purpose is to manage and coordinate the kafka cluster.

Protocols

- cluster metadata updates
- topic creation
- topic deletion
- partition reassignment
- preferred replica leader election
- topic partition expansion
- broker join
- broker failure
- controlled shutdown
- controller leader election

cluster metadata updates

A client can query any broker in the cluster about partition topology (the replica set and leader of the partition) through a `MetadataRequest`. This partition topology can change for many reasons. Producers and consumers always produce to and consume from the leader of the partition. The controller broadcasts the partition topology to every broker in the cluster through `UpdateMetadataRequests` so that brokers can accurately respond to clients.

topic creation

Topic creations can happen in one of three ways:

1. (deprecated) through direct client interaction with zookeeper by adding a `/brokers/topics/<topic>` znode.
2. through kafka with `CreateTopicsRequest`.
3. through kafka with `MetadataRequest` when `"auto.create.topics.enable"` is set to true.

In any case, the controller just watches for topic creations in zookeeper by watching for child changes to the `/brokers/topics` znode. The topic znode made from all of the 3 means of topic creation has already specified the number of partitions and the replica set for each replica. The controller reacts to topic creations by picking a leader from the given replica set, notifying the replicas of the topic creation, and updating the cluster's metadata. It also begins listening on partition changes to the topic by watching for data changes to the topic znode at `/brokers/topics/<topic>`.

topic deletion

Topic deletions can happen in one of two ways:

1. (deprecated) through direct client interaction with zookeeper by adding a `/admin/delete_topics/<topic>` znode.
2. through kafka with `DeleteTopicsRequest`.

In any case, the controller just watches for topic deletions in zookeeper by watching for child changes to the `/admin/delete_topics` znode. The znode itself has no data and the topic is extracted from the znode path. The controller reacts to topic deletions by driving two phases: make all replicas stop accepting requests, and make all replicas delete persisted data. Once the controller is notified that all replicas have successfully been deleted, the controller finishes the topic deletion protocol by removing the topic from zookeeper.

partition reassignment

Partition reassignments are made by an administrator. The administrator specifies the desired partition reassignment movements by writing to the `/admin/reassign_partitions` znode.

The controller just watches for partition reassignments in zookeeper by watching for data changes to the `/admin/reassign_partitions` znode. The znode can contain reassignments for multiple partitions. The controller actually initiates the partition reassignment movement for a given partition specified by the administrator if all of the criteria are met:

1. the reassigned partition's corresponding topic is not in the process of being deleted
2. the reassigned partition is not already in the process of being reassigned.
3. the replica set for each of the reassigned partitions differ from what's already there.

The actual partition reassignment process has the replica set of the partition go through an expansion and contraction phase and is well-documented in the code. The expansion phase expands the replica set to the (old replica set + new replica set), waits for them all to be in-sync with the leader, and transitions the leader to a replica in the new replica set if needed. The contraction phase removes the old replicas no longer in the new replica set. The controller detects that the (old replica set + new replica set) have become in-sync with the leader by temporarily watching for data changes to the `/brokers/topics/<topic>/partitions/<partition>/state` path for every partition being reassigned.

preferred replica leader election

Partition leadership can change when reassigning partitions or when the partition's leader fails. Over time, this can cause partition leadership imbalance in the cluster.

Preferred replica leader elections can happen in one of two ways:

1. automatically from the controller when `"auto.leader.rebalance.enable"` is set to true.
2. manually from direct administrator interaction with zookeeper by writing to the `/admin/preferred_replica_election` znode.

In either case, the controller watches for manually triggered preferred replica elections in zookeeper by watching for data changes to the `/admin/preferred_replica_election` znode. The `/admin/preferred_replica_election` znode contains a list of partitions to undergo preferred replica leader election. The controller does the partition leadership change based on the replica order defined in zookeeper. The controller transitions partition leadership to the first replica in the ordered replica set and updates the cluster's metadata if that replica is alive and is in-sync. The controller skips preferred replica leader elections for partitions whose topics are in the process of being deleted.

topic partition expansion

An administrator can expand a topic's partition count through direct interaction with zookeeper by writing the replica set for each partition to the `/brokers/topics/<topic>` znode just as with topic creation.

On topic creation, the controller watches for topic partition expansions in zookeeper by watching for data changes to the `/brokers/topics/<topic>` znode. The topic znode has already specified the number of partitions and the replica set for each replica. The controller reacts to the topic partition expansion by picking a leader from the given replica set, notifying the replicas of the partition creation, and updating the cluster's metadata.

broker join

The controller watches for broker joins in zookeeper by watching for child changes to the `/brokers/ids` znode. When a broker joins the cluster, the controller updates the cluster's metadata, informs the broker of the partitions to serve, resumes any partition reassignments that were suspended from the down broker, and tries to elect the joined broker as leader for partitions that have previously been offline.

broker failure

The controller watches for broker failures in zookeeper by watching for child changes to the `/brokers/ids` znode. When a broker fails, the controller informs the impacted replicas of the failure, picks new leaders for the impacted partitions, and updates the cluster's metadata.

controlled shutdown

A broker may gracefully shutdown through a process called controlled shutdown. This is to reduce the unavailability window on partitions owned by the shutting down broker compared to broker failures being detected from zookeeper session expirations. The shutting down broker informs the controller its intent to shutdown with a `ControlledShutdownRequest`. The broker's shutdown is blocked until it either:

1. receives a `ControlledShutdownResponse` from the controller indicating success
2. exhausts all of its retries.

The controller sends a `ControlledShutdownResponse` after any needed leadership movements have occurred and brokers on relevant replica sets have been notified of potential in-sync-replica shrinks. Note that this protocol is different from all the others in that an administrative operation is done through rpc from broker to controller instead of by triggering the controller through zookeeper. Background regarding this exception to the pattern is in KAFKA-340, KAFKA-817, and KAFKA-927.

controller leader election

Controller leader election is kafka's way of ensuring the kafka cluster has one active controller even when a controller fails or experiences a controlled shutdown. A kafka cluster can undergo controller leader election in one of four ways:

1. through direct administrative interaction with zookeeper by deleting the `/controller` znode.
2. through direct administrative interaction with zookeeper by writing a broker id to the `/controller` znode.
3. through controller broker failure.
4. through controlled shutdown of the controller.

In any case, the controller just watches for the need to undergo controller leader election in zookeeper by watching for data changes to the `/controller` znode. This ephemeral `/controller` znode just specifies the broker id of the current active controller. A leader is decided based on successful ownership of the ephemeral `/controller` znode in zookeeper. Upon becoming controller, the new controller increments the controller epoch in the `/controller_epoch` znode, registers zookeeper listeners, loads zookeeper metadata into its local state, proceeds with any in-progress partition reassignments and preferred replica leader elections, and updates the cluster's metadata. Upon controller resignation, the former controller deregisters its zookeeper listeners and clears its local controller state.

Current Design

`KafkaController` maintains a connection to every broker in the cluster. Each of these connections are supported by a `NetworkClient` running on a separate request send thread.

With one exception, all communication between the controller and a broker is from controller to broker. This includes `UpdateMetadataRequest`, `LeaderAndIsrRequest`, and `StopReplicaRequest`. The exception is the `ControlledShutdownRequest` sent from the shutting down broker to the controller.

With one exception, all administrative operation requests are communicated to the controller through zookeeper. Again the exception is the `ControlledShutdownRequest` sent from the shutting down broker to the controller.

Components

`KafkaController` today is made up of the following components:

- `ZkClient`
- `ControllerContext`

- `ControllerChannelManager`
- `ControllerBrokerRequestBatch`
- `RequestSendThread`
- `PartitionStateMachine`
- `ReplicaStateMachine`
- `ZookeeperLeaderElector`
- `TopicDeletionManager`
- `OfflinePartitionLeaderSelector`
- `ReassignedPartitionLeaderSelector`
- `PreferredReplicaPartitionLeaderSelector`
- `ControlledShutdownLeaderSelector`
- `PartitionsReassignedListener`
- `PreferredReplicaElectionListener`
- `IsrChangeNotificationListener`

ZkClient

Kafka brokers depend on org.I0Itec.zkclient.ZkClient. This client is an abstraction over the vanilla org.apache.zookeeper.ZooKeeper client in that it:

- offers permanent watches on nodes in zookeeper
- allows users to subscribe and unsubscribe listeners to data changes, child changes, and state changes with IZkDataListener, IZkChildListener, and IZkStateListener, respectively.
- automatically tears down and reinitializes the underlying ZooKeeper client and additionally re-establishes new sessions to zookeeper upon session expiration.

ZkClient has a single org.I0Itec.zkclient.ZkEventThread from which it triggers all notifications sequentially.

ControllerContext

This stores the controller's cluster state. It contains state like the topics in the cluster, which brokers are in the cluster, partition topology, partitions undergoing reassignment, and partitions undergoing preferred replica leader elections. This state is shared across threads and is protected by a `controllerLock`. The context also contains the `ControllerChannelManager`.

ControllerChannelManager

This component maintains a connection to every broker in the cluster. It holds a `NetworkClient` and message queue per broker connection. Each of these connections has a dedicated `RequestSendThread` operating the `NetworkClient` and message queue.

ControllerBrokerRequestBatch

This component batches up partially-built controller-to-broker `LeaderAndIsrRequests`, `StopReplicaRequests`, and `UpdateMetadataRequests`. Users update these batches at a per-request level stating additional partitions to add for certain brokers. Once ready, the batch gets turned into actual requests and get queued up using the `ControllerChannelManager`. `KafkaController`, `PartitionStateMachine`, and `ReplicaStateMachine` each maintain their own `ControllerBrokerRequestBatch`.

RequestSendThread

This component sends queued up messages to a broker in the cluster by operating its own `NetworkClient`. There are one of these for each controller-to-broker connection.

PartitionStateMachine

This component stores the state for every partition in the kafka cluster. Partition states include `NonExistentPartition`, `NewPartition`, `OnlinePartition`, and `OfflinePartition`. Most state transitions simply update the partition state in the `PartitionStateMachine's` local state. The exceptions are state transitions to the `OnlinePartition` state. This transition figures out the leader and in-sync-replica set for the partition, updates zookeeper, and updates the cluster's metadata.

valid states:

- `NonExistentPartition` - the partition either never existed before or was created and deleted.
- `NewPartition` - the partition has been created. It has a replica set, but the leader and isr have not yet been decided.
- `OnlinePartition` - the partition has a leader.
- `OfflinePartition` - the leader for a partition has died.

valid state transitions:

- `NonExistentPartition` -> `NewPartition`
- `NewPartition` -> `OnlinePartition`, `OfflinePartition`
- `OnlinePartition` -> `OnlinePartition`, `OfflinePartition`
- `OfflinePartition` -> `OnlinePartition`, `OfflinePartition`, `NonExistentPartition`

ReplicaStateMachine

This component stores the state for every replica in the kafka cluster. Replica states include `NonExistentReplica`, `NewReplica`, `OnlineReplica`, `OfflineReplica`, `ReplicaDeletionStarted`, `ReplicaDeletionSuccessful`, and

`ReplicaDeletionIneligible`. Most state transitions simply update the replica state in the `ReplicaStateMachine's` local state. The exceptions are `NewReplica` and `OfflineReplica` which interact with zookeeper to read or update the in-sync-replica set in zookeeper.
valid states:

- `NonExistentReplica` - the replica either never existed before or was deleted successfully.
- `NewReplica` - the replica is brand new either as a result of partition reassignment or topic creation. The replica can be a follower in this state.
- `OnlineReplica` - the replica has started and is eligible for becoming either a leader or follower for a partition.
- `OfflineReplica` - the replica has either gracefully been shutdown from controlled shutdown, failed, or has been kicked out of the replica set due to partition reassignment.
- `ReplicaDeletionStarted` - the replica has been instructed to begin deletion either from having been kicked out of the replica set due to partition reassignment.
- `ReplicaDeletionSuccessful` - the replica has been successfully deleted either from having been kicked out of the replica set due to partition reassignment.
- `ReplicaDeletionIneligible` - the replica to be deleted is either down or the `StopReplicaRequest` instructing deletion returned with an error.

valid state transitions:

- `NonExistentReplica` -> `NewReplica`
- `NewReplica` -> `OnlineReplica`, `OfflineReplica`
- `OnlineReplica` -> `OnlineReplica`, `OfflineReplica`
- `OfflineReplica` -> `OnlineReplica`, `OfflineReplica`, `ReplicaDeletionStarted`
- `ReplicaDeletionStarted` -> `ReplicaDeletionSuccessful`, `ReplicaDeletionIneligible`
- `ReplicaDeletionSuccessful` -> `NonExistentReplica`
- `ReplicaDeletionIneligible` -> `OnlineReplica`, `OfflineReplica`

ZookeeperLeaderElector

This component makes sure a kafka cluster has one active controller. It watches for data changes in the `/controller` znode and runs the controller election algorithm upon data change.

TopicDeletionManager

This component has a single `kafka.controller.DeleteTopicsThread` which in a loop tracks the progress of topic deletion through its two phases of stopping all replicas and deleting all replicas.

OfflinePartitionLeaderSelector

This component purely decides the leader for a newly created partition. It doesn't change any state.

ReassignedPartitionLeaderSelector

This component purely decides the leader for a reassigned partition. It doesn't change any state.

PreferredReplicaPartitionLeaderSelector

This component purely decides the leader for a partition undergoing preferred replica leader election. It doesn't change any state.

ControlledShutdownLeaderSelector

This component purely decides the leader for a partition such that it doesn't pick the broker undergoing controlled shutdown. It doesn't change any state.

PartitionsReassignedListener

This component contains the logic triggered by `ZkClient` when an administrator writes to the `/admin/reassign_partitions` znode.

PreferredReplicaElectionListener

This component contains the logic triggered by `ZkClient` when an administrator writes to the `/admin/preferred_replica_election` znode.

IsrChangeNotificationListener

Brokers keep track of all in-sync-replica set changes for partitions of which it is the leader. Brokers periodically write any changes observed as a sequential znode child to the `/isr_change_notification` znode. This component contains the logic triggered by `ZkClient` upon child change when a broker writes a child to `/isr_change_notification`. The controller simply updates its local cache of this change and updates the cluster's metadata.

Redesign Goal

The goal of this redesign is to improve controller performance, controller maintainability, and cluster reliability.

Problems with the Controller

1. synchronous per-partition zookeeper writes
2. sequential per-partition controller-to-broker requests
3. complicated concurrency semantics
4. poor controller code organization
5. no separation of control plane from data plane
6. controller-to-broker requests are not broker-generation-aware
7. ZkClient obstructs client state management

synchronous per-partition zookeeper writes

Synchronous zookeeper writes means that we wait an entire round trip before doing the next write. These synchronous writes are happening at a per-partition granularity in several places, so partition-heavy clusters suffer from the controller doing many sequential round trips to zookeeper.

- `PartitionStateMachine.electLeaderForPartition` updates `leaderAndIsr` in zookeeper on transition to `OnlinePartition`. This gets triggered per-partition sequentially with synchronous writes during controlled shutdown of the shutting down broker's replicas for which it is the leader.
- `ReplicaStateMachine` updates `leaderAndIsr` in zookeeper on transition to `OfflineReplica` when calling `KafkaController.removeReplicaFromIsr`. This gets triggered per-partition sequentially with synchronous writes for failed or controlled shutdown brokers.

sequential per-partition controller-to-broker requests

TODO: The impact of sequential requests needs to be further examined, as they are sent out by a separate RequestSenderThread and not in the way of a controlled shutdown.

It appears that the controller performs sequential per-partition requests for all three forms of controller-to-broker requests:
`StopReplicaRequest` gets sent per-partition sequentially to a shutting down broker even though `StopReplicaRequest` already accepts multiple partitions. On top of this, if you observe the request logs, you'll find that a broker undergoing controlled shutdown receives two

`StopReplicaRequests` for every partition that should be stopped. This magnifies the per-partition sequential request issue.

`LeaderAndIsrRequest` gets sent per-partition sequentially to all replicas of partitions that a controlled shutdown broker formerly led to notify the replica set of the newly elected leader. Requests also get sent per-partition sequentially to all replicas of partitions that a controlled shutdown broker formerly followed to notify the replica set of a potential change in the partition's in-sync replicas.

Any time a `LeaderAndIsrRequest` gets sent, a corresponding `UpdateMetadataRequest` is sent to the entire cluster. When considering the controlled shutdowns described above, we see that `UpdateMetadataRequest` is also sent out on a per-partition basis but this time to the entire cluster.

These sequential per-partition controller-to-broker requests magnify per-request overheads. For instance, on every per-partition UpdateMetadataRequest, the MetadataCache acquires a read-write lock and updates its aliveBrokers and aliveNodes even though they'll most likely be the exact same value throughout the controlled shutdown.

complicated concurrency semantics

Today `KafkaController` shares state across many threads. Threads that the controller needs to worry about are:

- IO threads handling controlled shutdown requests
- The `ZkClient` `org.I0Itec.zkclient.ZkEventThread` processing zookeeper callbacks sequentially
- The `TopicDeletionManager` `kafka.controller.DeleteTopicsThread`
- Per-broker `RequestSendThread` within `ControllerChannelManager`.

All of these threads with the exception of the per-broker `RequestSendThread` are accessing and modifying the `ControllerContext` state within the `ControllerContext.controllerLock`, so little parallelism is taking place anyway.

poor controller code organization

Logic and state is split poorly across `KafkaController`, `PartitionStateMachine`, and `ReplicaStateMachine`. Why this matters:

- It's difficult to answer questions like:
    - "where and when does zookeeper get updated?"
    - "where and when does a controller-to-broker request get formed?"
    - "what impact does a failing zookeeper update or controller-to-broker request have on the cluster state?"
- stunts development. open source is reluctant to make changes to the controller out of fear of breaking something.

- fallacies emerge like "it must not be broken since nobody is making changes to it".

Example pain points:

- scala setter overrides are very misleading and should be avoided. This is only used once in `ControllerContext` in a simple way but should still be removed.
- `KafkaController`, `PartitionStateMachine`, and `ReplicaStateMachine` each have their own `ControllerBrokerRequestBatch`. This prevents you from being able to batch up requests across these classes.
- There are a number of needless back-and-forth code flows between classes. The biggest offender is between `KafkaController` and `ControllerChannelManager` where `KafkaController` calls `ControllerChannelManager`'s `sendRequestsToBrokers` which calls `KafkaController`'s `sendRequest` which calls `ControllerChannelManager`'s `sendRequest`. Similar back-and-forth code flows happen upon broker failure between `KafkaController` and `ReplicaStateMachine`.

no separation of control plane from data plane

Today all requests (client requests, broker requests, controller requests) to a broker are put into the same queue. They all have the same priority. So a backlog in client requests will postpone the processing of requests from the controller.

This can have undesirable consequences. Imagine for instance the controller broadcasts to a replica set that leadership of the replica set has changed. The new leader starts accepting client requests. Meanwhile, the former replica set leader is busy processing a backlog of client requests before processing the controller's LeaderAndIsrRequest informing it of the leadership transfer. Some of the requests in the backlog may pertain to the partition undergoing leadership transfer. Specifically, messages from produce requests without stronger acknowledgment settings can get erased from log truncation (technically not categorized as data loss since these messages are beyond the "high watermark", but it's a bad, unexpected result nonetheless for the producer).

controller-to-broker requests are not broker-generation-aware

Broker generation here means an identifier of the broker that changes every time it joins the cluster. Controller-to-broker requests are not broker-generation-aware, meaning it's possible for a restarted broker to accidentally receive and act on requests intended for the broker's earlier generation, leaving the broker in a bad state.

For instance, if a broker restarts in the middle of its own controlled shutdown, the restarted broker may accidentally process its earlier generation's StopReplicaRequest sent from the controller for one of its follower replicas, leaving the replica offline while its remaining replicas may stay online.

ZkClient obstructs client state management

Client state management here means the ability to intervene when a state change on the client occurs such as connection loss or session expiration from zookeeper.

ZkClient re-establishes new sessions under the hood and processes all notifications (including state change notifications) sequentially from the ZkEventThread. This means that even if you subscribe an IZkStateListener to the ZkClient, they will get processed only after processing pending notifications in front of the state change notification in the ZkEventThread's queue. So, by the time the ZkEventThread starts processing the state change notification, a new session may have been established and writes intended to be on the old session have already happened on the new session. Without a means of intervention as the state changes occur, we are susceptible to cluster inconsistencies such as the one mentioned in KAFKA-3083.

Proposed Controller Improvements

1. use async zookeeper apis everywhere
2. improve controller-to-broker request batching
3. single-threaded event queue model
4. refactor cluster state management
5. prioritize controller requests
6. make controller-to-brokers requests broker-generation-aware
7. use vanilla ZooKeeper client for better client state management

use async zookeeper apis everywhere

The zookeeper client offers various means of performing a request: synchronous, asynchronous, and multi:

- synchronous calls mean we wait for one request to complete before starting another request.
- asynchronous calls means we don't have to wait for one request to complete before starting another request, and we get notified of the result through a callback.
- multi batches multiple requests into a single request over-the-wire, and the whole batch of requests get lumped under the same transaction. Something to keep in mind with multi is the maximum allowed request size limitation in zookeeper represented by `jute.maxbuffer`. Extra care needs to be taken to choose the right amount of batching in the multi request to make sure it doesn't exceed the maximum request size limit. This can be mitigated by batching up the multi operations.

[4] indicates that multi and batched multi provides the best performance, followed by async, and sync being the worst. So moving forward, the question is whether we should go down the async path or batched multi path. For this, we need to weigh pros and cons.

	sync	async	multi	batched multi
sync or async	sync	async	sync	sync
granularity	Per-request	Per-request	Per-batch. Batch has many sub-requests handled under one transaction.	Per-batch. Batch has many sub-requests handled under one transaction.
error mechanism	Exception	Callback error code	Exception specifying the specific sub-request causing the problem.	Exception specifying the specific sub-request causing the problem.
limitations	Possible to hit the `jute.maxbuffer` 1MB limit	Possible to hit the `jute.maxbuffer` 1MB limit	Easy to hit the `jute.maxbuffer` 1MB limit	Possible to hit the `jute.maxbuffer` 1MB limit
performance	Bad for many writes.	Good for pipelining many writes.	Optimal for many writes. Avoids request overhead in async.	Good compromise between async and multi while avoiding the multi's 1MB limitation.
zkclient ready	Yes	No: Async apis are offered in the raw zookeeper client. Some options are to switch to just using the raw zookeeper client or to extend ZkClient to reuse its existing raw zookeeper client.	Yes	Yes

Reaction to errors	Error is per-request, so you can independently handle the individual failure.	Error is per-request, so you can independently handle the individual failure.	Either the whole transaction succeeds or fails, so one failure causes all requests in the transaction to fail. Retries are costly.	Either the whole transaction succeeds or fails, so one failure causes all requests in the transaction to fail. Retries are costly.

Flavio's opinion: "You don't really need to batch with multi, you just need to make the calls asynchronous. In fact, unless you really need to make multiple updates transactional, the preferred way is to push updates asynchronously to keep the pipeline full." [5] It seems that he wants ZkUtils to completely migrate to using async calls with the raw zookeeper client.

The async apis look like the winner primarily due to the simplicity in handling per-request granularity successes and failures while still having substantially better write performance than sync.

We can safely run the async apis by waiting for the set of async calls to finish before moving on. The end result of pipelining async calls is that latency for N writes ends up being much less than N round trips. A more detailed analysis can be found in Appendix A.

improve controller-to-broker request batching

TODO: The impact of these sequential requests needs to be further examined, as they are sent out by a separate RequestSenderThread and not in the way of a controlled shutdown.

single-threaded event queue model

Switching to a single-threaded event queue model vastly simplifies the concurrency semantics. This single thread is the only thread accessing and modifying the controller local state, so we no longer have to pass a lock around different threads and classes.

Note that this is less drastic of a change than it sounds. The bulk of the existing controller work today already happens sequentially in the ZkClient's single `org.I0Itec.zkclient.ZkEventThread`. The per-broker `RequestSendThread` already just blindly sends requests that were prepared by the `org.I0Itec.zkclient.ZkEventThread` . So really all that's needed is to shift the work done by `org.I0Itec.zkclient.ZkEventThread`, the work done by the IO threads upon controlled shutdown, and the work done by the `kafka.controller.DeleteTopicsThread` into the same thread.

Since we can't elegantly push arbitrary work onto the `org.I0Itec.zkclient.ZkEventThread`, we can add a layer of indirection. We add a new ControllerThread which processes events held in an event queue. All work now gets delegated to this single thread.

Notifications processed by the `org.I0Itec.zkclient.ZkEventThread` can be processed by this single-threaded event queue model by transforming every notification into an event. The processing of the event will now perform all work previously done within the `org.I0Itec.zkclient.ZkEventThread`.

Controlled shutdowns can work their way into the single-threaded event queue model by just adding a controlled shutdown event to the event queue. Existing behavior has the IO thread wait indefinitely for completion. Some options for mimicking this behavior is to share a synchronization mechanism between the IO thread and the ControllerThread such as a single-element blocking queue or alternatively just put the request into a purgatory, but these possibilities are orthogonal to the single-threaded event queue model discussion.

Callbacks from the `RequestSendThread` can fit into this single-threaded event queue model by delegating the actual callback logic into an event which the ControllerThread can later process.

Topic deletion can get folded into the single-threaded event queue model in several ways. Topic deletion progress can pause due to down brokers or arbitrary exceptions in its two phases. If broker membership was the only concern, then the existing dedicated DeleteTopicsThread would be unnecessary - simply pause and resume based on broker joins, failures, or shutdowns. However, topic deletion can pause for other reasons such as error codes coming from responses during its two phases. So we need some mechanism to eventually retry deletion of the offending replicas until success. While technically possible to piggyback this retry logic into the event processing of other events, progress on retrying topic deletion would depend on frequency of events making it into the event queue. A simpler approach would be to schedule a repeated task that appends an event into the event queue telling the ControllerThread to retry pending topic deletions. This proposal prefers the latter approach.

This single-threaded event queue approach has the added benefit of having short ZkClient callbacks and should reduce the impact of a longstanding issue described in KAFKA-1155.

refactor cluster state management

One option to reduce the complexity of figuring out which cluster resides where and when they get modified is to get rid of the state machines altogether and to just define the actions to be taken when handling a controller event.
Pros:

- local state and state manipulations are easier to follow: just read the function handling the controller event.
- it's a lower-level organization than the state machines, so you get better control over what happens when.

Cons:

- state transitions are now implicit
- It's easier now to miss edge case logic

Another option is to do a hybrid approach of the existing state machines and the option described above. This would end up looking like the GroupCoordinator, where states and state transitions are made explicit, but there is no explicit state machine class internally performing the state manipulation logic. An external class (in this case, the controller) would hold all of the cluster state, decide when a transition should occur, and define the state manipulation logic.

Pros:

- state transitions are explicit
- local state and state manipulations are easier to follow: just read the function handling the controller event.
- it's a lower-level organization than the state machines, so you get better control over what happens when.
- explicit state transitions should make it easier to notice edge cases

prioritize controller requests

We want to separate the control plane from the data plane. To do this, we want to prioritize controller requests. This allows brokers to react more proactively to controller requests when faced with a backlog of requests. Clients will also appreciate controller request prioritization because their requests will behave as they expect even when the broker is under stress.

Request prioritization can happen at the network layer with the `RequestChannel`. The `RequestChannel` can categorize the request as regular or prioritized based on the request id. If the incoming request id matches that of `UpdateMetadataRequest`, `LeaderAndIsrRequest`, and `StopReplicaRequest`, the request can get prioritized.

There are several ways to implement request prioritization:

1. Add a prioritized request queue to supplement the existing request queue in the `RequestChannel` and add request prioritization-aware logic to both the `sendRequest` and `receiveRequest` operations of `RequestChannel`. `sendRequest` puts the request into the respective queue based on whether the request is prioritized or not. `receiveRequest` can optimistically check the prioritized request queue and otherwise fallback to the regular request queue. One subtlety here is whether to do a timed poll on just the regular request queue or on both the prioritized request queue and regular request queue sequentially. Only applying the timed poll to the regular request queue punishes a prioritized request that arrives before a regular request but moments after the

prioritized request check. Applying the timed poll to both queues sequentially results in a guaranteed latency increase on a regular request.
2. Replace `RequestChannel`'s existing request queue with a prioritization-aware blocking queue. This approach avoids the earlier stated subtlety by allowing the timed poll to apply to either prioritized or regular requests in low-throughput scenarios while still allowing queued prioritized requests to go ahead of queued regular requests.

This document prefers the second implementation as it avoids the earlier stated subtle issue of punishing late arriving prioritized requests.

This has been broken out into KAFKA-4453 and already has a pending patch.

make controller-to-brokers requests broker-generation-aware

Broker generation here means an identifier of the broker that changes every time it joins the cluster. All controller-to-broker requests should include the broker generation. If the recipient broker notices the request's generation doesn't match its own generation, it rejects the request.

Some options for the generation are:

- a guid generated by the broker that gets propagated to the controller
- the czxid from the broker's zookeeper ephemeral node

Using the czxid is a natural fit since it's a unique, monotonically increasing identifier of the broker that changes every time it joins the cluster and the controller anyways reads the relevant /brokers/ids/<id> znode upon broker join to discover the broker's rack and endpoint information.

use vanilla ZooKeeper client for better client state management

Client state management here means the ability to intervene when a state change on the client occurs such as connection loss or session expiration from zookeeper.

Unlike ZkClient, synchronous calls will actually bubble up ConnectionLossException and SessionExpiredException to the user upon a read or write attempt to zookeeper, allowing us to intervene if the client either hits connection loss or session expiration.

But as stated earlier, we want to use the vanilla ZooKeeper client for its async apis. We can still get notified of request errors from state changes such as connection loss or session expiration from the return codes passed to callbacks, and similarly we can react to state changes upon processing an event by looking at the event's reported state.

Using the vanilla ZooKeeper clients lets us act on these state changes when the state notification is received instead of when ZkClient's ZkEventThread has finally reached the state change notification.

Before diving into what intervention should occur during a state change, the following is a quick overview of relevant session states and state transitions.
valid states:

- NOT_CONNECTED - the initial state of the client
- CONNECTING - the client is establishing a connection to zookeeper
- CONNECTED - the client has established a connection and session to zookeeper
- CLOSED - the session has closed or expired

valid state transitions:

- NOT_CONNECTED -> CONNECTING
- CONNECTING -> CONNECTED
- CONNECTING -> CLOSED
- CONNECTED -> CONNECTING
- CONNECTED -> CLOSED

While a client can locally make the decisions that it has lost a connection to zookeeper as well as locally decide that it wants to close its session, only the zookeeper ensemble can decide that the client's session has expired.

When a client receives a notification of connection loss (client is in the CONNECTING state), it means the client cannot get any notifications from zookeeper. When disconnected, the controller should simply pause whatever tasks it was working on since another broker could have taken over as controller without its knowing. From the disconnected state, the client can either re-establish a connection (state transition to CONNECTED) or have its session expire (state transition to CLOSED). When transitioning to the CONNECTED state, the controller should resume its tasks. However, when transitioning to the CLOSED state, the broker knows it is no longer the controller and should discard any of its pending tasks.

Appendix A: Zookeeper Performance

Environment

The following experiments are run against a 5-machine zookeeper ensemble using a single org.apache.zookeeper.ZooKeeper client run in the same datacenter as the zookeeper ensemble. Each experiment is run 5 times.

Experiments

- set-data-sync: synchronously write the same random sequence of bytes to N znodes
- get-data-sync: synchronously read N znodes
- set-data-async: asynchronously write the same random sequence of bytes to N znodes
- get-data-async: asynchronously read N znodes

- multi-set-data-sync: write atomic batches (of size B) of N znodes synchronously
- multi-check-and-set-data-async: atomically check a znode and write a batch (of size B) of N znodes asynchronously

Experiment results applied to 500,000 znodes

	Run 1 (ms)	Run 2 (ms)	Run 3 (ms)	Run 4 (ms)	Run 5 (ms)	Avg (ms)
set-data-sync	247435	251086	254658	258412	255292	253376.6
get-data-sync	63938	64227	64654	61650	66290	64151.8
multi-set-data-sync (batch size 10)	34161	35084	27158	35015	35285	33340.6
multi-set-data-sync (batch size 50)	17210	17553	15481	17702	17686	17126.4
multi-set-data-sync (batch size 100)	12148	11565	11210	10197	11421	11308.2
multi-set-data-sync (batch size 500)	7541	7844	7551	7721	7165	7564.4
multi-set-data-sync (batch size 1000)	6971	6702	7049	7063	6758	6908.6
multi-set-data-sync (batch size 5000)	5576	5220	5538	5255	5571	5432.0
multi-set-data-sync	4943	5268	5683	4882	5078	5170.8

(batch size 10000)						
multi-set-data-sync (batch size 50000)	ConnectionLoss(jute.maxbuffer)	ConnectionLoss(jute.maxbuffer)	ConnectionLoss(jute.maxbuffer)	ConnectionLoss(jute.maxbuffer)	ConnectionLoss(jute.maxbuffer)	n/a
set-data-async	19698	16156	17196	20527	20226	18760.6
get-data-async	8067	14025	7097	6734	15600	10304.6
multi-check-and-set-data-async (batch size 1)	OOM	OOM	OOM	OOM	OOM	n/a
multi-check-and-set-data-async (batch size 10)	4438	4009	4384	4141	4382	4270.8
multi-check-and-set-data-async (batch size 50)	3708	4252	4191	4136	4324	4122.2
multi-check-and-set-data-async (batch size 100)	3969	3878	3727	3757	3790	3824.2
multi-check-and-set-data-async (batch size 500)	3789	3834	3774	3800	3779	3795.2
multi-check-and-set-data-async (batch size	3765	3801	3730	3757	3812	3773.0

1000)						
multi-check-and-set-data-async (batch size 5000)	3238	3125	3004	3283	3025	3135.0
multi-check-and-set-data-async (batch size 10000)	3334	3788	3294	2907	3220	3308.6
multi-check-and-set-data-async (batch size 50000)	ConnectionLoss(jute.maxbuffer)	ConnectionLoss(jute.maxbuffer)	ConnectionLoss(jute.maxbuffer)	ConnectionLoss(jute.maxbuffer)	ConnectionLoss(jute.maxbuffer)	n/a

Experiment results applied to 1,000,000 znodes

	Run 1 (ms)	Run 2 (ms)	Run 3 (ms)	Run 4 (ms)	Run 5 (ms)	Avg (ms)
set-data-sync	514700	501081	513054	515765	528159	514551.8
get-data-sync	127084	133080	134900	129372	124262	129739.6
multi-set-data-sync (batch size 10)	69061	68500	68382	57656	69768	66673.4
multi-set-data-sync (batch size 50)	34599	33808	34949	34264	35189	34561.8
multi-set-data-sync (batch size 100)	23029	20220	22429	21936	22876	22098.0
multi-set-d	16007	14660	15107	14464	14528	14953.2

ata-sync (batch size 500)						
multi-set-data-sync (batch size 1000)	14112	13168	14055	12970	12461	13353.2
multi-set-data-sync (batch size 5000)	10764	10455	10800	10487	10258	10552.8
multi-set-data-sync (batch size 10000)	10046	9800	9571	11504	9486	10081.4
multi-set-data-sync (batch size 50000)	ConnectionLoss(jute.maxbuffer)	ConnectionLoss(jute.maxbuffer)	ConnectionLoss(jute.maxbuffer)	ConnectionLoss(jute.maxbuffer)	ConnectionLoss(jute.maxbuffer)	n/a
set-data-async	OOM	OOM	OOM	OOM	OOM	n/a
get-data-async	OOM	OOM	OOM	OOM	OOM	n/a
multi-check-and-set-data-async (batch size 10)	OOM	OOM	OOM	OOM	OOM	n/a

References

[1] https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Controller+Internals
[2] https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Controller+Redesign
[3] https://github.com/apache/kafka/pull/1149
[4] http://zookeeper-user.578899.n2.nabble.com/sync-vs-async-vs-multi-performances-td7284355.html

[5]
https://issues.apache.org/jira/browse/KAFKA-3038?focusedCommentId=15085300&page=com.atlassian.jira.plugin.system.issuetabpanels:comment-tabpanel#comment-15085300