Producer API Proposal

Here is a proposal for refactoring the producer API. There are several goals:

1. Make it more natural to store keys in the messages. Currently the ProducerData maps a key to a list of values, which is odd and doesn't map well to the data model in Message.
2. Return back the offset of the message that was just written.
3. Enable asynchonous I/O, even in the sync case (more on this below).
4. Improve the partitioner api

This are also two major internal improvements that would be good to make:

1. Clean up the producer logic which is currently quite convoluted
2. Move the producer to use NIO and a Selector instead of blocking sockets and support multiplexing requests.
   We don't need to do these things all at once, and we may not need to do any of them in 0.8, but it seems good to lay out all the things we want to do here. Some discussion of possible phasing is given below.

Here is some details on each of these above items:

The send() call currently takes one or more ProducerData objects. Each such object has an optional key, a topic, and one or more values. I don't recall the exact rationale for this, but it is a little bit odd. Now that we support keys in the Message class it is even more odd. A more natural model would have each ProducerData item correspond to one Message (i.e. an optional key, a value, and a topic).

Now that we have a proper response object for the produce request it would be nice to give the producer back the offset and partition of the message that was just added to the log. This offset is a good way to refer to the message's position in the log and represents the "point-in-time" of the append.

NIO would have a number of advantages which I discuss in more detail below.

The Partitioner API currently is a method partition(key: K, numberOfPartitions: Int). This is a pretty good API, but misses one important case. In previous versions we had the ability to configure a broker list (which was admittedly a hack), we used this to effectively direct 100% of messages to a single broker from each producer. The advantage of this setup was that each producer only needed one connection. This worked both ways, on the broker side we only needed to support NUM_PRODUCERS/NUM_BROKERS connections per broker. Effectively we no longer have a way to scale the total number of connections in the case where you don't care about key-based partitioning. To support this in a non-hacky way I think you need to things: (1) the ability to timeout unused connections, (2) the ability to implement a partitioner that directs all messages to a single broker (maybe switching brokers every 30 seconds or so for load balancing). The problem is that the current API doesn't give you any information about nodes, only about partitions, so even if you always give a constant partition, this partition will be on different brokers for different topics. We could fix this by expanding the Partitioner interface to also take the current leader list to allow implementing such a partitioner.

Proposed Producer API

Producer:

```
trait Producer {

  /* Send one or more messages and get back a list of corresponding responses, one per message.
   * This API is meant to be asynchonous, so it never directly throws exceptions.
   * Instead it returns a list of lazy Future-like response values which contain the result or error.
   */
  def send(message: ProducerMessage*): Seq[ProduceResponse]
}
}
```

A ProduceMessage is analagous to the ProducerData class we currently have (i.e. just a placeholder for the preserialized data):

```
 case class ProducerMessage(topic: String, key: K, value: V)
```

The ProduceResponse is a lazy object that contains the offset and any error that was thrown:

```
 class ProduceResponse {

  /* Wait for the response to complete and return the result. This method will
   * either return the partition and offset of the corresponding
   * message or else throw an exception (if there was an error).
   * If the timeout is hit with no response available, it will throw a TimeoutException.
   */
  def await(timeout: Long = 0): PartitionAndOffset

  /* Check if the request is complete without blocking
   */
  def isComplete(): Boolean

  /* Execute the given callback when the response has arrived. Mutliple callbacks may be specified.
   * Although we guarantee that the callback will execute and that r.await will not block
   * we don't guarantee what thread will execute the callback.
   */
  def whenComplete((r: ProduceResponse) => Unit)

}
```

Note that currently send() is a blocking calls and so returning a lazy object might seem odd. However the move to NIO would make this untrue.

Some Limitations And An Alternative

The primary limitation of the proposed API is that it has no way for specifying per-request parameters. We allow our produce request to specify the number of acks, and the timeout, but there is no place to put this. The right way to fix this would be to have send() take a ProducerRequest object that contained the ProducerMessages but also had fields for the various per-request configs. Having request-level object seems like a good future-proof way to have a place to put any future optional parameters we decided we might need to add to the request.

Such an API would look like this:

```
 val request = new ProducerRequest()
                .addMessage("topic_1", "my_key1", "my_value1")
                .addMessage("topic_2", "my_key2", "my_value2")
                .timeout(1000)     // timeout after 1 second
                .awaitCommitted()  // i.e. ISR = -1, would also support await(n) to give an explicit number of
brokers
                .compression(NoCompressionCodec)
 val responses = client.send(request)
 for(response <- responses)
   println(response.waitFor)
```

In general I think it is a good design to always have this kind of API because you can easily evolve it by adding more optional parameters. With our current API in order to add a new parameter we would need to make a new method with more parameters.

The problem with this approach is that the async producer blurs the user's control over what goes into which request. So if the user set different parameters on a few of these request objects and then we had to batch them into a single API request what parameter would we choose? If we went down this path we would need a rule for combining per-request configurations (i.e. we enforce the least batch size and the minimum timeout). Depending on what the rules were this might or might not be intuitive.

Rationale for NIO

A few people have questioned this, so it seems worthwhile to layout what this buys us.

Currently we have phenomenal *throughput* when messages are batched together, but our throughput when sending a single message at a time is not great (it's not terrible either, but it could be a lot better).

Previously our send had no acknowledgement which meant that often the send was effectively asynchonous. However now that is no longer the case we always block on responses. This means that if you use the async producer and send a large batch of messages you will effectively do a set of serial requests to each server.

So there are a few kinds of paralellism we can get out of this:

1. The ability to send multiple messages to a server without waiting for the response. This allows multithreaded use of the producer.send without each thread blocking the others on the server's append.
2. The ability to send messages to multiple servers in parallel rather than serially

Blocking I/O is definitely simpler to use in the case where there is only a single connection to a single server (as we originally had), but I don't think it is simpler in the case where there is multiple connections to multiple servers. Now all our network I/O for both the consumer, the replicas, and producer connects to multiple servers. Non-blocking I/O has been pretty good on the server, and I think we should move to it for the clients too.

Here is a sketch of how this would look in code. Instead of having a single connection that took Send and Receive objects, we would have a multi-connection which represented a connection to the whole cluster of brokers:

```
class KafkaSelector {

  /* Add a new connection associated with the given node id */
  def connect(node: Int, host: String, port: Int)

  /* Remove a connection */
  def disconnect(node: Int)

  /* Add some Send objects to be sent to the given node ids. Return any completed Receives
   * If the given timeout is reached return even if no new receives are ready
   */
  def poll(sends: Seq[(Int, Send)], timeout: Long): Seq[Recieve]
}
```

This would allow the producer to simultaneously send on all sockets with only a single thread. It would also allow a single consumer thread to effectively poll on many sockets. It also allows the producer to multiplex requests over a single socket (i.e. have many requests in flight before a response is received).