# Writing a Driver for Kafka

## Writing a Driver for Kafka

This is an attempt to document Kafka's wire format and implementation details to encourage others to write drivers in various languages. Much of this document is based off of the Kafka Wire Format article by Jeffrey Damick and Taylor Gautier.

### Status of this Document

I'm currently in the process of verifying many of the things said here, to make sure they're actually a result of the protocol and not some quirk of our driver. I've tried to flag those with "FIXME" notes, but I'm sure I've missed a few.

I really want to make this into the document that I wish we had at Datadog when we first started working on Kafka driver code. Corrections and comments would be greatly appreciated. Please drop me an email at dave@datadoghq.com.

### Ground Rules

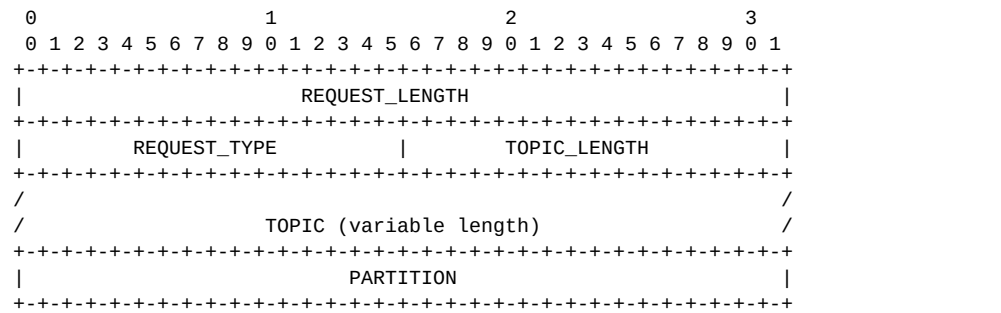
If you haven't read the design doc, read it. There are a few things that are outdated, but it's a great overview of Kafka.

Some really high level takeaways to get started:

- Kafka has topics, and topics have numbered partitions starting from 0. A topic can be created at runtime just by writing to it, but the number of partitions per topic is determined by broker configuration.
- Kafka stores messages on disk, in a series of large, append-only log files broken up into segments. Each topic+partition is a directory of these segment files. For more details, see What are Segment Files.
- An offset is just the byte offset in a given log for a topic+partition. The messages don't have any other unique identifier. They're simply stored back to back in the segment files, and you ask for them by their byte offset.
- When producing messages, the driver has to specify what topic and partition to send the message to. When requesting messages, the driver has to specify what topic, partition, *and offset* it wants them pulled from.
- While you can request "old" messages if you know their topic, partition, and offset, Kafka does not have a message index. You cannot efficiently query Kafka for the N-1000th message, or ask for all messages written between 30 and 35 minutes ago.
- Kafka tends to do the simplest thing possible and relies on smarter clients to keep bookkeeping. The broker does not keep track of what the client has read. More advanced setups use ZooKeeper to help with this tracking, but that is currently beyond the scope of this document.
- The protocol is a work in progress, and new point releases can introduce backwards incompatibile changes.
- The broker runs on port 9092 by default.
- If you are testing with multiple brokers on the same machine, you'll need to change both the port that they listen on (in the config file), as well as the JMX port they use. To specify a different JMX port, set the environment property JMX_PORT when starting Kafka.

### Basic Objects

**Request Header (all single non-multi requests begin with this)**



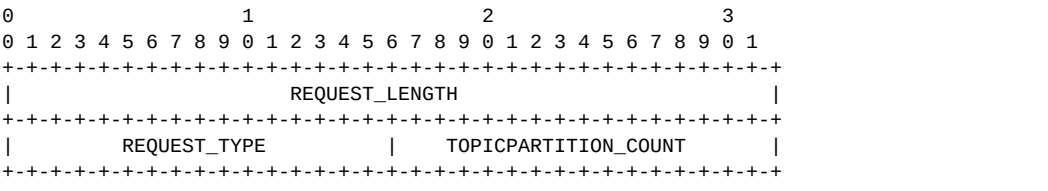
```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      REQUEST_LENGTH                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         REQUEST_TYPE          |        TOPIC_LENGTH           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
/                                                               /
/                    TOPIC (variable length)                    /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           PARTITION                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
REQUEST_LENGTH = int32 // Length in bytes of entire request (excluding this field)
REQUEST_TYPE   = int16 // See table below
TOPIC_LENGTH   = int16 // Length in bytes of the topic name

TOPIC = String // Topic name, ASCII, not null terminated
               // This becomes the name of a directory on the broker, so no
               // chars that would be illegal on the filesystem.

PARTITION = int32 // Partition to act on. Number of available partitions is
                  // controlled by broker config. Partition numbering
                  // starts at 0.
```

| REQUEST_TYPE | VALUE | DEFINITION |
|---|---|---|
| PRODUCE | 0 | Send a group of messages to a topic and partition. |
| FETCH | 1 | Fetch a group of messages from a topic and partition. |
| MULTIFETCH | 2 | Multiple FETCH requests, chained together |
| MULTIPRODUCE | 3 | Multiple PRODUCE requests, chained together |
| OFFSETS | 4 | Find offsets before a certain time (this can be a bit misleading, please read the details of this request). |

Very similar to the Request-Header is the multi-request header used for requesting more than one topic-partition combo at a time. Either for multi-produce, or multi-fetch.

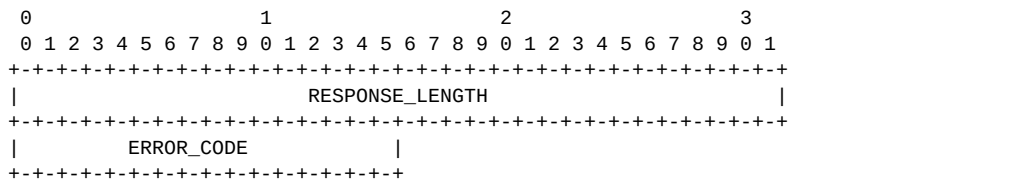
**Multi-Request Header (more than one topic-partition combo)**



```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      REQUEST_LENGTH                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         REQUEST_TYPE          |    TOPICPARTITION_COUNT       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
REQUEST_LENGTH       = int32 // Length in bytes of entire request (excluding this field)
REQUEST_TYPE         = int16 // See table above
TOPICPARTITION_COUNT = int16 // number of unique topic-partition combos in this request
```

**Response Header (all responses begin with this 6 byte header)**



```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        RESPONSE_LENGTH                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         ERROR_CODE            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

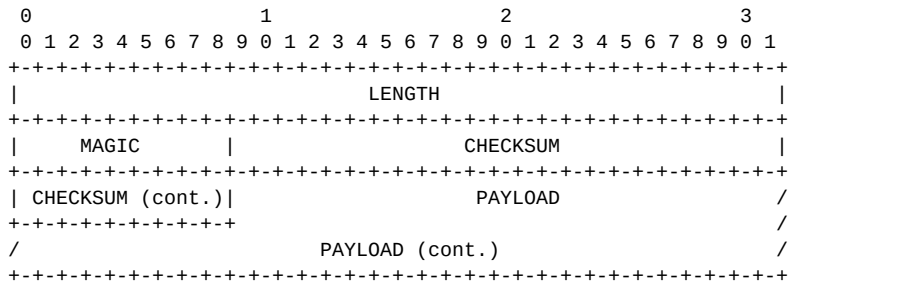
```
RESPONSE_LENGTH = int32 // Length in bytes of entire response (excluding this field)
ERROR_CODE = int16 // See table below.
```

| ERROR_CODE | VALUE | DEFINITION |
|---|---|---|
| Unknown | -1 | Unknown Error |
| NoError | 0 | Success |
| OffsetOutOfRange | 1 | Offset requested is no longer available on the server |
| InvalidMessage | 2 | A message you sent failed its checksum and is corrupt. |
| WrongPartition | 3 | You tried to access a partition that doesn't exist (was not between 0 and (num_partitions - 1)). |
| InvalidFetchSize | 4 | The size you requested for fetching is smaller than the message you're trying to fetch. |

FIXME: Add tests to verify all these codes.

FIXME: Check that there weren't more codes added in 0.7.

**Message (Kafka 0.6 and earlier)**



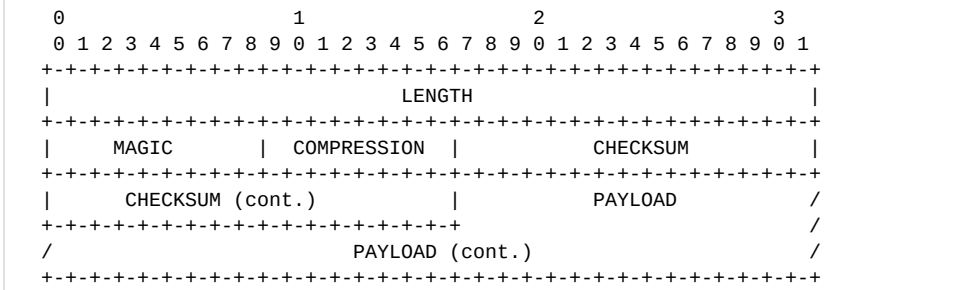
```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            LENGTH                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     MAGIC       |                  CHECKSUM                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| CHECKSUM (cont.)|                   PAYLOAD                  /
+-+-+-+-+-+-+-+-+-+                                           /
/                         PAYLOAD (cont.)                     /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
LENGTH   = int32 // Length in bytes of entire message (excluding this field)
MAGIC    = int8  // 0 is the only valid value
CHECKSUM = int32 // CRC32 checksum of the PAYLOAD
PAYLOAD  = Bytes[] // Message content
```

The offsets to request messages are just byte offsets. To find the offset of the next message, take the offset of this message (that you made in the request), and add LENGTH + 4 bytes (length of this message + 4 byte header to represent the length of this message).

Starting with version 0.7, Kafka added an extra field for compression:

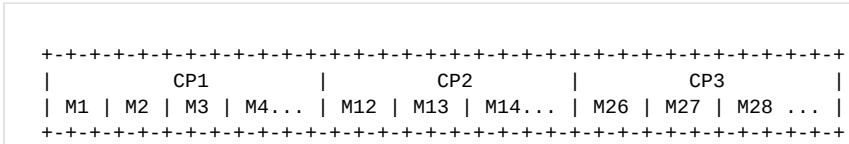
**Message (Kafka 0.7 and later)**




```
LENGTH = int32 // Length in bytes of entire message (excluding this field)
MAGIC = int8 // 0 = COMPRESSION attribute byte does not exist (v0.6 and below)
             // 1 = COMPRESSION attribute byte exists (v0.7 and above)
COMPRESSION = int8 // 0 = none; 1 = gzip; 2 = snappy;
                   // Only exists at all if MAGIC == 1
CHECKSUM = int32  // CRC32 checksum of the PAYLOAD
PAYLOAD = Bytes[] // Message content
```

Note that compression is end-to-end. Meaning that the Producer is responsible for sending the compressed payload, it's stored compressed on the broker, and the Consumer is responsible for decompressing it. Gzip gives better compression ratio, snappy gives faster performance.

Let's look at what compressed messages act like:




In this scenario, let's say that `M1`, `M2`, etc. represent complete, *uncompressed* messages (including headers) that the user of your library wants to send. What your driver needs to do is take `M1`, `M2`... up to some predetermined number/size, concatenate them together, and then compress them using gzip or snappy. The result (`CP1` in this case) becomes the `PAYLOAD` for the *compressed* message `CM1` that your library will send to Kafka.

It also means that we have to be careful about calculating the offsets. To Kafka, `M1`, `M2`, don't really exist. It only sees the `CM1` you send. So when you make calculations for the offset you can fetch next, you have to make sure you're doing it on the boundaries of the compressed messages, not the inner messages.

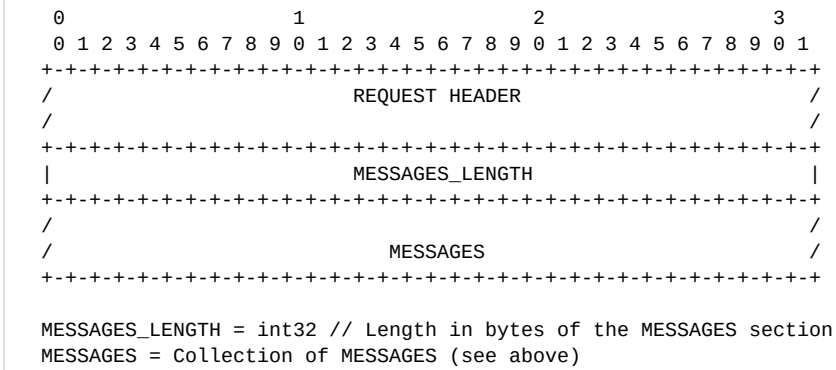
FIXME: Haven't implemented compression yet, need to verify this is correct.

# Interactions

## Produce

To produce messages from the driver and send to Kafka, use the following format:

**Produce Request**

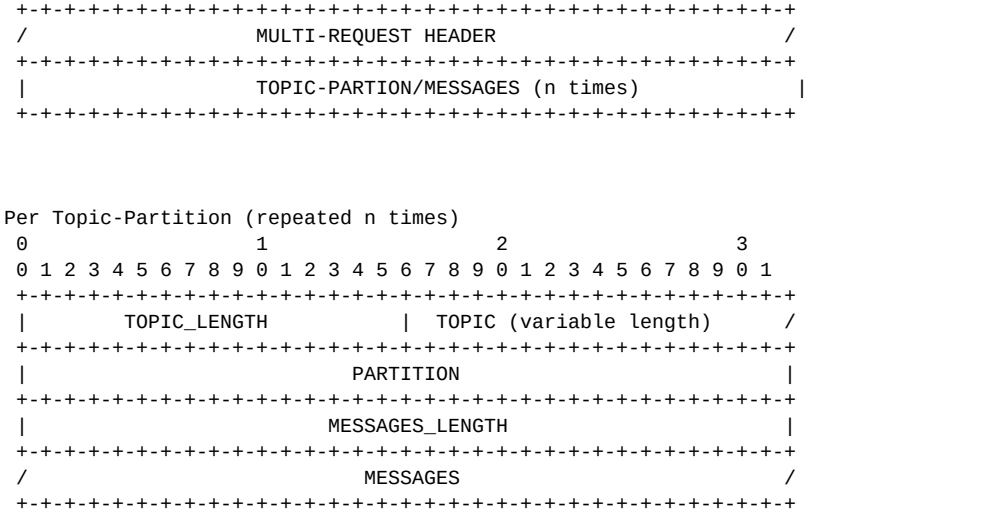



There is no response to a PRODUCE Request. There is currently no way to tell if the produce was successful or not. This is being worked.

## Multi-Produce

The multi-produce request has a different header, with the (topic-length/topic/message_length/messages) repeated many times.

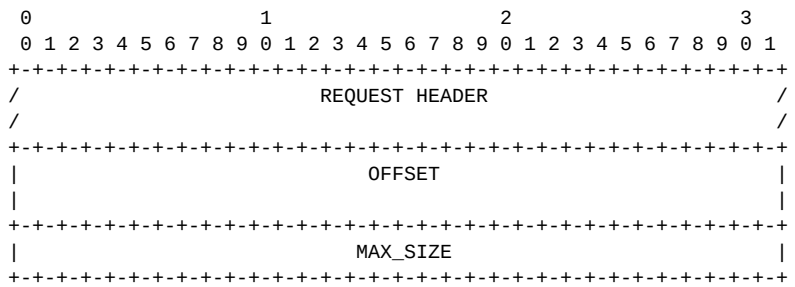
**Multi-Produce Request**




## Fetch

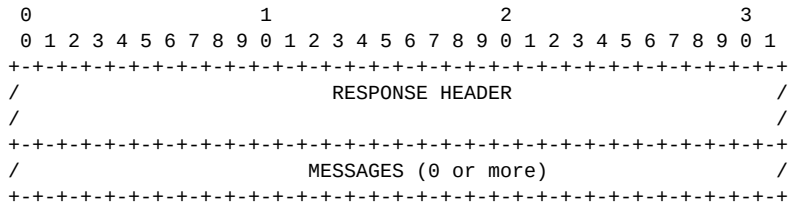
Reading messages from a specific topic/partition combination.

**Fetch Request**



```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
/                         REQUEST HEADER                        /
/                                                               /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             OFFSET                            |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            MAX_SIZE                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

REQUEST_HEADER = See REQUEST_HEADER above
OFFSET   = int64 // Offset in topic and partition to start from
MAX_SIZE = int32 // MAX_SIZE of the message set to return
```

**Fetch Response**



```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
/                         RESPONSE HEADER                       /
/                                                               /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
/                       MESSAGES (0 or more)                    /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Edge case behavior:

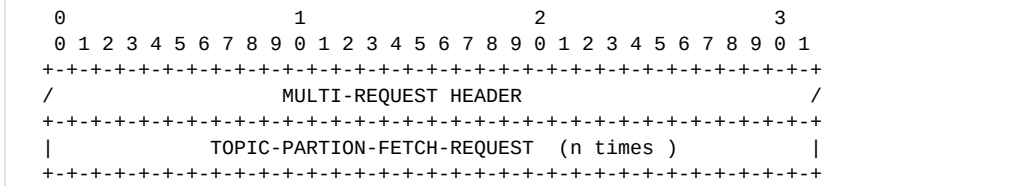
- If you request an offset that does not exist for that topic/partition combination, you will get an OffsetOutOfRange error. While Kafka keeps messages persistent on disk, it also deletes old log files to save space.
- FIXME: VERIFY – If you request a fetch from a partition that does not exist, you will get a WrongPartition error.
- FIXME: VERIFY – If the MAX_SIZE you specify is smaller than the largest message that would be fetched, you will get an InvalidFetchSize error.
- FIXME: VERIFY – What happens when you ask for an offset that's in the middle of a message? It just sends you the chunk without checking?
- FIXME – Try invalid topic, invalid partition reading
- FIXME – Look at InvalidMessageSizeException

Normal, but possibly unexpected behavior:

- If you ask the broker for up to 300K worth of messages from a given topic and partition, it will send you the appropriate headers followed by a 300K chunk worth of the message log. If 300K ends in the middle of a message, you get half a message at the end. If it ends halfway through a message header, you get a broken header. This is not an error, this is Kafka pushing complexity outward to the driver to make the broker simple and fast.
- Kafka stores its messages in log files of a configurable size (512MB by default) called segments. A fetch of messages will not cross the segment boundary to read from multiple files. So if you ask for a fetch of 300K's worth of messages and the offset you give is such that there's only one message at the end of that segment file, then you will get just one message back. The next time you call fetch with the following offset, you'll get a full set of messages from the next segment file. Basically, don't make any assumptions about how many messages are remaining from how many you got in the last fetch.

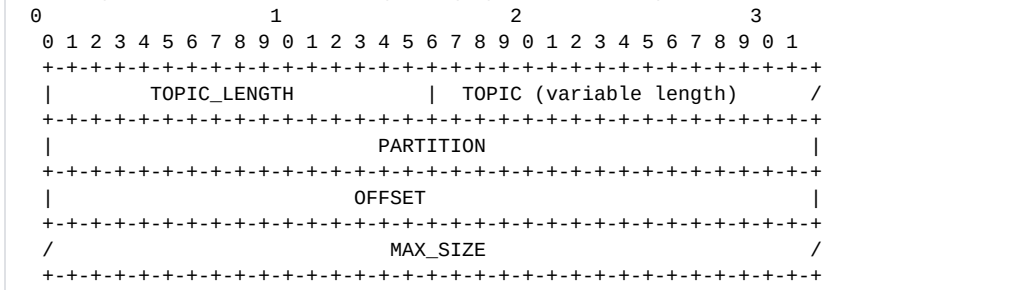
## Multi-Fetch

**Multi-Fetch Request**



```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
/                   MULTI-REQUEST HEADER                        /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             TOPIC-PARTION-FETCH-REQUEST  (n times )           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

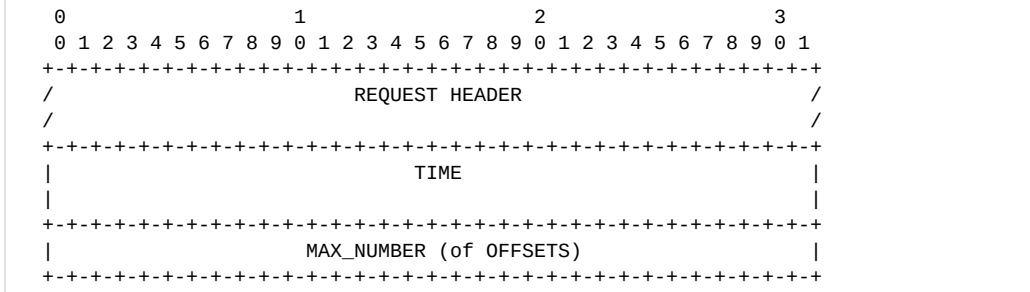
```
REQUEST_HEADER = See MULTI_REQUEST_HEADER above
OFFSET   = int64 // Offset in topic and partition to start from
MAX_SIZE = int32 // MAX_SIZE of the message set to return
The TOPIC_LENGTH, TOPIC, PARTITION, MESSAGES_LENGTH are documented above for size.
```



```
Per Topic-Partition-Fetch- Request (repeated n times)
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        TOPIC_LENGTH           |  TOPIC (variable length)      /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            PARTITION                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         OFFSET                                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
/                            MAX_SIZE                           /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

## Offsets

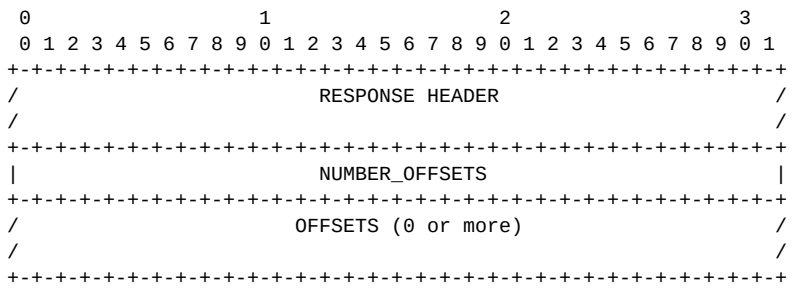
**Offsets Request**



```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
/                         REQUEST HEADER                        /
/                                                               /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                              TIME                             |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     MAX_NUMBER (of OFFSETS)                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
TIME = int64 // Milliseconds since UNIX Epoch.
             // -1 = LATEST
             // -2 = EARLIEST
MAX_NUMBER = int32 // Return up to this many offsets
```

**Offsets Response**




```
NUMBER_OFFSETS = int32 // How many offsets are being returned
OFFSETS = int64[] // List of offsets
```

This one can be deceptive. It is *not* a way to get the the precise offset that occurred at a specific time. Kafka doesn't presently track things at that level of granularity, though there is a proposal to do so. To understand how this request really works, you should know how Kafka stores data. If you're unfamiliar with segment files, please see What are Segment Files.

What Kafka does here is return up to MAX_NUMBER of offsets, sorted in descending order, where the offsets are:

1. The first offset of every segment file for a given partition with a modified time less than `TIME`.
2. If the last segment file for the partition is not empty and was modified earlier than `TIME`, it will return both the first offset for that segment and the high water mark. The high water mark is not the offset of the last message, but rather the offset that the next message sent to the partition will be written to.

There are special values for `TIME` indicating the earliest (-2) and latest (-1) time, which will fetch you the first and last offsets, respectively. Note that because offsets are pulled in descending order, asking for the earliest offset will always return you a list with a single element.

Because segment files are quite large and fine granularity is not possible, you may find yourself using this call mostly to get the beginning and ending offsets.

### What are segment files?

Say your Kafka broker is configured to store its log files in `/tmp/kafka-logs` and you have a topic named "dogs", with two partitions. Kafka will create a directory for each partition:

```
/tmp/kafka-logs/dogs-0
/tmp/kafka-logs/dogs-1
```

Inside each of these partition directories, it will store the log for that topic+parition as a series of segment files. So for instance, in `dogs-0`, you might have:

```
00000000000000000000.kafka
00000000000536890406.kafka
00000000001073761356.kafka
```

Each file is named after the offset represented by the first message in that file. The size of the segments are configurable (512MB by default). Kafka will write to the current segment file until it goes over that size, and then will write the next message in new segment file. The files are actually slightly larger than the limit, because Kafka will finish writing the message – a single message is never split across multiple files.

## ZooKeeper

Kafka relies on ZooKeeper in order to coordinate multiple brokers and consumers. If you're unfamiliar with ZooKeeper, just think of it as a server that allows you to atomically create nodes in a tree, assign values to those nodes, and sign up for notifications when a node or its children get modified. Nodes can be either be permanent or ephemeral, the latter meaning that the nodes will disappear if the process that created them disconnects (after some timeout delay).

While creating the nodes we care about, you'll often need to create the intermediate nodes that they are children of. For instance, since offsets are stored at `/consumers/[consumer_group]/offsets/[topic]/[broker_id]-[partition_id]`, something has to create `/consumers`, `/consumers/[consumer_group]`, etc. All nodes have values associated with them in ZooKeeper, even if Kafka doesn't use them for anything. To make debugging easier, the value that should be stored at an intermediate node is the ID of the node's creator. In practice that means that the first Consumer you create will need to make this skeleton structure and store its ID as the value for `/consumers`, `/consumers/[consumer_group]`, etc.

ZooKeeper has Java and C libraries, and can be run as a cluster.

## Basic Responsibilities

Kafka Brokers, Consumers, and Producers all have to coordinate using ZooKeeper. The following is a high level overview of their ZooKeeper interactions. We assume here that a Consumer only consumes one topic

Kafka Broker

- When starting up:
  - Publish its `brokerid` – a simple integer ID that uniquely identifies it.
  - Publish its location, so that Producers and Consumers can find it.
  - Publish all topics presently in the Broker, along with the number of partitions for each topic.
- Whenever a new topic is created:
  - Publish the topic, along with the number of partitions in the topic

Producer

- Read the locations for all Brokers with a given topic, so that we know where to send messages to.

Consumer

- Publish what ConsumerGroup we belong to.
- Publish our unique ID so other Consumers can see us.
- Determine which partitions on which Brokers this Consumer is responsible for, and publish its ownership of them.
- Publish what offset we've successfully read up to for every partition that we're responsible for.

Every Consumer belongs to exactly one ConsumerGroup. A Consumer can read from multiple brokers and partitions, but every unique combination of (ConsumerGroup, Broker, Topic, Partition) is read by only one and only one Consumer. ConsumerGroups allow you to easily support topic or queue semantics. If your Consumers are all in separate ConsumerGroups, each message goes to every Consumer. If all your Consumers are in the same ConsumerGroup, then each message goes to only one Consumer.

Well, for the most part. The orchestration Kafka uses guarantees at least once delivery, but has edge cases that may yield duplicate delivery even in a queue (one ConsumerGroup) arrangement.

## Kafka Broker

All these nodes are written by the Kafka Broker. Your client just needs to be able to read this broker data and understand its limitations.

| Role | ZooKeeper Path | Type | Data Description |
|---|---|---|---|
| ID Registry | `/brokers/ids/[0..N]` | Ephemeral | String in the format of "creator:host:port" of the broker. |
| Topic Registry | `/brokers/topics/[topic]/[0..N]` | Ephemeral | Number of partitions that topic has on that Broker. |

So let's take the example of the following hypothetical broker:

- Broker ID is 2 (`brokerid=2` in the Kafka config file)
- Running on IP 10.0.0.12
- Using port 9092
- Topics:
  - "dogs" with 4 partitions
  - "mutts" with 5 partitions

Then the broker would register the following:

- `/brokers/ids/2 = 10.0.0.12-1324306324402:10.0.0.12:9092`
- `/brokers/topics/dogs/2 = 4`
- `/brokers/topics/mutts/2 = 5`

Some things to note:

- Broker IDs don't have to be sequential, but they do have to be integers. They are a config setting, and not randomly generated. If a Kafka server goes offline for some reason and comes back an hour later, it should be reconnecting with the same Broker ID.
- The ZooKeeper hierarchy puts individual brokers under topics because Producers and Consumers will want to put a watch on a specific topic node, to get notifications when new brokers enter or leave the pool.
- The Broker's description is formatted such that it's `creator:host:port`. The host will also up as part of the creator because of the version of UUID that Kafka's using, but don't rely on that behavior. Always split on ":" and extract the host that will be the second element.
- These nodes are ephemeral, so if the Broker crashes or is disconnected from the network, it will automatically be removed. But this removal is not instantaneous, and it might show up for a few seconds. This can cause errors when a broker crashes and is restarted, and subsequently tries to re-create its still existent Broker ID registry node.

## Producer

Reads:

- `/brokers/topics/[topic]/[0..N]`, so that it knows what Broker IDs are available for this topic, and how many partitions they have.
- `/brokers/ids/[0..N]`, to find the address of the Brokers, so it knows how to connect to them.

Watches:

- `/brokers/topics/[topic]`, so that it knows when Brokers enter and exit the pool.
- `/brokers/ids`, so that it can update the Broker addresses in case you bring down a Broker and bring it back up under a different IP/port.

Producers are fairly straightforward (with one caveat), and a Producer never has to write anything to ZooKeeper. The basic operation goes like this:

1. A Producer is created for a topic.
2. The Producer reads the Broker-created nodes in `/brokers/ids/[0..N]` and sets up an internal mapping of Broker IDs => Kafka connections.
3. The Producer reads the nodes in `/brokers/topics/[topic]/[0..N]` to find the number of partitions it can send to for each Broker.
4. The Producer takes every Broker+Partition combination and puts them in an internal list.
5. When a Producer is asked to send a message set, it picks from one of it's Broker+Partition combinations, looks up the appropriate Broker address, and sends the message set to that Broker, for that topic and partition. The precise mechanism for choosing a destination is undefined, but debugging would probably be easier if you ordered them by Broker+Partition (e.g. "0-3") and used a hash function to pick the index you wanted to send to. You could also just make it randomly choose.

The Producer's internal mappings change when:

- When a Broker leaves the pool or a new Broker enters it.
- The number of partitions a Broker publishes for a topic changes.

The latter is actually *extremely* common, which brings us to the only tricky part about Producers – dealing with new topics.

### Creating New Topics

Topics are not pre-determined. You create them just by sending a new message to Kafka for that topic. So let's say you have a number of Brokers that have joined the pool and don't list themselves in `/brokers/topics/[topic]/[0..N]` for the topic you're interested in. They haven't done so because those topics don't exist on those Brokers yet. But our Producer knows the Brokers themselves exist, because they are in the Broker registry at `/brokers/ids/[0..N]`. We definitely need to send messages to them, but what partitions are safe to send to? Brokers can be configured differently from each other and topics can be configured on an individual basis, so there's no way to infer the definitive answer by looking at what's in ZooKeeper.

The solution is that for new topics where the number of available partitions on the Broker is unknown, you should just send to partition 0. Every Broker will at least have that one partition available. As soon as you write it and the topic comes into existence on the Broker, the Broker will publish all available partitions in ZooKeeper. You'll get notified by the watch you put on `/brokers/topics/[topic]`, and you'll add the new Broker+Partitions to your destination pool.

## Consumer

FIXME: Go over all the registration stuff that needs to happen.

### Rebalancing

Occurs: When Brokers or Consumers enter or leave the pool.

Objectives:

- All Consumers in a ConsuerGroup will come to a consensus as to who is consuming what.
- Each Broker+Topic+Partition combination is consumed by one and only one Consumer, even if it means that some Consumers don't get anything at all.
- A Consumer should try to have as many partitions on the same Broker as possible, so sort the list by [Broker ID]-[Partition] (0-0, 0-1, 0-2, etc.), and assign them in chunks.
- Consumers are sorted by their Consumer IDs. If there are three Consumers, two Brokers, and three partitions in each, the split might look like:
    - Consumer A: [0-0, 0-1]
    - Consumer B: [0-2, 1-0]
    - Consumer C: [1-1, 1-2]
- If the distribution can't be even and some Consumers must have more partitions than others, the extra partitions always go to the earlier consumers on the list. So you could have a distribution like 4-4-4-4 or 5-5-4-4, but never 4-4-4-5 or 4-5-4-4.