Kafka Detailed Consumer Coordinator Design

WARN: This is an obsolete design. The design that's implemented in Kafka 0.9.0 is described in this wiki.

The following is a draft design that uses a high-available consumer coordinator at the broker side to handle consumer rebalance. By migrating the rebalance logic from the consumer to the coordinator we can resolve the consumer split brain problem and help thinner the consumer client. With this design we hope to incorporates:

KAFKA-167: Move partition assignment to the broker

KAFKA-364: Add ability to disable rebalancing in ZooKeeper consumer

And solves:

KAFKA-242: Subsequent calls of ConsumerConnector.createMessageStreams cause Consumer offset to be incorrect

1. Motivation

The motivation of moving to a new set of consumer client APIs with broker side co-ordination is laid out here. Some back of the envelope calculations along with experience of building a prototype for this suggest that with a single coordinator we should support up to 5K consumers (the bottleneck should be the number of socket channels a single machine can have). Every broker will be the co-ordinator for a subset of consumer groups, thereby, providing the ability to scale the number of consumer groups linearly with the size of the cluster.

2. Group management protocol

Rebalancing is the process where a group of consumer instances (belonging to the same group) co-ordinate to own a mutually exclusive set of partitions of topics that the group has subscribed to. At the end of a successful rebalance operation for a consumer group, every partition for all subscribed topics will be owned by a single consumer instance. The way rebalancing works is as follows. One of the brokers is elected as the coordinator for a subset of the consumer groups. It will be responsible for triggering rebalancing attempts for certain consumer groups on consumer group membership changes or subscribed topic partition changes. It will also be responsible for communicating the resulting partition-consumer ownership configuration to all consumers of the group undergoing a rebalance operation.

Consumer

1. On startup or on co-ordinator failover, the consumer sends a ClusterMetadataRequest to any of the brokers in the "bootstrap.brokers" list. In the ClusterMetadataResponse, it receives the location of the co-ordinator for it's group.
2. The consumer sends a RegisterConsumer request to it's co-ordinator broker. In the RegisterConsumerResponse, it receives the list of topic partitions that it should own.
3. At this time, group management is done and the consumer starts fetching data and (optionally) committing offsets for the list of partitions it owns.

Co-ordinator

1. Upon election or startup, the co-ordinator reads the list of groups it manages and their membership information from zookeeper. If there is no previous group membership information, it does nothing until the first consumer in some group registers with it.
2. Upon election, if the co-ordinator finds previous group membership information in zookeeper, it waits for the consumers in each of the groups to re-register with it. Once all known and alive consumers register with the co-ordinator, it marks the rebalance completed.
3. Upon election or startup, the co-ordinator also starts failure detection for all consumers in a group. Consumers that are marked as dead by the co-ordinator's failure detection protocol are removed from the group and the co-ordinator marks the rebalance for a group completed by communicating the new partition ownership to the remaining consumers in the group.
4. In steady state, the co-ordinator tracks the health of each consumer in every group through it's failure detection protocol.
5. If the co-ordinator marks a consumer as dead, it triggers a rebalance operation for the remaining consumers in the group. Rebalance is triggered by killing the socket connection to the remaining consumers in the group. Consumers notice the broken socket connection and trigger a co-ordinator discovery process (#1, #2 above). Once all alive consumers re-register with the co-ordinator, it communicates the new partition ownership to each of the consumers in the RegisterConsumerResponse, thereby completing the rebalance operation.
6. The co-ordinator tracks the changes to topic partition changes for all topics that any consumer group has registered interest for. If it detects a new partition for any topic, it triggers a rebalance operation (as described in #5 above). It is currently not possible to reduce the number of partitions for a topic. The creation of new topics can also trigger a rebalance operation as consumers can register for topics before they are created.

3. Failure detection protocol

4. Configuration

Config Options to Brokers/Consumers

The following config options are added to brokers, these options will only be used by the broker when it elects as the coordinator:

1. Rebalancing threadpool size
2. Minimum viable value for consumer session timeout
3. Maximum number of timed out heartbeats to consumers
4. Offset committing (to ZK) interval

The following config options are added to consumers

1. Bootstrap broker list
2. Session timeout

In addition, the following config options are moved from consumer side to broker side

1. Maximum rebalance retries
2. Rebalance backoff time (in milliseconds)

Paths Stored in ZooKeeper

Most of the original ZK paths storage are kept, in addition to the **coordinator path** (stores the current coordinator info):

```
/consumers/coordinator --> broker_id (ephemeral; created by coordinator)
```

Besides, some of the original ZK paths are removed, including:

```
/consumers/groups/ids
```

And some of the ephemeral ZK paths are changed to persistent:

```
/consumers/groups/owners
```

5. On Coordinator Startup

Every server will create a coordinator object as its member upon startup. The consumer coordinator keeps the following fields (all the request formats and their usage will be introduced later in this page):

1. A ZK based elector using the coordinator path mentioned above.
2. For each group, the registry metadata containing the list of its consumer member's registry metadata (including consumer's session timeout value, topic count, offset commit option, etc), current consumed offset for each topic, etc.
3. For each topic, the consumer groups that have subscribed to the topic.
4. A list of groups that has wildcard topic subscriptions.
5. A list of rebalance handler threads, each associated with a configurable fixed size list of blocking queue storing the rebalance tasks assigned to the handler.
6. A scheduler thread for sending PingRequest to all the consumers.
7. A committer thread for periodically committing each group's consumed offsets to Zookeeper.
8. Three request purgatories, one each for PingRequest, StopFetcherRequest and StartFetcherRequest request expiration watchers.

The elector, upon initialization, will immediately try to elect as the coordinator. If someone else has become the coordinator, it will listen to the coordinator path for data change, and try to re-elect whenever the current elector resigns (i.e. the data on the path is deleted).

Whenever it elects to become the leader, it will trigger the startup procedure of the coordinator described below:

```
coordinatorStartup :

1. Register session expiration listener

2. Read all the topics from ZK and initialize the subscribed groups for each topic.

3. Register listeners for topics and their partition changes

3.1 Subscribe TopicChangeListener to /brokers/topics

3.2 Subscribe TopicPartitionChangeListener to each /brokers/topics/[topic]

4. Start up the committer and the scheduler

5. Initialize the threadpool of rebalancing handlers
```

The scheduler for ping requests is a delayed scheduler, using a priority queue of timestamps to keep track of the outstanding list of PingRequest, whenever it sends a request it needs to initialize an expire watcher in the PingRequestPurgatory for the request.

```
scheduler.run :

While isRunning

  1. Peek the head consumer from the priority queue

  2. If the consumer.scheduledTime <= current_time() try to send the PingRequest, otherwise sleep for (consumer.
scheduledTime - current_time()) and then sends it

  2.1 Sends the PingRequest via the SocketServer of the broker (the corresponding processor Id and selection
key is remembered in the consumer registry)

  2.2 Set the timeout watcher in PingRequestPurgatory for the consumer

  3. Remove the consumer from the head of the queue and put the consumer with consumer.scheduledTime +=
(consumer.session_timeout_ms * 1/3) back to the queue
```

Whenever a rebalance handler sends a Stop/StartFectherRequest to a consumer's channel, it also needs to initializes an expire watcher for the request to the corresponding Stop/StartFetcherRequestPurgatory.

When the processor received the corresponding Stop/StartFetcherResponse or the PingResponse from the channel, it needs to clear the watcher (we will go back and talk about this timeout protocol in more details later in this page). In addition, for Stop/StartFetcherResponse, it will also check if all the consumers inside the rebalancing group have their watchers cleared. If yes, notify the rebalance thread that the Stop/StartFetcherRequest have been completed handled and responded.

The timeout watcher mechanism will be implemented using the Purgatory component. For a detailed description of the request expire/satisfy purgatory, please read here.

6. On Topic Partition Changes Detection

When the ZK wachers are fired notifying topic partition changes, the coordinator needs to decide which consumer groups are affected by this change and hence need rebalancing.

Handle Topic Change

```
TopicChangeListener.handleChildChange :

1. Get the newly added topic (since /brokers/topics are persistent nodes, no topics should be deleted even if
there is no consumers any more inside the group)

2. For each newly added topic:

2.1 Subscribe TopicPartitionChangeListener to /brokers/topics/topic

2.2 For groups that have wildcard interests, check whether any group is interested in the newly added topics;
if yes, add the group to the interested group list of this topic

2.3 Get the set of groups that are interested in this topic, and if the group's rebalance bit is not set, set
it and try to rebalance the group by assigning a rebalance task to some rebalance handler's queue randomly.

* To avoid concurrent rebalancing of the same group by two different handler threads, we need to enforce
assigning the task to the same handler's queue if it is already assigned to that handler. Details will be
described in the implementation section.
```

Handle Topic Partition Change

```
TopicPartitionChangeListener.handleChildChange :

1. Get the set of groups that are interested in this topic, and if the group's rebalance bit is not set, set it
and try to rebalance the group
```

7. On Consumer Startup

Upon creation, the consumer will get a list of

```
  {brokerId : (host, port)}
```

as part of the config info from its properties file. It can then consult to any one of the brokers to get the host/port of all brokers and the ID of the coordinator.

Once the consumer finds out the address of the coordinator, it will try to connect to the coordinator. When the connection is set up, it will send the RegisterConsumerRequest to the coordinator.

After that, the consumer will try to keep reading new requests from the coordinator. Hence the consumer does not need to embed a socket server.

```
consumerStartup (initBrokers : Map[Int, (String, String)]):

1. In a round robin fashion, pick a broker in the initialized cluster metadata, create a socket channel with
that broker

1.1. If the socket channel cannot be established, it will log an error and try the next broker in the
initBroker list

1.2. The consumer will keep retrying connection to the brokers in a round robin fashioned it is shut down.

2. Send a ClusterMetadataRequest request to the broker and get a ClusterMetadataResponse from the broker

3. From the response update its local memory of the current server cluster metadata and the id of the current
coordinator

4. Set up a socket channel with the current coordinator, send a RegisterConsumerRequest and receive a
RegisterConsumerResponse

5. If the RegisterConsumerResponse indicates the consumer registration is successful, it will try to keep
reading rebalancing requests from the channel; otherwise go back to step 1
```

ClusterMetadataRequest

The ClusterMetadataRequest will be sent by the consumer to any of the brokers in the cluster.

Note that in this design we are still considering the format of 0.7, although new wire format has already been used in 0.8. Our implementation will keeps this in mind and make it easy to adapt to new format.

```
{
  size: int32                // the size of this request, not including the first 4 bytes for this field
  request_type_id: int16     // the request type id, distinguish Fetch/Produce/Offset/Consult/etc requests
                             // do not need any data for the cluster metadata request
}
```

ClusterMetadataResponse

```
{
  size: int32                                          // the size of this response, not including the first
4 bytes for this field
  error_code: int16                                    // global error code for this request, if any
  coordinator_id: int32                                // broker id of the coordinator
  num_brokers: int16                                   // number of brokers in the following list
  brokers_info: [<broker_struct>]                      // current broker list in the cluster
}

broker_struct =>
{
  id: int32
  host: string                                         // we adapt the protocol of 0.8 for strings, where
the size of the string is stored at the beginning, followed by the bytes
  port: int32
}
```

RegisterConsumerRequest

```
{
  size: int32                        // the size of this request
  request_type_id: int16             // the type of the request, currently only one type is allowed:
ConnectConsumerRequest
  group_id: string                   // group that the consumer belongs to
  consumer_id: string                // consumer id
  topic_count: string                // interested topic and number of streams, can be wildcard
  auto_commit: boolean               // indicator of whether autocommit is enabled
  auto_offset_rest: string           // indicator of what to do if an offset is out of range, currently either
smallest or largest
  session_timeout_ms: int32          // session timeout in milliseconds
}
```

RegisterConsumerResponse

Upon receiving the RegisterConsumerRequest, the coordinator will check the following:

1. Consumer's ID is unique within the same group
2. Consumer's specified session_timeout is larger than its minimum allowed session timeout value

If all the checks are successful, then the co-ordinator accepts the registration and sends a RegisterConsumerReponse with no error message; otherwise it includes an appropriate error code in the RegisterConsumerReponse. It does not close the connection but let the consumer who receives the error code to close the connection.

```
{
  size: int32                        // the size of this response
  error_code: int16                  // global error code for this request, if any
                                     // no data needed
}
```

When a new group id is discovered by the coordinator (i.e. this group does not have a registry entry in the coordinator yet), it needs to handle the newly added group.

Handle New Group

```
handleNewGroup :

1. Create a new group registry with zero consumer members in coordinator
```

In addition, whenever a new consumer is accepted by the coordinator, the coordinator needs to do the following:

Handle New Consumer

```
handleAddedConsumer :

1. Add the ping request task for this consumer to the scheduler.

2. Add the consumer registry to the group registry's member consumer registry list.

3. Read the subscribed topic/count of this consumer.

3.1 If it is a wildcard topic/count subscription, then add its group to the list of groups with wildcard topic
subscriptions.

3.2 For each topic that it subscribed to, add its group to the list of groups that has subscribed to the topic.

4. Try to rebalance the group
```

8. On Consumer Failure Detection

Upon successfully registering a consumer, the coordinator will add the consumer's PingRequest to the ping request scheduler's queue, which will then try to keep track if the consumer is still alive or not.

1. The scheduler will try to send the PingRequest to the consumer every 1/3 * consumer.session_timeout, and set the timeout watcher waiting for 2/3 * consumer.session_timeout.

2. Similarly, when the processor threads sends the stop/start fetcher requests to the consumer it also expect to receive a response within 2/3 * consumer.session_timeout.
3. If the watcher for PingRequest expires, the coordinator marks that ping request as failed.
4. If the PingRequest has failed more than the maximum number of ping retries, the coordinator will marks the consumer as failed.
    a. It removes the consumer's registry information from its memory and close the connection.
    b. It issues a rebalance request for the group of the failed consumer.
5. Even if the consumer is going through a soft failure (GC), it will disconnect from the co-ordinator within 1/3*session_timeout and stop its fetchers until it has re-registered with the current co-ordinator.

Handle Consumer Failure

```
handleConsumerFailure :

1. Remove its ping request task from the scheduler.

2. Close its corresponding socket from the broker's SocketServer.

3. Remove its registry from its group's member consumer registry list.

4. If there is an ongoing rebalance procedure by some of the rebalance handlers, let it fail directly.

5. If the group still has some member consumers, try to rebalance the group.
```

9. On Rebalancing

The rebalancing threads keep block-reading from their corresponding rebalance request queue. Hence each handles the rebalancing tasks for a non-overlapping subset of groups (e.g. through hashing functions).

For each rebalance request for a specific group it triggers the following rebalance logic:

```
rebalance (group) :

1. Get the topics that are interested by the group. For each topic:

1.1. Get the number of partitions by reading from ZK

1.2. Get the number of threads for each topic from the meta information of consumers in the group it kept in
memory

1.3. Compute the new ownership assignment for the topic

3. Check if a rebalance is necessary by trying to get the current ownership from ZK for each topic.

3.1 If there is no registered ownership info in ZK, rebalance is necessary

3.2 If some partitions are not owned by any threads, rebalance is necessary

3.3 If some partitions registered in the ownership map do not exist any longer, rebalance is necessary

3.4 If ownership map do not match with the newly computed one, rebalance is necessary

3.5 Otherwise rebalance is not necessary

4. If rebalance is necessary, do the following:

4.1 For each consumer in the group, send the StopFetcherRequest to the socket server's corresponding processor

4.2 Then wait until socket server has reported that all the StopFetcherReponse have been received or a timeout
has expired

4.3 Enforce committing the current consumed offsets information to Zookeeper

4.4 For each consumer in the group, send the StartFetcherRequest to the socket server's corresponding processor

4.5 Then wait until socket server has reported that all the StartFetcherReponse have been received or a timeout
has expired

4.6 Update the new assignment of partitions to consumers in the Zookeeper

5. If a timeout signal is received in either 4.2 or 4.5 from the socket server, wait for config.
rebalanceBackoffMs and retry rebalancing again.
```

If the handler cannot finish rebalance successfully with *config.maxRebalanceRetries* retries, it will throw a *ConsumerRebalanceFailedException* to the log.

10. On Coordinator Failover

Whenever the current coordinator's hosted server dies, other coordinator's elector will realize that through the ZK listener and will try to re-elect to be the leader, and whoever wins will trigger the coordinator startup procedure.

When the dead server comes back, it's elector will atomically reconnect to ZK and trigger the *handleNewSession* function:

```
handleNewSession :

1. Reset its state by clearing its in memory metadata.

2. Re-register the session expiration listener (this is because ZkClient does not re-register itself once
fired).

3. Try to re-elect to be the coordinator by directly calling the elect function of its coordinatorElector.
```

As for consumers, if a consumer has not heard any request (either Ping or Stop/StartFetcher) from the coordinator for session_timeout, it will suspects that the coordinator is dead. Then it will stops its fetcher, close the connection and re-execute the startup procedure, trying to re-discover the new coordinator.

11. On Consumer Handling Coordinator Request

There are three types of requests that a consumer can receive from the coordinator: PingRequest, StopFetcherRequest and StartFetcherRequest. For each type of request, the consumer is expected to respond within 2/3 * session_timeout.

PingRequest

Coordinator uses PingRequest to check if the consumer is still alive, and consumer need to respond with the current offset for each consuming partition if auto_commit is set.

```
{
  size: int32                      // the size of this request
  request_type_id: int16           // the type of the request, distinguish Ping/Stop/StartFetcher requests
  consumer_id: string              // id string of the consumer
  num_retries: int32               // indicate this is the #th ping retry
}
```

PingResponse

```
{
  size: int32                      // the size of this response
  request_type_id: int16           // the type of the request, distinguish Ping/Stop/StartFetcher requests
  error_code: int16                // global error code for this request, if any
  consumer_id: string              // id string of the consumer, this is only used by the purgatory
  num_partition: int16             // number of partitions that the consumer is consuming
  offset_info: [<offset_struct>]  // the offset info for each partition consumed by this consumer, its size
should be exactly num_partition
                                   // if auto_commit is set to false the last two fields should not exist
}

offset_struct =>
{
  topic: string
  partition: string
  offset: int64
}
```

Handling PingRequest

```
handlePingRequest (request: PingRequest):

1. If shutdown is initiated:

1.1 If autoCommit is not turned on, can shut down immediately after responding this PingRequest

1.2 Otherwise, need to wait for the next PingRequest to arrive after responding this PingRequest to make sure
the previous PingResponse with the offset info has been received by the coordinator.

2. If the fetchers of the consumer has stopped unexpectedly due to out-of-range error, close the current
connection with the coordinator, stop fetchers, and try to reconnect to the coordinator in order to trigger
rebalance.

3.1 If autoCommit is not enabled, just sends a PingResponse with no num_partition and offset_info

3.2 Otherwise read topicRegistry and sends a PingResponse with the created num_partitions and offset_info
```

Handling PingResponse

```
handlePingResponse (response: PingResponse):

1. Get the offset information for each consumed partition, and record the offset information to the group's
registry if possible.

2. Clear the corresponding expiration watcher in PingRequestPurgatory.
```

StopFetcherRequest

When the coordinator decides to rebalance a group, it will send StopFetcherRequest to every consumer in the group to let them stop their fetchers.

```
{
  size: int32                                 // the size of this request
  request_type_id: int16                      // the type of the request, distinguish Ping/Stop/StartFetcher
requests
  consumer_id: string                         // id string of the consumer, this is only used by the purgatory
}
```

Handling StopFetcherRequest

```
StopFetcherRequest (request: StopFetcherRequest):

1. Close all fetchers and clean their corresponding queues

2. Send the response, if autoCommit is turned on attached the current consumed offsets to the response also.

3. Clear the corresponding expiration watcher in StopFetcherRequestPurgatory, and if all the watchers have been
cleared notify the rebalance handler.
```

StopFetcherResponse

```
{
  size: int32                      // the size of this response
  request_type_id: int16           // the type of the request, distinguish Ping/Stop/StartFetcher requests
  error_code: int16                // global error code for this request, if any
  consumer_id: string              // id string of the consumer, this is only used by the purgatory
  num_partition: int16             // number of partitions that the consumer is consuming
  offset_info: [<offset_struct>]   // the offset info for each partition consumed by this consumer, its size
should be exactly num_partition
                                   // if auto_commit is set to false the last two fields should not exist
}
```

StartFetcherRequest

When the coordinator have received the StopFetcherResponse from all the consumers, it will send the StartFetcherRequest to the consumers along with the starting offset for each partition got from the StopFetcherResponses. Note that the StartFetcherRequest also contains the current cluster metadata, which is used to update the consumer's server cluster metadata in order to avoid unnecessary rebalancing attempts.

```
{
  size: int32                                 // the size of this request
  request_type_id: int16                      // the type of the request, distinguish Ping/Start/StartFetcher
requests
  consumer_id: string                         // id string of the consumer
  num_threads: int16                          // the number of threads that have assigned partitions
  ownership_info: [<ownership_struct>]        // the detailed info of the ownership
  num_brokers: int16                          // number of brokers in the following list
  brokers_info: [<broker_struct>]             // current broker list in the cluster}

ownership_struct =>
{
  consumer_thread_id: string                  // consumer thread id
  num_partitions: int16                       // the number of partitions assigned to the consumer
  assigned_parts: [<partition_struct>]        // the detailed info of assigned partitions, along with the starting
offset
}

partition_struct =>
{
  topic: string
  partition: string                           // note that this partition string already contains the broker id
  offset: int64
}
```

Handling StartFetcherRequest

```
handleStartFetcherRequest (request: StartFetcherRequest):

1. If the consumer's topic interests are wildcard, re-construct topicThreadIdAndQueues and
KafkaMessageAndMetadataStreams

2. Read the assigned partitions along with offsets from assigned_parts

3. Update topic registry with the newly assigned partition map

4. Start fetchers with the new topic registry

5. Send back the StartFetcherResponse
```

StartFetcherResponse

```
{
  size: int32                      // the size of this response
  request_type_id: int16           // the type of the request, distinguish Ping/Stop/StartFetcher requests
  error_code: int16                // global error code for this request, if any
  consumer_id: string              // id string of the consumer, this is only used by the purgatory
}
```

Handling StartFetcherResponse

```
handleStartRequest (response: StartFetcherResponse):

1. Clear the corresponding expiration watcher in StartFetcherRequestPurgatory, and if all the watchers have
been cleared notify the rebalance handler.
```

12. Implementation

12.1 kafka.api

The following classes are used to describe the request/registry format described in Section 6 and Section 10:

- *ClusterMetadataRequest / ClusterMetadataResponse* (it is only for reading response from channel, with corresponding writing class as kafka.server.ClusterInfoSend)
- *HeartbeatRequest / HeartbeatResponse* (similar to above, its corresponding write-to-channel class is kafka/consumer/OffsetInfoSend)
- *PartitionAndOffset*: represents one assigned partition info (topic, partition, offset)
- *RegisterConsumerRequest* (it does not have a wrapper class for reading from channel since it is very simple; its corresponding response format is in kafka/server/ConfirmationSend)
- *GroupRegistry*: it contains a list of ConsumerRegistry of the consumers that belongs to this group. In addition, it contains the following metadata:
    - offsetPerTopic: the offset information per topic-partition that is consumed by the group so far, updated by either receiving PingResponses or StopFetcherResponses from consumers of the group.
    - offsetUpdated (Boolean) : indicate if the offset information of the group has been updated since the last time it gets written to ZK. Periodically, the coordinator's offset committer thread will loop through all the groups, and writes their offset info to ZK if this bit is set; once written, the thread will reset this bit.
    - rebalanceInitiated (Boolean) : indicate if the group is waiting in some rebalancer's queue for rebalancing. If it is set, then no more rebalance requests will be generated and pushed to the rebalancers' queues. It will be set upon pushing to any rebalancer's queue, and be reset by the rebalancer once its get dequeued.
    - currRebalancer : indicate if the current rebalancer handling this group's rebalance request (if no one is handling it then it will be None). Upon adding a new rebalance request, the coordinator will first try to see if it already has some rebalancer assigned to it; if yes, the coordinator is enforced to push this rebalance request to the same queue in order to avoid concurrent rebalancing of the same group. Otherwise the coordinator will try to assign it to any one in order to achieve load balancing. It will be assigned the rebalancer's id once gets pushed to its queue, and only be reset once the rebalancer has completed the rebalance task.
    - rebalanceLock/rebalanceCondition: used by the rebalancer to wait on everyone to respond the stop/start fetcher request.
    - rebalanceFailed: indicate if the current rebalance trial has failed. It can be set if 1) some stop/start fetcher request has expired by the purgatory's expiration reaper; 2) the handleConsumerFailure function in case this consumer's group is under going rebalance in order to achieve fastfail. It will be reset at the beginning of every rebalance trial and checked by the rebalancer after waiting for everyone to respond the stop/start fetcher request.
- *ConsumerRegistry*: it contains the meta information of the consumer which will be sent through the RegisterConsumerRequest; in addition, it contains the corresponding processorId and selectionKey that is used in the SocketServer to remember the consumer's corresponding channel. These two fields are not sent in RegisterConsumerRequest but only get recorded by the coordinator upon accepting the consumer's registry.

- *StartFetcherRequest / StartFetcherResponse* (similar to above, its corresponding write-to-channel class is kafka/consumer/OffsetInfoSend)

- *ConsumerPartitionOwnershipInfo*: stores a listof PartitionAndOffset that is assigned to one consumer thread. This class is used in the StartFetcherRequest for representing the assigned partition info to each consumer.

- *StopFetcherRequest / StopFetcherResponse* (similar to above, its corresponding write-to-channel class is kafka/server/ConfirmationSend)

- All the above requests' request ids are added to RequestKeys.scala

12.2 kafka.cluster

Broker and *Cluster* add writeTo(buffer) and sizeInBytes functions to be used in ClusterInfoSend.

12.3 kafka.common

Add some more error code to *ErrorMapping.*

12.4 kafka.network

- *SocketServer*: hacked to be able to recognize coordinator-specific requests, which would need coordinator's metadata to handle; also hacked to be able to pro-actively send messages by assuming no partial read/write is under going (which is valid for the coordinator's usage). **In the future this hacking needs to be removed when we redesign SocketServer after 0.8 for interleaving reads/writes**.
- *NonBlockingChannel*:

12.5 kafka.utils

ZkElection: election module based on Zookeeper.

Also some utility functions are added to *Utils* and *ZkUtils*.

12.6 kafka.consumer

- *OffsetInfoSend*: offset information format wrapper, match with for HeartbeatResponse and StopFetcherResponse.

- *SimpleConsumerConnector*: The simplified consumer client to replace the original ZookeeperConsumerConnector, also inherited from the ConsumerConnector interface. It differs with the ZookeeperConsumerConnector as the following:
  - Remove all functionalities that depends on Zookeeper, remove the ZkClient (hence its Fetcher/FetcherRunnable classes also removed ZkClient)
  - Simplified WildcardStreamsHandler, now it onl used to remember the template type of the message, but no need to connect to Zookeeper to keep track of topic changes.
  - Add a ConsumerCoordinatorConnector thread which is used to communicate with coordinator via RPC and indicates the consumer to stop/start fetchers.

- *WildcardStreamsHandler*: for consumers subscribing to wildcard topics, it will construct a WildcardStreamsHandler, which is used to re-initialize the mapping from chunk queues to streams on receiving StartFetcherRequest

- *ConsumerCoordinatorConnector*: it is used to
  - Register to the coordinator and maintains the channel with the coordinator.
  - Handles lost-connection/broken pipe event by re-reconnecting to the new coordinator.
  - Receive and handle HearbeatRequest and /Start/StopFetcherRequest through its CoordinatorRequestHandler.
  - **In the future it needs to be able to pro-actively sends CommitOffsetRequest to the coordinator in order to actively call commitoffset().**

- *CoordinatorRequestHandler*: it contains the handler function to handle different types of requests from the coordinator by the ConsumerCoordinatorConnector. Its logic is described in Section 10.

12.7 kafka.server

- *KafkaServer* contains a new *ConsumerCoordinator* instance, which will be initialized upon server startup if ZookeeperEnabled is true.

- *ConsumerCoordinator* contains the following members and metadata structures (just following Section 4):
  - A ZKClient that is different from the one that KafkaServer used.
  - ZkElection: a Zookeeper based election module, which will trigger the startup call back procedure of the ConsumerCoordinator upon success election. Hence at the same time period only one ConsumerCoordinator of the servers will be "active".
  - Array[RebalanceRequestHandler]: a list of rebalance handler threads which is used for processing the rebalancing tasks for groups, each has a BlockingQueue[String] storing assigned rebalance tasks.
  - A KafkaScheduler heartbeat request scheduling thread which periodically sends heartbeat request to all consumers (frequency based on consumer's session timeout value) that is registered to this coordinator.
  - A KafkaScheduler offset committing thread which periodically writes the current consumed offset for each group to Zookeeper using the coordinator's ZKClient.
  - A HeartbeatRequestPurgatory storing expiration watchers for sent HeartbeatRequest (in implementation we rename "ping" to "heartbeat")
  - A StopFetcherRequestPurgatory and a StartFetcherRequestPurgatory storing expiration watchers for Stop /StartFetcherRequestPurgatory.

  - consumerGroupsPerTopic: a list of groups which have subscribed to each topic.
  - groupsWithWildcardTopics: a list of groups which have at least one consumer that have wildcard topic subscription.

    - groupRegistries: a list of group registries, each contains the group metadata for rebalancing usage and its member consumers' registered metadata.

and the main API functions include:

    - startup: the startup logic described in Section 4.
    - shutdown: called by the KafkaServer upon shutting down, will first close the elector, shutdown the heartbeat request scheduler, offset committer and rebalance handlers.
    - addRebalanceRequest: assign one rebalance task for a specific group to one of the rebalance handler's queue.
        - If the group has been assigned to a rebalance handler and have not been finished rebalancing (i.e, currRebalancer != None), enforce assign this task also to this handler to avoid concurrent rebalancing of the same group.
        - Otherwise choose the rebalance handler in a round-robin fashion for load balance.
    - sendHeartbeatToConsumer: called by the heartbeat request scheduler, used to send the HeartbeatRequest to the consumer if 1) the previous request has been responded and 2) the consumer still exist in the group's registry
    - handleHeartbeatRequestExpired: increment the number of failed response in the consumer's registry, and if it has exceeded the maximum allowed failures, mark the consumer as failed and trigger handleConsumerFailure
    - handleNewGroup: logic described in Section 6
    - handleNewGroupMember: logic of handle new consumer described in Section 6
    - handleConsumerFailure: logic described in Section 7
    - A list of utility functions used to access the group registry data outside ConsumerCoordinator, which is synchronized to avoid read/write conflicts.

- *RebalanceRequestHandler*: logic of the rebalance handler thread:
    - run: while not shutting down, keep trying to get the next rebalance task from its queue, clear the group's rebalanceInitiated bit and process the rebalance logic (described in Section 8).
        - Before each rebalance attempt, first reset the rebalanceFailed bit in the group's registry, which could be set by other thread during the rebalance procedure.
        - Before each rebalance attempt, read the current group's consumer registry list snapshot, which will not change during the rebalance procedure even if its underlying registry data has changed.
        - Upon successfully complete the rebalance, reset the currRebalancer to None, indicating that no one is now assigned to this group for rebalancing.
    - sendStopFetcherRequest: if the consumer's channel has closed while trying to send the request, immediately mark this attempt as failed.
    - sendStartFetcherRequest: read the offset from Zookeeper before constructing the request, and if the consumer's channel has closed while trying to send the request, immediately mark this attempt as failed.
    - isRebalanceNecessary: compare the new assignment with the current assignment written in Zookeeper to decide if rebalancing is necessary.
    - waitForAllResponses: waiting for all Stop/StartFetcherResponse to received or timeout.

- *KafkaRequestHandlers*: add handle functions for coordinator-specific responses (i.e. HeartbeatResponse, StopFetcherResponse, StartFetcherResponse), need a different function signature since it requires knowledge of the coordinator. **In 0.8 with api/handlers this needs to refactoring**.

- *HeartbeatRequestPurgatory*: inherted from RequestPurgatory, on expire call the handleHeartbeatRequestExpired function of ConsumerCoordinator.

- *StartFetcherRequestPurgatory:* inherted from RequestPurgatory, on expire call the handleStopOrStartFetcherRequestExpired function of ConsumerCoordinator.

- *StopFetcherRequestPurgatory:* inherted from RequestPurgatory, on expire call the handleStopOrStartFetcherRequestExpired function of ConsumerCoordinator. **In 0.8 branch with RequestOrReponse class these two can be combined into one**.

13. Open Issues / Future work

Implementation

- Active CommitOffset on Consumers: Currently consumers can no longer actively call commitOffset() to commit their current consumed offsets to Zookeeper, but can only wait for HeartbeatRequests to respond with their offsets. In order to support this the consumers must be not only able to read requests from the coordinator channel but also write requests to the channel. In addition, the SocketServer that the coordinator depends on for RPC communication must also be able to concurrently read to and write from channels.

- Conflict of HeartbeatRequest with Stop/StartFetcherResponse: When the heartbeat scheduler of the coordinator tries to send the HeartbeatRequest to consumers, it is possible that the channel for the consumer has some partial read Stop/StartFetcherResponse, since the current SocketServer can only read or write alone at the same time, the scheduler can only drop this request and wait for the next time. On the consumer's side, it would probably suspect the coordinator has failed and re-connect to the new one, causing unnecessary rebalancing attempts. The solution of the previous issue should be able to solve this also: enabling SocketServer to be able to concurrently read to and write from channels.

- Incorporating with 0.8 branch: There are still several implementation details that are based on 0.7, which needs to be incorporated with 0.8 after 0.8 is stable (these are marked as TODOs in the code):
    - Combine Request and Response into RequestOrResponse: this can allow StartFetcherPurgatory and StopFetcherPurgatory to be combined also.
    - Request format protocol: in 0.8 we need to add some more fields such as versionId, correlationId and ClientId to the requests.
    - Refactor RequestHandlers: in 0.8 the RequestHandlers are in kafka.api with SocketServer enhanced with request and response queues. Currently we hacked the RequestHandlers to treat coordinator request/response as special cases against other request types such as fetcher/produce/offset, and also hacked SocketServer to be aware of these coordinator-related special case requests; after 0.8 we should go back to the principal that SocketServer is ignorant of Kafka. More detailed discussions can be found here.

- Virtual Failure in Unit Test: in order to test the failure detection functionality of the coordinator/consumer, we also need to simulate scenarios when the coordinator/consumer is virtually failed (e.g. undergoing a long GC or blocked by IO). Currently it is hacked in the Coordinator code to simulate such cases, but we definitely needs some methods to not touch the implementation but enable them in some testing frameworks.

- Deleting Group Registries from Coordinator: currently the group registry entry will never be removed from the coordinator even if the group has no member consumers any more. With ConsoleConsumer implementation which creates a random-named group to register, it may cause a lot of "empty" group registry entries in some long running coordinator. We need to think about the semantics of "when" and "how" we can delete a consumer group in the coordinator.

Design

- Dependency on Initial Broker List: When all the brokers listed in the properties file as known brokers are gone when a consumer starts/resumes, the consumer cannot find the coordinator and thus cannot be added to the group to start consuming. This case should be treated as an operational error since the migration of broker cluster should be incremental and adapt to consumer properties file.

- Latency Spike during Coordinator Failover: Since consumers no longer register themselves in Zookeeper, when a new coordinator stands up, it needs to wait for all the consumer to re-connect to it, causing a lot of channel connectioned and rebalancing requests to the rebalance handlers. Hence the consumers will not start their fetchers after be connected to the coordinator until it receives the StartFetcherRequest, some groups's consumers might wait longer time than usually to start consuming.

- Multiple Coordinator: In the future, it might be possible that we need each broker to act as a coordinator, with consumer groups evenly distributed amongst all the consumer coordinators. This will enable supporting >5K consumers since right now the number of open socket connections a single machine can handle limits the scale of the consumers.

- Rebalance Retrial Semantics needs Rethinking: currently the rebalance retrial semantics still follows the original design: whenever the current attempt has failed, sleep for some backoff time and retry again; when the maximum number of retries as reached, throw an exception. With the coordinator based rebalancing mechanism, it is not clear if the maximum rebalance retry still make sense any more: probably the rebalance handler should keep retrying until the rebalance has succeeded, and also the backoff time should be dependent to the consumers' session timeout value.