A Guide To The Kafka Protocol

Introduction

This document covers the protocol implemented in Kafka 0.8 and beyond. It is meant to give a readable guide to the protocol that covers the available requests, their binary format, and the proper way to make use of them to implement a client. This document assumes you understand the basic design and terminology described here.

The protocol used in 0.7 and earlier is similar to this, but we chose to make a one time (we hope) break in compatibility to be able to clean up cruft and generalize things.

Overview

The Kafka protocol is fairly simple, there are only six client requests APIs.

1. Metadata - Describes the currently available brokers, their host and port information, and gives information about which broker hosts which partitions.
2. Send - Send messages to a broker
3. Fetch - Fetch messages from a broker, one which fetches data, one which gets cluster metadata, and one which gets offset information about a topic.
4. Offsets - Get information about the available offsets for a given topic partition.
5. Offset Commit - Commit a set of offsets for a consumer group
6. Offset Fetch - Fetch a set of offsets for a consumer group

Each of these will be described in detail below.

Preliminaries

Network

Kafka uses a binary protocol over TCP. The protocol defines all apis as request response message pairs. All messages are size delimited and are made up of the following primitive types.

The client initiates a socket connection and then writes a sequence of request messages and reads back the corresponding response message. No handshake is required on connection or disconnection. TCP is happier if you maintain persistent connections used for many requests to amortize the cost of the TCP handshake, but beyond this penalty connecting is pretty cheap.

The client will likely need to maintain a connection to multiple brokers, as data is partitioned and the clients will need to talk to the server that has their data. However it should not generally be necessary to maintain multiple connections to a single broker from a single client instance (i.e. connection pooling).

The server guarantees that on a single TCP connection, requests will be processed in the order they are sent and responses will return in that order as well. The broker's request processing allows only a single in-flight request per connection in order to guarantee this ordering. Note that clients can (and ideally should) use non-blocking IO to implement request pipelining and achieve higher throughput. i.e., clients can send requests even while awaiting responses for preceding requests since the outstanding requests will be buffered in the underlying OS socket buffer. All requests are initiated by the client, and result in a corresponding response message from the server except where noted.

The server has a configurable maximum limit on request size and any request that exceeds this limit will result in the socket being disconnected.

Partitioning and bootstrapping

Kafka is a partitioned system so not all servers have the complete data set. Instead recall that topics are split into a pre-defined number of partitions, P, and each partition is replicated with some replication factor, N. Topic partitions themselves are just ordered "commit logs" numbered 0, 1, ..., P.

All systems of this nature have the question of how a particular piece of data is assigned to a particular partition. Kafka clients directly control this assignment, the brokers themselves enforce no particular semantics of which messages *should* be published to a particular partition. Rather, to publish messages the client directly addresses messages to a particular partition, and when fetching messages, fetches from a particular partition. If two clients want to use the same partitioning scheme they must use the same method to compute the mapping of key to partition.

These requests to publish or fetch data must be sent to the broker that is currently acting as the leader for a given partition. This condition is enforced by the broker, so a request for a particular partition to the wrong broker will result in an the NotLeaderForPartition error code (described below).

How can the client find out which topics exist, what partitions they have, and which brokers currently host those partitions so that it can direct its requests to the right hosts? This information is dynamic, so you can't just configure each client with some static mapping file. Instead all Kafka brokers can answer a metadata request that describes the current state of the cluster: what topics there are, which partitions those topics have, which broker is the leader for those partitions, and the host and port information for these brokers.

In other words, the client needs to somehow find one broker and that broker will tell the client about all the other brokers that exist and what partitions they host. This first broker may itself go down so the best practice for a client implementation is to take a list of two or three urls to bootstrap from. The user can then choose to use a load balancer or just statically configure two or three of their kafka hosts in the clients.

The client does not need to keep polling to see if the cluster has changed; it can fetch metadata once when it is instantiated cache that metadata until it receives an error indicating that the metadata is out of date. This error can come in two forms: (1) a socket error indicating the client cannot communicate with a particular broker, (2) an error code in the response to a request indicating that this broker no longer hosts the partition for which data was requested.

1. Cycle through a list of "bootstrap" kafka urls until we find one we can connect to. Fetch cluster metadata.
2. Process fetch or produce requests, directing them to the appropriate broker based on the topic/partitions they send to or fetch from.
3. If we get an appropriate error, refresh the metadata and try again.

Partitioning Strategies

As mentioned above the assignment of messages to partitions is something the producing client controls. That said, how should this functionality be exposed to the end-user?

Partitioning really serves two purposes in Kafka:

1. It balances data and request load over brokers
2. It serves as a way to divvy up processing among consumer processes while allowing local state and preserving order within the partition. We call this semantic partitioning.

For a given use case you may care about only one of these or both.

To accomplish simple load balancing a simple approach would be for the client to just round robin requests over all brokers. Another alternative, in an environment where there are many more producers than brokers, would be to have each client chose a single partition at random and

publish to that. This later strategy will result in far fewer TCP connections.

Semantic partitioning means using some key in the message to assign messages to partitions. For example if you were processing a click message stream you might want to partition the stream by the user id so that all data for a particular user would go to a single consumer. To accomplish this the client can take a key associated with the message and use some hash of this key to choose the partition to which to deliver the message.

Batching

Our apis encourage batching small things together for efficiency. We have found this is a very significant performance win. Both our API to send messages and our API to fetch messages always work with a sequence of messages not a single message to encourage this. A clever client can make use of this and support an "asynchronous" mode in which it batches together messages sent individually and sends them in larger clumps. We go even further with this and allow the batching across multiple topics and partitions, so a produce request may contain data to append to many partitions and a fetch request may pull data from many partitions all at once.

The client implementer can choose to ignore this and send everything one at a time if they like.

Versioning and Compatibility

The protocol is designed to enable incremental evolution in a backward compatible fashion. Our versioning is on a per-api basis, each version consisting of a request and response pair. Each request contains an API key that identifies the API being invoked and a version number that indicates the format of the request and the expected format of the response.

The intention is that clients would implement a particular version of the protocol, and indicate this version in their requests. Our goal is primarily to allow API evolution in an environment where downtime is not allowed and clients and servers cannot all be changed at once.

The server will reject requests with a version it does not support, and will always respond to the client with exactly the protocol format it expects based on the version it included in its request. The intended upgrade path is that new features would first be rolled out on the server (with the older clients not making use of them) and then as newer clients are deployed these new features would gradually be taken advantage of.

Currently all versions are baselined at 0, as we evolve these APIs we will indicate the format for each version individually.

The Protocol

Protocol Primitive Types

The protocol is built out of the following primitive types.

Fixed Width Primitives

int8, int16, int32, int64 - Signed integers with the given precision (in bits) stored in big endian order.

Variable Length Primitives

bytes, string - These types consist of a signed integer giving a length N followed by N bytes of content. A length of -1 indicates null. string uses an int16 for its size, and bytes uses an int32.

Arrays

This is a notation for handling repeated structures. These will always be encoded as an int32 size containing the length N followed by N repetitions of the structure which can itself be made up of other primitive types. In the BNF grammars below we will show an array of a structure foo as [foo].

Notes on reading the request format grammars

The BNFs below give an exact context free grammar for the request and response binary format. For each API I will give the request and response together followed by all the sub-definitions. The BNF is intentionally not compact in order to give human-readable name (for example I define a production for ErrorCode even though it is just an int16 in order to give it a symbolic name). As always in a BNF a sequence of productions indicates concatenation, so the MetadataRequest given below would be a sequence of bytes containing first a VersionId, then a ClientId, and then an array of TopicNames (each of which has its own definition). Productions are always given in camel case and primitive types in lower case. When there are multiple possible productions these are separated with '|' and may be enclosed in parenthesis for grouping. The top-level definition is always given first and subsequent sub-parts are indented.

Common Request and Response Structure

All requests and responses originate from the following grammar which will be incrementally describe through the rest of this document:

```
RequestOrResponse => Size (RequestMessage | ResponseMessage)
  Size => int32
```

Field	Description
MessageSize	The MessageSize field gives the size of the subsequent request or response message in bytes. The client can read requests by first reading this 4 byte size as an integer N, and then reading and parsing the subsequent N bytes of the request.

Requests

Requests all have the following format:

```
RequestMessage => ApiKey ApiVersion CorrelationId ClientId RequestMessage
  ApiKey => int16
  ApiVersion => int16
  CorrelationId => int32
  ClientId => string
  RequestMessage => MetadataRequest | ProduceRequest | FetchRequest | OffsetRequest |
OffsetCommitRequest | OffsetFetchRequest
```

Field	Description
ApiKey	This is a numeric id for the API being invoked (i.e. is it a metadata request, a produce request, a fetch request, etc).
ApiVersion	This is a numeric version number for this api. We version each API and this version number allows the server to properly interpret the request as the protocol evolves. Responses will always be in the format corresponding to the request version. Currently the supported version for all APIs is 0.
CorrelationId	This is a user-supplied integer. It will be passed back in the response by the server, unmodified. It is useful for matching request and response between the client and server.
ClientId	This is a user supplied identifier for the client application. The user can use any identifier they like and it will be used when logging errors, monitoring aggregates, etc. For example, one might want to monitor not just the requests per second overall, but the number coming from each client application (each of which could reside on multiple servers). This id acts as a logical grouping across all requests from a particular client.

The various request and response messages will be described below.

Responses

```
Response => CorrelationId ResponseMessage
CorrelationId => int32
ResponseMessage => MetadataResponse | ProduceResponse | FetchResponse | OffsetResponse
| OffsetCommitResponse | OffsetFetchResponse
```

Field	Description
CorrelationId	The server passes back whatever integer the client supplied as the correlation in the request.

The response will always match the paired request (e.g. we will send a MetadataResponse in return to a MetadataRequest).

Message sets

One structure common to both the produce and fetch requests is the message set format. A message in kafka is a key-value pair with a small amount of associated metadata. A message set is just a sequence of messages with offset and size information. This format happens to be used both for the on-disk storage on the broker and the on-the-wire format.

A message set is also the unit of compression in Kafka, and we allow messages to recursively contain compressed message sets to allow batch

compression.

N.B., MessageSets are not preceded by an int32 like other array elements in the protocol.

```
MessageSet => [Offset MessageSize Message]
  Offset => int64
  MessageSize => int32
```

Message format

```
Message => Crc MagicByte Attributes Key Value
  Crc => int32
  MagicByte => int8
  Attributes => int8
  Key => bytes
  Value => bytes
```

Field	Description
Offset	This is the offset used in kafka as the log sequence number. When the producer is sending messages it doesn't actually know the offset and can fill in any value here it likes.
Crc	The CRC is the CRC32 of the remainder of the message bytes. This is used to check the integrity of the message on the broker and consumer.
MagicByte	This is a version id used to allow backwards compatible evolution of the message binary format. The current value is 0.
Attributes	This byte holds metadata attributes about the message. The lowest 2 bits contain the compression codec used for the message. The other bits should be set to 0.
Key	The key is an optional message key that was used for partition assignment. The key can be null.
Value	The value is the actual message contents as an opaque byte array. Kafka supports recursive messages in which case this may itself contain a message set. The message can be null.

Compression

Kafka supports compressing messages for additional efficiency, however this is more complex than just compressing a raw message. Because individual messages may not have sufficient redundancy to enable good compression ratios, compressed messages must be sent in special batches (although you may use a batch of one if you truly wish to compress a message on its own). The messages to be sent are wrapped (uncompressed) in a MessageSet structure, which is then compressed and stored in the Value field of a single "Message" with the appropriate compression codec set. The receiving system parses the actual MessageSet from the decompressed value. The outer MessageSet should contain only one compressed "Message" (see KAFKA-1718 for details).

Kafka currently supports two compression codecs with the following codec numbers:

Compression	Codec
None	0
GZIP	1
Snappy	2

The APIs

This section gives details on each of the individual APIs, their usage, their binary format, and the meaning of their fields.

Metadata API

This API answers the following questions:

- What topics exist?
- How many partitions does each topic have?
- Which broker is currently the leader for each partition?
- What is the host and port for each of these brokers?

This is the only request that can be addressed to any broker in the cluster.

Since there may be many topics the client can give an optional list of topic names in order to only return metadata for a subset of topics.

The metadata returned is at the partition level, but grouped together by topic for convenience and to avoid redundancy. For each partition the metadata contains the information for the leader as well as for all the replicas and the list of replicas that are currently in-sync.

Topic Metadata Request

```
TopicMetadataRequest => [TopicName]
  TopicName => string
```

Field	Description
TopicName	The topics to produce metadata for. If empty the request will yield metadata for all topics.

Metadata Response

The response contains metadata for each partition, with partitions grouped together by topic. This metadata refers to brokers by their broker id. The brokers each have a host and port.

```
MetadataResponse => [Broker][TopicMetadata]
  Broker => NodeId Host Port  (any number of brokers may be returned)
   NodeId => int32
   Host => string
   Port => int32
  TopicMetadata => TopicErrorCode TopicName [PartitionMetadata]
   TopicErrorCode => int16
  PartitionMetadata => PartitionErrorCode PartitionId Leader Replicas Isr
   PartitionErrorCode => int16
   PartitionId => int32
   Leader => int32
   Replicas => [int32]
   Isr => [int32]
```

Field	Description
Leader	The node id for the kafka broker currently acting as leader for this partition. If no leader exists because we are in the middle of a leader election this id will be -1.
Replicas	The set of alive nodes that currently acts as slaves for the leader for this partition.
Isr	The set subset of the replicas that are "caught up" to the leader
Broker	The node id, hostname, and port information for a kafka broker

Produce API

The produce API is used to send message sets to the server. For efficiency it allows sending message sets intended for many topic partitions in a single request.

The produce API uses the generic message set format, but since no offset has been assigned to the messages at the time of the send the producer is free to fill in that field in any way it likes.

Produce Request

```
ProduceRequest => RequiredAcks Timeout [TopicName [Partition MessageSetSize
MessageSet]]
  RequiredAcks => int16
  Timeout => int32
  Partition => int32
  MessageSetSize => int32
```

Field	Description
RequiredAcks	This field indicates how many acknowledgements the servers should receive before responding to the request. If it is 0 the server will not send any response (this is the only case where the server will not reply to a request). If it is 1, the server will wait the data is written to the local log before sending a response. If it is -1 the server will block until the message is committed by all in sync replicas before sending a response. For any number > 1 the server will block waiting for this number of acknowledgements to occur (but the server will never wait for more acknowledgements than there are in-sync replicas).
Timeout	This provides a maximum time in milliseconds the server can await the receipt of the number of acknowledgements in RequiredAcks. The timeout is not an exact limit on the request time for a few reasons: (1) it does not include network latency, (2) the timer begins at the beginning of the processing of this request so if many requests are queued due to server overload that wait time will not be included, (3) we will not terminate a local write so if the local write time exceeds this timeout it will not be respected. To get a hard timeout of this type the client should use the socket timeout.
TopicName	The topic that data is being published to.
Partition	The partition that data is being published to.
MessageSetSize	The size, in bytes, of the message set that follows.
MessageSet	A set of messages in the standard format described above.

Produce Response

```
ProduceResponse => [TopicName [Partition ErrorCode Offset]]
  TopicName => string
  Partition => int32
  ErrorCode => int16
  Offset => int64
```

Field	Description
Topic	The topic this response entry corresponds to.
Partition	The partition this response entry corresponds to.
ErrorCode	The error from this partition, if any. Errors are given on a per-partition basis because a given partition may be unavailable or maintained on a different host, while others may have successfully accepted the produce request.
Offset	The offset assigned to the first message in the message set appended to this partition.

Fetch API

The fetch API is used to fetch a chunk of one or more logs for some topic-partitions. Logically one specifies the topics, partitions, and starting offset at which to begin the fetch and gets back a chunk of messages. In general, the return messages will have offsets larger than or equal to the starting offset. However, with compressed messages, it's possible for the returned messages to have offsets smaller than the starting offset. The number of such messages is typically small and the caller is responsible for filtering out those messages.

Fetch requests follow a long poll model so they can be made to block for a period of time if sufficient data is not immediately available.

As an optimization the server is allowed to return a partial message at the end of the message set. Clients should handle this case.

One thing to note is that the fetch API requires specifying the partition to consume from. The question is how should a consumer know what partitions to consume from? In particular how can you balance the partitions over a set of consumers acting as a group so that each consumer

gets a subset of partitions. We have done this assignment dynamically using zookeeper for the scala and java client. The downside of this approach is that it requires a fairly fat client and a zookeeper connection. We haven't yet created a Kafka API to allow this functionality to be moved to the server side and accessed more conveniently. A simple consumer client can be implemented by simply requiring that the partitions be specified in config, though this will not allow dynamic reassignment of partitions should that consumer fail. We hope to address this gap in the next major release.

Fetch Request

```
FetchRequest => ReplicaId MaxWaitTime MinBytes [TopicName [Partition FetchOffset
MaxBytes]]
  ReplicaId => int32
  MaxWaitTime => int32
  MinBytes => int32
  TopicName => string
  Partition => int32
  FetchOffset => int64
  MaxBytes => int32
```

Field	Description
ReplicaId	The replica id indicates the node id of the replica initiating this request. Normal client consumers should always specify this as -1 as they have no node id. Other brokers set this to be their own node id. The value -2 is accepted to allow a non-broker to issue fetch requests as if it were a replica broker for debugging purposes.
MaxWaitTime	The max wait time is the maximum amount of time in milliseconds to block waiting if insufficient data is available at the time the request is issued.
MinBytes	This is the minimum number of bytes of messages that must be available to give a response. If the client sets this to 0 the server will always respond immediately, however if there is no new data since their last request they will just get back empty message sets. If this is set to 1, the server will respond as soon as at least one partition has at least 1 byte of data or the specified timeout occurs. By setting higher values in combination with the timeout the consumer can tune for throughput and trade a little additional latency for reading only large chunks of data (e.g. setting MaxWaitTime to 100 ms and setting MinBytes to 64k would allow the server to wait up to 100ms to try to accumulate 64k of data before responding).
TopicName	The name of the topic.
Partition	The id of the partition the fetch is for.
FetchOffset	The offset to begin this fetch from.
MaxBytes	The maximum bytes to include in the message set for this partition. This helps bound the size of the response.

Fetch Response

```
FetchResponse => [TopicName [Partition ErrorCode HighwaterMarkOffset MessageSetSize
MessageSet]]
  TopicName => string
  Partition => int32
  ErrorCode => int16
  HighwaterMarkOffset => int64
  MessageSetSize => int32
```

Field	Description
TopicName	The name of the topic this response entry is for.
Partition	The id of the partition this response is for.
HighwaterMarkOffset	The offset at the end of the log for this partition. This can be used by the client to determine how many messages behind the end of the log they are.
MessageSetSize	The size in bytes of the message set for this partition

MessageSet	The message data fetched from this partition, in the format described above.

Offset API

This API describes the valid offset range available for a set of topic-partitions. As with the produce and fetch APIs requests must be directed to the broker that is currently the leader for the partitions in question. This can be determined using the metadata API.

The response contains the starting offset of each segment for the requested partition as well as the "log end offset" i.e. the offset of the next message that would be appended to the given partition.

We agree that this API is slightly funky.

Offset Request

```
OffsetRequest => ReplicaId [TopicName [Partition Time MaxNumberOfOffsets]]
  ReplicaId => int32
  TopicName => string
  Partition => int32
  Time => int64
  MaxNumberOfOffsets => int32
```

Field	Decription
Time	Used to ask for all messages before a certain time (ms). There are two special values. Specify -1 to receive the latest offset (i.e. the offset of the next coming message) and -2 to receive the earliest available offset. Note that because offsets are pulled in descending order, asking for the earliest offset will always return you a single element.

Offset Response

```
OffsetResponse => [TopicName [PartitionOffsets]]
  PartitionOffsets => Partition ErrorCode [Offset]
  Partition => int32
  ErrorCode => int16
  Offset => int64
```

Offset Commit/Fetch API

These APIs allow for centralized management of offsets. Read more Offset Management. As per comments on KAFKA-993 these API calls are not fully functional in releases until Kafka 0.8.1.1. It will be available in the 0.8.2 release.

Consumer Metadata Request

The offsets for a given consumer group are maintained by a specific broker called the offset coordinator. i.e., a consumer needs to issue its offset commit and fetch requests to this specific broker. It can discover the current offset coordinator by issuing a consumer metadata request.

```
ConsumerMetadataRequest => ConsumerGroup
  ConsumerGroup => string
```

Consumer Metadata Response

```
ConsumerMetadataResponse => ErrorCode CoordinatorId CoordinatorHost CoordinatorPort
  ErrorCode => int16
  CoordinatorId => int32
  CoordinatorHost => string
  CoordinatorPort => int32
```

Offset Commit Request

```
v0 (supported in 0.8.1 or later)
OffsetCommitRequest => ConsumerGroupId [TopicName [Partition Offset Metadata]]
  ConsumerGroupId => string
  TopicName => string
  Partition => int32
  Offset => int64
  Metadata => string


v1 (supported in 0.8.2 or later)
OffsetCommitRequest => ConsumerGroupId ConsumerGroupGenerationId ConsumerId [TopicName
[Partition Offset TimeStamp Metadata]]
  ConsumerGroupId => string
  ConsumerGroupGenerationId => int32
  ConsumerId => string
  TopicName => string
  Partition => int32
  Offset => int64
  TimeStamp => int64
  Metadata => string

v2 (supported in 0.8.3 or later)
OffsetCommitRequest => ConsumerGroup ConsumerGroupGenerationId ConsumerId
RetentionTime [TopicName [Partition Offset Metadata]]
  ConsumerGroupId => string
  ConsumerGroupGenerationId => int32
  ConsumerId => string
  RetentionTime => int64
  TopicName => string
  Partition => int32
  Offset => int64
  Metadata => string
```

In v0 and v1, the time stamp of each partition is defined as the commit time stamp, and the offset coordinator will retain the committed offset until its commit time stamp + offset retention time specified in the broker config; if the time stamp field is not set, brokers will set the commit time as the receive time before committing the offset, users can explicitly set the commit time stamp if they want to retain the committed offset longer on the broker than the configured offset retention time.

In v2, we removed the time stamp field but add a global retention time field (see KAFKA-1634 for details); brokers will then always set the commit time stamp as the receive time, but the committed offset can be retained until its commit time stamp + user specified retention time in the commit request. If the retention time is not set, the broker offset retention time will be used as default.

Offset Commit Response

```
v0, v1 and v2:
OffsetCommitResponse => [TopicName [Partition ErrorCode]]]
  TopicName => string
  Partition => int32
  ErrorCode => int16
```

Offset Fetch Request

Per the comment on KAFKA-1841 - OffsetCommitRequest API - timestamp field is not versioned RESOLVED , v0 and v1 are identical on the wire, but v0 (supported in 0.8.1 or later) reads offsets from zookeeper, while v1 (supported in 0.8.2 or later) reads offsets from kafka.

```
OffsetFetchRequest => ConsumerGroup [TopicName [Partition]]
  ConsumerGroup => string
  TopicName => string
  Partition => int32
```

Offset Fetch Response

```
OffsetFetchResponse => [TopicName [Partition Offset Metadata ErrorCode]]
  TopicName => string
  Partition => int32
  Offset => int64
  Metadata => string
  ErrorCode => int16
```

Note that if there is no offset associated with a topic-partition under that consumer group the broker does not set an error code (since it is not really an error), but returns empty metadata and sets the offset field to -1.

Constants

Api Keys

The following are the numeric codes that the ApiKey in the request can take for each of the above request types.

API name	ApiKey Value
ProduceRequest	0
FetchRequest	1
OffsetRequest	2
MetadataRequest	3
Non-user facing control APIs	4-7
OffsetCommitRequest	8
OffsetFetchRequest	9
ConsumerMetadataRequest	10

Error Codes

We use numeric codes to indicate what problem occurred on the server. These can be translated by the client into exceptions or whatever the

appropriate error handling mechanism in the client language. Here is a table of the error codes currently in use:

Error	Code	Description
NoError	0	No error--it worked!
Unknown	-1	An unexpected server error
OffsetOutOfRange	1	The requested offset is outside the range of offsets maintained by the server for the given topic/partition.
InvalidMessage	2	This indicates that a message contents does not match its CRC
UnknownTopicOrPartition	3	This request is for a topic or partition that does not exist on this broker.
InvalidMessageSize	4	The message has a negative size
LeaderNotAvailable	5	This error is thrown if we are in the middle of a leadership election and there is currently no leader for this partition and hence it is unavailable for writes.
NotLeaderForPartition	6	This error is thrown if the client attempts to send messages to a replica that is not the leader for some partition. It indicates that the clients metadata is out of date.
RequestTimedOut	7	This error is thrown if the request exceeds the user-specified time limit in the request.
BrokerNotAvailable	8	This is not a client facing error and is used mostly by tools when a broker is not alive.
ReplicaNotAvailable	9	If replica is expected on a broker, but is not (this can be safely ignored).
MessageSizeTooLarge	10	The server has a configurable maximum message size to avoid unbounded memory allocation. This error is thrown if the client attempt to produce a message larger than this maximum.
StaleControllerEpochCode	11	Internal error code for broker-to-broker communication.
OffsetMetadataTooLargeCode	12	If you specify a string larger than configured maximum for offset metadata
OffsetsLoadInProgressCode	14	The broker returns this error code for an offset fetch request if it is still loading offsets (after a leader change for that offsets topic partition).
ConsumerCoordinatorNotAvailableCode	15	The broker returns this error code for consumer metadata requests or offset commit requests if the offsets topic has not yet been created.
NotCoordinatorForConsumerCode	16	The broker returns this error code if it receives an offset fetch or commit request for a consumer group that it is not a coordinator for.

Some Common Philosophical Questions

Some people have asked why we don't use HTTP. There are a number of reasons, the best is that client implementors can make use of some of the more advanced TCP features--the ability to multiplex requests, the ability to simultaneously poll many connections, etc. We have also found HTTP libraries in many languages to be surprisingly shabby.

Others have asked if maybe we shouldn't support many different protocols. Prior experience with this was that it makes it very hard to add and test new features if they have to be ported across many protocol implementations. Our feeling is that most users don't really see multiple protocols as a feature, they just want a good reliable client in the language of their choice.

Another question is why we don't adopt XMPP, STOMP, AMQP or an existing protocol. The answer to this varies by protocol, but in general the problem is that the protocol does determine large parts of the implementation and we couldn't do what we are doing if we didn't have control over the protocol. Our belief is that it is possible to do better than existing messaging systems have in providing a truly distributed messaging system, and to do this we need to build something that works differently.

A final question is why we don't use a system like Protocol Buffers or Thrift to define our request messages. These packages excel at helping you to managing lots and lots of serialized messages. However we have only a few messages. Support across languages is somewhat spotty (depending on the package). Finally the mapping between binary log format and wire protocol is something we manage somewhat carefully and this would not be possible with these systems. Finally we prefer the style of versioning APIs explicitly and checking this to inferring new values as nulls as it allows more nuanced control of compatibility.