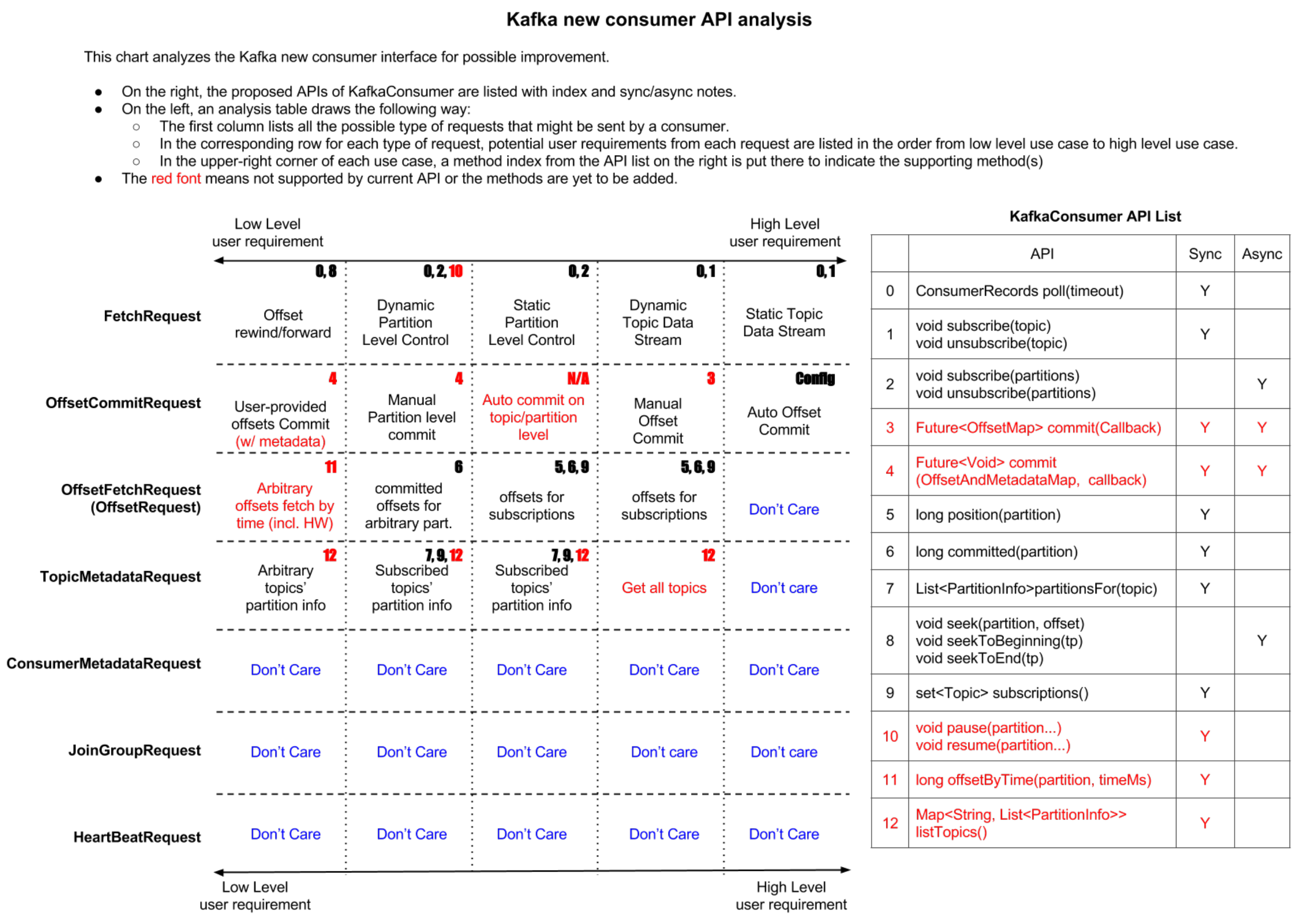
New consumer API change proposal

API and user requirement analysis



Kafka new consumer API analysis

This chart analyzes the Kafka new consumer interface for possible improvement.

- On the right, the proposed APIs of KafkaConsumer are listed with index and sync/async notes.
- On the left, an analysis table draws the following way:
  - The first column lists all the possible type of requests that might be sent by a consumer.
  - In the corresponding row for each type of request, potential user requirements from each request are listed in the order from low level use case to high level use case.
  - In the upper-right corner of each use case, a method index from the API list on the right is put there to indicate the supporting method(s)
- The red font means not supported by current API or the methods are yet to be added.

	Low Level user requirement →				High Level user requirement
FetchRequest	0, 8 Offset rewind/forward	0, 2, 10 Dynamic Partition Level Control	0, 2 Static Partition Level Control	0, 1 Dynamic Topic Data Stream	0, 1 Static Topic Data Stream
OffsetCommitRequest	4 User-provided offsets Commit (w/ metadata)	4 Manual Partition level commit	N/A Auto commit on topic/partition level	3 Manual Offset Commit	Config Auto Offset Commit
OffsetFetchRequest (OffsetRequest)	11 Arbitrary offsets fetch by time (incl. HW)	6 committed offsets for arbitrary part.	5, 6, 9 offsets for subscriptions	5, 6, 9 offsets for subscriptions	Don't Care
TopicMetadataRequest	12 Arbitrary topics' partition info	7, 9, 12 Subscribed topics' partition info	7, 9, 12 Subscribed topics' partition info	12 Get all topics	Don't care
ConsumerMetadataRequest	Don't Care	Don't Care	Don't Care	Don't Care	Don't Care
JoinGroupRequest	Don't Care	Don't Care	Don't Care	Don't care	Don't care
HeartBeatRequest	Don't Care	Don't Care	Don't Care	Don't Care	Don't Care
	Low Level user requirement ←				High Level user requirement

KafkaConsumer API List

	API	Sync	Async
0	ConsumerRecords poll(timeout)	Y	
1	void subscribe(topic) void unsubscribe(topic)	Y	
2	void subscribe(partitions) void unsubscribe(partitions)		Y
3	Future<OffsetMap> commit(Callback)	Y	Y
4	Future<Void> commit (OffsetAndMetadataMap, callback)	Y	Y
5	long position(partition)	Y	
6	long committed(partition)	Y	
7	List<PartitionInfo>partitionsFor(topic)	Y	
8	void seek(partition, offset) void seekToBeginning(tp) void seekToEnd(tp)		Y
9	set<Topic> subscriptions()	Y	
10	void pause(partition...) void resume(partition...)	Y	
11	long offsetByTime(partition, timeMs)	Y	
12	Map<String, List<PartitionInfo>> listTopics()	Y	

The chart above analyzes the new consumer API requirements by looking at the requirements that user might have for each request. We are trying to use this chart to exhaustively list

1. All the functions consumer can possibly achieve.
2. All the potential user requirements

The goal is to have a clean and intuitive APIs design with good reasoning for each user requirement (each grid in the above chart).

Threading Model and Sync/Async Semantics

Currently new consumer follows a single threaded model. While it provides simplicity by saving some synchronization efforts, it is also enforces user to care about the things that they don't need to care about. Several examples are:

1. When user try to commit offsets asynchronously, without calling a poll(), offset won't be committed.
2. If user does not call poll() frequently enough, broker will consider the consumer dead. If user set session timeout to be larger, the failure detection will be also longer.
3. callbacks will not fire.

We fixed some problems in KAFKA-2123, which introduced a task queue and will temporarily reuse the caller thread for an "execution thread" to run against the queue. But it does not solve the fundamental issue which is user have to essentially provide a dedicated thread keep calling poll even if they actually don't want to consume data. So although new consumer is claimed to be single threaded, it is very likely most user have to write their own wrapper with multiple threads.

Async semantic also becomes confusing with the single-threaded model. We are now enforcing user to do something after they fire an async call, which seems to defeat one important purpose of async - user lose the freedom of doing something else after fire an async call. From user point of view, it is as if they are still blocked on that call because they have to keep calling poll().

The main benefit of single threaded model is that we can detect client failure when they stop consuming data. We want to change the threading model a bit so it can still have this benefit but solve the issues we mentioned above.

Proposed Solution

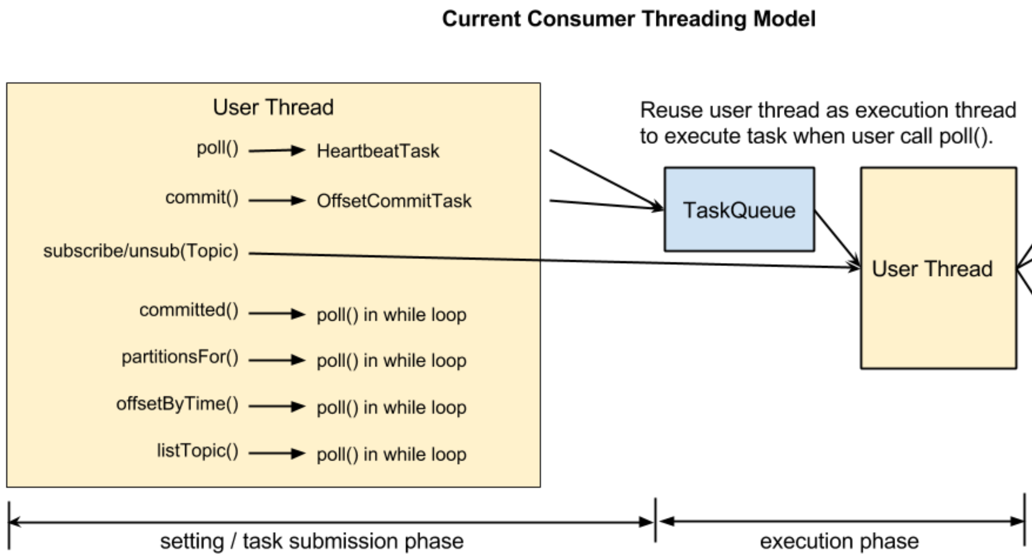
API Proposal

```
public interface Consumer<K, V> extends Closeable {

    /**
     * Return the topic partition the consumer is currently subscribing to.
     */
    Set<TopicPartition> subscriptions();
    /**
     * Subscribe to a specified topics. This method is for user who uses
     * Kafka based group management. This is a blocking call and will return
     * when consumer successfully subscribed to the topic. Exception will be thrown
     * when subscription fails.
         */
    void subscribe(String... topics);
    /**
     * Subscribe to a partition explicitly. This method is for users who want to
     * have self group management. This is a non blocking call and will take effect
     * when user do the next poll().
     */
    void subscribe(TopicPartition... partitions);
    /**
     * Similar to subscribe(String... topics), this method is for users who uses
     * Kafka based group management. It is a blocking call and will return
     * when consumer successfully unsubscribed from the topics. Exception will be
     * thrown when subscription fails.
     */
    void unsubscribe(String... topics);
    /**
     * Unsubscribe from a partition explicitly. This method is for users who want
     * to have self group management. This is a non blocking call and will take
     * effect when user do the next poll().
     */
    void unsubscribe(TopicPartition... partitions);
    /**
     * This method will try to get data from Kafka brokers. It will block for at most
     * timeout if there is no data available, and will return immediately when there
     * are fetched messages.
     */
    ConsumerRecords<K, V> poll(long timeout);
    /**
     * Commit offset for all the partitions this consumer is consuming from. The committed
     * offsets will be the offsets after last poll(). This function is a non-blocking method.
     * If user wants to commit offsets synchronously, user can call commit().get().
     * If user wants to commit offsets asynchronously, user can use the callback.
     */
    Future<Map<TopicPartition, Long>> commit(ConsumerCommitCallback callback);
    /**
     * Similar to commit(ConsumerCommitCallback), except this methods allows user to commit specified
     * offsets with optional metadata.
     */
    Future<Map<TopicPartition, Long>> commit(Map<TopicPartition, Long> offsets, ConsumerCommitCallback
callback);
    /**
```

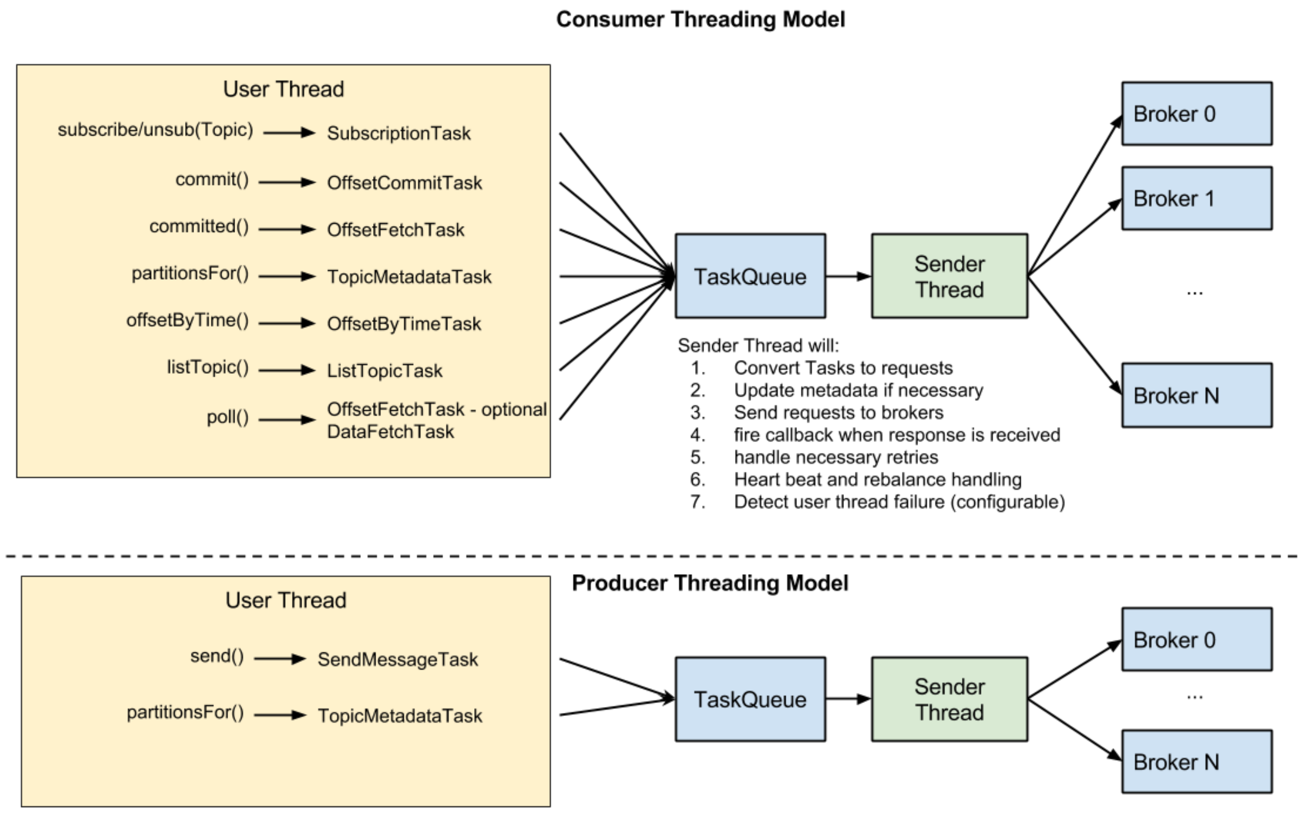
```
     * Temporarily stop consuming from the specified partitions.
     * This is a non-blocking call and will take effect from the next poll()
     */
    void pause(TopicPartition... partitions);
    /**
     * Resume consumption from partitions previously called on pause().
     * This is a non-blocking call and will take effect from the next poll().
     */
    void resume(TopicPartition... partitions);
    /**
     * Seek to a specified offset for a partition. This is a non-blocking call and will take effect
     * from the next poll().
     */
    void seek(TopicPartition partition, long offset);
    /**
     * Seek to the earliest available offsets of the partitions.
     * This method is a non-blocking call and will only take effect from the next poll().
     */
    void seekToBeginning(TopicPartition... partitions);
    /**
     * Seek to the latest offsets of the partitions.
     * Similar to seekToBeginning(TopicPartition...), this is a non-blocking call that will take effect
     * after the next poll().
     */
    void seekToEnd(TopicPartition... partitions);
    /**
     * Return the current offset for the partition this consumer is subscribing to.
     * Exception will be thrown when partition is not in subscriptions.
     */
    long position(TopicPartition partition);
    /**
     * Return the committed offsets of a subscribed partition.
     * This is a synchronous call.
     */
    OffsetAndMetadata committed(TopicPartition partition);
    /**
     * Return the latest offset appended to the log before the specified time.
     * This is a blocking call.
     * If time=-1, the method returns the earliest offset.
     * If time=-2, the method returns the latest offset.
     * Implementation wise, the latest offset can be acquired from the LEO piggybacked
     * in fetch response. So we don't need to talk to broker unless the last fetch response
     * is certain time ago(e.g. 1 second). For the partition the consumer is not consuming from,
     * we still need to talk to broker.
     */
    OffsetAndMetadata offsetByTime(TopicPartition partition, long time)
    /**
     * Return the metrics.
     */
    Map<MetricName, ? extends Metric> metrics();
    /**
     * Return the partition information of a topic.
     * This is a synchronous call.
     */
    List<PartitionInfo> partitionsFor(String topic);
    /**
     * Return all the topic partition information in the cluster.
     * This is a synchronous call.
     */
    Map<TopicPartition, List<PartitionInfo>> listTopics();
    /**
     * @see KafkaConsumer#close()
     */
    void close();

}
```

Threading-Model Proposal

The major gaining of using a single-threaded model is to associate the consumer liveliness with the actual data fetching. The downside of this issue is that the current threading model is sort of "hijacking" the user thread to act as an execution thread when user thread calls poll().



We want to propose a threading model very similar to what we are using in new producer which has been proven to be welcomed by users. At the same time, we will keep the benefit of single-threaded model with little efforts.



The major changes in this threading model are:

1. Asynchronous calls will follow the convention and easy to implement.
2. poll() becomes very intuitive - meaning user wants some data.

To solve the liveliness association with data fetching. We can let sender thread detect how long has it been since the user thread last called poll() - generating a DataFetchTask. If user thread hasn't been fetching for session.timeout, the sender thread can choose to stop heartbeat. This feature can be a boolean config to turn on/off, e.g. **liveliness.detection.enabled**. If it is turned on, the liveliness definition will be the same as current new consumer. If it is turned off, the liveliness definition will be the same as old high level consumer.

Sync or Async, that's a question....

Personally, I think a call should be sync if

1. User expects information back, or
2. subsequent action depends on the function call.

Otherwise a method can be async.

The assumption is that if user called a method and try to get some information back, they will use that information for further actions. For methods that might have both use cases, we provide both sync/async interface, e.g. commit().

The proposed API follows this reasoning, but I'm not sure if they are correctly defined. So we can discuss about that if there are further concerns.

With the above threading model, implementation of sync or async will be clean.

Throwing Exceptions Or Not?

In current implementation, consumer will try to handle exceptions if possible. For example, if user try to subscribe to a non-existing topic, the consumer will just wait until the topic to be existed. However, after talking to several users, they actually want to know if a topic does not exist. In that case, throwing exception might be better than handle that for user. Because if user wants to ignore it, they can always catch exception and retry, whereas if we handle it for user, some user who cares about non-existing topic might not know they subscribed to a wrong topic. The above interface followed this reasoning. However, arguably with some checking interface, it might be reasonable to say, if user cares about if a topic exists or not, they can always call listTopics() before subscribing. So I am also not sure if we should throw exceptions in that case or not.