Client Rewrite

Over several years of usage we have run into a number of complaints about the scala client. This is a proposal for addressing these complaints.

This would likely be done in the timeframe of a 0.9 release.

Current Problems and Fixes

Here is a dump of all the problems we have seen and some solutions:

- Move clients to Java to fix scala problems
  - Javadoc
  - Scala version non-compatability
  - Readability by non-scala users
  - Scary stack traces
  - Leakage of scala classes/interfaces into java api
- Code cleanup and embeddability
  - Both producer and consumer code are extremely hard to understand
  - Redo the request serialization layer to avoid all the custom request definition objects
  - Eliminate the "simple" consumer api and have only a single consumer API with the capabilities of both
  - Remove all threads from the consumer
  - Have a separate client jar with no depedencies
- Generalize APIs
  - Producer
    - Give back a return value containing error code, offset, etc
  - Consumer
    - Enable static partition assignment for stateful data systems
    - Enable consumer-driven offset changes.
- Better support non-java consumers
  - Move to a high-level protocol for consumer group management to centralize complexity on the server for all clients
- Improve performance and operability
  - Make the producer fully async to to allow issuing sends to all brokers simultaneously and having multiple in-flight requests simultaneously. This will dramatically reduce the impact of latency on throughput (which is important with replication).
  - Move to server-side offset management will allow us to scale this facility which is currently a big scalability problem for high-commit rate consumers due to zk non scalability.
  - Server-side group membership will be more scalable with number of partitions then the current consumer co-ordination protocol
  - Improve inefficiencies in compression code

The idea would be to roll out the new api as a separate jar, leaving the existing client intact but deprecated for one or two releases before removing the old client. This should allow a gradual migration.

Proposed Producer API

```
SendResponse send(KafkaMessage... message);


Usage:
Producer producer = new Producer(new ProducerConfig(props));
SendResponse r = producer.send(new KafkaMessage(topic, key, message));
r.onCompletion(new Runnable() {System.out.println("All done")})
r.getOffset()
r.getError()
```

Changes from current API

- The key and message in KafkaMessage would have type Object instead of parameterized types. The parameterized types have not played well with the fact that different topics may take different types and since the serializer is instantiated via reflection (at runtime) the parametric types add no actual type safety.
- The producer will always attempt to batch data and will always immediately return a SendResponse which acts as a Future to allow the client to await the completion of the request.

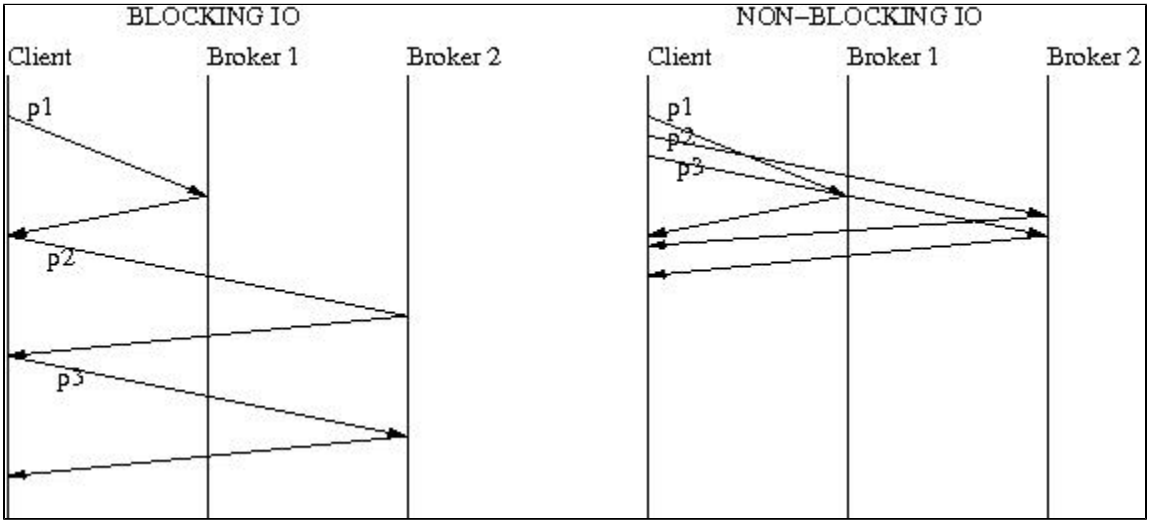
Implementation

As today we will have a single background thread that does message collation, serialization, and compression. However this thread will now run an event loop to simultaneously send requests and receive responses from all brokers.

In 0.8, the producer throughput is limited due to the synchronous interaction of the producer with the broker. At a time, there can only be one request in flight per broker and that hurts the throughput of the producer severely. The problem with this approach also is that it doesn't take advantage of parallelism on the broker where requests for different partitions can be handled by different request handler threads on the broker, thereby increasing the throughput seen by the producer.

Key idea behind the design

Request pipelining allows multiple requests to the same broker to be in flight. The idea is to do I/O multiplexing using select(), epoll(), wait() and wait for at least one readable or writable socket instead of doing blocking reads and writes. So if a producer sends data to 3 partitions on 2 brokers, the difference between blocking sends/receives and pipelined sends/receives can be understood by looking at the following picture



Design requirements

1. Ordering should be maintained per partition, if there are no retries.
2. Existing producer API should remain completely asynchronous.
3. No APIs changes are necessary to roll this out.

Proposed design

There are 2 threads, depending on how metadata requests are handled.

1. Client thread - partitions
2. Send thread - refreshes metadata, serializes, collates, batches, sends data and receives acknowledgments

Event Queuing

There are 2 choices here -

One queue per partition

There will be one queue with partition "-1" per topic. The events with null key will enter this queue. There will be on "undesignated" partition queue per topic. This is done to prevent fetching metadata on the client thread when the producer sends the first message for a new topic.

Pros:

1. More isolation within a topic. Keys that are high traffic will not affect keys that are lower traffic within a topic. This will protect important low throughput topics like audit data.
2. Easier to handle the most common error which is leader transition. It is easier with one queue per partition since you can just ignore the queues for partitions that don't have leaders yet.

Cons:

1. More queues means more memory overhead, especially for tools like MirrorMaker.
2. Handling time based event expiration per partition queue complicates the code to some extent.

One queue for all topics and partitions.

Pros:
Less memory overhead since there is only one queue data structure for all events

Cons:

1. Less isolation. One high throughput topic or partition can cause data for other topics to be dropped. This especially hurts audit data and can be easily avoided by having multiple queues.
2. Complicates batching since you will have to find a way to skip over partitions that don't have leaders and avoid dequeueing their data until a leader can be discovered.

Partitioning

Partitioning can happen before the event enters the queue. The advantage is that the event does not require re-queuing if one queue per partition approach is used. Events with null key will be queued to the <topic>-"-1" queue.

Metadata discovery

When a producer sends the first message for a new topic, it enters the <topic>-undesignated queue. The metadata fetch happens on the event thread. There are 2 choices on how the metadata fetch request will work -

The metadata fetch is a synchronous request in the event loop.

Pros:
Simplicity of code

Cons:
If leaders only for a subset of partitions have changed, a synchronous metadata request can potentially hurt the throughput for other topics/partitions.

Metadata fetch is non blocking

Pros:
More isolation, better overall throughput

Serialization

One option is doing this before the event enters the queue and after partitioning. Downside is potentially slowing down the client thread if compression or serialization takes long. Another option is to just do this in the send thread, which seems like a better choice.

Batching and Collation

Producer maintains a map of broker to list of partitions that the broker leads. Batch size is per partition. For each broker, if the key is in write state, the producer's send thread will poll the queues for the partitions that the broker leads. Once the batch is full, it will create a ProducerRequest with the partitions and data, compress the data, and writes the request on the socket. This happens in the event thread while handing new requests. The collation logic gets a little complicated if there is only one queue for all topics/partitions.

Compression

When the producer send thread polls each partition's queue, it compresses the batch of messages that it dequeues.

Event loop

```
while(isRunning)
{

// configure any new broker connections

// select readable keys

// handle responses

// select writable keys

// handle topic metadata requests, if there are non-zero partitions in error

// handle incomplete requests

// handle retries, if any

// handle new requests

}
```

Consumer API

Here is an example of the proposed consumer API:

```
Consumer consumer = new Consumer(props);
consumer.addTopic("my-topic"); // consume dynamically assigned partitions
consumer.addTopic("my-other-topic", 3); //
long timeout = 1000;
while(true) {
  List<MessageAndOffset> messages = consumer.poll(timeout);
  process(messages);
  consumer.commit(); // alternately consumer.commit(topic) or consumer.commit(topic, partition)
}
```

As before the consumer group is set in the properties and for simplicity each consumer can belong to only one group.

addTopic is used to change the set of topics the consumer consumes. If the user gives only the topic name it will automatically be assigned a set of partitions based on the group membership protocol below. If the user specifies a partition they will be statically assigned that partition.

The actual partition assignment can be made pluggable (see the proposal on group membership below) by setting an assignment strategy in the properties.

This api allows alternative methods for managing offsets as today by simply disabling autocommit and not calling commit().

Implementation

The client would be entirely single threaded.

The poll() method executes the network event loop, which includes the following:

- Checks for group membership changes
- Sends a heartbeat to the controller if needed
- Issues a fetch request to all brokers for which the consumer currently is consuming
- Reads any available fetch responses
- Issues metadata requests when requests fail to get the new cluster topology

The timeout the user specifies will be purely to ensure we have a mechanism to give control back to the user even when no messages are delivered. It is up to the user to ensure poll() is called again within the heartbeat frequency set for the consumer group. Internally the timeout on our select() may uses a shorter timeout to ensure the heartbeat frequency is met even when no messages are delivered.

Consumer Group Membership APIs

We will introduce a set of RPC apis for managing partition assignment on the consumer side. This will be based on the prototype here. This set of APIs is orthogonal to the APIs for producing and consuming data, it is responsible for group membership.

api	sender	description	issue to
ListGroups	client	Returns metadata for one or more groups. If issued with a list of GroupName it returns metadata for just those groups. If no GroupName is given it returns metadata for all active groups. This request can be issued to any server.	any server
CreateGroup	client	Create a new group with the given name and the specified minimum heartbeat frequency. Return the id/host/port of the server acting as the controller for that group. If the ephemeral flag is set the group will disappear when the last client exits.	any server
DeleteGroup	client	Deletes a non-emphemeral group (but only if it has no members)	controller
RegisterConsumer	client	Ask to join the given group. This happens as part of the event loop that gets invoked when the consumer uses the poll() API	controller
Heartbeat	client	This heartbeat message must be sent by the consumer to the controller with an SLA specified in the CreateGroup command. The client can and should issues these more frequently to avoid accidentally timing out.	controller

Heartbeat

```
Heartbeat
  Version            => int16
  CorrelationId      => int64
  ClientId           => string
  SessionTimeout     => int64
```

HeartbeatResponse

```
Version               => int16
CorrelationId         => int64
ControllerGeneration  => int64
ErrorCode             => int16 // error code is non zero if the group change is to be initiated
```

CreateGroup

```
Version                  => int16
CorrelationId            => int64
ClientId                 => string
ConsumerGroup            => string
SessionTimeout           => int64
Topics                   => [string]
```

CreateGroupResponse

```
Version                   => int16
CorrelationId             => int64
ConsumerGroup             => string
ErrorCode                 => int16
```

DeleteGroup

```
Version                  => int16
CorrelationId            => int64
ClientId                 => string
ConsumerGroup            => [string]
```

DeleteGroupResponse

```
Version                   => int16
CorrelationId             => int64
DeletedConsumerGroups     => [string]
ErrorCode                 => int16
```

RegisterConsumer

```
Version                   => int16
CorrelationId             => int64
ClientId                  => string
ConsumerGroup             => string
ConsumerId                => string
```

RegisterConsumerResponse

```
Version                   => int16
CorrelationId             => int64
ClientId                  => string
ConsumerGroup             => string
PartitionsToOwn           => [{Topic Partition Offset}]
  Topic                   => string
  Partition               => int16
  Offset                  => int64
ErrorCode                 => int16
```

ListGroups

```
Version                   => int16
CorrelationId             => int64
ClientId                  => string
ConsumerGroups            => [string]
```

ListGroupsResponse

```
Version                   => int16
CorrelationId             => int64
ClientId                  => string
GroupsInfo                => [{GroupName, GroupMembers, Topics, ControllerBroker, SessionTimeout}]
  GroupName               => string
  GroupMembers            => [string]
  Topics                  => [string]
  ControllerBroker        => Broker
     Broker               => BrokerId Host Port
     BrokerId             => int32
     Host                 => string
     Port                 => int16
ErrorCode                 => int16
```

RewindConsumer

```
Version                    => int16
CorrelationId              => int64
ClientId                   => string
ConsumerGroup              => string
NewOffsets                 => [{Topic Partition Offset}]
  Topic                    => string
  Partition                => int16
  Offset                   => int64
```

RewindConsumerResponse

```
Version                    => int16
CorrelationId              => int64
ConsumerGroup              => string
ActualOffsets              => [{Topic Partition Offset}]
  Topic                    => string
  Partition                => int16
  Offset                   => int64
ErrorCode                  => int16
```

Consumer Group Membership Protocol

The use of this protocol would be as follows:

On startup or when co-ordinator heartbeat fails -

- on startup the consumer issues a list_groups(my_group) to a random broker to find out the location of the controller for its group (each server is the controller for some groups)
- knowing where the controller is, it connects and issues a RegisterConsumer RPC, then it awaits a RegisterConsumerResponse
- on receiving the RegisterConsumer request the server sends an error code in the HeartbeatResponse to all alive group members and waits for a new RegisterConsumer request from each consumer, except the one that sent the initial RegisterConsumer.
- on receiving an error code in the HeartbeatResponse, all consumers stop fetching, commit offsets, re-discover the controller through list_groups (my_group) and send a RegisterConsumer request to the controller
- on receiving RegisterConsumer from all consumers of the group membership change, the controller sends the RegisterConsumerResponse to all the new group members, with the partitions and the respective offsets to restart consumption from. If there are no previous offsets for the consumer group, -1 is returned. The consumer starts fetching from earliest or latest, depending on consumer configuration
- on receiving the RegisterConsumerResponse the consumer is now able to start consuming its partitions and must now start sending heartbeat messages back to the controller with the current generation id.

When new partitions are added to existing topics or new topics are created -

- on discovering newly created topics or newly added partitions to existing topics, the controller sends an error code in the HeartbeatResponse to all alive group members and waits for a new RegisterConsumer request from each consumer, except the one that sent the initial RegisterConsumer.
- on receiving an error code in the HeartbeatResponse, all consumers stop fetching, commit offsets, re-discover the controller through list_groups (my_group) and send a RegisterConsumer request to the controller
- on receiving RegisterConsumer from all consumers of the group membership change, the controller sends the RegisterConsumerResponse to all the new group members, with the new partitions and the respective offsets to restart consumption from. For newly added partitions and topics, the offset is set to smallest (0L).
- on receiving the RegisterConsumerResponse the consumer is now able to start consuming its partitions and must now start sending heartbeat messages back to the controller with the current generation id.

Offset Rewind

```
 while(true) {
  List<MessageAndMetadata> messages = consumer.poll(timeout);
  process(messages);
  if(rewind_required) {
     List<PartitionOffset> partitionOffsets = new ArrayList<PartitionOffset>();
     partitionOffsets.add(new PartitionOffset(topic, partition, offset));
     rewind_offsets(partitionOffsets);
  }
}
```

- the consumer stops fetching data, sends a RewindConsumer request to the controller and awaits a RewindConsumerResponse
- on receiving a RewindConsumer request, the controller sends an error code in the HeartbeatResponse to the current owners of the rewound partitions
- on receiving an error code in the HeartbeatResponse, the affected consumers stop fetching, commit offsets and send the RegisterConsumer request to the controller
- on receiving a RegisterConsumer request from the affected members of the group, the controller records the new offsets for the specified partitions and sends the new offsets to the respective consumers
- on receiving a RegisterConsumerResponse, the consumers start fetching data from the specified offsets

Low-level RPC API

TBD we need to design an underlying client network abstraction that can be used to send requests. This should ideally handle metadata and addressing by broker id, multiplex over multiple connections, and support both blocking and non-blocking operations.

Something like

```
KafkaConnection connection = new KafkaConnection(bootstrapUrl, socket_props);
List<Responses> connection.poll(timeout, Request...requests);
```

This is an async api so the responses responses returned have no relationship to the request sent. This is meant to be called from within an event loop--i.e. each call to poll corresponds to one iteration of the select() loop. poll() with no request simply checks for any new responses to previous requests or non-client-initiated communication.

Misc Notes and questions

Currently we do a lot of logging and metrics recording in the client. I think instead of handling reporting of metrics and logging we should instead incorporate this feedback in the client apis and allow the user to log or monitor this in the manner they chose. This will avoid dependence on a particular logging or metrics library and seems like a good policy.

It is not clear how to support changing the offset either manually or programmatically while consumption is happening. Simply using the api to set the offset will likely collide with commit calls from the active consumer. Perhaps this is not needed?

How many jars should we have? I think we could do either

- Option A: kafka-clients.jar and kafka-server.jar or
- Option B: kakfa-common.jar, kafka-producer.jar, kafka-consumer.jar, and kafka-server.jar

I prefer Option A as it is simpler if we add an AdminApi--we won't need to introduce a whole new jar for it.