A Byte's Journey Through Kafka

Micah Whitacre

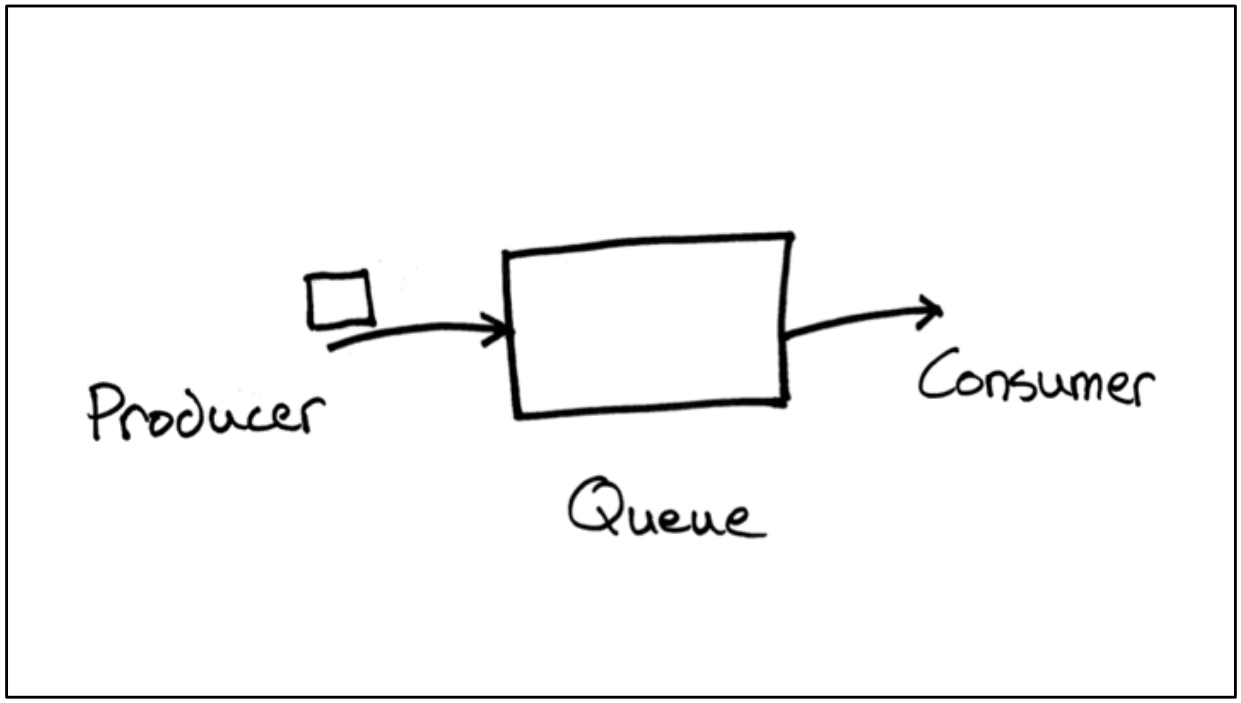
A Byte's Journey Through Kafka

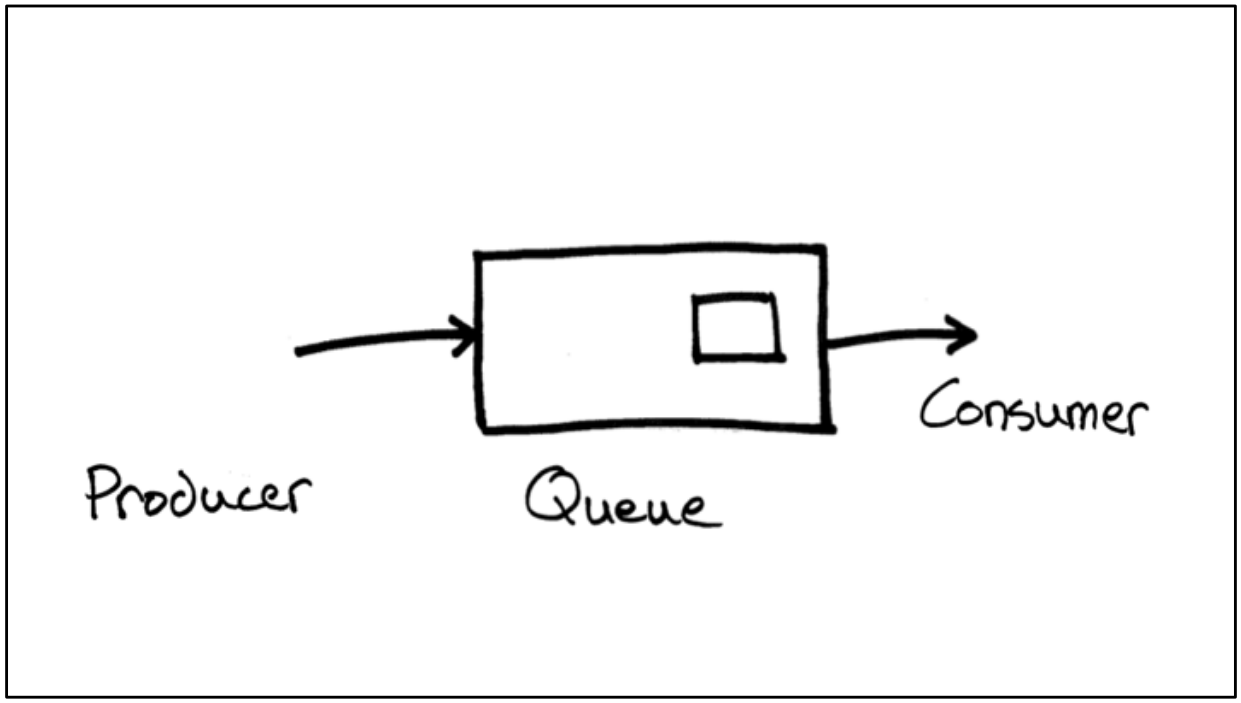
Author: Micah Whitacre
Good Art: T. Racing
Bad Art: Micah Whitacre

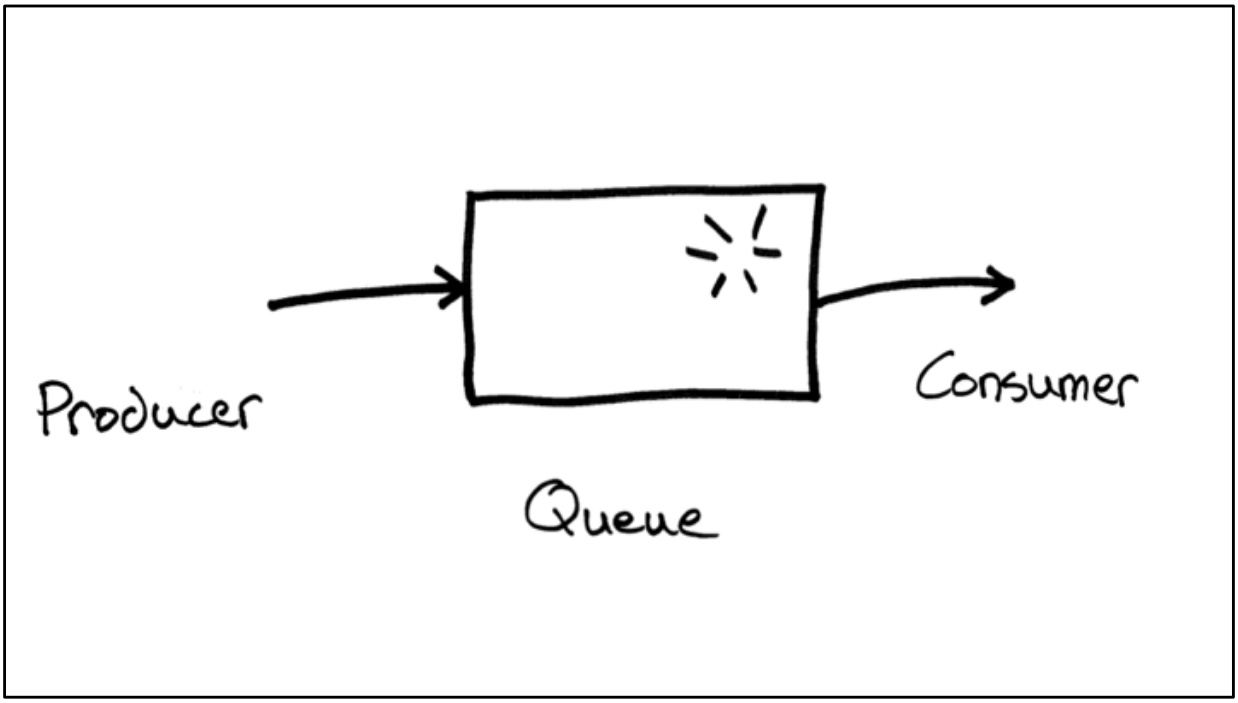
KAFKA

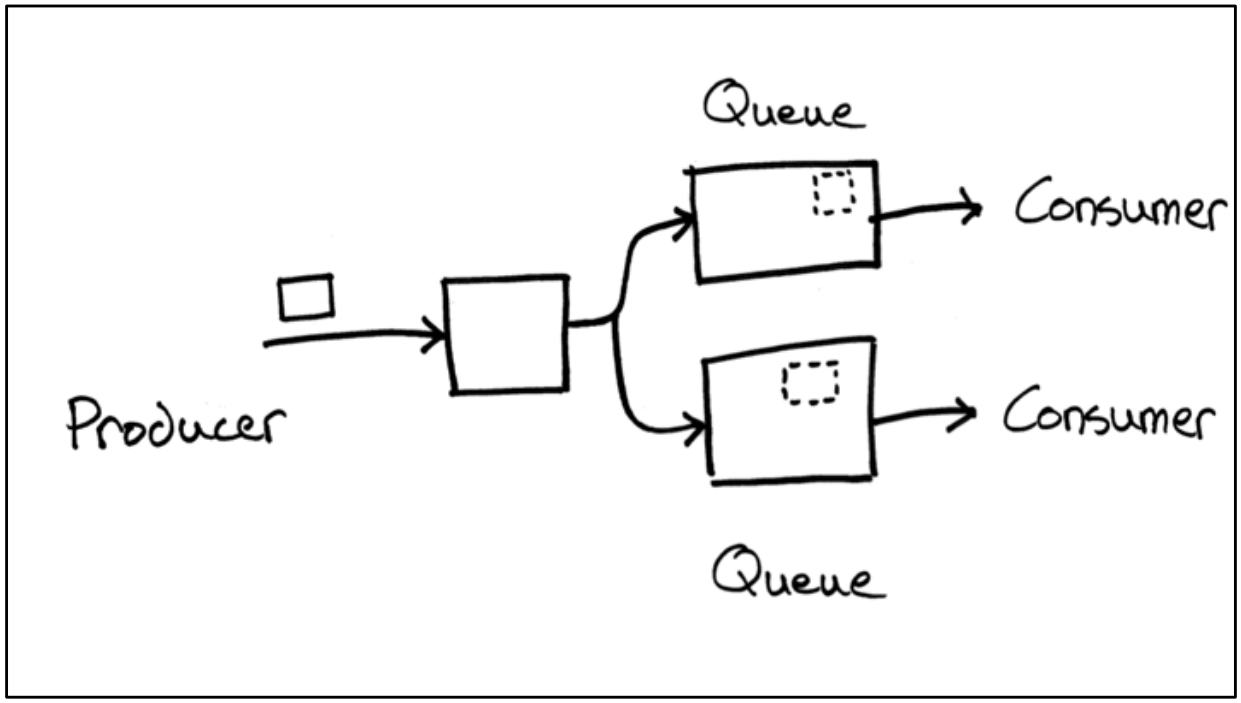
THAT'S A MESSAGE QUEUE RIGHT ??

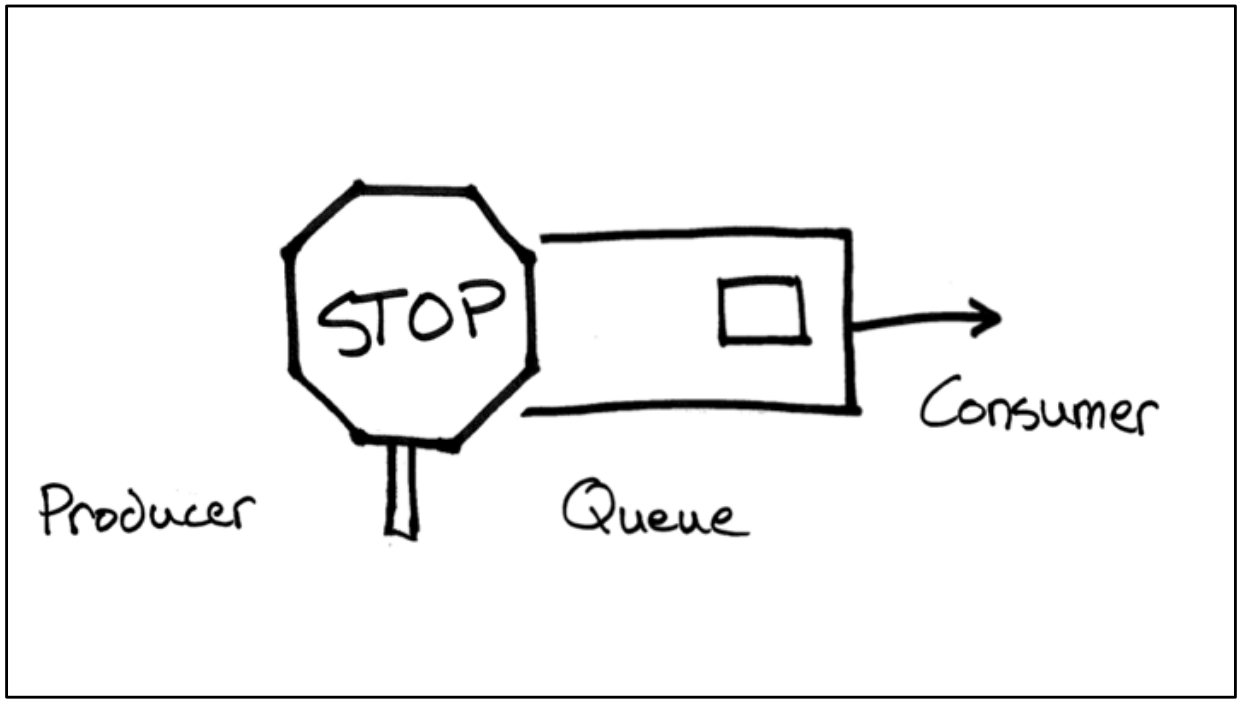

KAFKA
THAT'S A MESSAGE QUEUE RIGHT ??
EHH ... KINDA

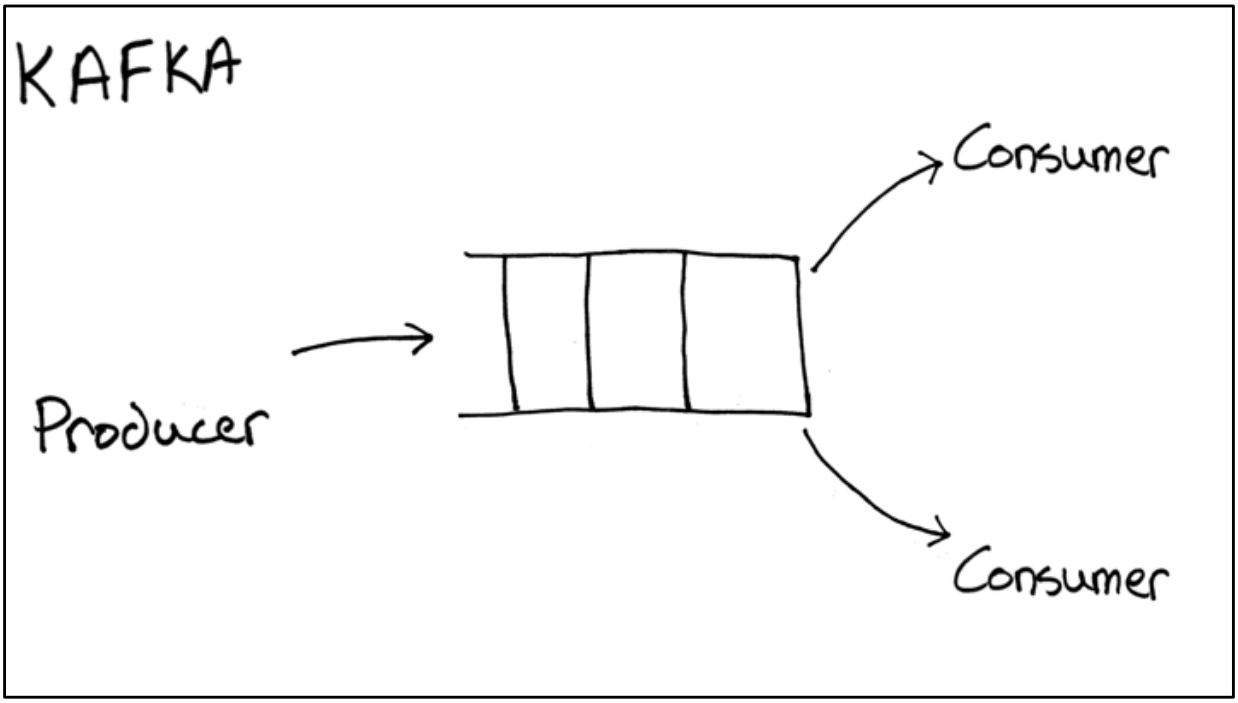

Producer
Consumer
Queue


Consumer
Producer
Queue

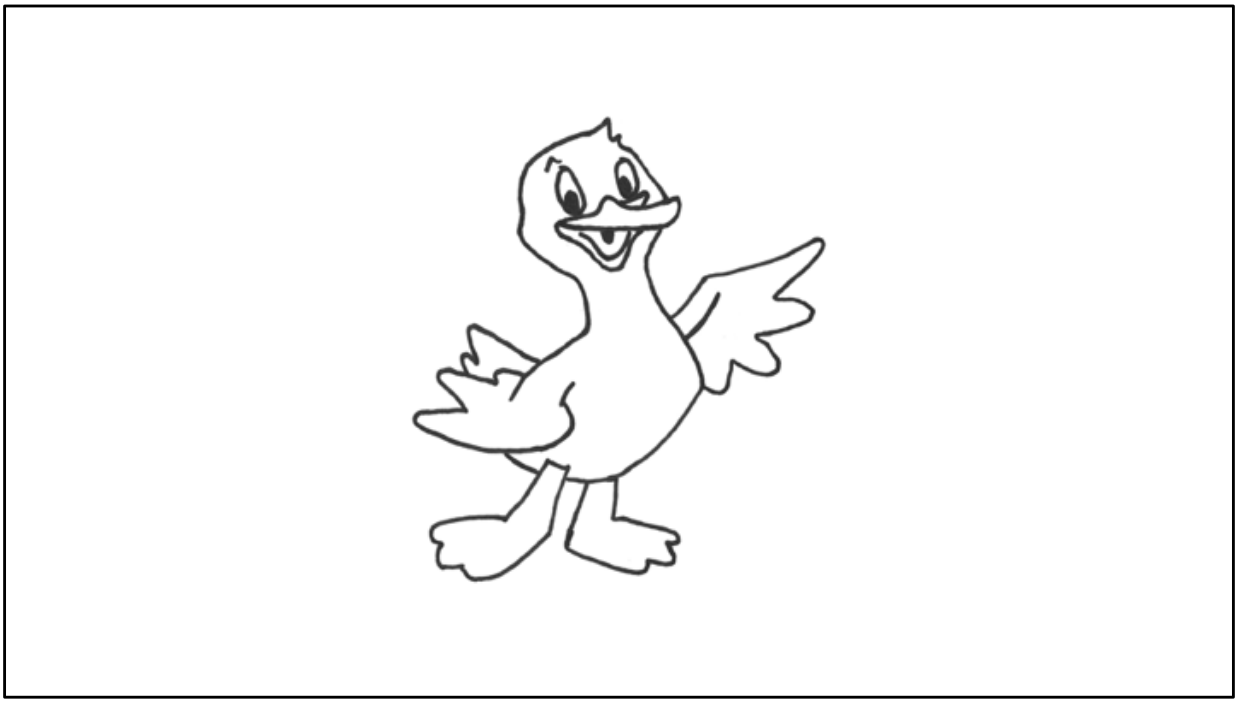

Producer
Consumer
Queue


Queue
Consumer
Producer
Consumer
Queue

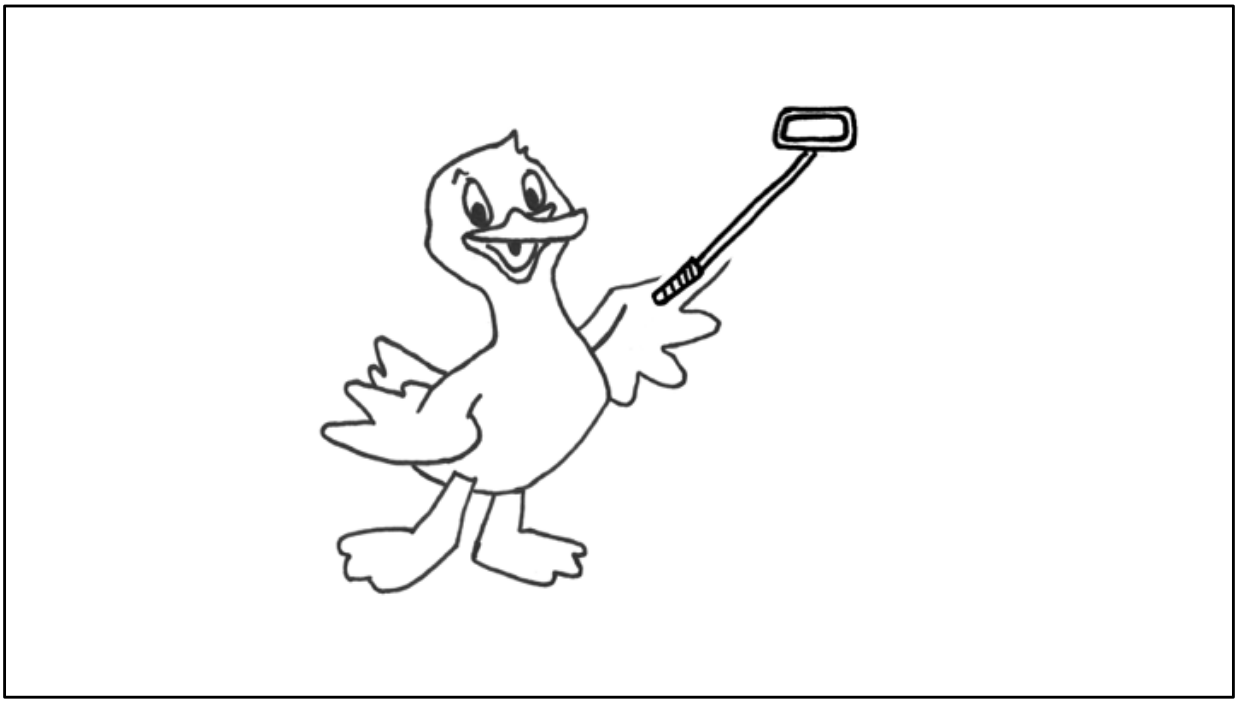

STOP
Consumer
Producer
Queue


KAFKA
Consumer
Producer
Consumer

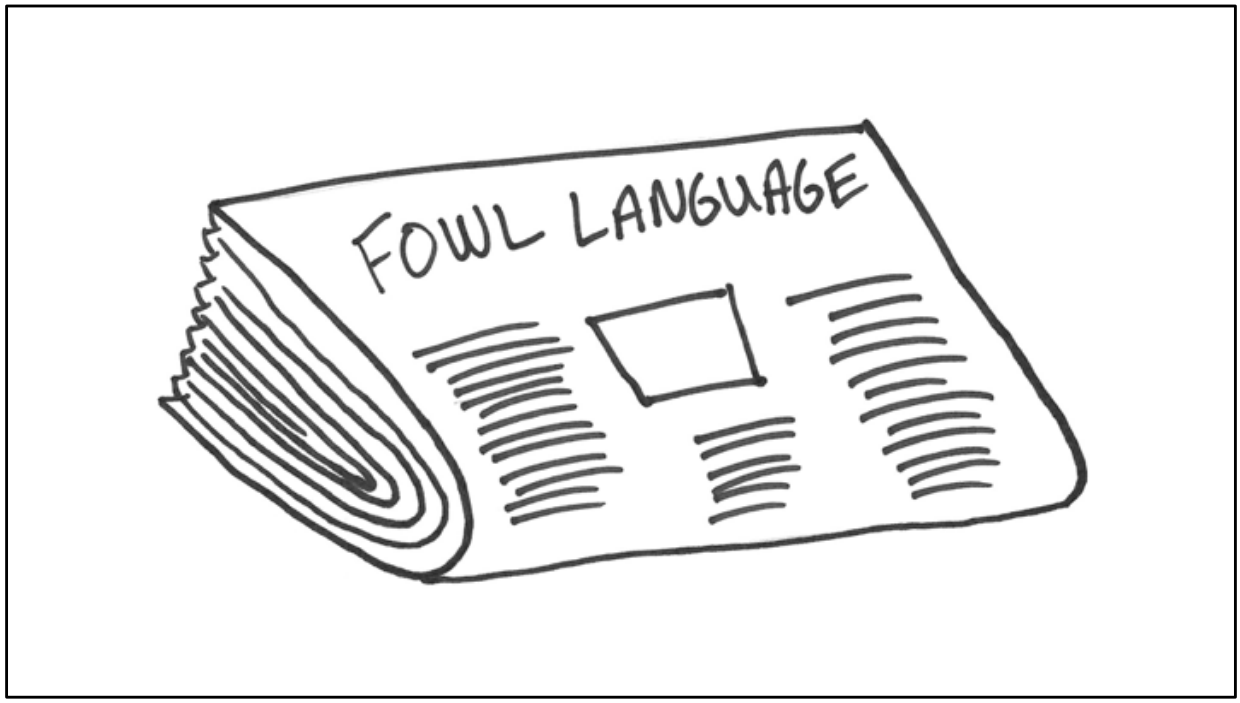

Dilly Makes A
POST!



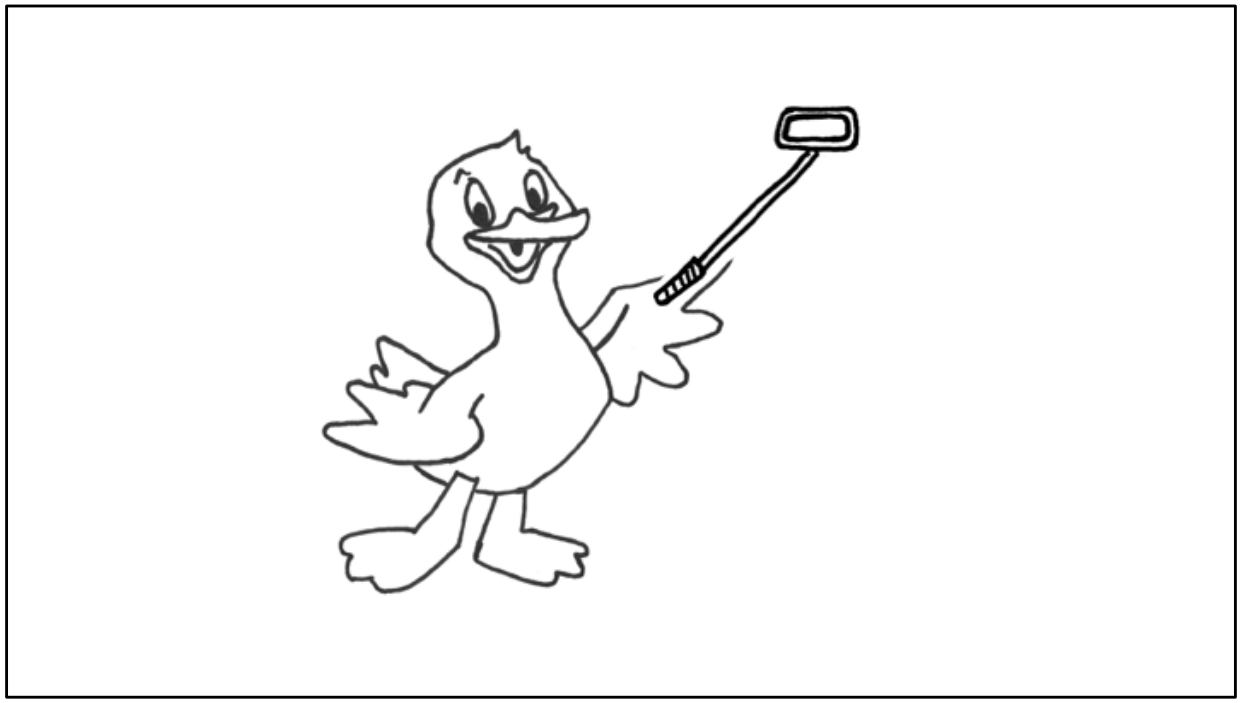
Meet Dilly! Dilly just recently graduated from school and did so while achieving no actual marketable skills. Dilly was therefore forced to be creative and fell back to doing what she knew best. If you ask other people, they will be shocked to find out Dilly's job and that it is even a real thing. Dilly on the other hand will tell you that her job is the most important job in the world as it brings joy, happiness, and let's Dilly tell the world her opinion whether they want to hear it or not. You see Dilly is a "social media influencer".



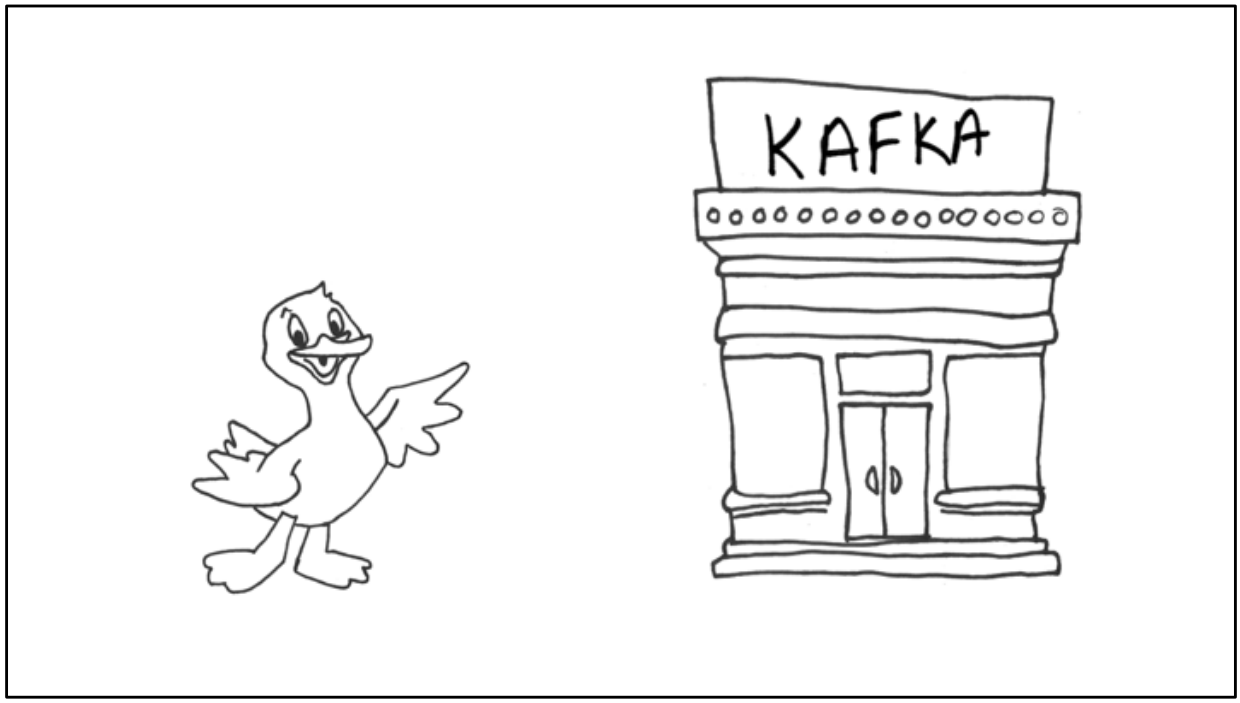
Dilly loves to share pictures, stories, and videos of all of her favorite things. While some people question how you can actually make money at this Dilly was reasonably successful and so far has had two major notable events in her career. The first is that Dilly helped pioneer the use of "duck lips" in selfies. Of course that was completely accidental as it just naturally happens based on her natural attributes. The second notable event however was not a happy moment. Instead Dilly was caught up in a scandal.



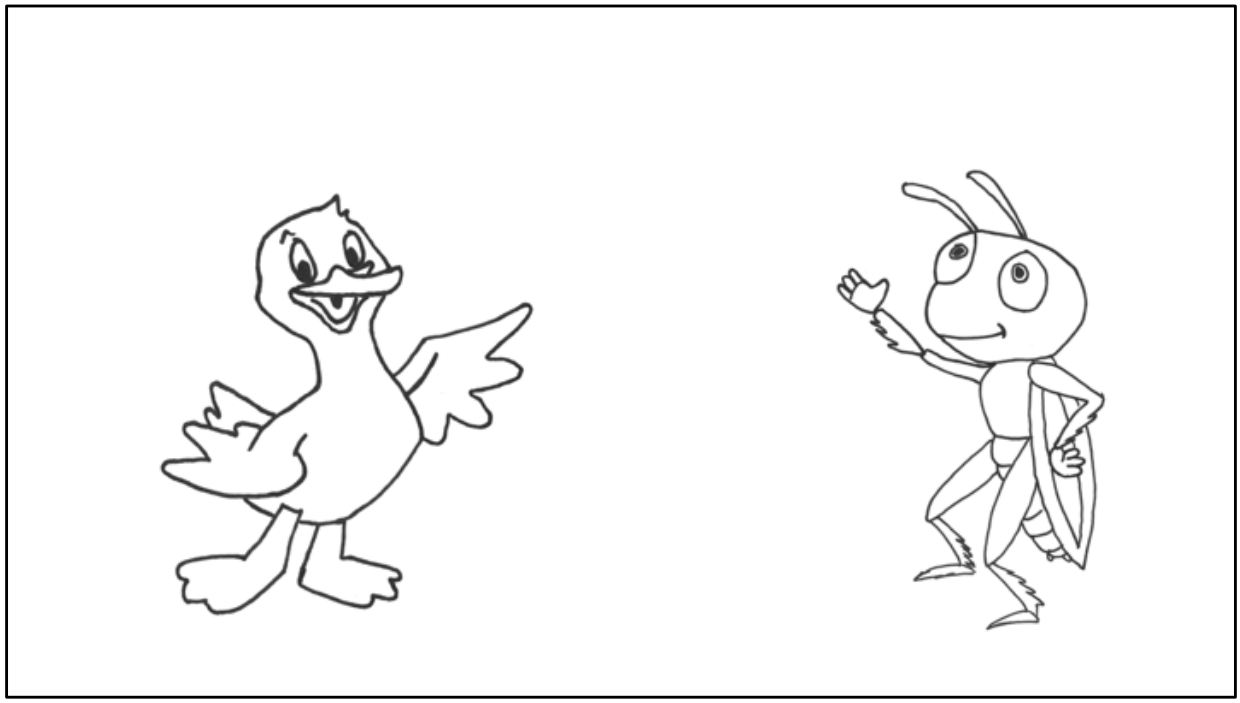
In a moment of blind rage at her favorite contestant being kicked off a reality tv show, Dilly posted a message so foul and offensive that sailors blushed and truckers cried. Newspapers printed headlines proclaiming her misdeed and all of the big names in social media quickly banned her account quickly ending her sponsorship deals and stopping her from being able to communicate with her fans. Dilly was heartbroken at the result and quickly scrambled to figure out what to do next with her life.



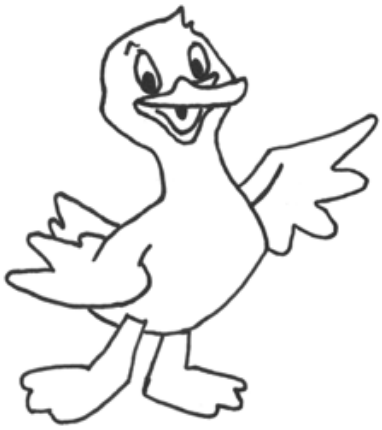
After battling periods of anxiety at having to potentially get a real job, Dilly quickly threw herself back into her "work" filling her phone with pics of food, outfits, toast that looked like famous people, and of course selfies. But despite generating so much content she struggled to figure out how to distribute it to her fans now that she wasn't allowed to post on social media. Dilly then decided to walk around town hoping to find something. She asked the butcher and the baker but they were no help. It wasn't until she asked the candlestick maker who happened to run a shop on Etsy that she got a hint about a new way to distribute her content.



The candlestick maker was “in the know” and had heard about the internal infrastructure used by a lot of silicon valley companies to distributed messages called Kafka. So Dilly went off on an adventure to visit Kafka to learn about how she could distribute her stuff to the world as she know that tens of people had to be distraught not to be seeing her selfies.

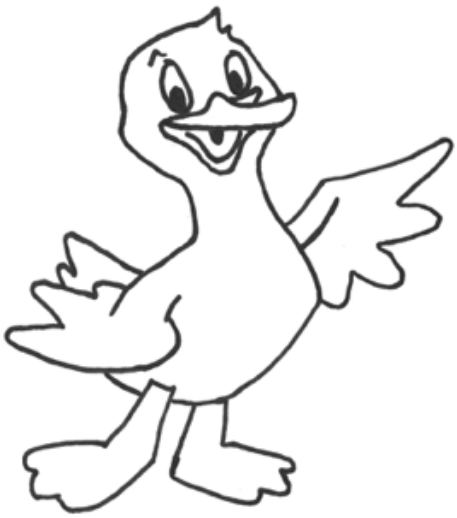


Upon entering the store Dilly was greeted by the helpful bug named, KIP. “Can you help me distribute my pics and thoughts to the world?” asked Dilly. “Of course!”, KIP quickly responded. “But I don’t know all of the people who should get the messages, is that going to be a problem?”, asked Dilly. “Not at all. Kafka is designed to support multiple producers and multiple consumers with minimal coordination between the two. We can easily support all of your current fans along with many more in the future.” replied KIP. “But how does it work?” asked Dilly.



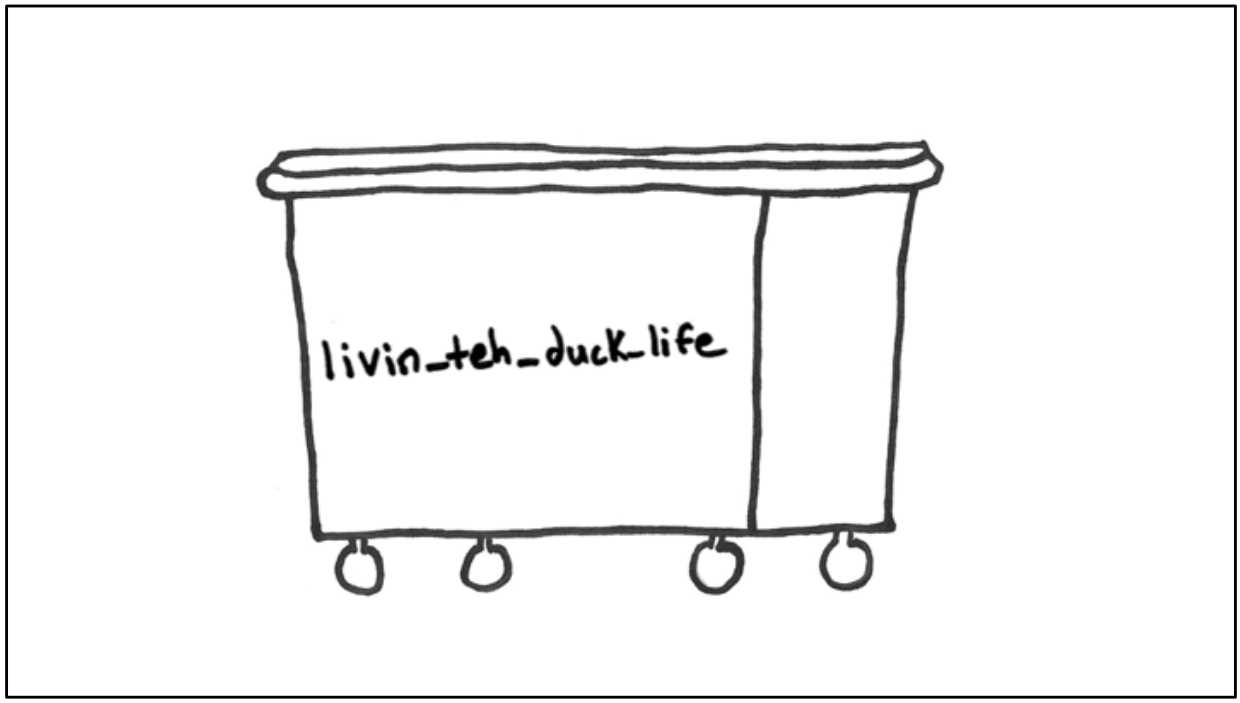
```
Properties producerProps = new Properties();
String brokers = "broker1:6667,broker2:6667,broker3:6667");
producerProps.setProperty(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, brokers);
```

As KIP was excited about Kafka he eagerly jumped right into explaining how the system worked. "Well Kafka is made up of a cluster of brokers which are used to store all the data, manage the flood of messages coming from important producers like you, and answering requests from interested consumers. To get started the first thing we need to do is help connect you to the cluster. You don't need to know about all of the brokers in the cluster but instead just a few. They will then help you connect to the rest of the cluster." KIP was so excited he did not even realize that Dilly had begun to zone out and look at shiny objects around the room.

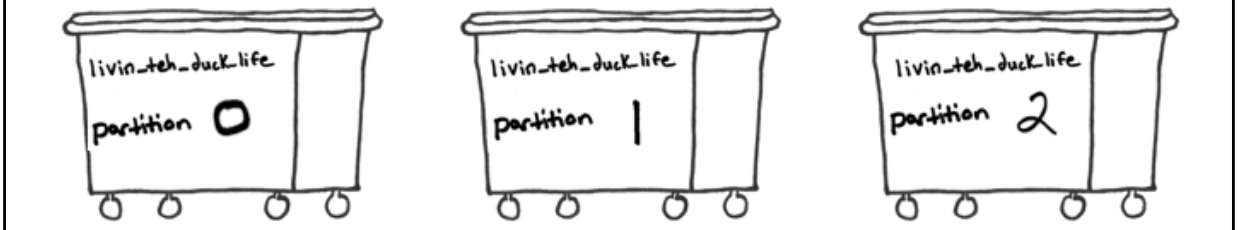


```
String topicName = "livin_teh_duck_life";
```

KIP continued to explain, “For your fans to see your posts we need to create a topic that they can subscribe to. The topic name is important as it can’t be changed and if you ever do try and change it, your fans will stop receiving your messages.” “I have the best idea for a topic name!!!” Dilly exclaimed and quickly wrote it down on the form. “Now can I send my messages?”, asked Dilly.

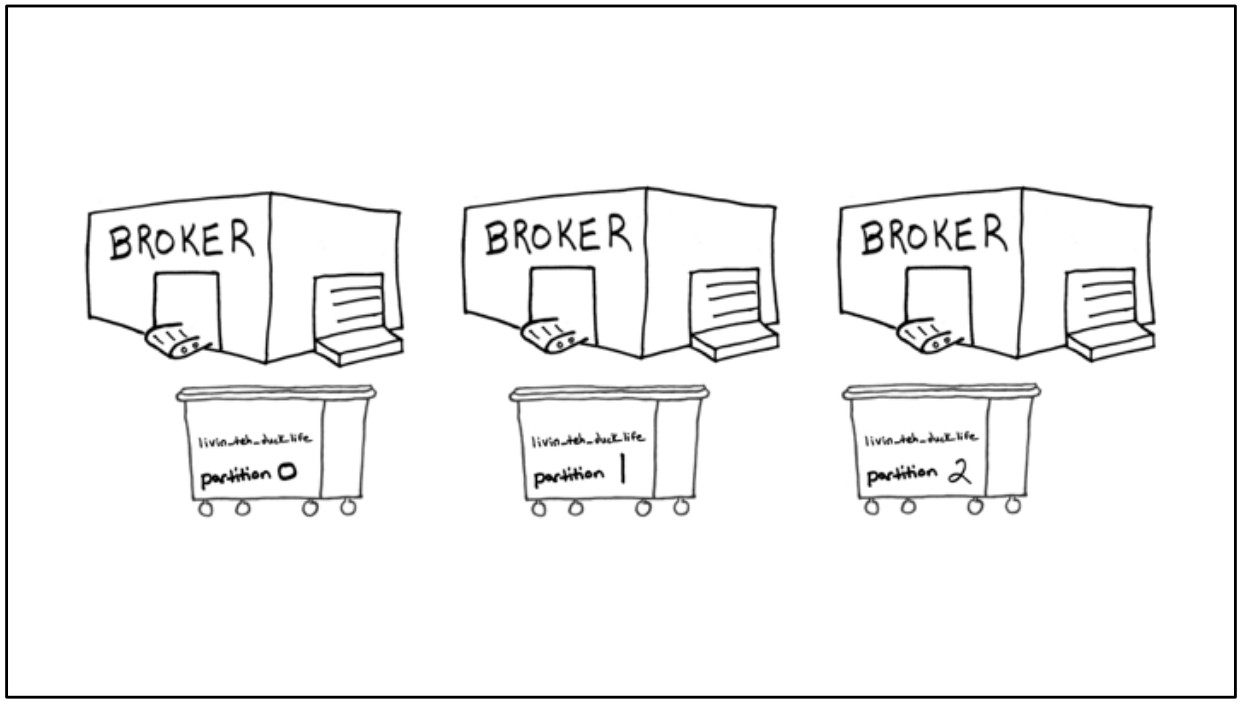


”Not quite yet.” responded KIP. “Kafka is going to store all your messages in but there are some limitations to how many messages we can store since we physically have to put them somewhere. We have various containers like this one and we have bigger ones yet but the rule is the container cannot be bigger than our building. Are you going to send lots of messages?” asked KIP. “Not too many. Likely just 20,000 per second.” answered Dilly.

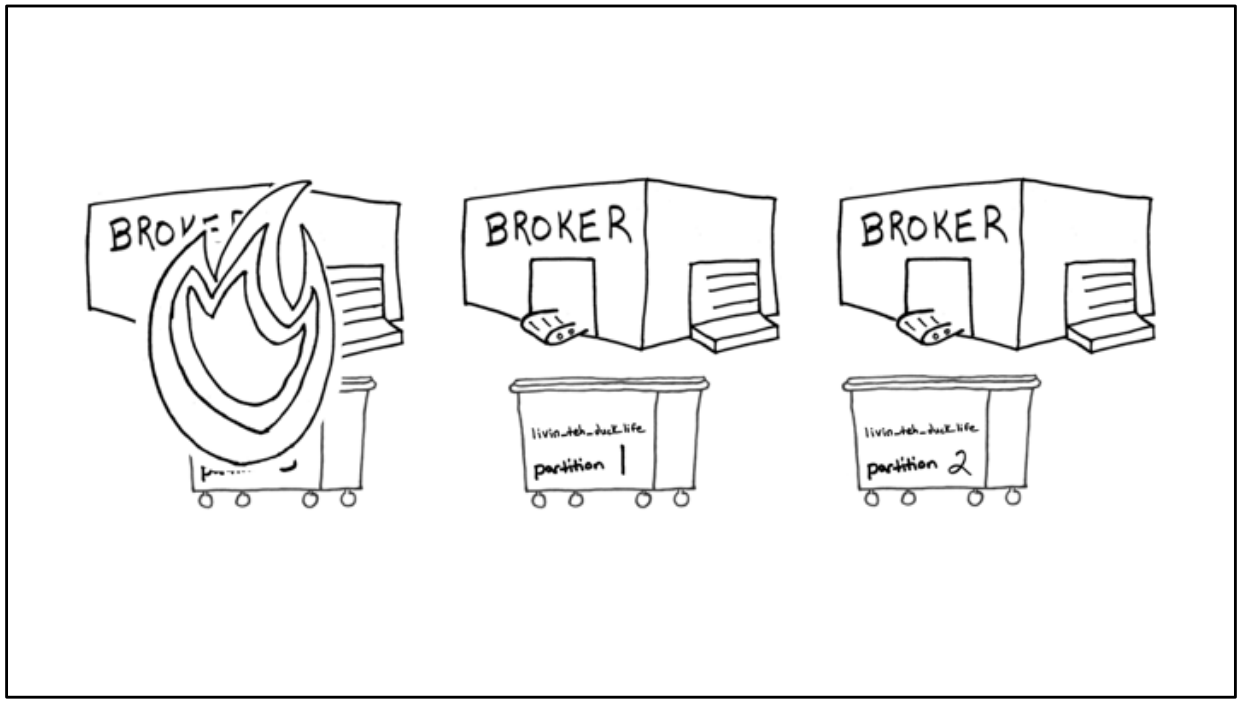



```
int numPartitions = 3;
```

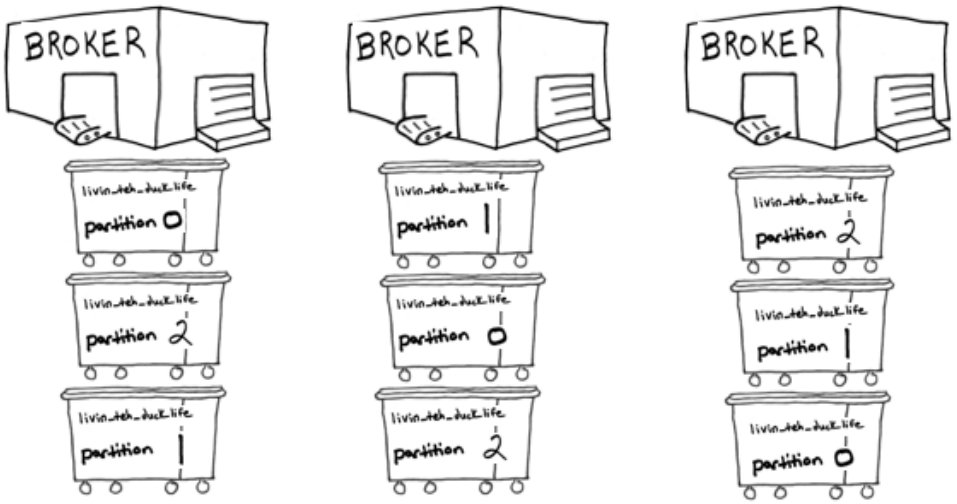
“Oh boy that is going to be a lot of messages. It would be best to split them all up into smaller groups called a partition so that instead of having one really big container we have more smaller ones that more easily fit inside out building.” suggested KIP.



”Splitting up the messages into partitions means we do not have to worry about ever filling up a single broker.” said KIP cheerfully not realizing that Dilly had started to ignore him and was practicing selfie poses. “Are these messages important? “ asked KIP. Dilly immediately snapped to attention and answered, “oh they are super important!!! Without these messages my fans will not know about the hot new trends in fashion, the newest places to go eat, and worst of all without my posts they might start to think for themselves forming their own opinions without me telling them if they like something or not.”

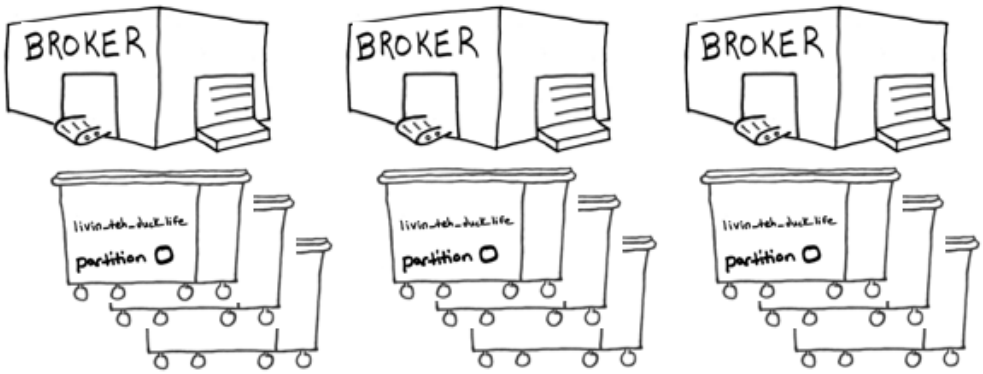


“Umm ok…well we take every precaution to secure and protect our facilities but in the case of accidents and failures, any messages on that broker when it fails will be lost” responded KIP. “That is not acceptable”, responded Dilly who quickly turned about ready to storm out. “Wait!!!” exclaimed KIP. “Let me finish. We have ways of handling those failures as well called replicas.”



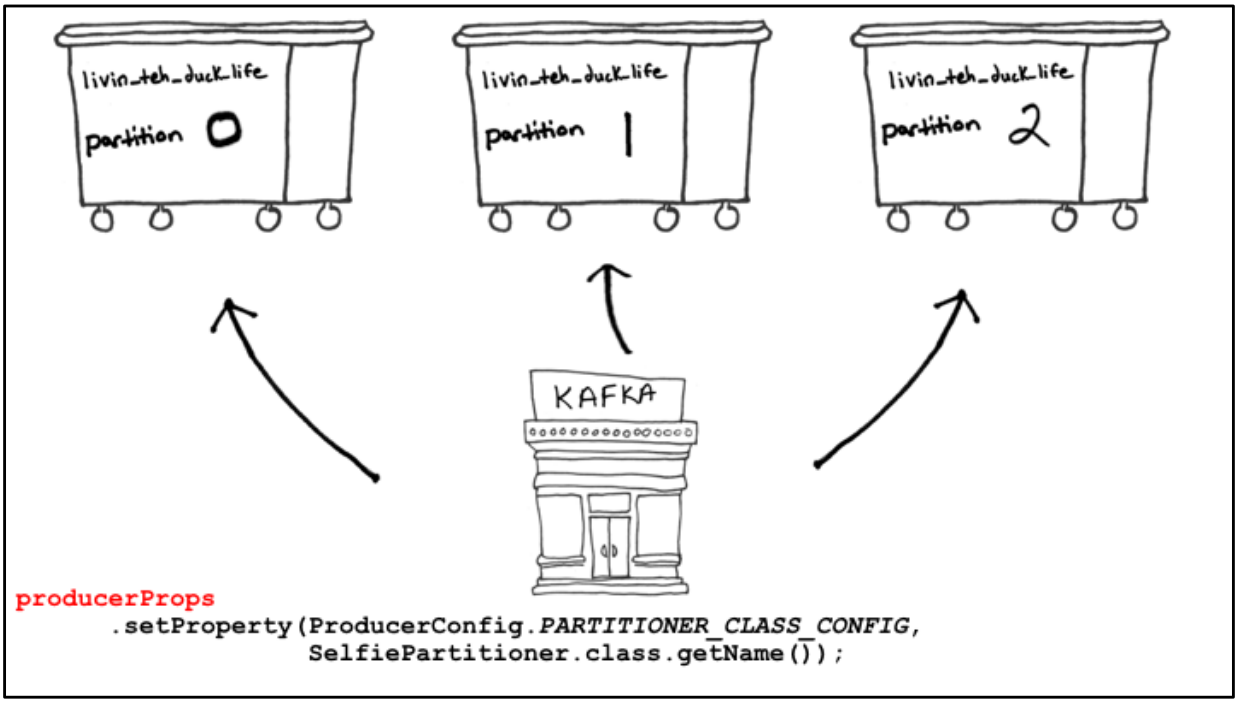

```
short numReplicas = 3;
Map<String, String> topicConfig = new HashMap<>();
topicConfig.put("min.insync.replicas", "2");
```

"Kafka allows you to configure replicas of each partition that will be stored at a different broker. This way if we lose a broker we'll be able to recover the messages from a different location. Additionally to prevent data loss in the event of failure we can set default configuration that will enforce that a message must be committed to a certain number of replicas." said KIP in a rush to calm Dilly down. "That all sounds great. Let's do it", said Dilly.

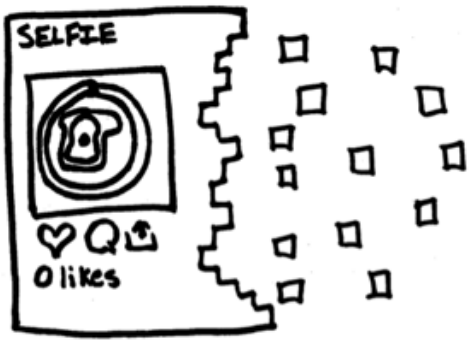



```
NewTopic newTopic = new NewTopic(topicName, numPartitions, numReplicas)
    .configs(topicConfig);
AdminClient client = AdminClient.create(producerProps);
CreateTopicsResult topic = client.createTopics(Collections.singleton(newTopic));
//Block waiting for result
topic.all().get();
```

KIP quickly readied the paperwork based off of all of the information provided by Dilly and submitted the request to Kafka to create the topics. Now the request took some time to complete as it required coordination among the brokers but eventually KIP was able to tell Dilly that her order has been completed. “Can I send a post now?” questioned Dilly and a very patient KIP responded ”Not quite yet.”

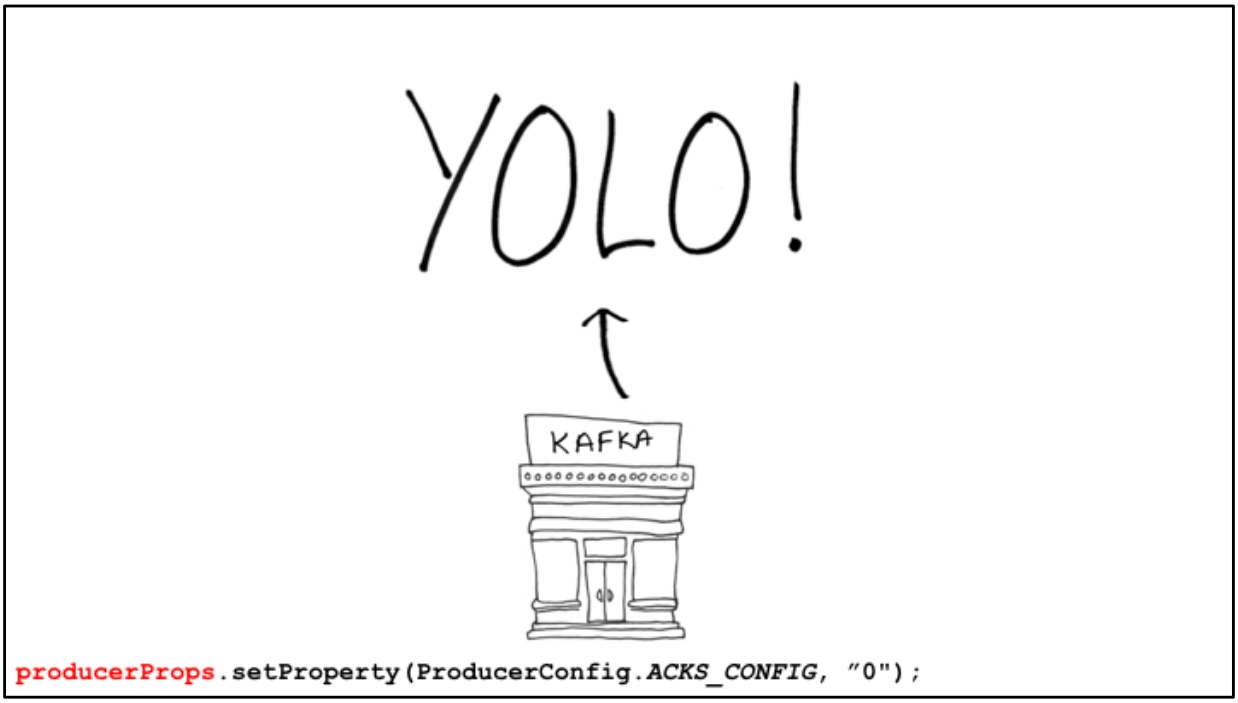


KIP realized Dilly was on the verge of losing interest and quickly started to explain that Dilly had to decide how the posts should be organized amongst the different partitions. "Depending on the message and its key you can decide which partition should receive the post." explained KIP. "Well how do I choose?", asked Dilly. "Well it depends… the ordering of your posts is only guaranteed inside of a single partition so if that is important then you need to make sure similar posts get sent to the same partition to preserve that ordering. Otherwise you could decide to send posts with the similar content like food to one partition, or posts with the same hashtags to another. The goal is to distribute the posts across each partition evenly so that one does not become lopsided while the others do not do any work." Upon finishing the explanation KIP was greeted with a blank stare from Dilly and just decided, "how about we just randomly distribute the messages across the partitions?" "Whatever!", answered Dilly.



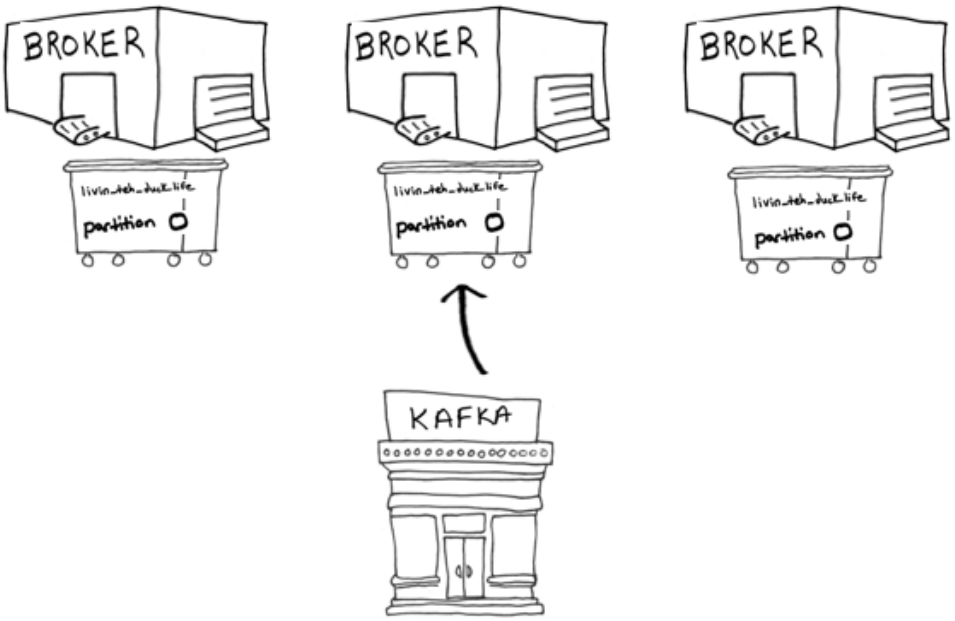

```
producerProps
      .setProperty(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
                   SelfieHashTagSerializer.class.getName());
producerProps
      .setProperty(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
                   SelfieSerializer.class.getName());
```

”To be able to send the messages, we need to be able to break them down into smaller pieces for more efficient transmission and storage.”, continued KIP. “YOU ARE GOING TO DESTROY MY POSTS BY EXPLODING IT INTO A LOT OF LITTLE PIECES”, screeched Dilly. “No no no, not destroy, we’ll be able to reconstruct your posts easily we just need to be able to transmit them.” said KIP hoping to avoid a tantrum. ”Now that we know how to direct your messages and know how to break them down for transmission, we need to talk about what sort of guarantees you need on committing the message to the broker. There are lots of options but some are more costly than others.”



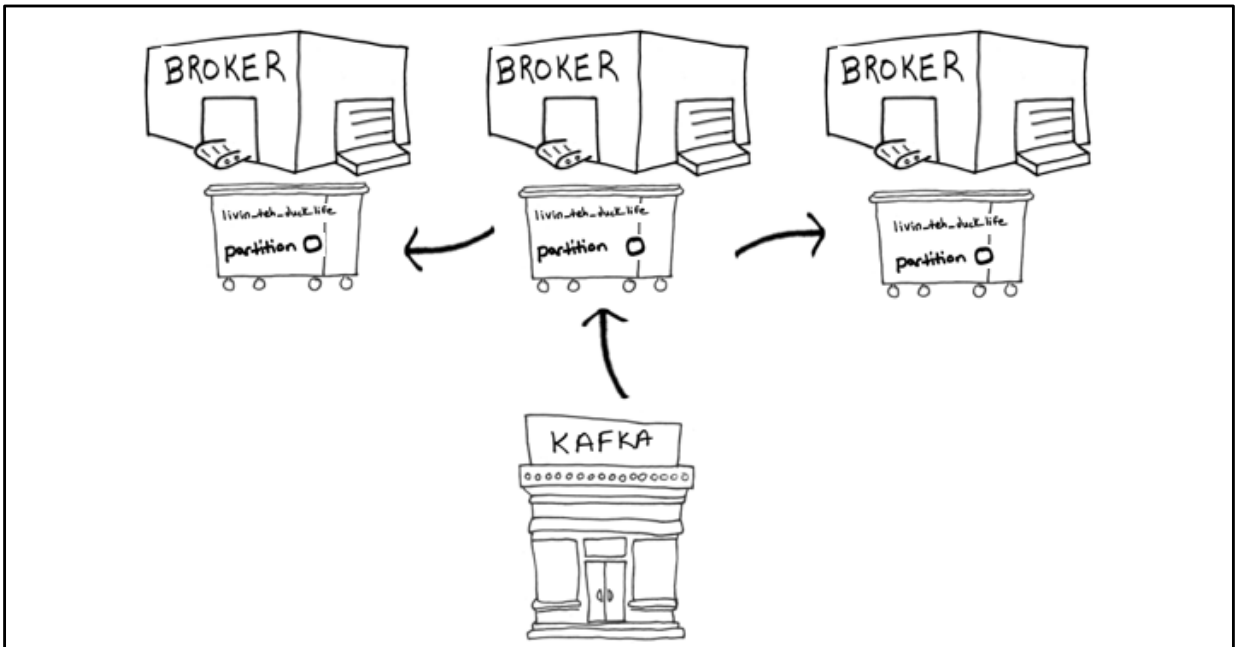

```
producerProps.setProperty(ProducerConfig.ACKS_CONFIG, "0");
```

”The first delivery option provides the least guarantees but the fastest service.” said KIP. “You simply drop your message off here with us and then continue on with your other activities for the day. We will do whatever we can to transmit the message to the broker but there are no guarantees and if you come on Tuesday’s when my brother Gary is working the message might end up in the trash because he’s so scatter brained.”



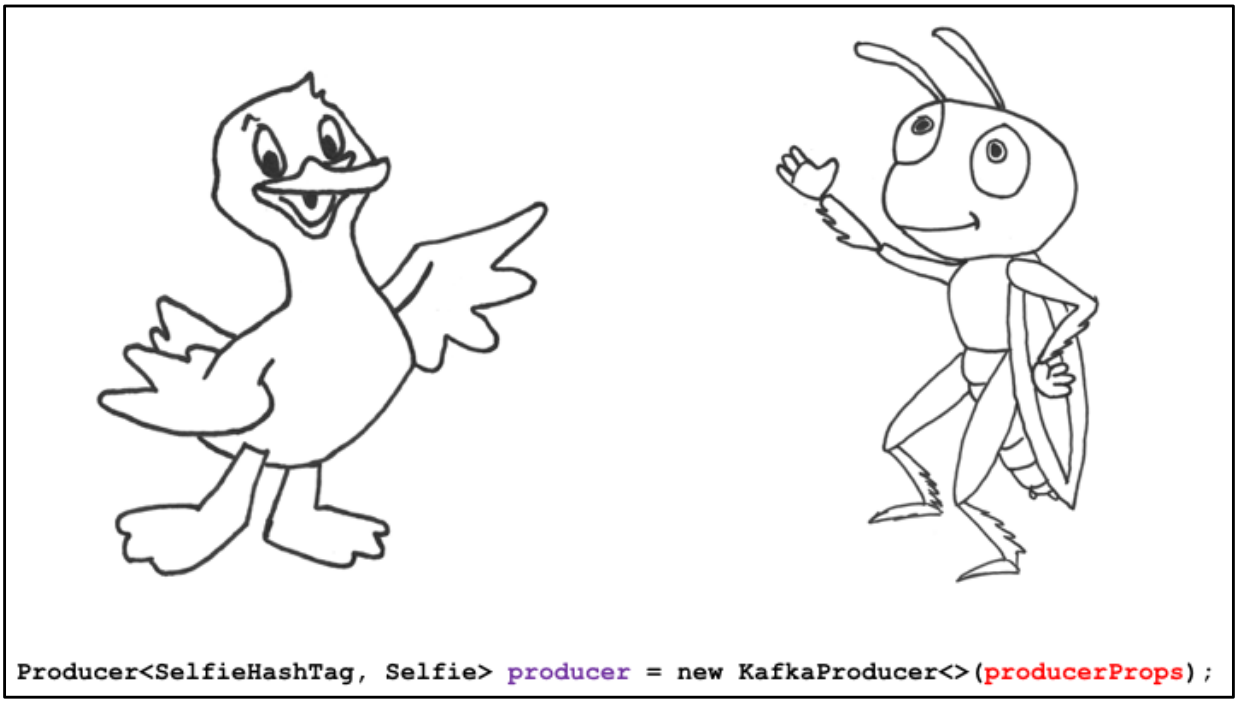

```
producerProps.setProperty(ProducerConfig.ACKS_CONFIG, "1");
```

”The next delivery option takes a little bit longer but provides the guarantee that the message gets committed to a single broker. You’ll need to wait in the store a bit longer for the acknowledgement to come back but you won’t have to wait too long. However if something were to happen to the broker before the message could get replicated to the other brokers then it will be lost and you won’t even know it.




```
producerProps.setProperty(ProducerConfig.ACKS_CONFIG, "all");
```

"The last option for sending a message is to require the message to be committed to each of the replicas that are in sync. So the message gets written to the leader and it then waits until it is sure each of the replicas has pulled the message before it tells you that the message was successful. You'll need to wait in the store for that acknowledgement but don't worry it is usually very fast but we also have a nice seating area and coffee bar for you to enjoy while you wait." finished KIP. After completing the explanation of all of the options, KIP realized that Dilly was completely zoned out and just went with the highest guarantees of "all" simply to avoid getting yelled at by Dilly later if a message got lost.

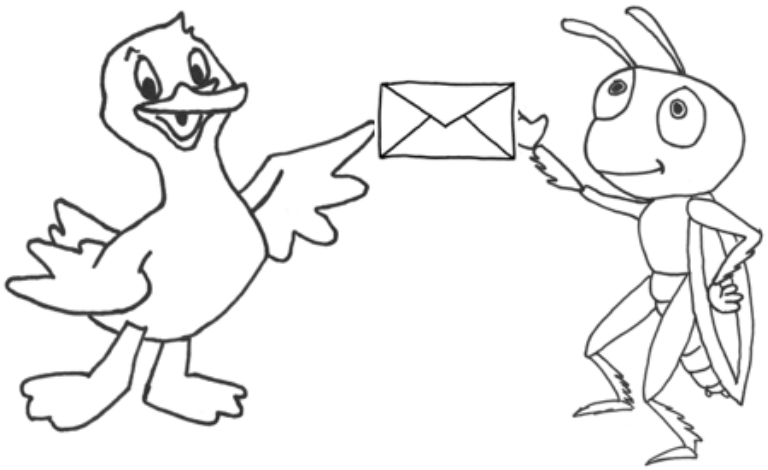


```
Producer<SelfieHashTag, Selfie> producer = new KafkaProducer<>(producerProps);
```

KIP quickly finished the next set of paperwork for Dilly and said, ”Now we are ready for you to send your messages.” Dilly became super excited and giddy with anticipation and quickly started scanning through the photos on her phone trying to figure out what to send.

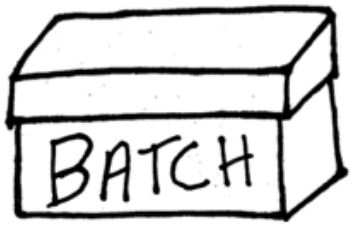

```
SelfieHashTag hashTag = new SelfieHashTag("#blessed");
Selfie selfie = new Selfie("/img/duck_lips_100.jpg");
```

Deciding that none of the thousands of pics on her phone would do, she quickly snapped a new pic on her phone with her selfie stick (knocking lots of expensive merchandize off the shelves around her and nearly hitting KIP in the head). Dilly quickly figured out the right key for the message using her favorite hashtag and selecting the right filter for the selfie.



```
ProducerRecord<SelfieHashTag, Selfie> record =
      new ProducerRecord<>(topicName, hashTag, selfie);
Future<RecordMetadata> result = producer.send(record);
```

Dilly quickly bundled up the key and message into a easily a message that could be sent through Kafka. KIP gleefully took the message as this was the first business of the day and gave Dilly back a receipt for her to use to track to make sure the message got sent successfully and committed to the cluster. "When does the message get sent?" asked Dilly.

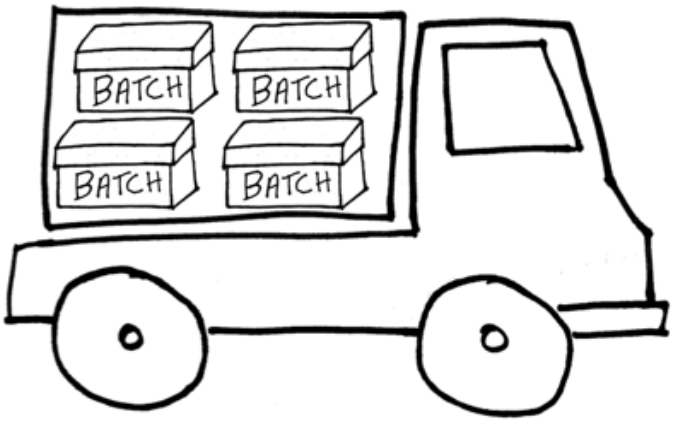



```
producerProps.setProperty(ProducerConfig.LINGER_MS_CONFIG, "5000");
String batchSize = String.valueOf(5 * 1024 * 1024);
producerProps.setProperty(ProducerConfig.BATCH_SIZE_CONFIG, batchSize);
```

“That depends” answered KIP. ”We are all about efficiency here at Kafka so we do not actually send each individual message by itself. Instead we bundle lots of messages up together into a batch. Each batch is intended to go to a single partition and therefore only to a single broker. If you send a lot of posts that go to a lot of different partitions they could be going to lots of different brokers in the cluster. Since you might be sending lots of messages very quickly we’ll hold the batch for a certain amount of time. However we don’t want the batch to get too heavy so we also set a maximum size. If the batch fills up or we are past our wait time we’ll send the batch to the broker.” “What happens if a message is bigger than a batch?” asked Dilly. “Well in that case we’ll just start sending it directly to the broker and hopefully it is not bigger than the broker’s maximum message size, as then the message will fail.”

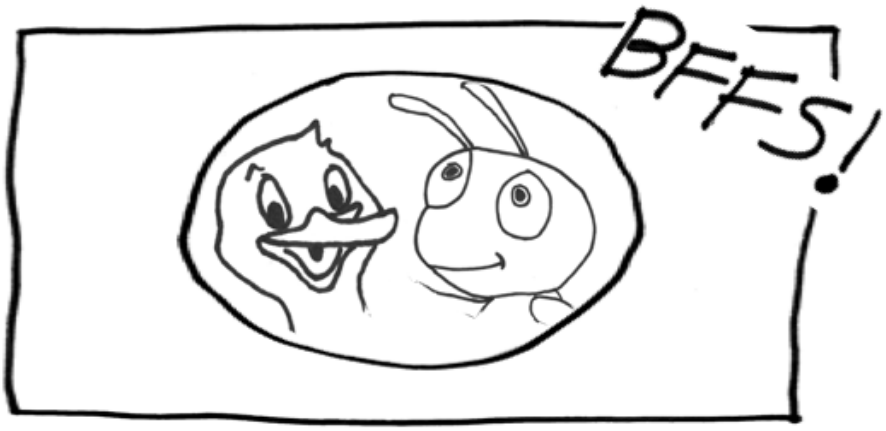

```
producerProps.setProperty(ProducerConfig.COMPRESSION_TYPE_CONFIG, "snappy");
```

KIP continues, “As mentioned, we are all about efficiency here at Kafka so besides batching your messages up we also support compression. In this case we compress the entire batch for efficient transit so assuming lots of your posts are the same we are able to send and store a lot smaller of a message.” “So compression is like spanx?”, asked Dilly. It was now KIP’s turn to give a blank stare.



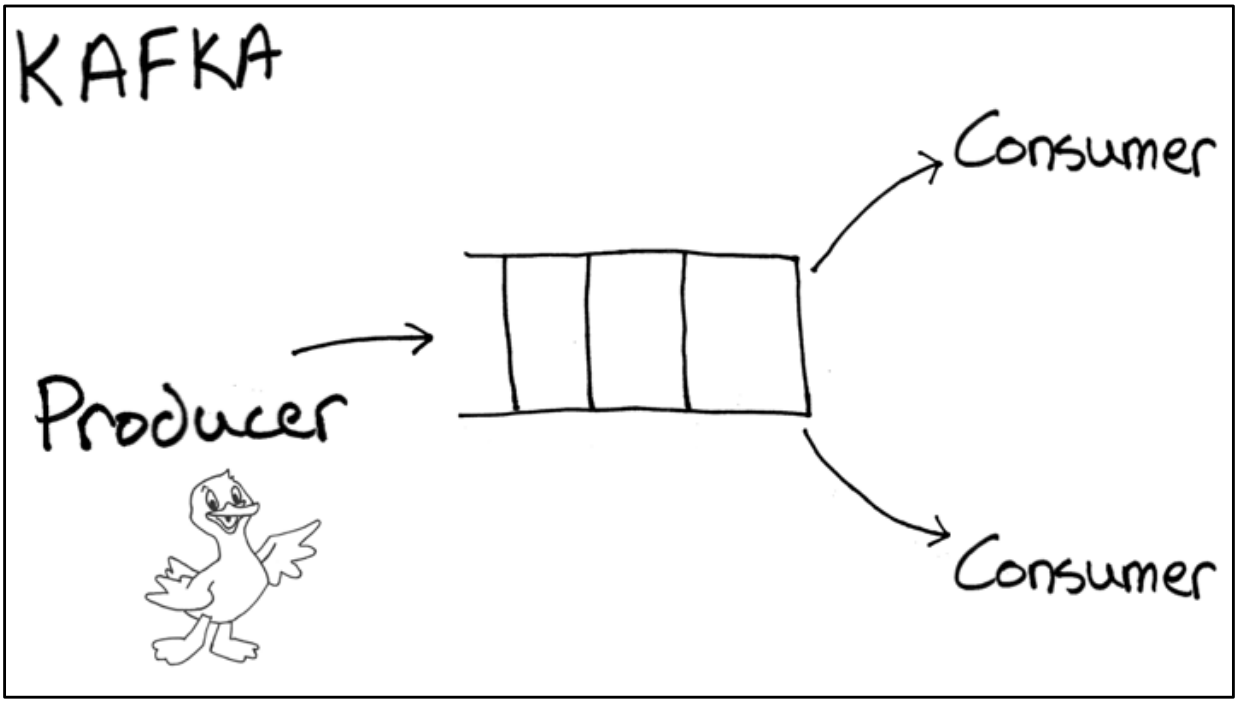

```
//Wait for messages to be sent synchronously
producer.flush();
```

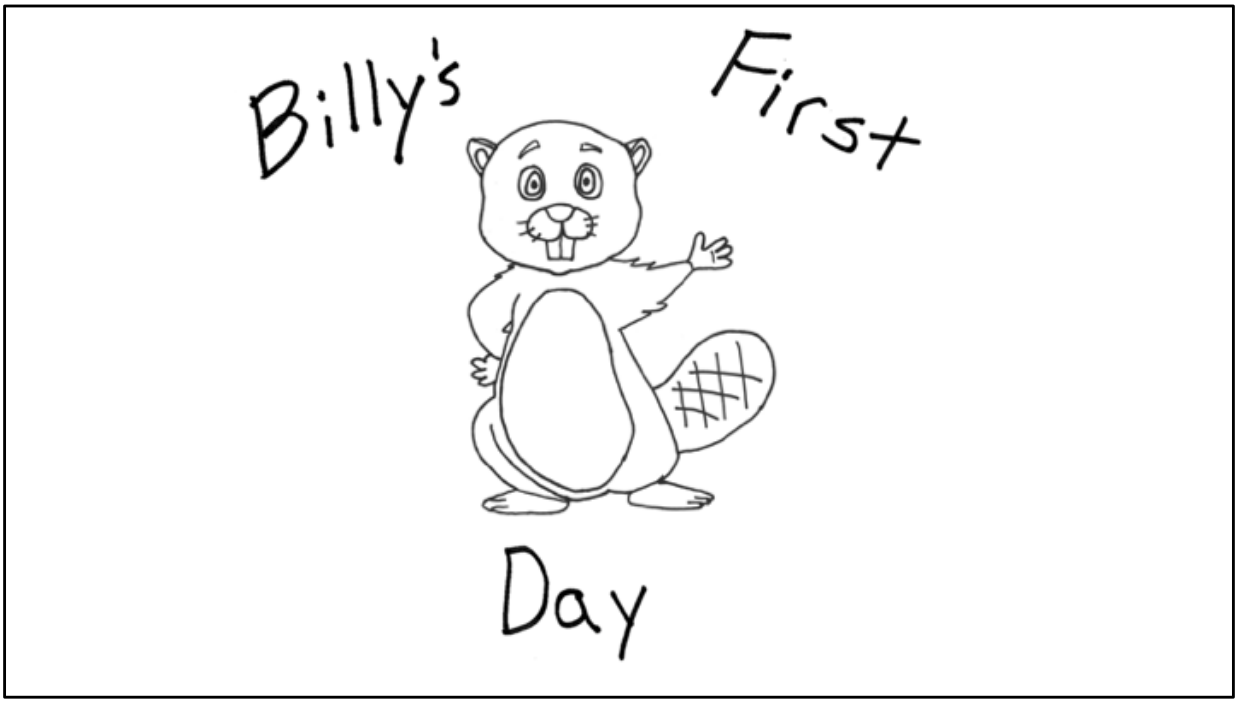
”what if I want to force all my messages to get sent at once?” asked Dilly. ”We have the option to let you control when messages leave the store and get sent to the brokers but try not to abuse this option too much. Failing to utilize the efficiencies of the batching and compression means things can slow down from a higher number of requests, inefficiency of storage, and more of your time waiting around the store instead of being out there creating posts.” offered KIP

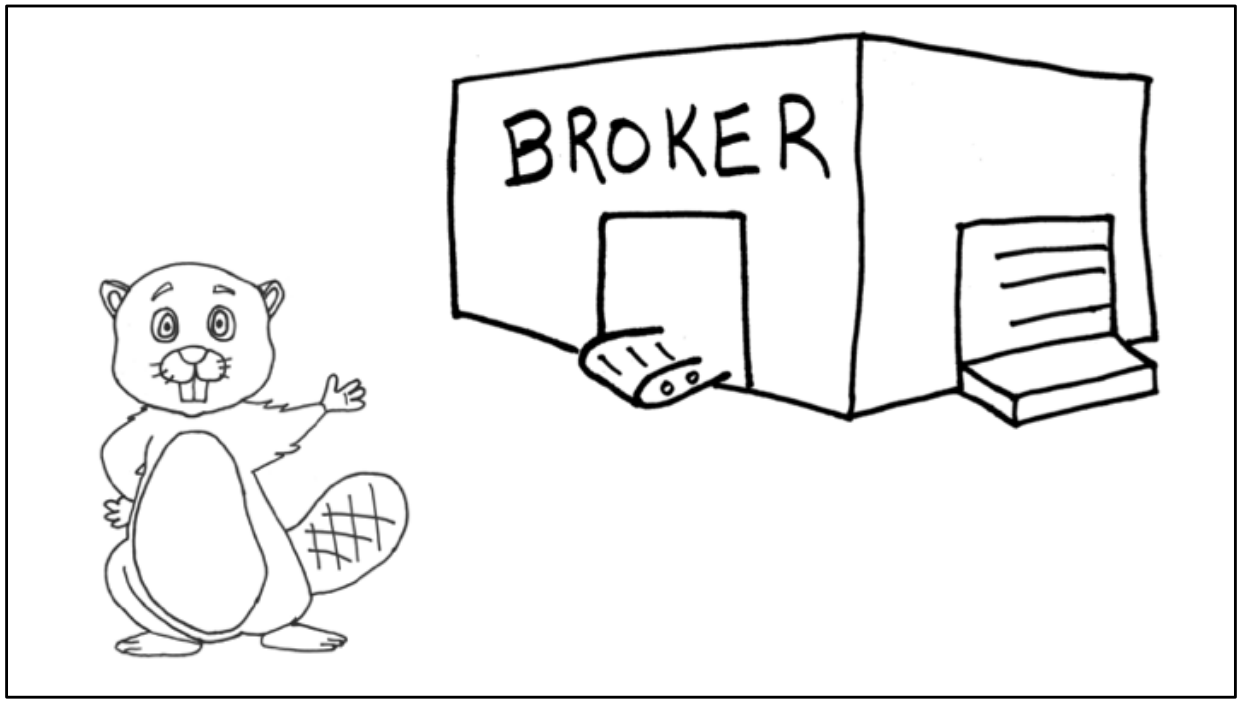



```
//Will block for message to be committed or throw an exception
RecordMetadata recordMetadata = result.get();
int partition = recordMetadata.partition();
long offset = recordMetadata.offset();
System.out.println("Partition: " + partition + " Offset: " + offset);
```

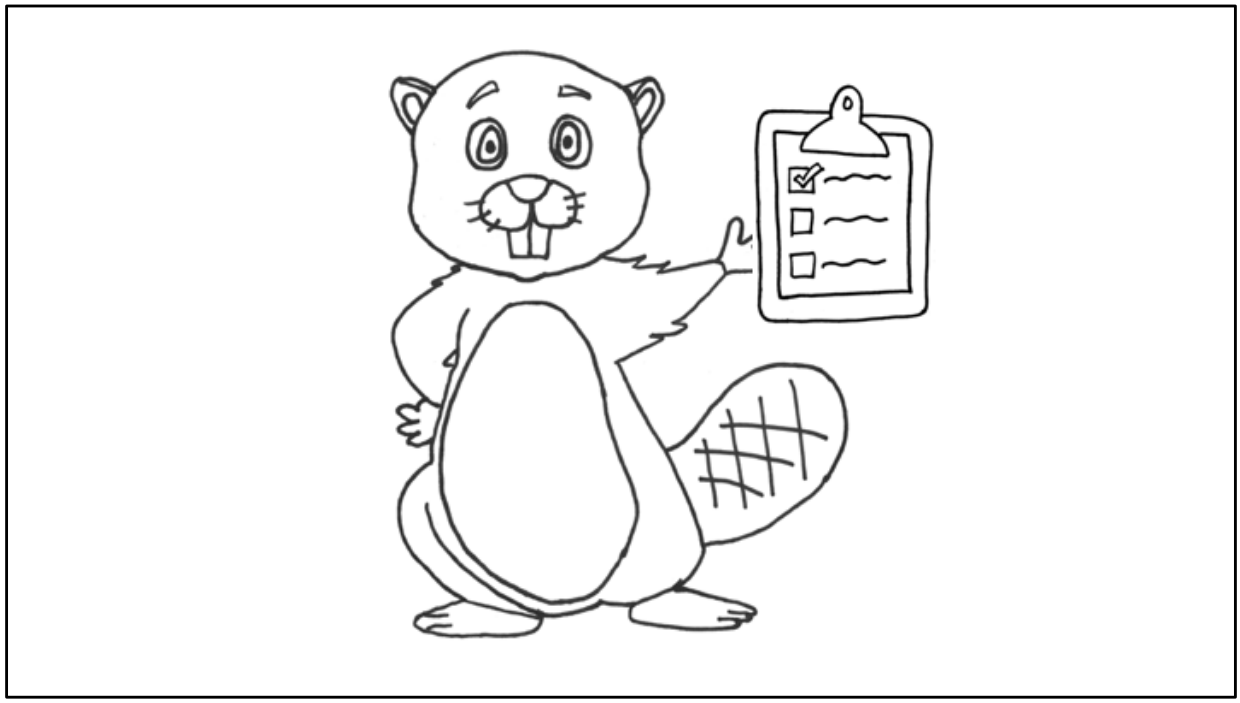
While KIP was busy explaining the concepts of compression and batching, Dilly's first post to Kafka had completed and was successful. Dilly was so excited and quickly ran off to find more things to post and rebuild her social media empire. Before she left however she quickly proclaimed that KIP was her new BFF and solidified it by making a post declaring it as so. KIP wasn't sure how to feel about it but was happy to have setup a brand new producer so was happy to go along with it and to finally get Dilly out of the store.


KAFKA
Consumer
Producer
Consumer


Billy's First Day



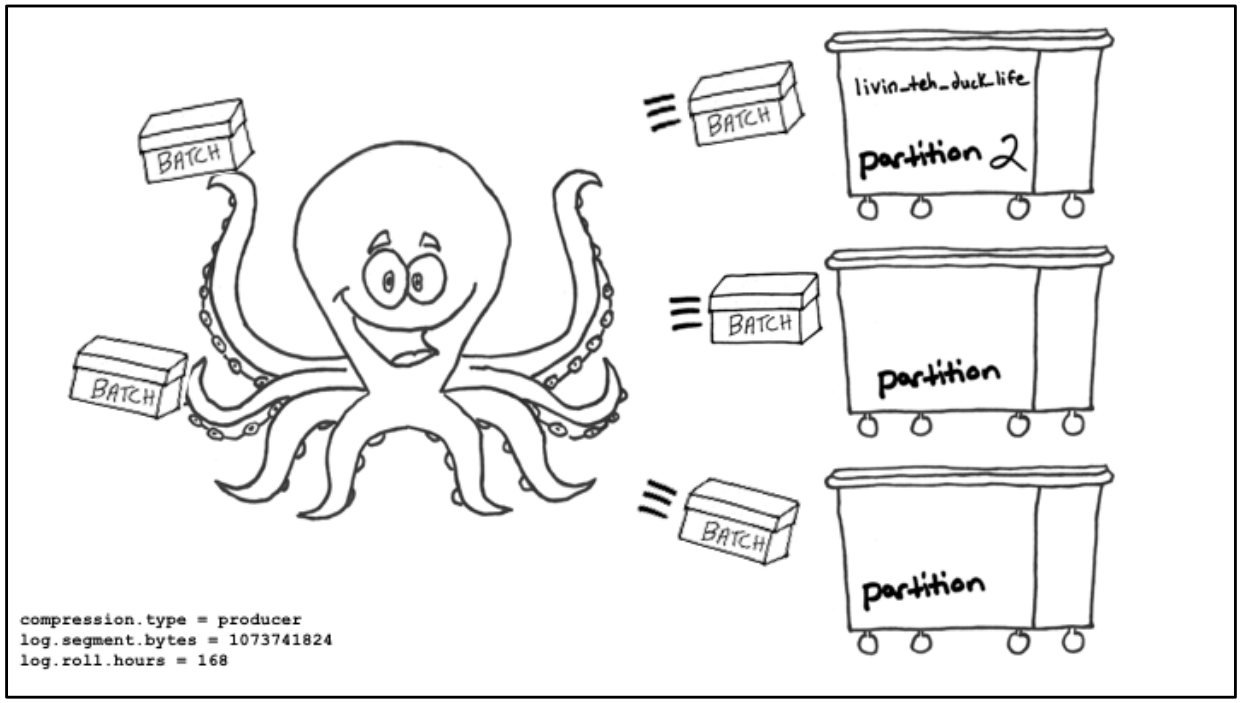
Billy was excited to start his first day at his new job. Billy had just been promoted to be in charge of a broker in the Kafka cluster. Billy knew it was going to be a lot of responsibility but had spent all night reading the manual to make sure he was prepared.



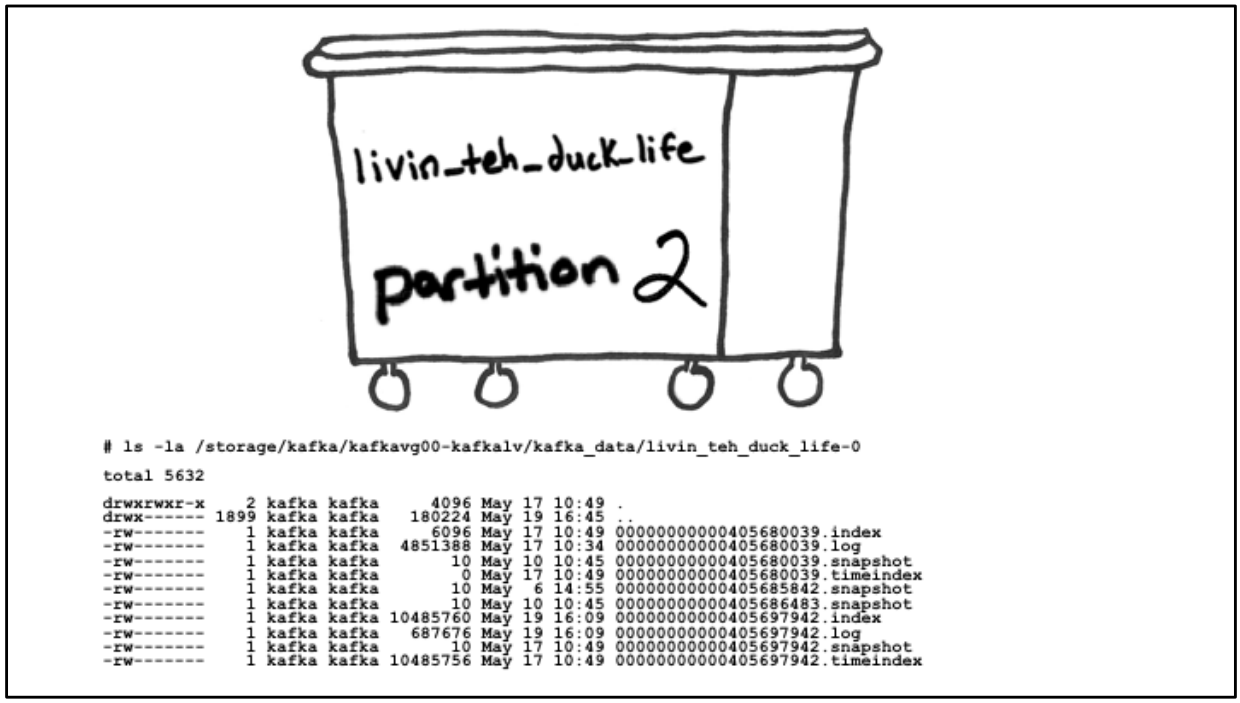
Billy knew that the broker was a very important component of the Kafka cluster. Therefore to start the day he quickly created a TODO list of all the activities and tasks he had to make sure were getting completed. He knew that if any of them feel behind or started to fail, his broker would start to fall apart and all the important messages being produced and distributed would be delayed or worse lost.

```
$ ./kafka-topics.sh --bootstrap-server broker:6667 --topic  livin_teh_duck_life -describe

Topic: livin_teh_duck_life PartitionCount:3 ReplicationFactor:3 Configs:unclean.leader.election.enable=false
      Topic: livin_teh_duck_life   Partition: 0     Leader: 2   Replicas: 2,3,1   Isr: 1,2,3
      Topic: livin_teh_duck_life   Partition: 1     Leader: 3   Replicas: 3,1,2   Isr: 1,2,3
      Topic: livin_teh_duck_life   Partition: 2     Leader: 1   Replicas: 1,2,3   Isr: 1,2,3
```

Billy knew that the most important task for the broker is being a leader for it's assigned partitions. Billy wanted to make sure he was doing his fair share and wasn't letting the cluster down. Looking up partitions with his broker id as the leader, Billy quickly took an inventory of all the partitions for which his broker was responsible. As manager of the broker Billy didn't have time to perform all of the leader responsibilities himself. Instead he knew he had to delegate to his super helpful coworker, Inky.

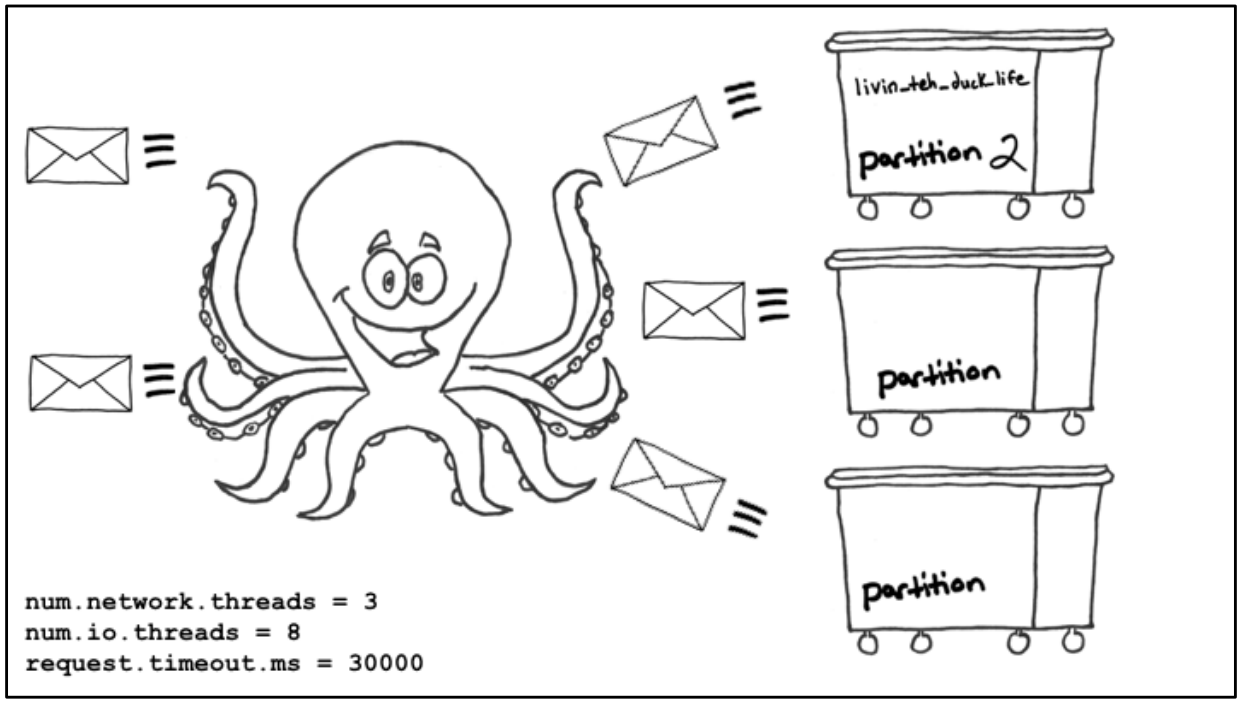


Inky is one of the most important workers at the broker. Inky was responsible for handling all of the requests into and out the broker. Receiving batches from the producers is a lot of work and as the leader of a partition all writes must be sent to Inky first. Inky has to sort the batches out into their various partitions. In some cases Inky might have to repack the batches using a different compression for more efficiencies but in most cases he can just reuse the already compressed batches. Inky also has to be very neat and orderly about his work. Therefore Inky makes sure when appending the batches to files the files never get too big and that they roll regularly.

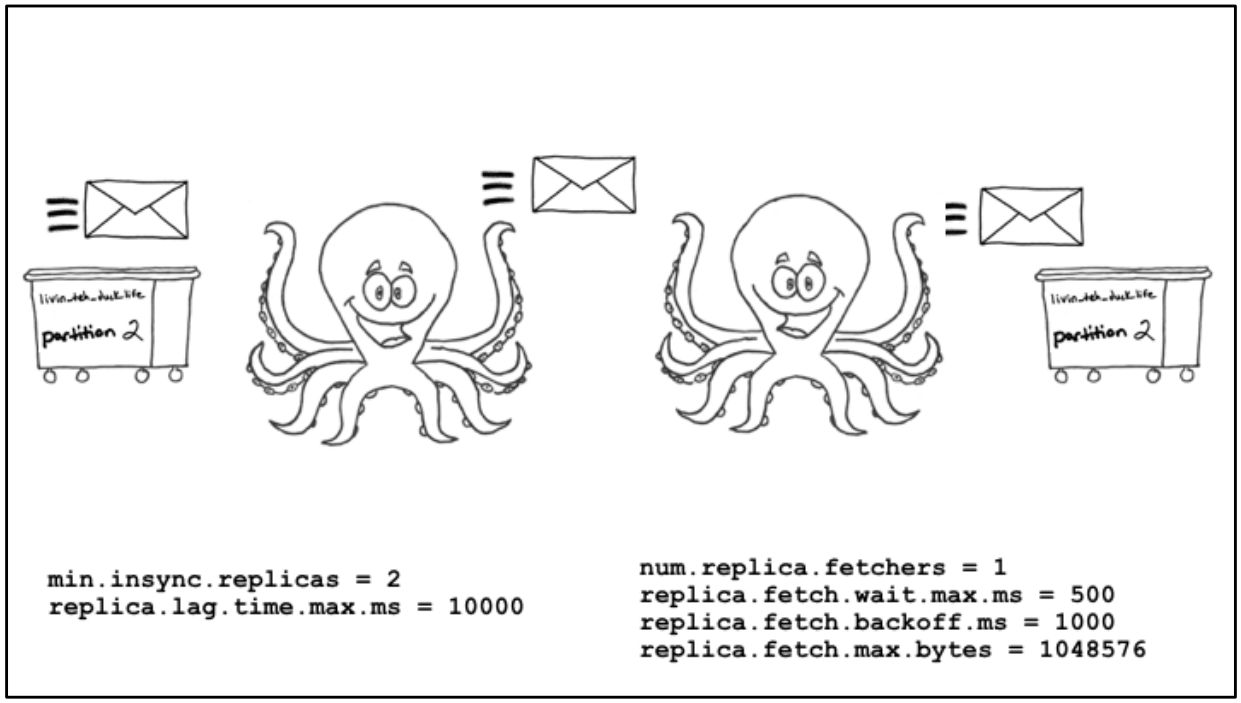



```
# ls -la /storage/kafka/kafkavg00-kafkalv/kafka_data/livin_teh_duck_life-0

total 5632

drwxrwxr-x    2 kafka kafka     4096 May 17 10:49 .
drwx------ 1899 kafka kafka   180224 May 19 16:45 ..
-rw-------    1 kafka kafka     6096 May 17 10:49 00000000000405680039.index
-rw-------    1 kafka kafka  4851388 May 17 10:34 00000000000405680039.log
-rw-------    1 kafka kafka       10 May 10 10:45 00000000000405680039.snapshot
-rw-------    1 kafka kafka        0 May 17 10:49 00000000000405680039.timeindex
-rw-------    1 kafka kafka       10 May  6 14:55 00000000000405685842.snapshot
-rw-------    1 kafka kafka       10 May 10 10:45 00000000000405686483.snapshot
-rw-------    1 kafka kafka 10485760 May 19 16:09 00000000000405697942.index
-rw-------    1 kafka kafka   687676 May 19 16:09 00000000000405697942.log
-rw-------    1 kafka kafka       10 May 17 10:49 00000000000405697942.snapshot
-rw-------    1 kafka kafka 10485756 May 17 10:49 00000000000405697942.timeindex
```

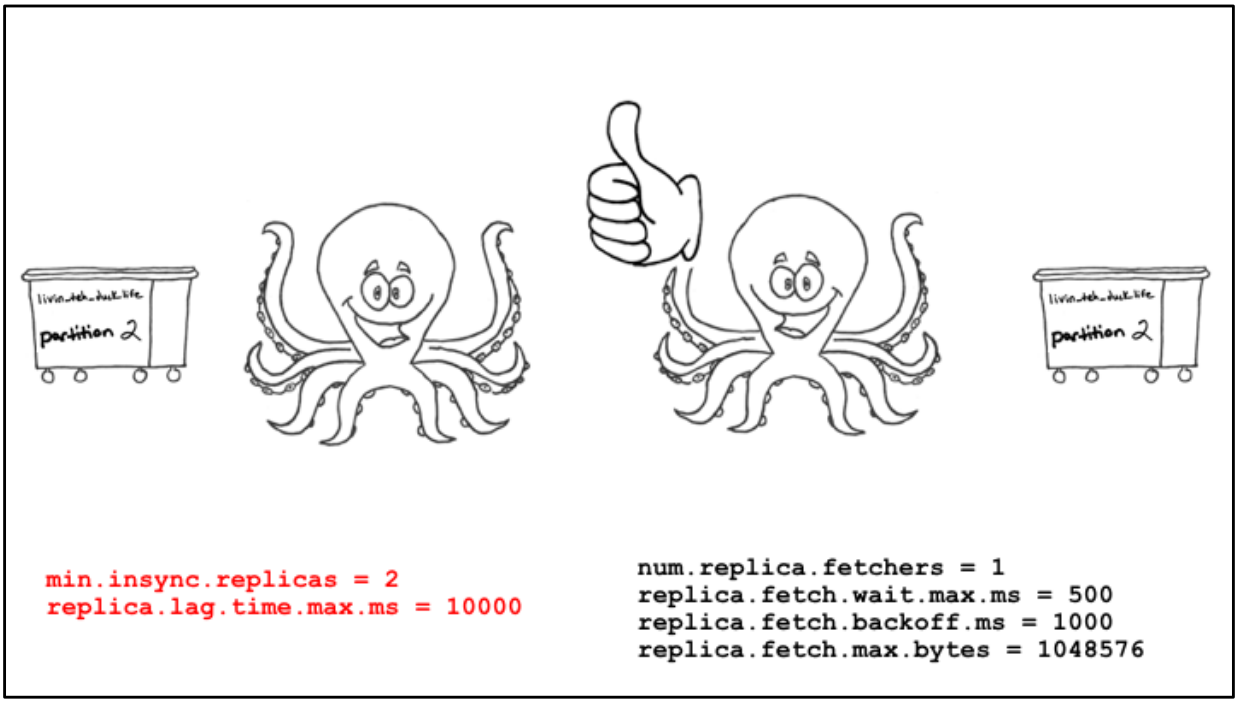
Inky is very meticulous. Each partition gets stored in a particularl place. Inky also keeps very thorough records of what gets stored for each partition. All of the important messages for a partition are appended to log files which as mentioned get rolled based on size and time. Inky also knows that some requests come for specific messages based on their offsets in the partition. To make the lookups more efficient Inky stores the mapping of offsets to messages in an index file so that he can seek quickly through the log file to find the message. Inky also knows that sometimes people want to look up messages by time instead of offset so another index is maintained to map when a message was written based on its timestamp. While all of this is a lot of hard work, Inky does not mind because it makes the next part of his job easier.



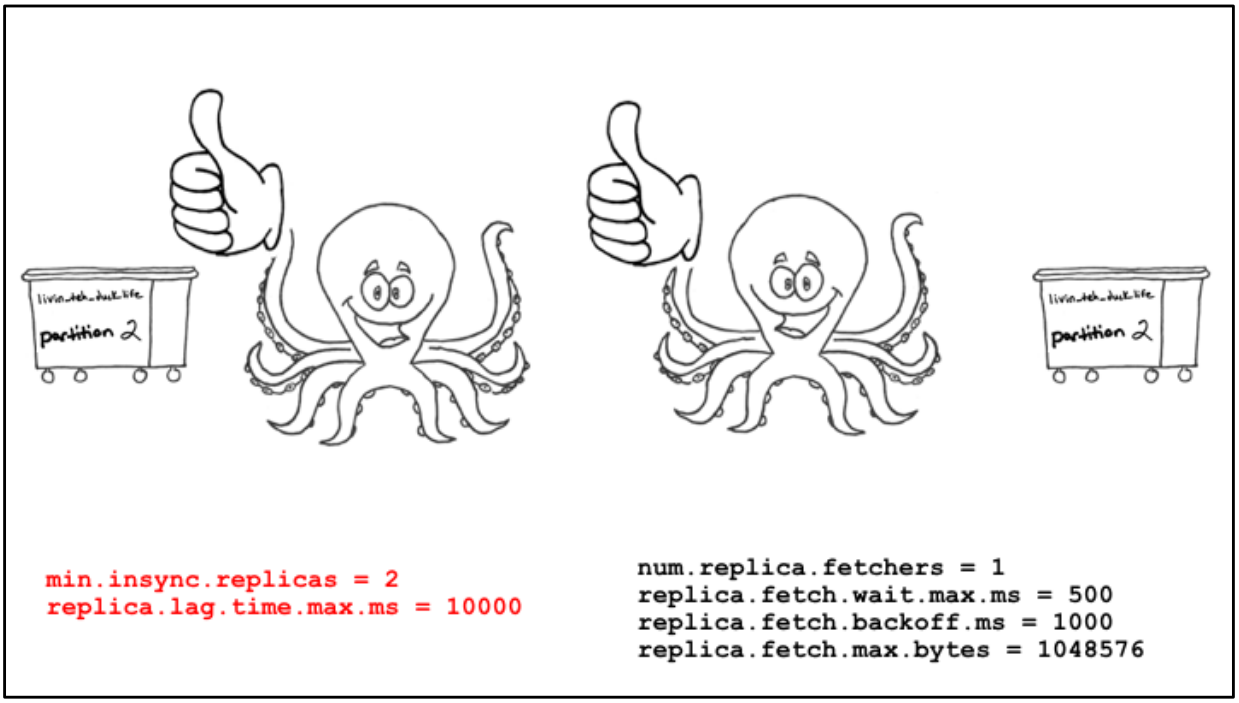
As the leader of a partition, Inky is not only responsible for storing he batches but also responsible for answering all of the requests for data from that partition. The requests are to start reading at a certain point in the logs and to start returning batches. Storing the files in logs and indices makes this part of Inky's job a lot easier. Inky can jump right to the specific messages being retrieved. Inky can also rely on some other friends to help answer the requests because the must be answered in a quick amount of time or else the requester might give up.



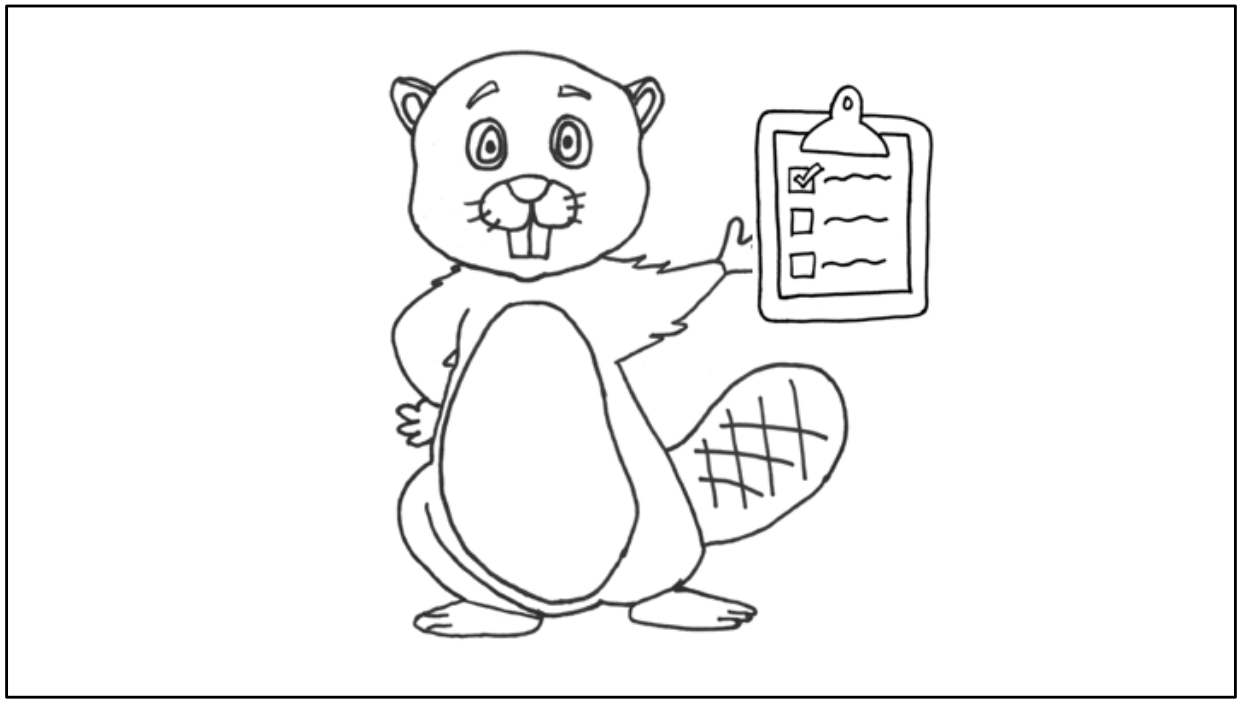
One set of requests that Inky must handle are the requests from his cousin Blinky. Blinky works at a different broker and is responsible for keeping all of the replica partitions at that broker in sync. Blinky is specifically responsible for pulling messages from all of the partitions for which Inky is a leader. If there is too much work for Blinky to do and he can't keep up then more replica fetchers can be used to distribute the work.



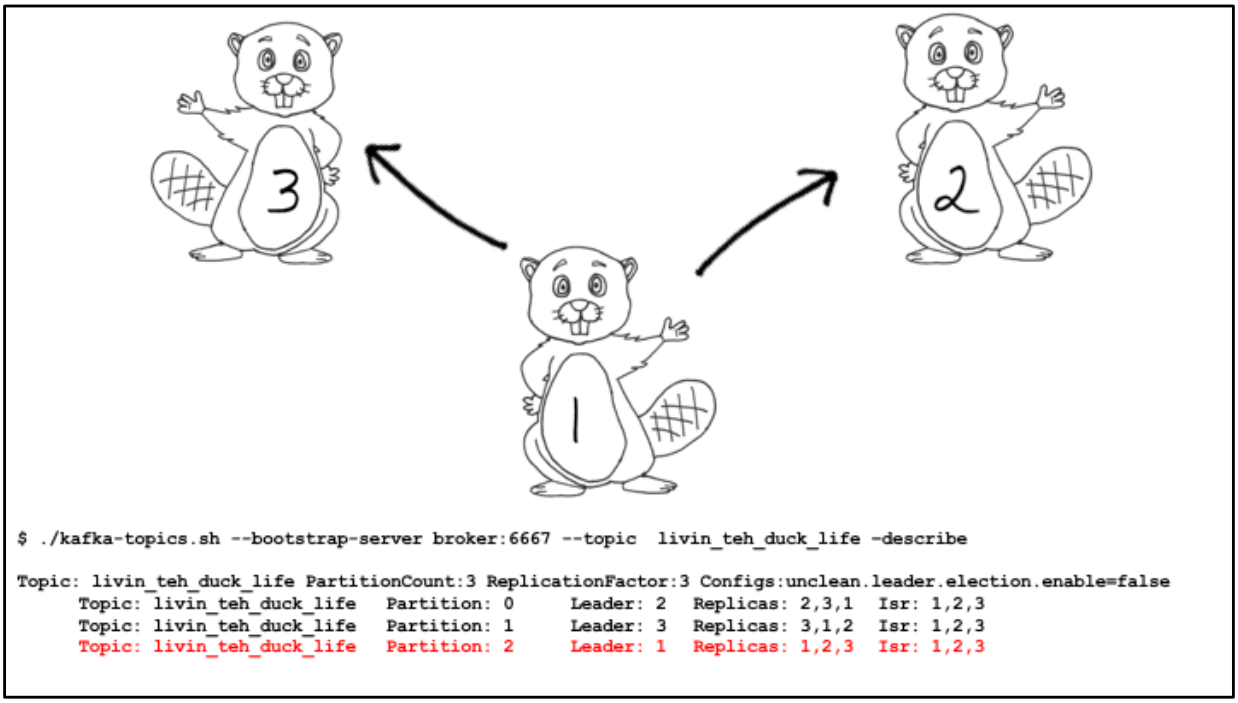
Just like Inky must be really fast about his work, Blinky has to be really fast as well. To support the delivery guarantees of a message being committed to multiple brokers, Blinky must pull the message and store them within a specified amount of time to be considered in sync. Once Blinky has pulled a batch of messages and successfully stored the message it sends a message back to Inky letting him know that the replica is all caught up.



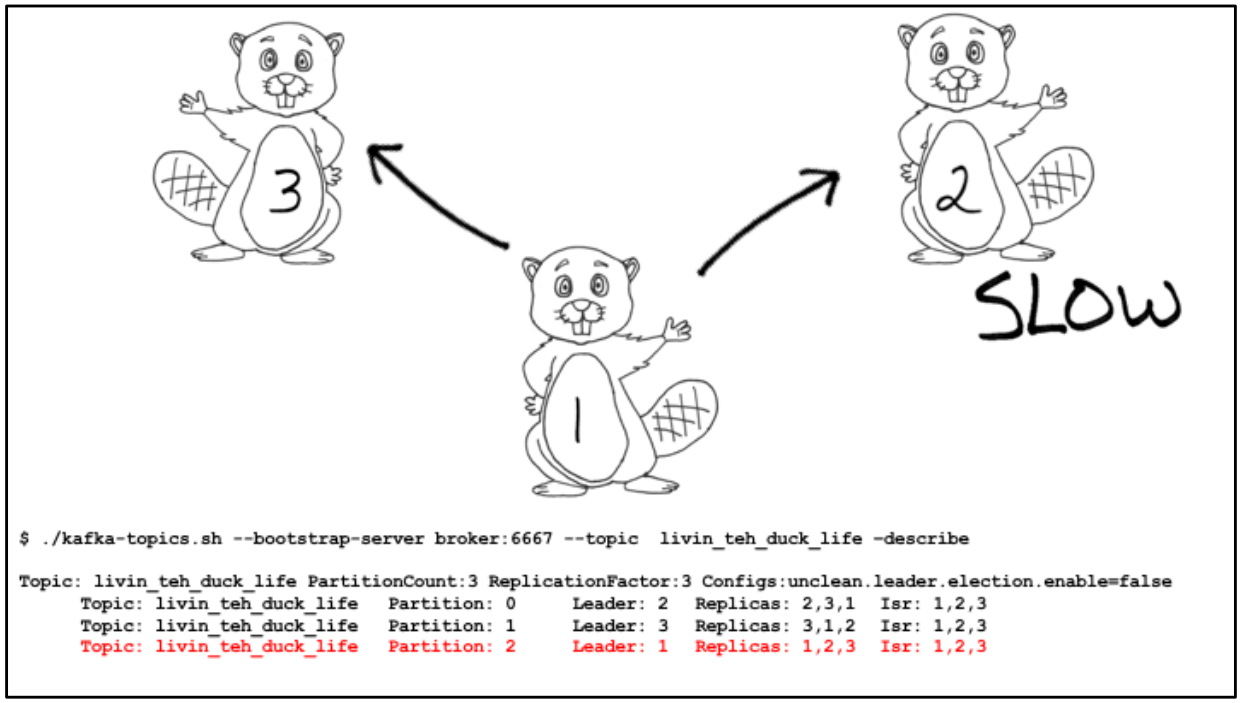
Inky is very grateful for the message because then he can communicate to the message producer that the requirements of the message being delivered has been satisfied, making a very happy customer. Inky does not like dealing with unhappy customers.



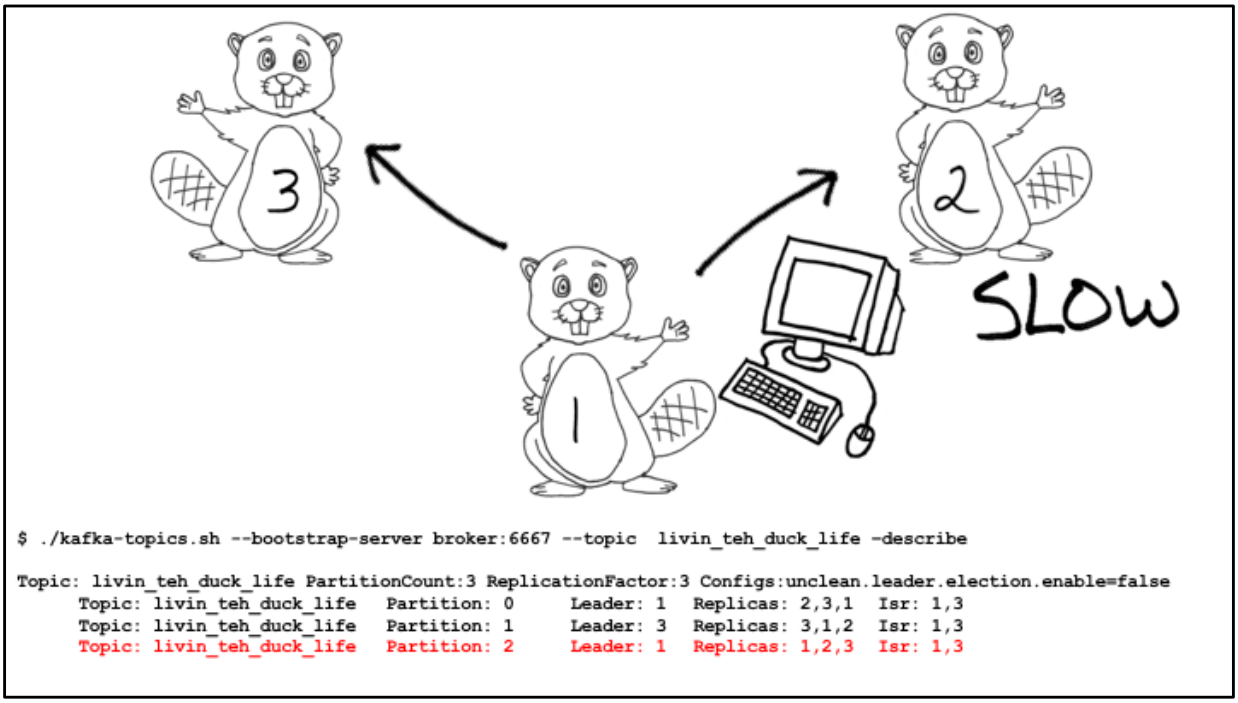
After observing Inky being hard at work and seeing that everything was flowing smoothly, Billy was excited to check one thing off of his list. Billy looked at the next thing on his list which was practicing gameday resiliency drills with his peer brokers. This is an important task for Billy as with large Kafka clusters there is always the opportunity for failures to occur and Billy does not want anything to affect Kafka's ability to receive and deliver messages.



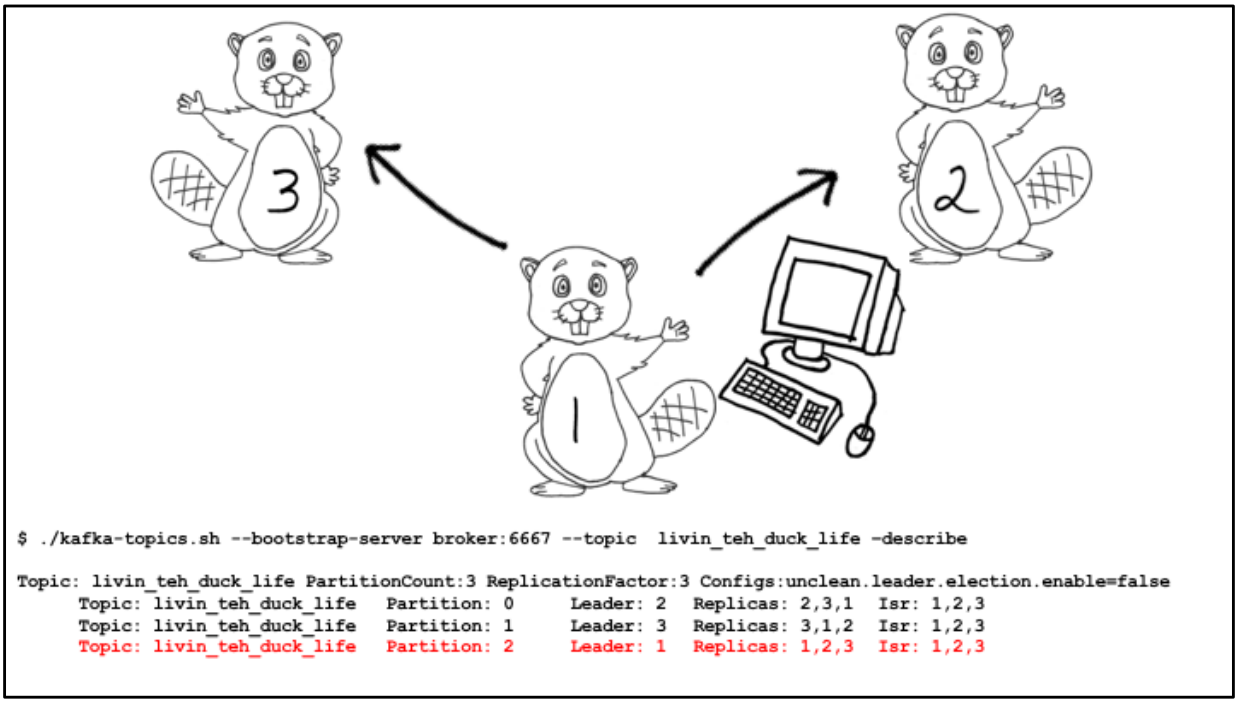
Billy starts the drill by identifying some of his peer brokers in the cluster. Billy says hello to his cousins Bobby and Barry. Normally Billy gets along great with these cousins sending messages between all of them as quickly as possible which ensure that while Billy is the leader for partition 2, Bobby and Barry was staying in the set of "in sync replicas". This is important to make sure no messages are lost.



Sometimes however things do not go completely smooth. We all know of that one relative that likes to goof off, takes too long in the supermarket or gets distracted staring at clouds. Well Bobby was that relative. Sometimes Bobby starts to slow down and can't keep up with all the messages that are getting written to Billy.

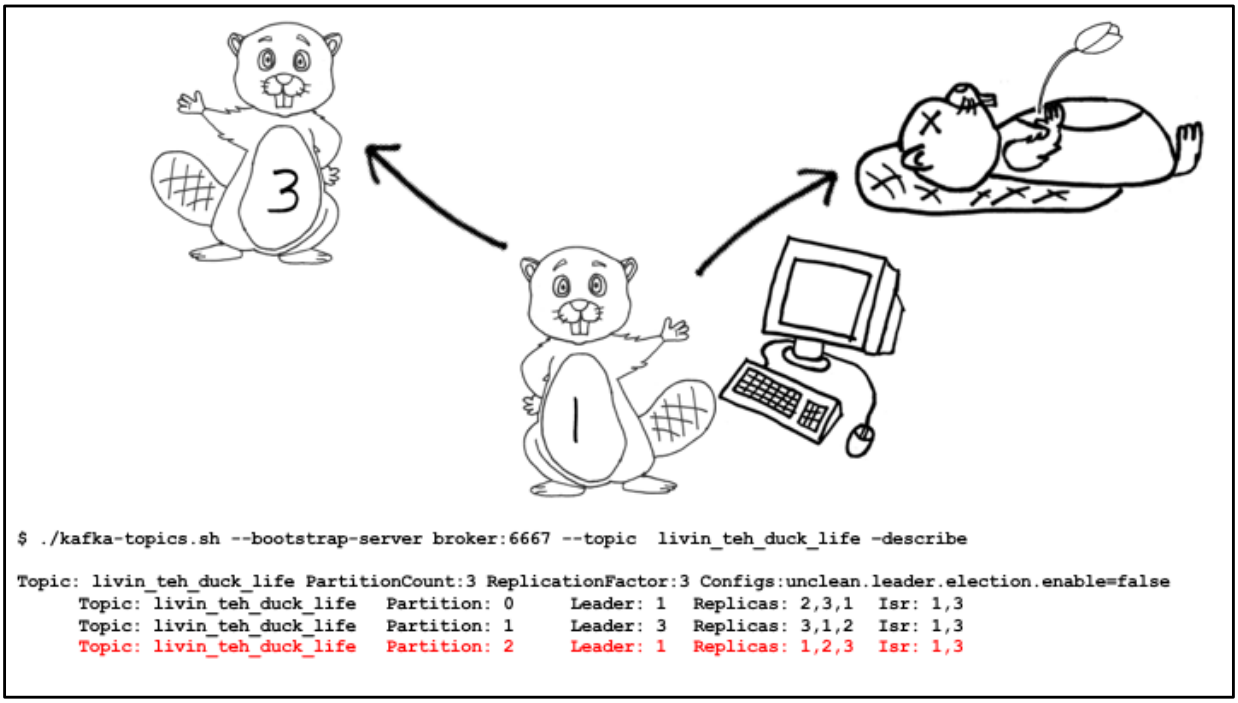


When Billy detects that Bobby is not keeping up, Billy updates their tracking software called Zookeeper to indicate that Bobby is no longer in sync. This is important as some messages are required to be committed to each in sync replica and if Bobby isn't keeping up then it will delay giving a good response.



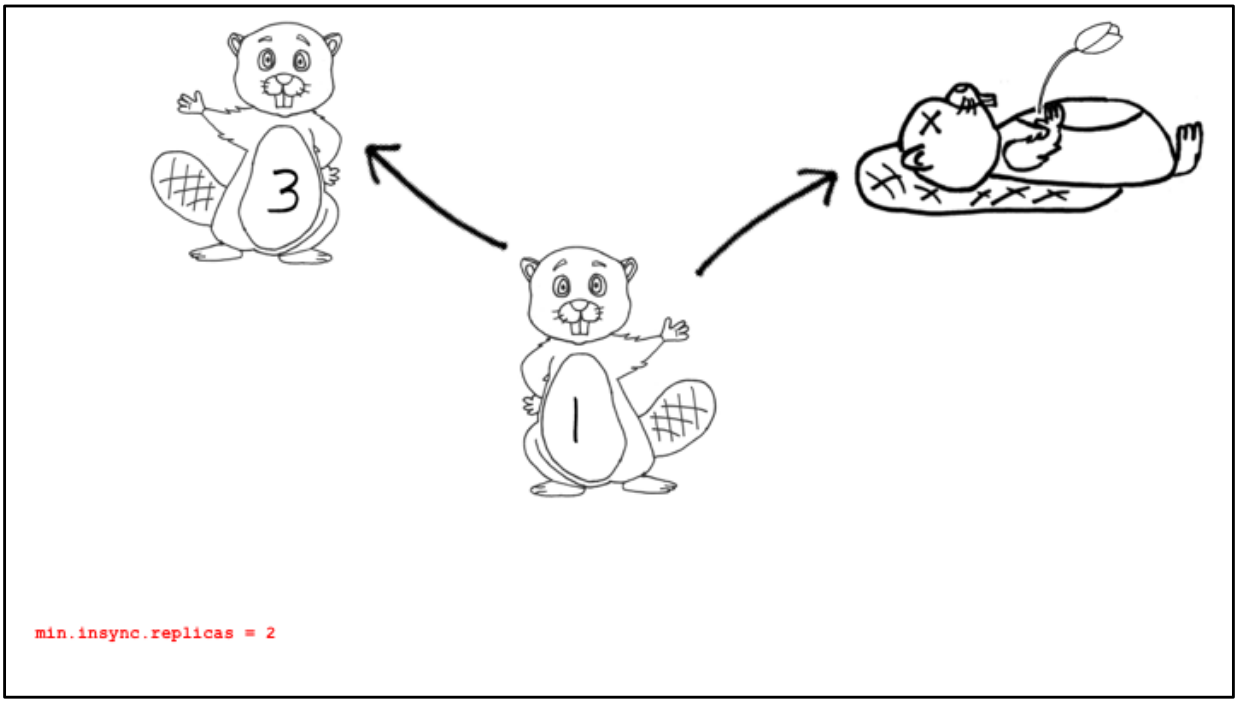

```
$ ./kafka-topics.sh --bootstrap-server broker:6667 --topic  livin_teh_duck_life -describe

Topic: livin_teh_duck_life PartitionCount:3 ReplicationFactor:3 Configs:unclean.leader.election.enable=false
      Topic: livin_teh_duck_life   Partition: 0     Leader: 2   Replicas: 2,3,1   Isr: 1,2,3
      Topic: livin_teh_duck_life   Partition: 1     Leader: 3   Replicas: 3,1,2   Isr: 1,2,3
      Topic: livin_teh_duck_life   Partition: 2     Leader: 1   Replicas: 1,2,3   Isr: 1,2,3
```

When Bobby starts to pay attention again and begins to keep up replicating messages, Billy will update the Zookeeper system adding Bobby back into the in sync replicas.

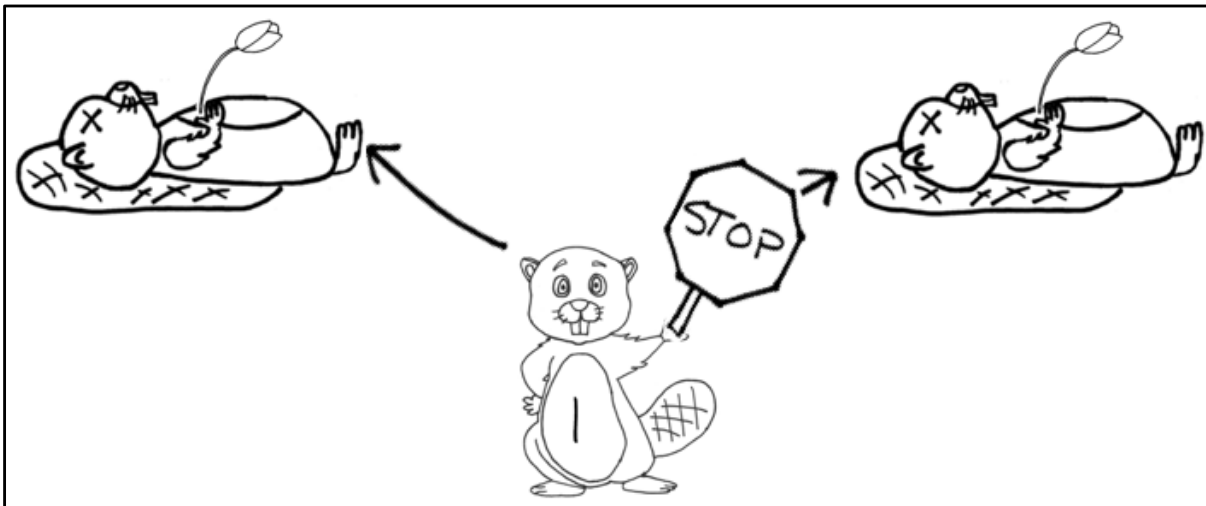



```
$ ./kafka-topics.sh --bootstrap-server broker:6667 --topic  livin_teh_duck_life -describe

Topic: livin_teh_duck_life PartitionCount:3 ReplicationFactor:3 Configs:unclean.leader.election.enable=false
      Topic: livin_teh_duck_life   Partition: 0     Leader: 1   Replicas: 2,3,1   Isr: 1,3
      Topic: livin_teh_duck_life   Partition: 1     Leader: 3   Replicas: 3,1,2   Isr: 1,3
      Topic: livin_teh_duck_life   Partition: 2     Leader: 1   Replicas: 1,2,3   Isr: 1,3
```

While Bobby sometimes goofs off and struggles to keep up with the messages there are occurrences when Bobby instead takes a “nap”. We have to say “nap” because this is a children’s book and we can’t say dead. Billy and Barry however need to keep on working sending and receiving messages.

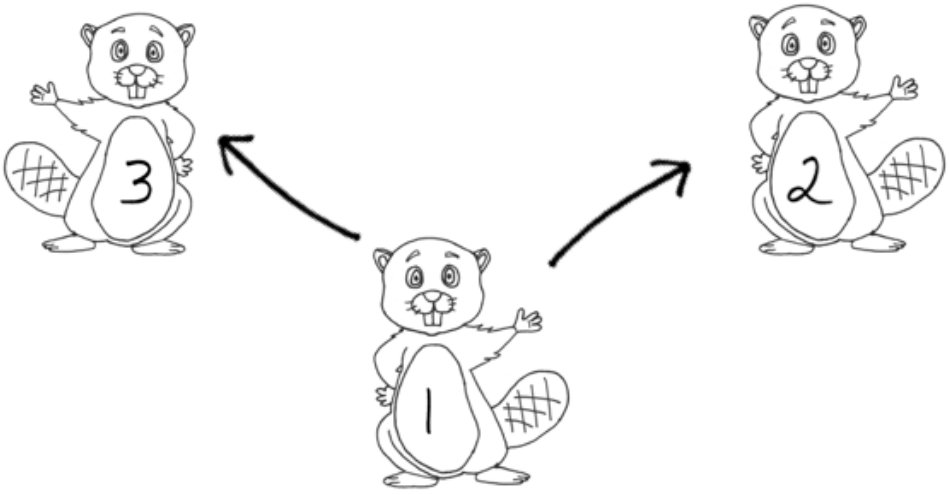


Billy and Barry are able to keep working and successfully send and receive messages because based on the configuration it is only required that two of the replicas remain in sync. This helps to avoid losing messages while Bobby continues to take his “nap”.



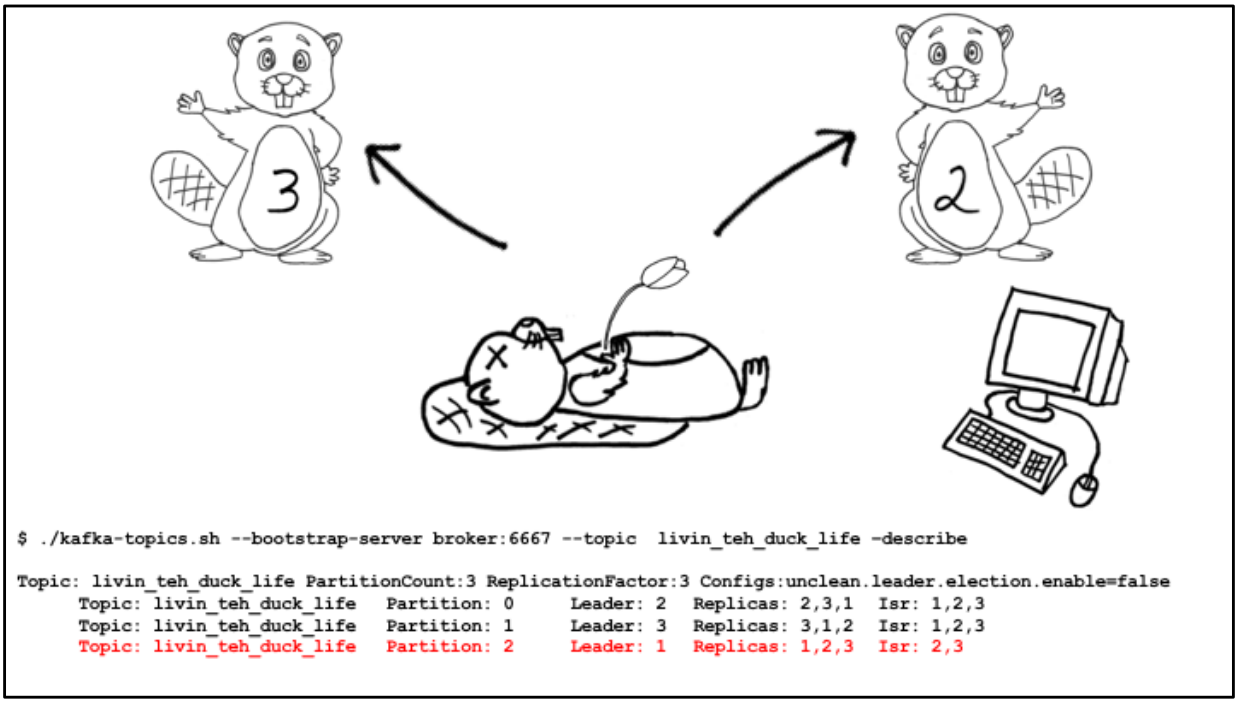

```
min.insync.replicas = 2

Topic: livin_teh_duck_life Partition: 2 Leader: 1 Replicas: 1,2,3   Isr: 1
```

The problem however becomes really critical if Barry ever takes a “nap” at the same time as Bobby. Since the minimum number of replicas cannot be satisfied then Billy has to start rejecting all messages for that specific partition. This is because the Kafka cluster prioritizes consistency over availability.



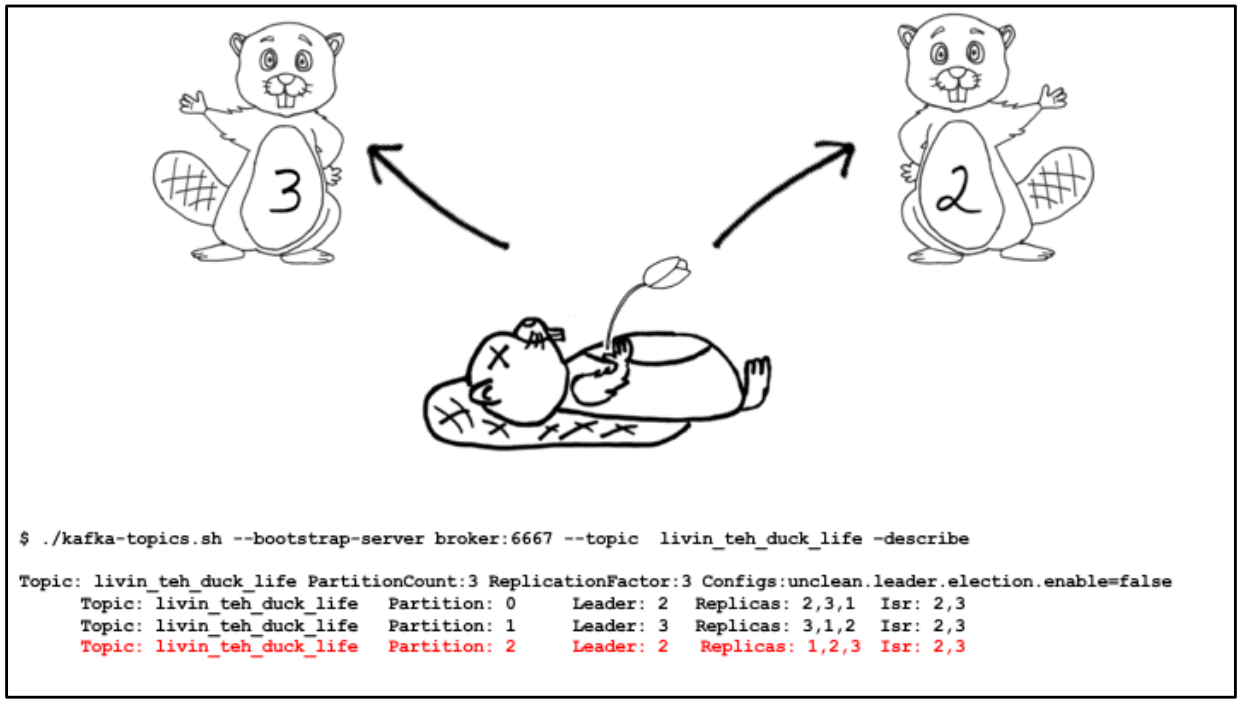

```
$ ./kafka-topics.sh --bootstrap-server broker:6667 --topic  livin_teh_duck_life -describe

Topic: livin_teh_duck_life PartitionCount:3 ReplicationFactor:3 Configs:unclean.leader.election.enable=false
      Topic: livin_teh_duck_life   Partition: 0     Leader: 2   Replicas: 2,3,1   Isr: 1,2,3
      Topic: livin_teh_duck_life   Partition: 1     Leader: 3   Replicas: 3,1,2   Isr: 1,2,3
      Topic: livin_teh_duck_life   Partition: 2     Leader: 1   Replicas: 1,2,3   Isr: 1,2,3
```

Once Bobby and Barry awake from their “nap” and quickly catch up on all the messages they hadn’t seen yet, Billy is able to mark them as in sync and start to receive new messages. Billy is grateful this is just a practice with Bobby and Barry but he is not done yet testing all the different failure scenarios.



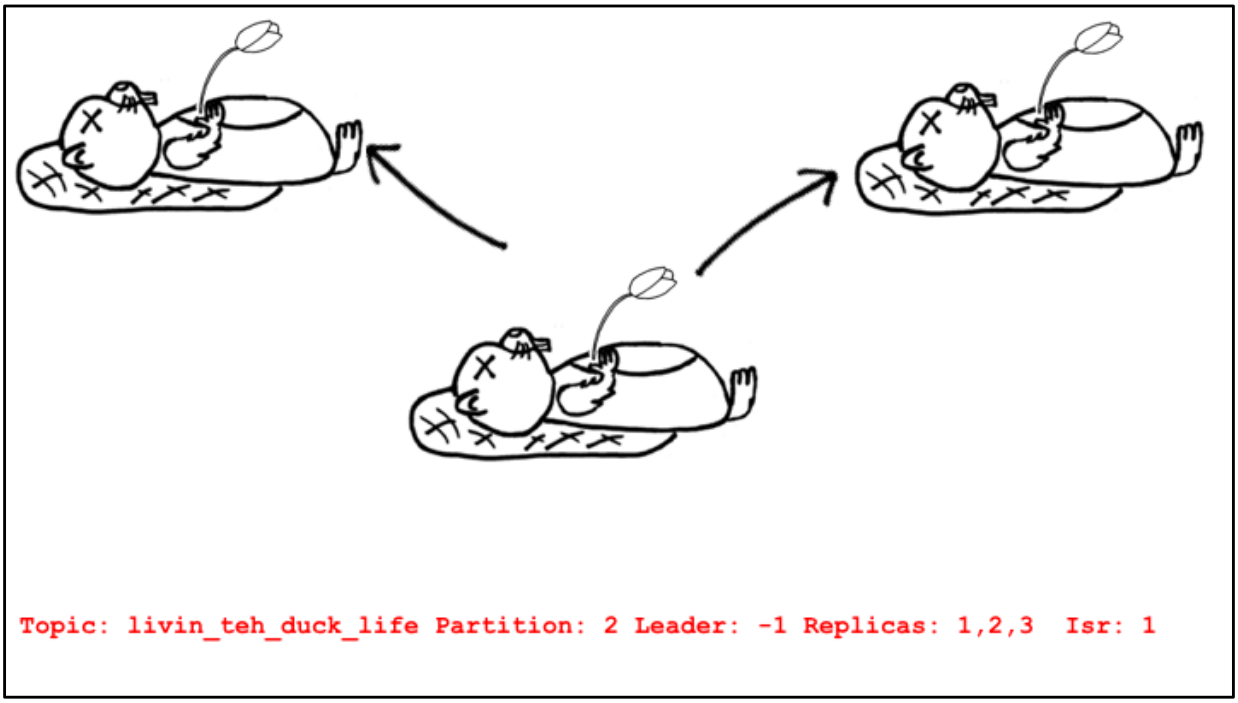

```
$ ./kafka-topics.sh --bootstrap-server broker:6667 --topic  livin_teh_duck_life -describe

Topic: livin_teh_duck_life PartitionCount:3 ReplicationFactor:3 Configs:unclean.leader.election.enable=false
      Topic: livin_teh_duck_life   Partition: 0     Leader: 2   Replicas: 2,3,1   Isr: 1,2,3
      Topic: livin_teh_duck_life   Partition: 1     Leader: 3   Replicas: 3,1,2   Isr: 1,2,3
      Topic: livin_teh_duck_life   Partition: 2     Leader: 1   Replicas: 1,2,3   Isr: 2,3
```

Billy needs to practice what happens when he as the leader of a partition decides to take a “nap”. Since the leader of a partition is responsible for receiving all new batches and for answering requests it is important that a new leader be elected quickly. The operating procedure among the brokers it not to utilize majority vote but instead promote any of the in sync replicas can be elected to be the new leader. This helps to minimize the amount of storage redundancy needed throughout the cluster.

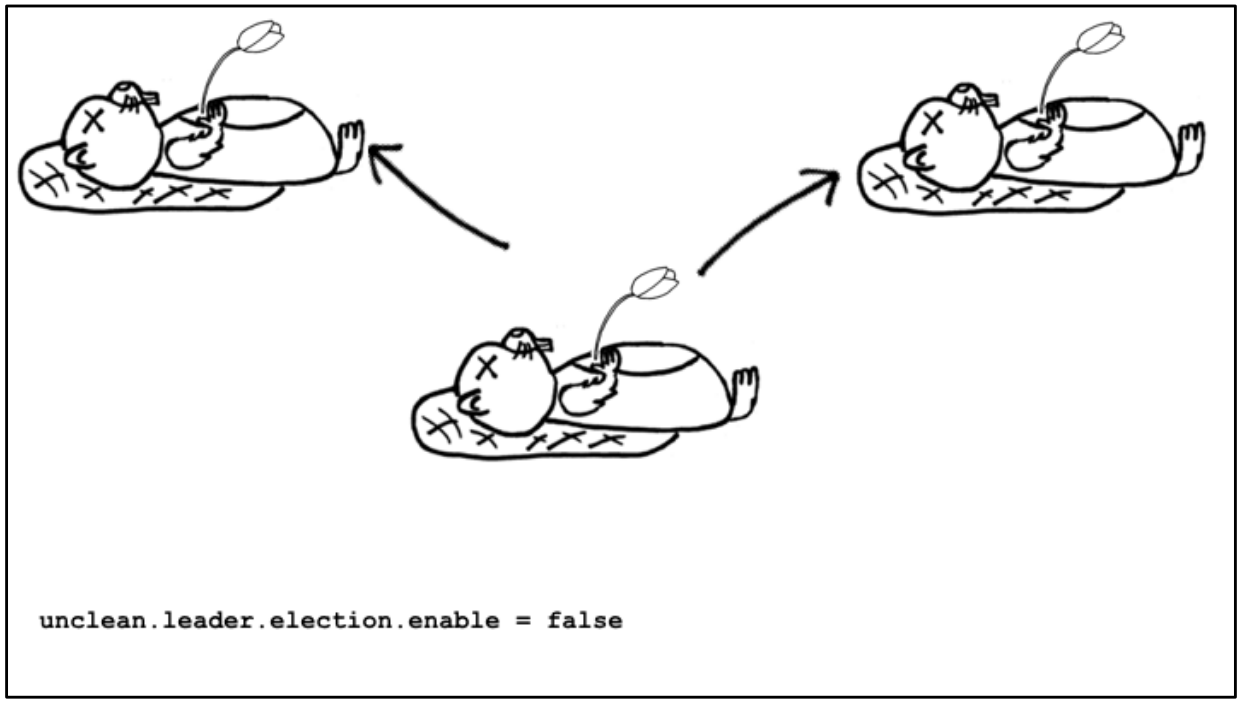



```
$ ./kafka-topics.sh --bootstrap-server broker:6667 --topic  livin_teh_duck_life -describe

Topic: livin_teh_duck_life PartitionCount:3 ReplicationFactor:3 Configs:unclean.leader.election.enable=false
      Topic: livin_teh_duck_life   Partition: 0     Leader: 2   Replicas: 2,3,1   Isr: 2,3
      Topic: livin_teh_duck_life   Partition: 1     Leader: 3   Replicas: 3,1,2   Isr: 2,3
      Topic: livin_teh_duck_life   Partition: 2     Leader: 2   Replicas: 1,2,3   Isr: 2,3
```

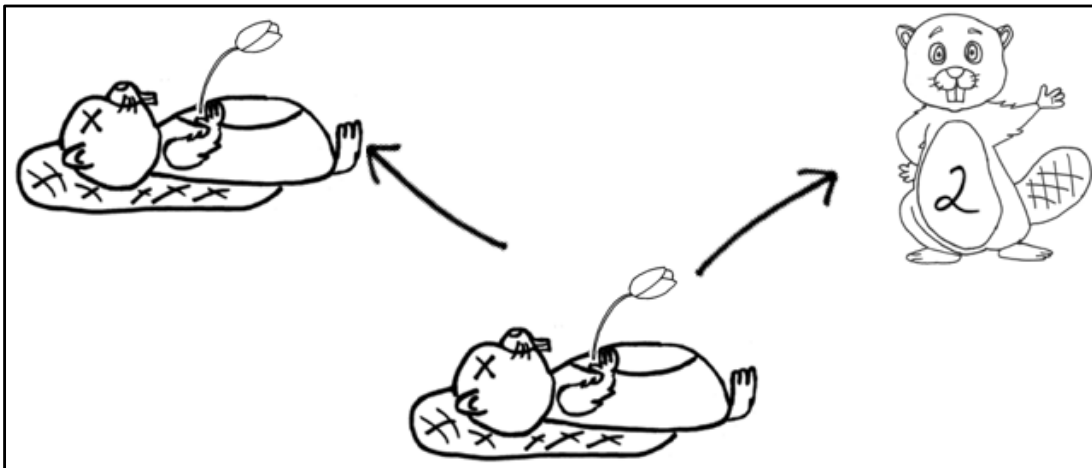
So with Billy taking a “nap”, Bobby is now elected to be the leader of the partition.



Billy, Bobby and Barry of course have to also practice drills for what happens when they all take “naps” at the same time. In this scenario Billy was the last leader of the partition and therefore the last member of the in sync replicas. However when he is “napping’ there is no active leader and therefore messages can’t be replicated and new messages cannot be received. Now we are in a really tricky situations.

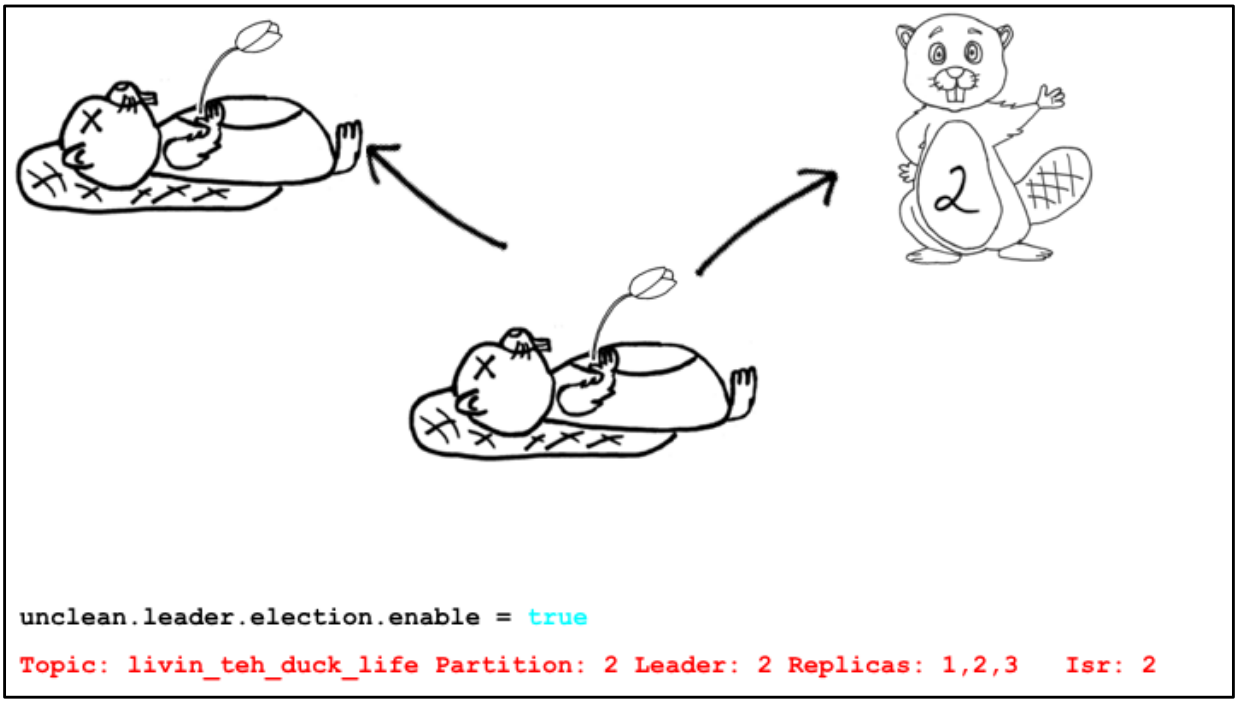


Since Billy, Bobby, and Barry all prioritize consistency they ensure their configuration is setup to make sure a new leader isn't elected that might not have had all of the messages.



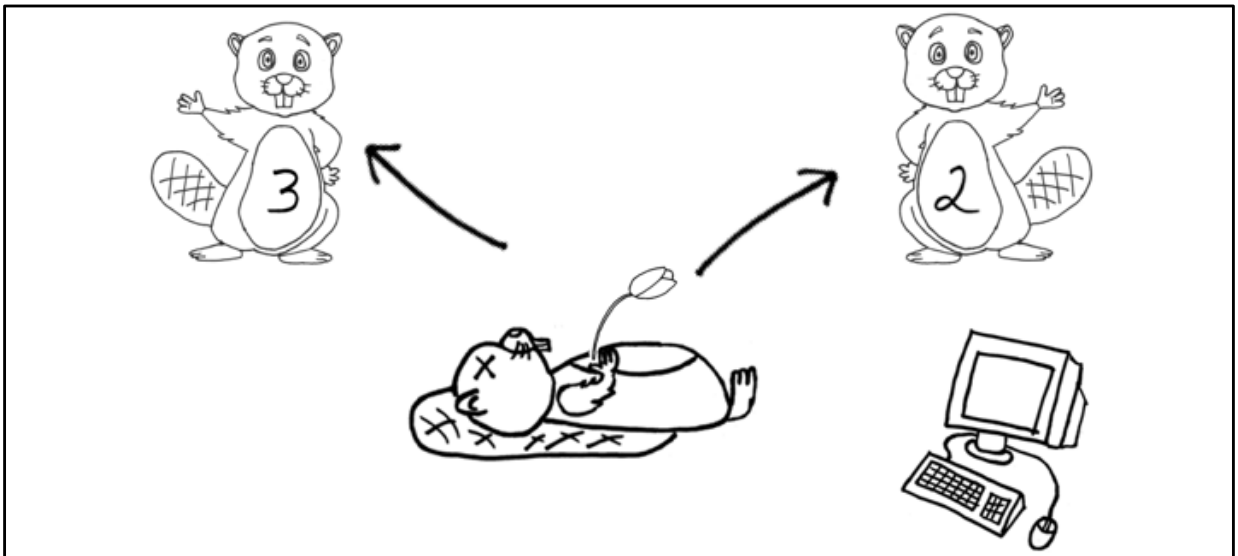

```
unclean.leader.election.enable = false
Topic: livin_teh_duck_life Partition: 2 Leader: -1 Replicas: 1,2,3  Isr: 1
```

The impact of this decision is that if Bobby wakes up from his “nap” but Billy is still asleep then Bobby cannot be elected to be the new leader, so no new messages can be received.




```
unclean.leader.election.enable = true
Topic: livin_teh_duck_life Partition: 2 Leader: 2 Replicas: 1,2,3   Isr: 2
```

In extreme situations, the setting for that specific topic can be adjust to allow Bobby to be elected leader. This will result in data loss as Bobby did not have all of the same messages that Billy had. This is because when either Bobby or Billy wake up they will throw away any messages they might have to sync with the leader.




```
unclean.leader.election.enable = false
```

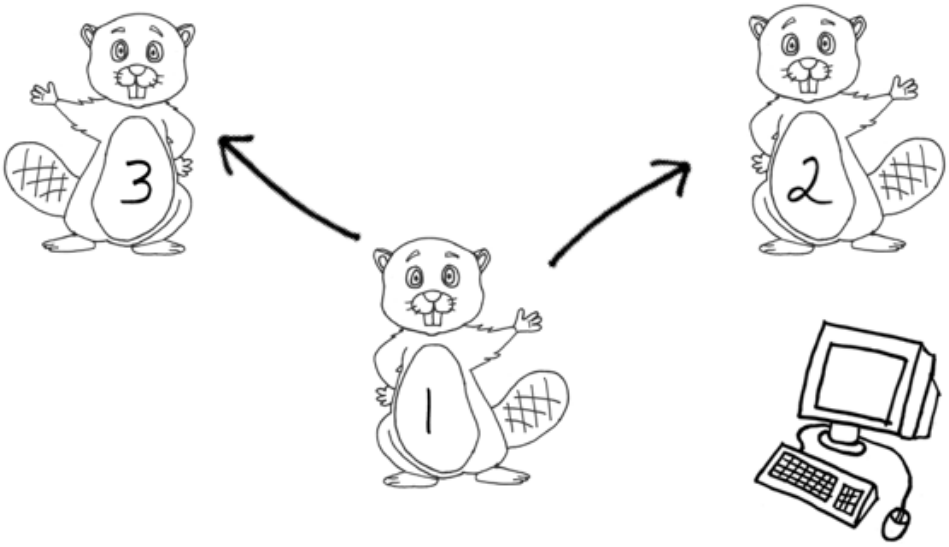
```
Topic: livin_teh_duck_life Partition: 2 Leader: -1 Replicas: 1,2,3  Isr: 1
```

In the situations where Billy will just not wake up from his “nap”, but Bobby and Barry are awake then with a lot of coordination and checking the executives who oversee the brokers can force the election of a new leader. Forcing the election of a new leader requires thorough checking of the each broker’s partition replica. They are checking to make sure the brokers are either in sync with each other or finding out which broker has the latest data to determine who should be promoted to be the new leader.


```
unclean.leader.election.enable = false
```

```
Topic: livin_teh_duck_life Partition: 2 Leader: 2 Replicas: 1,2,3   Isr: 2,3
```

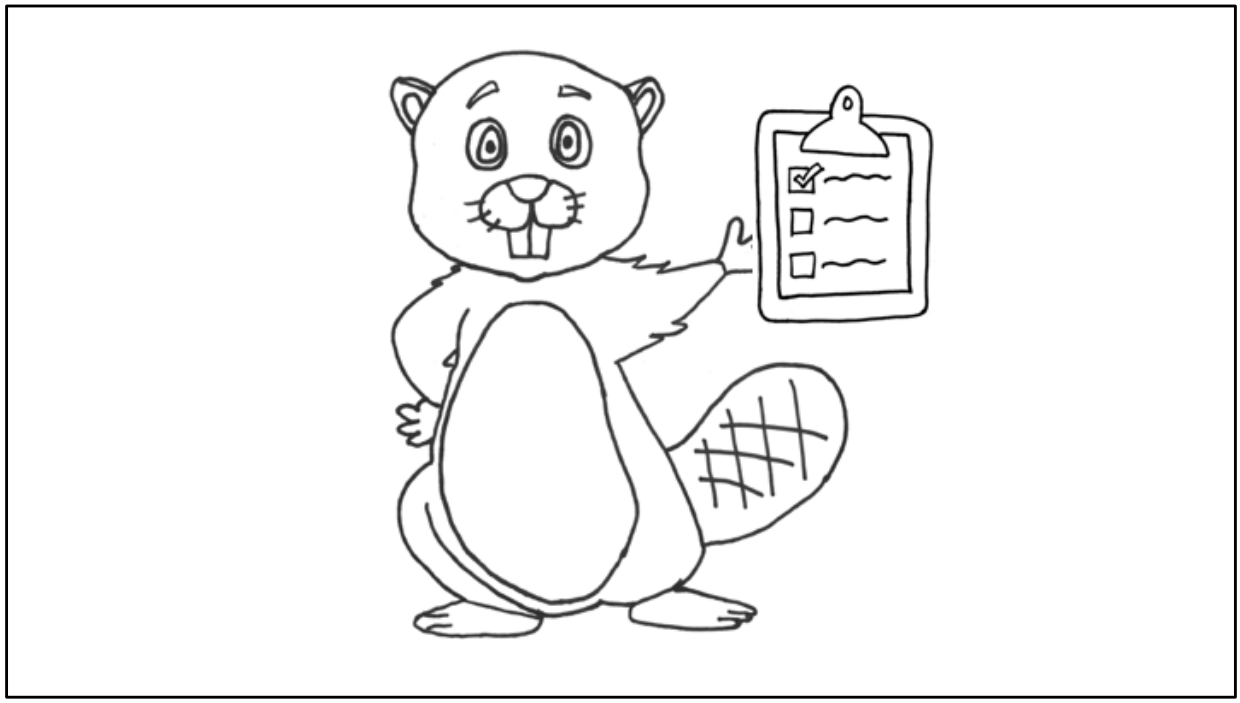
Once the broker with the latest data is determined, the executives can go into Zookeeper and promote Bobby to be the leader of the partition. This is a very tedious and manual workplan but it is important because it will ensure that there is no data loss for the partition.




```
unclean.leader.election.enable = false
```

```
Topic: livin_teh_duck_life Partition: 2 Leader: 2 Replicas: 1,2,3 Isr: 1,2,3
```

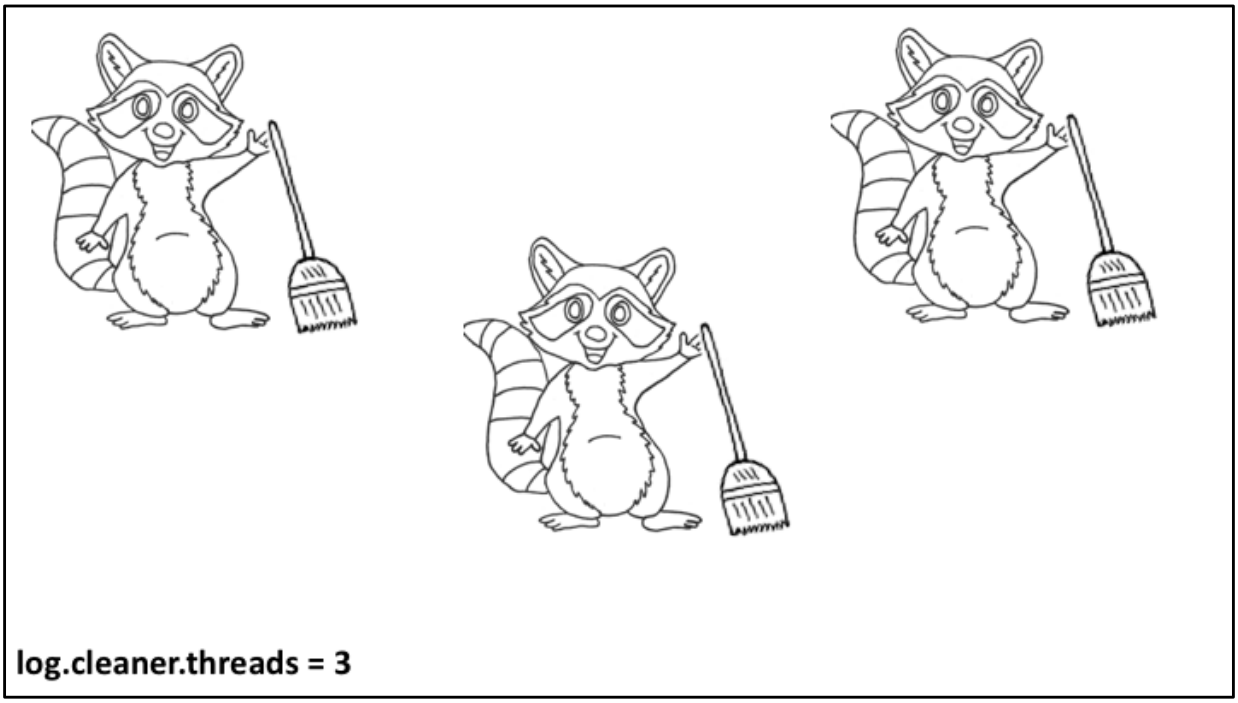
Eventually when Billy wakes up from his “nap” he can rejoin the group of in sync replicas letting Bobby take over the leadership responsibilities for awhile. These drills and operations are important for Billy to practice so he knows what do when failure occurs.



Billy is once again excited to cross another thing off his list. One of the last things on Billy's list is to make sure that everything stays nice and tidy on the broker. This is important because the broker can only store a certain amount of messages and Billy wants to make sure there is enough space for new messages coming in. Billy relies on his trusty cleaning crew led by Ricky to keep the broker clean.

log.cleaner.enabled = true

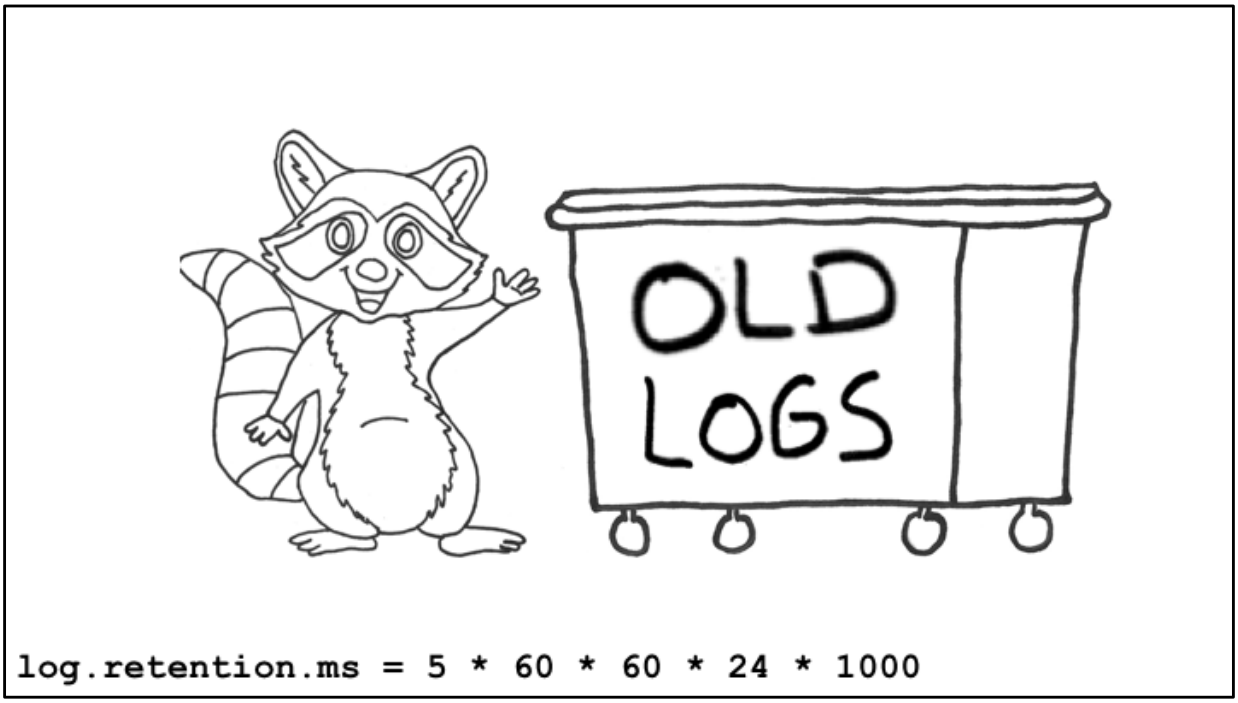
Ricky loves to keep things clean. Ricky knows that if he does not do his job then the broker will quickly fill up and have to shut down because there will be no room for Inky and Billy to do their work.



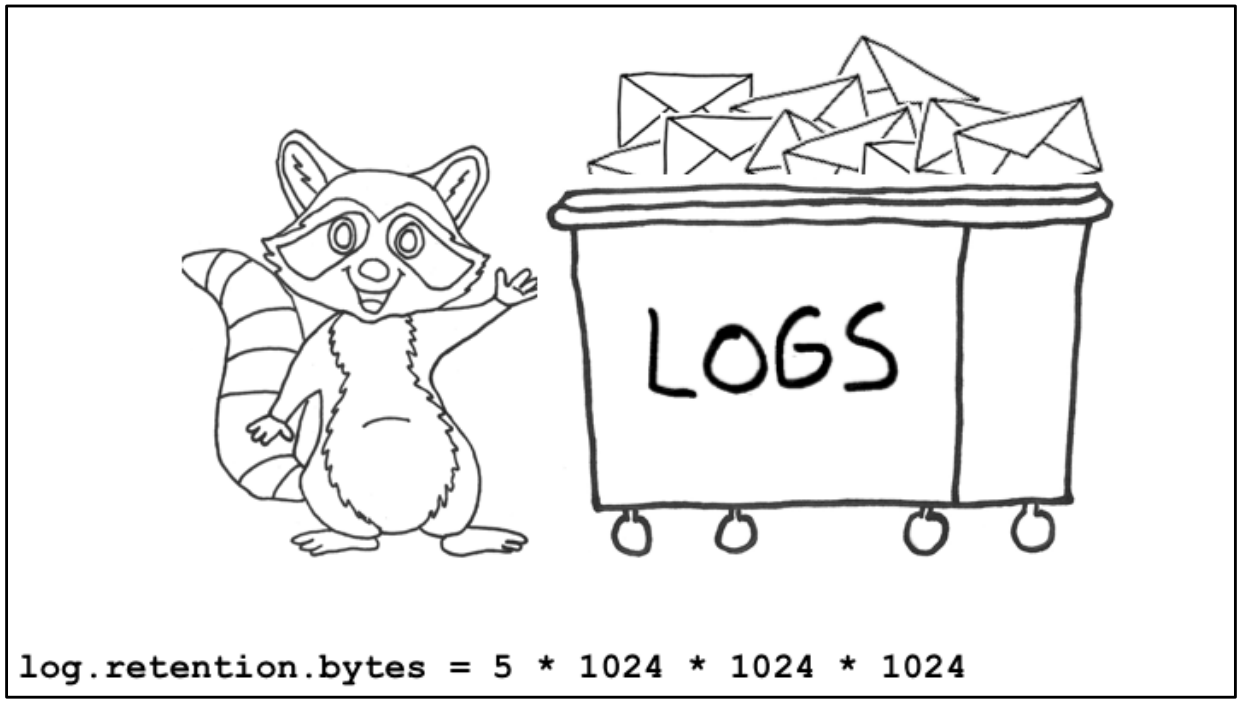
Ricky however doe not have to work alone. Ricky works with a great team who each divide up the work around the cluster. The team is responsible for cleaning up each partition getting rid of the log and index files.

log.cleanup.policy = delete

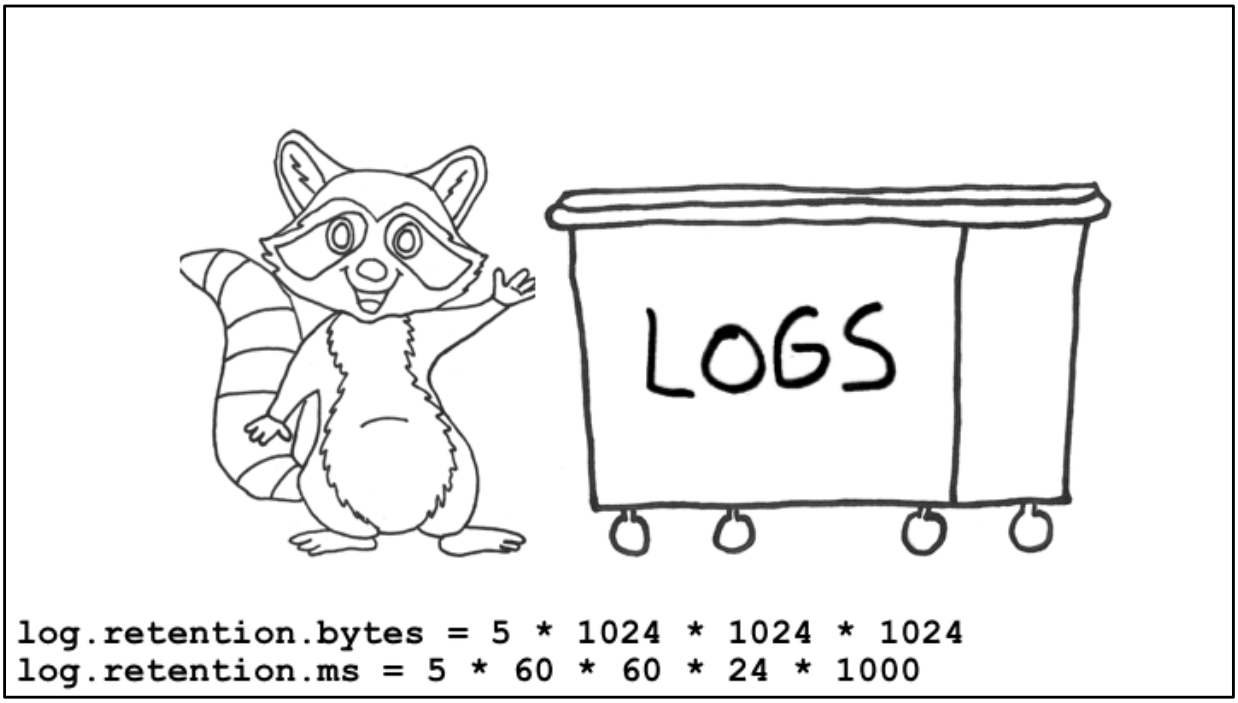
Ricky has to be very particular about how to perform the cleanup, ensuring that he doesn't destroy any messages that should be kept. The broker has a default policy for each topic for how Ricky is supposed to clean up but the policy can be overridden at the individual topic level. The broker Ricky works at has a policy to just delete any messages that should be cleaned up.



Now that Ricky knows what he is supposed to do with messages that should be cleaned, Ricky has to determine if a message is eligible to be cleaned up. Ricky might determine to clean up messages that are old. In this case Ricky checks each log file in a partition. All of the messages in that log file must be older than the limit for Ricky to decide to throw them away. This is helpful because if consumers wanting these messages get behind or only ask for the messages infrequently then they will have a known amount of time to ensure they don't miss a message.



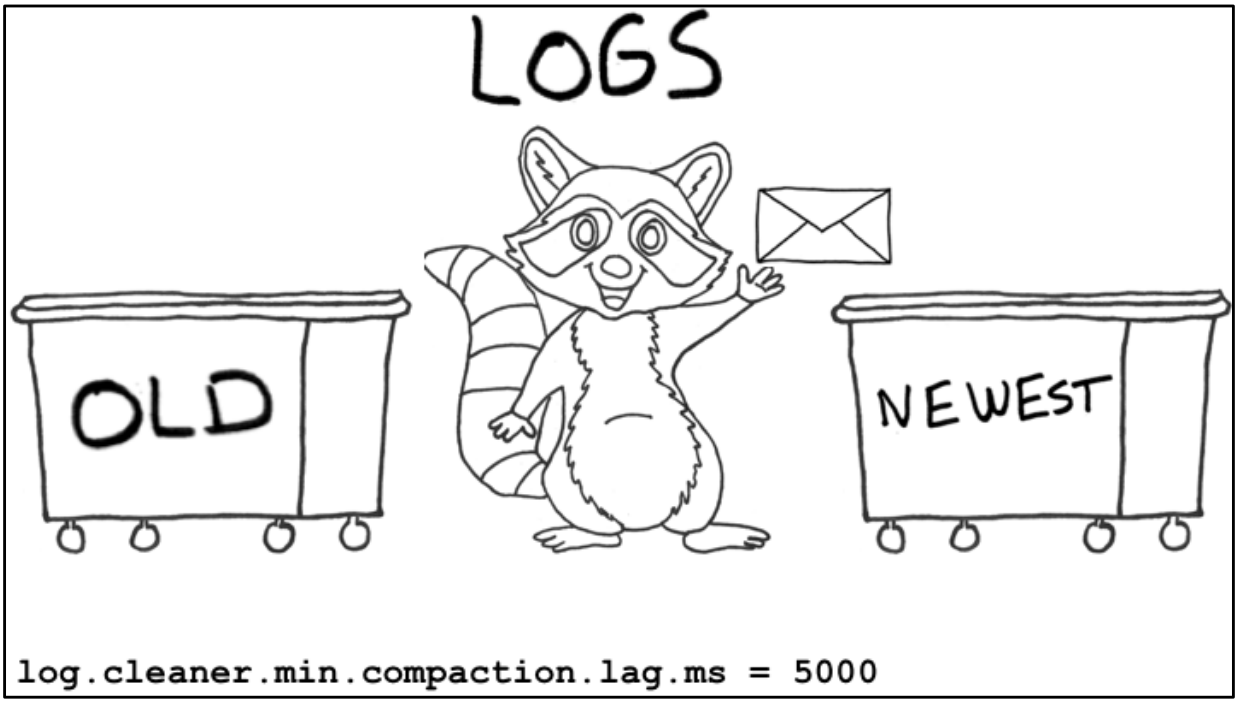
Ricky can also decide to delete messages when they are taking up too much space. Ricky finds this option helpful because it helps to prevent topics that are really really busy from taking up too much room and filling up the broker. This option does raise the risk of pushing too much data too fast for a consumer to keep up before the messages get deleted.



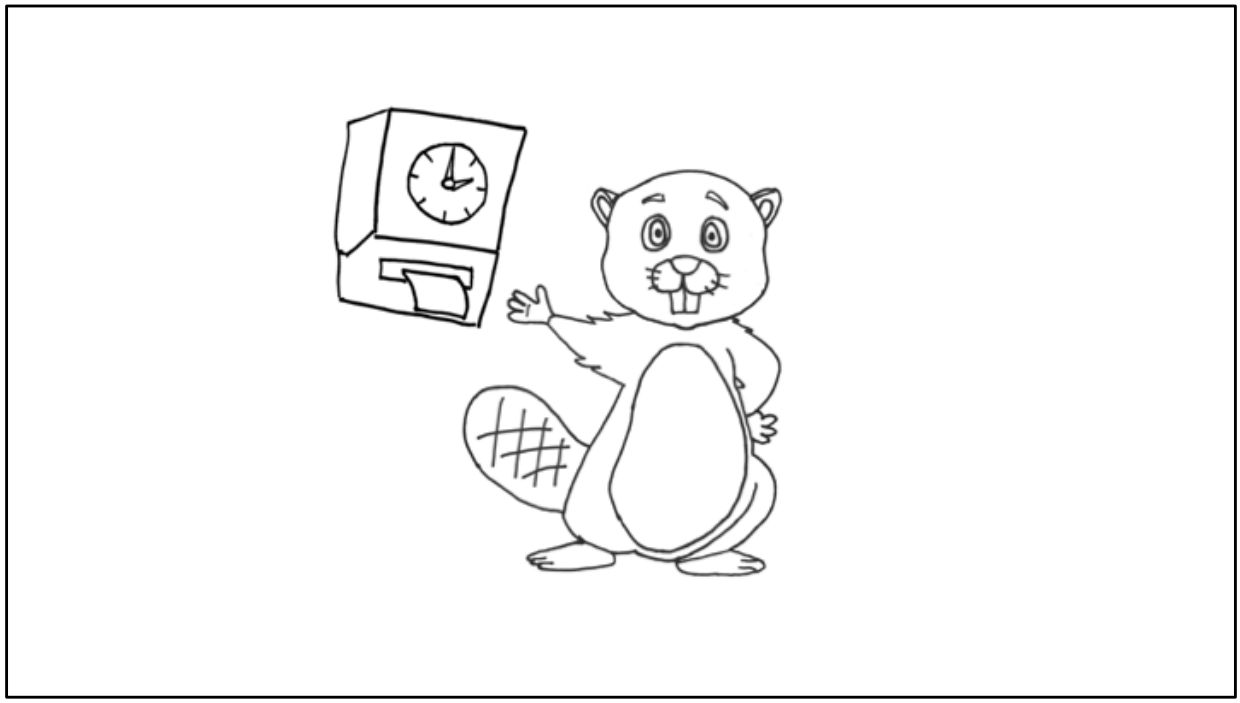
For some of the more complicated topics, Ricky might have to keep the logs cleaned based on both size and time.

log.cleanup.policy = compact

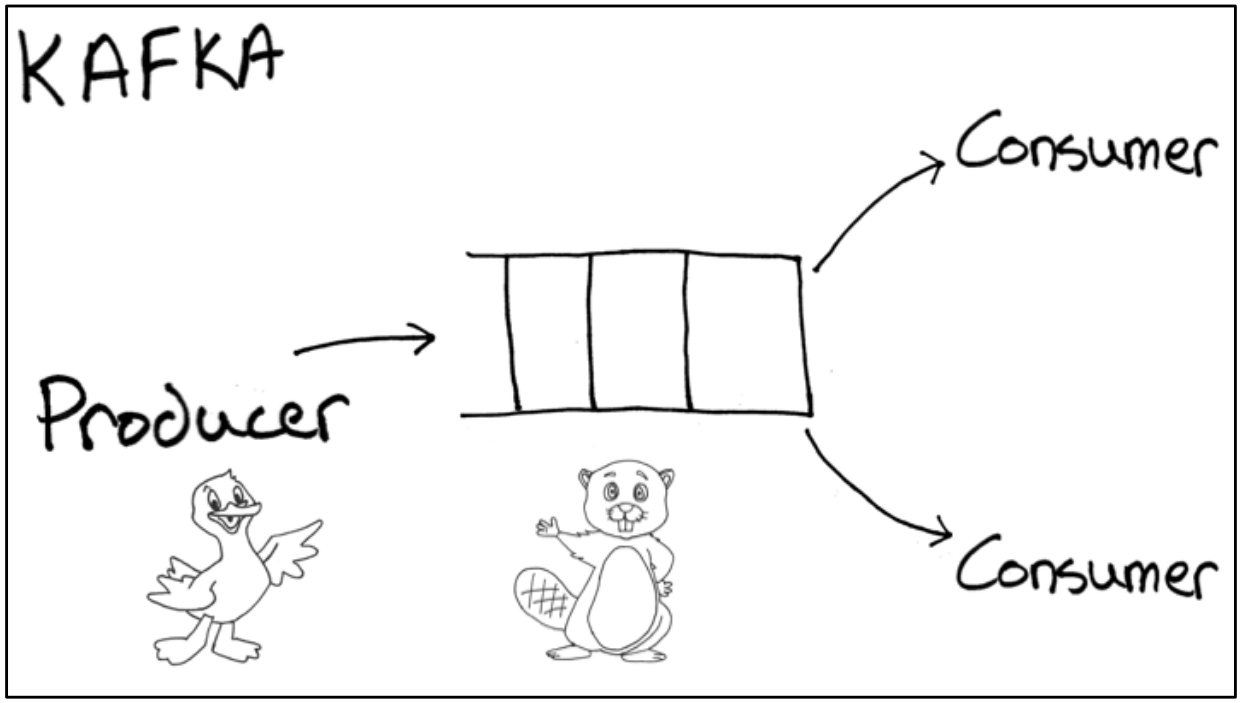
Besides deleting messages, Ricky can help to keep the broker clean by sorting through all of the messages for a topic, throwing away some and keeping others.



Ricky has a big job ahead of him. Ricky is supposed to sort through all of the messages in a partition and only keep the newest message for each key. This means that consumers can see the latest update for a given key without having to see all the previous updates or worry about the value for a given key being deleted. This option helps Ricky keep the broker clean but only really helps if the producer guarantees all the messages for the same key always go to the same partition. Otherwise multiple versions of the same key will be stored in each partition. Ricky loves his job and is happy to help Billy out.

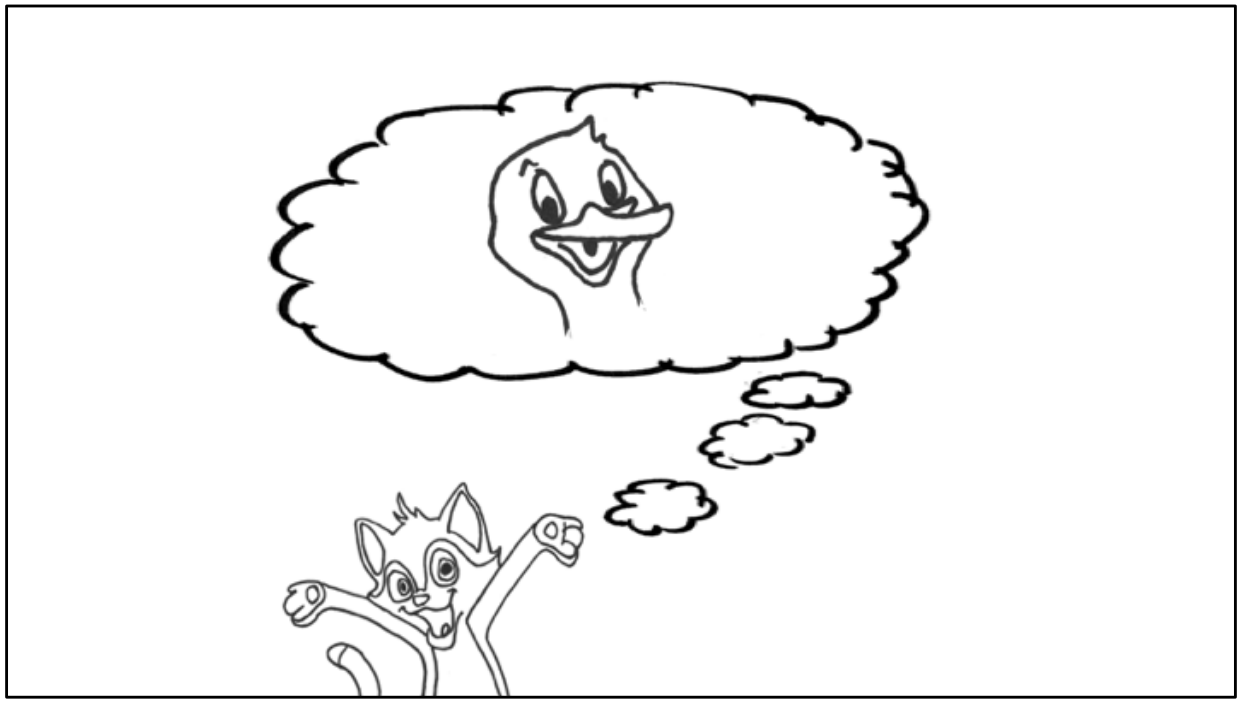


With all of the items on his checklist completed, Billy is excited but exhausted after his first day. He clocks out for the day over confident for the day that he's now an expert at all of the tasks for the broker. Billy doesn't realize that while he's got the main tasks done, when he comes back tomorrow he'll have to deal with transactions, quotas, ACLs, and group coordination. But we won't tell Billy that just yet because we want him to have a good night sleep.

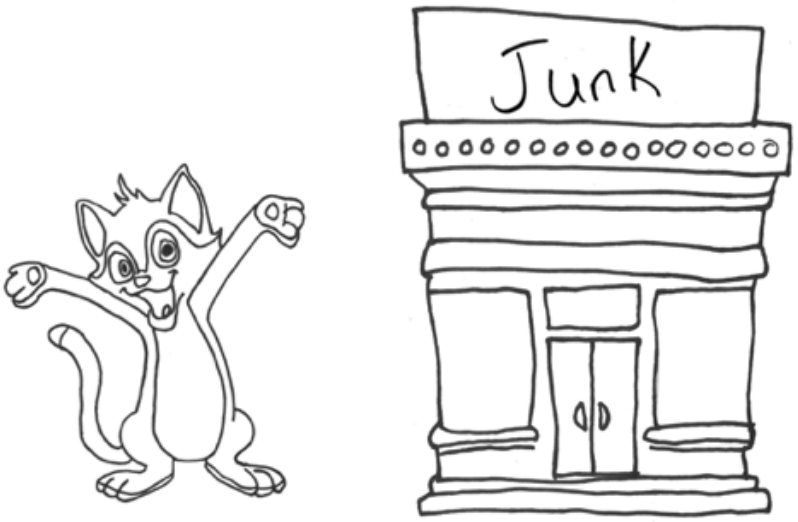

KAFKA
Consumer
Producer
Consumer


Kitty
Opens
Shop

Kitty just graduated from school and she she started to think about what she wanted to do with her life. The only thing she knew in her heart is that she wanted to be rich.

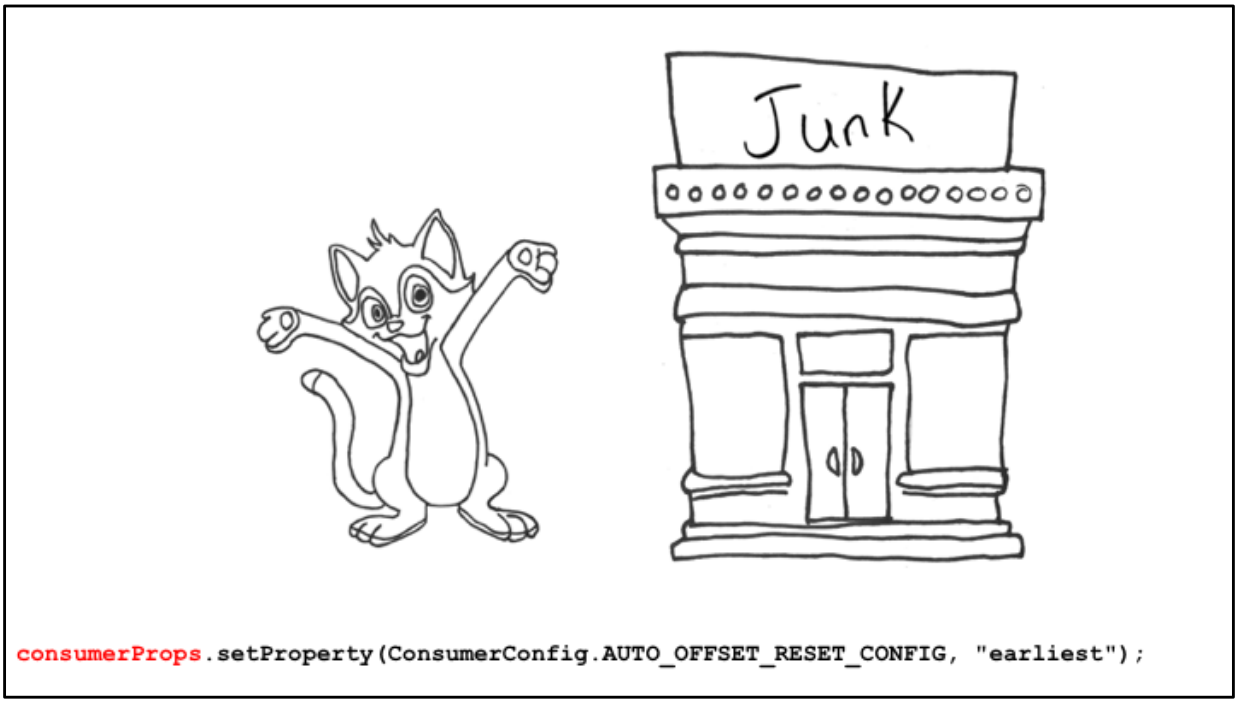


In talking with her friends Kitty began to think about her friend Dilly who was making a living as a social media influencer. Kitty knew that the court of public opinion was fickle and wanted a more stable foundation upon which to found her business and become rich. Kitty however was inspired by Dilly and combined with her education from the Mel Brooks Spaceball School of Business, she began to implement what she learned which was, “Merchandising, Merchandising, Merchandising”.



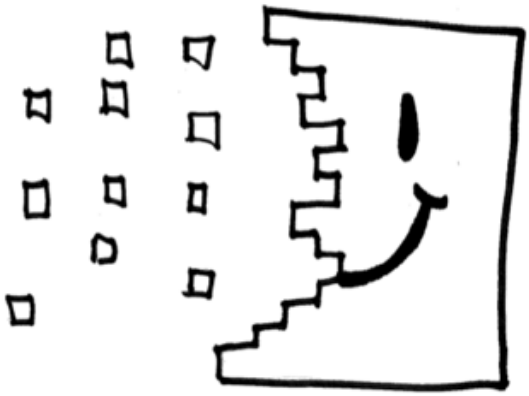

```
Properties consumerProps = new Properties();
String brokers = "broker1:6667,broker2:6667,broker3:6667");
consumerProps.setProperty(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, brokers);
consumerProps.put(ConsumerConfig.GROUP_ID_CONFIG, "kittys_junk_shop");
```

So Kitty decided to open a shop for selling tshirts, mugs, fidget spinners, and whatever else she could think of all based on popular posts on social media. She was going to sell other people's content to make a pop culture buck. You see while school had taught her about how to setup a business it had not exactly emphasized ethics. Kitty quickly found the distribution center behind the social media posts and started to subscribe to their messages.




```
consumerProps.setProperty(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
```

Kitty knew that there was a lot of good content out there so she made sure to default her offset to the earliest available offset in the log to read all of the messages available in Kafka. This setting would be used for each new topic she subscribed to ensuring she did not miss a thing.



```
consumerProps.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
                  SelfieHashTagDeserializer.class.getName());
consumerProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
                  SelfieDeserializer.class.getName());
```

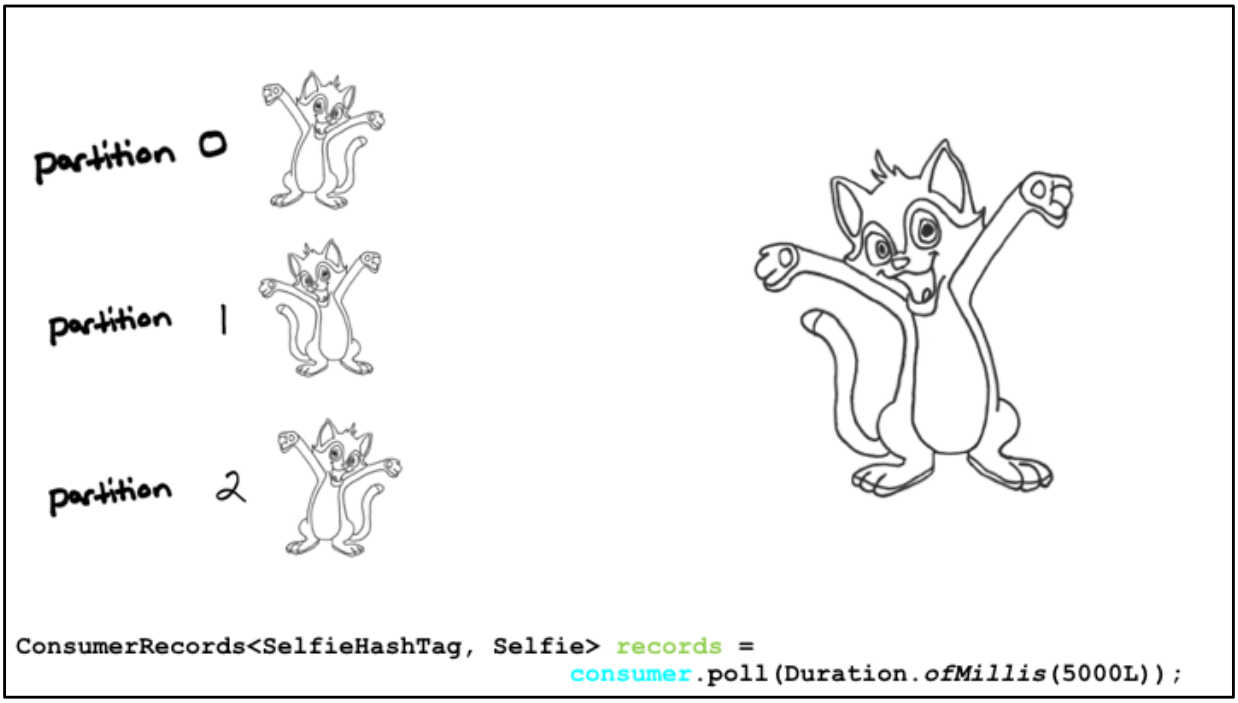
Kitty also had to configure how to unpack and interpret each of the messages being sent. This meant that she'd be able to benefit from the efficient storage and compression created by the producer and brokers and only have to reassemble the posts once they had arrived at the store.


```
Consumer<SelfieHashTag, Selfie> consumer = new KafkaConsumer<>(consumerProps);
```

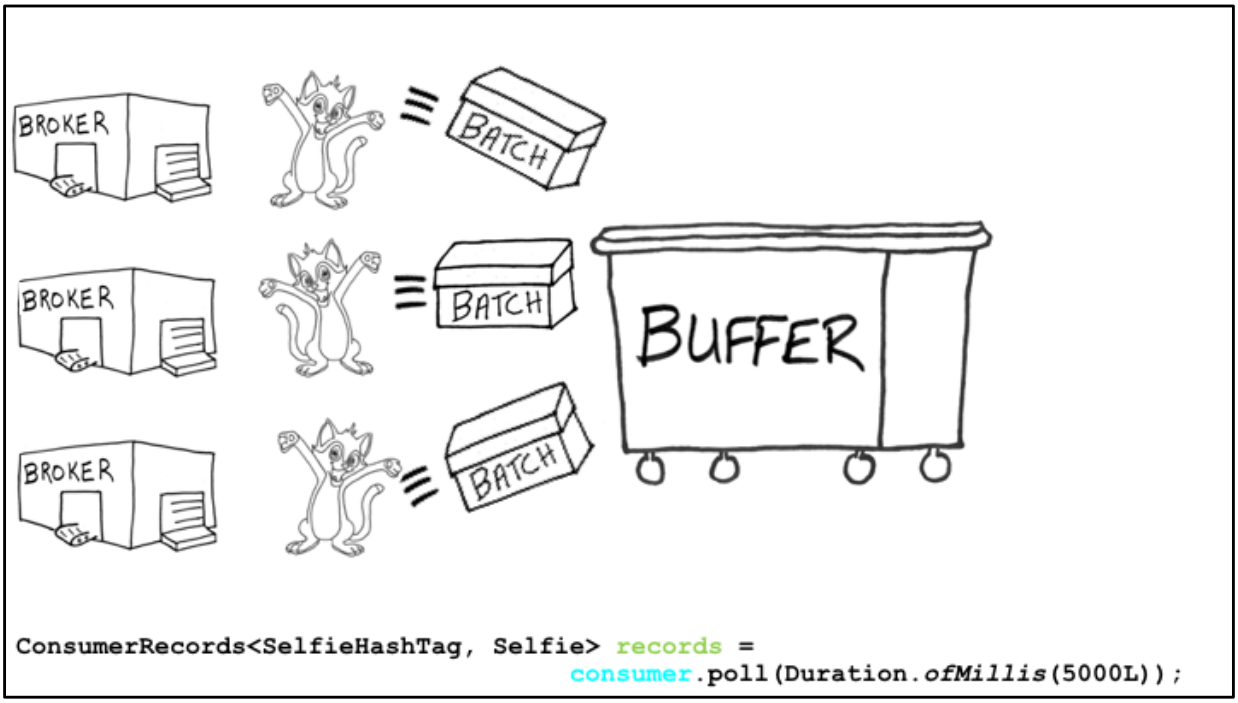
After Kitty has configured how to talk to the cluster and how to read the messages she was ready to open shop.

```
String topicName = "livin_teh_duck_life";
consumer.subscribe(Collections.singleton(topicName));
```

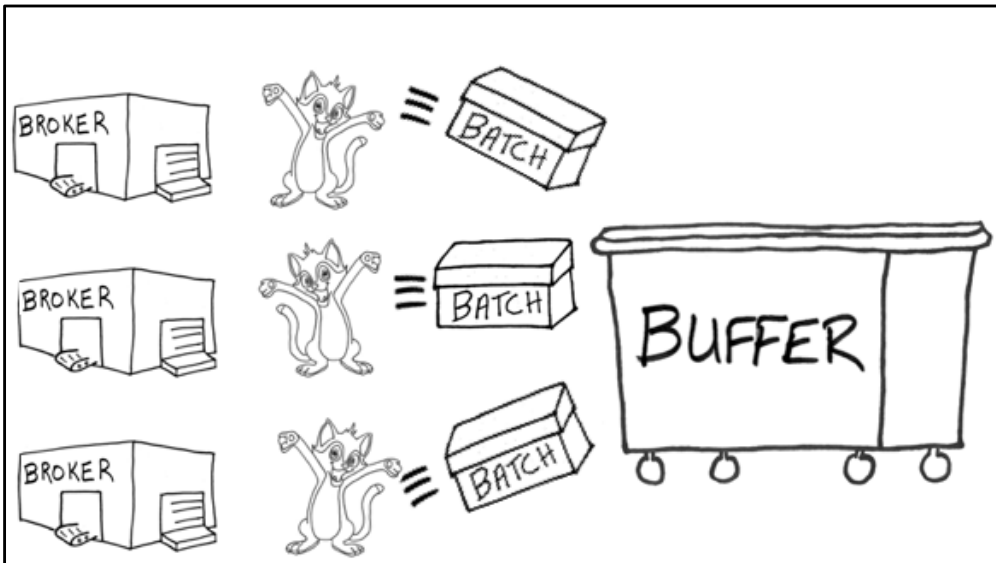
After opening shop, Kitty had to decide which influencers she'd print merchandise for. She decided to start small and since she was friends with Dilly subscribed to what Dilly was posting. The best part was that Dilly didn't know she was subscribed so Kitty could unsubscribe at any time without hurting her feelings.



After subscribing to the topic of Dilly's messages, Kitty just had to sit back and wait for the messages to arrive. Kitty could be lazy like that because she had hired helpers to work in the stockroom. The workers are responsible for discovering the number of partitions for the topic and then dividing up the work based on which broker is the leader for those particular partitions.

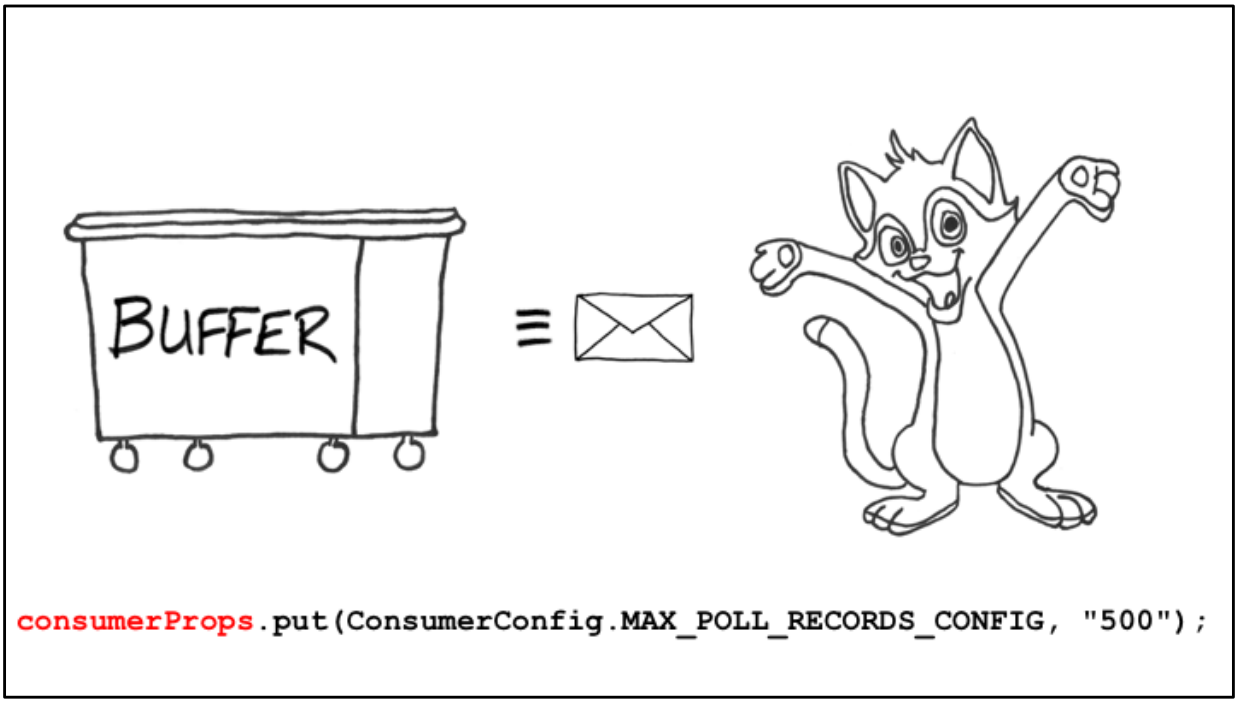


The workers are really fast, running out to the individual brokers and fetching batches of messages. The batches of messages get retrieved in the order they were written. The workers then return to Kitty's store and dump the batches into a single buffer until it gets full. This means there is no guarantee of fairness between the workers but instead priority is given to whomever is the fastest and can put the most messages in the buffer before it is full.

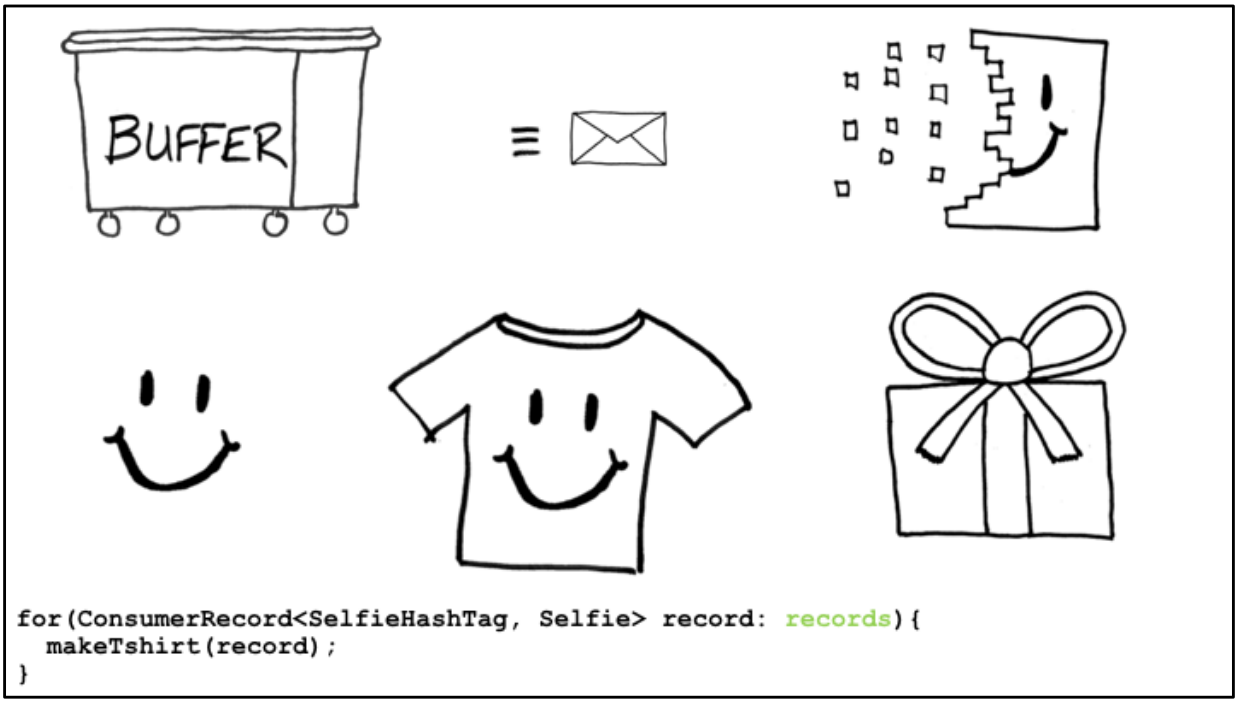



```
consumerProps.put(ConsumerConfig.FETCH_MAX_BYTES_CONFIG, "1048576");
consumerProps.put(ConsumerConfig.FETCH_MIN_BYTES_CONFIG, "1");
consumerProps.put(ConsumerConfig.FETCH_MAX_WAIT_MS_CONFIG, "500");
```

The workers do try to be as efficient as possible trying to minimize the number of trips from the broker but also not waiting too long.



Once the buffer starts to get messages, Kitty can start to do her work taking out a few messages at a time from the buffer.



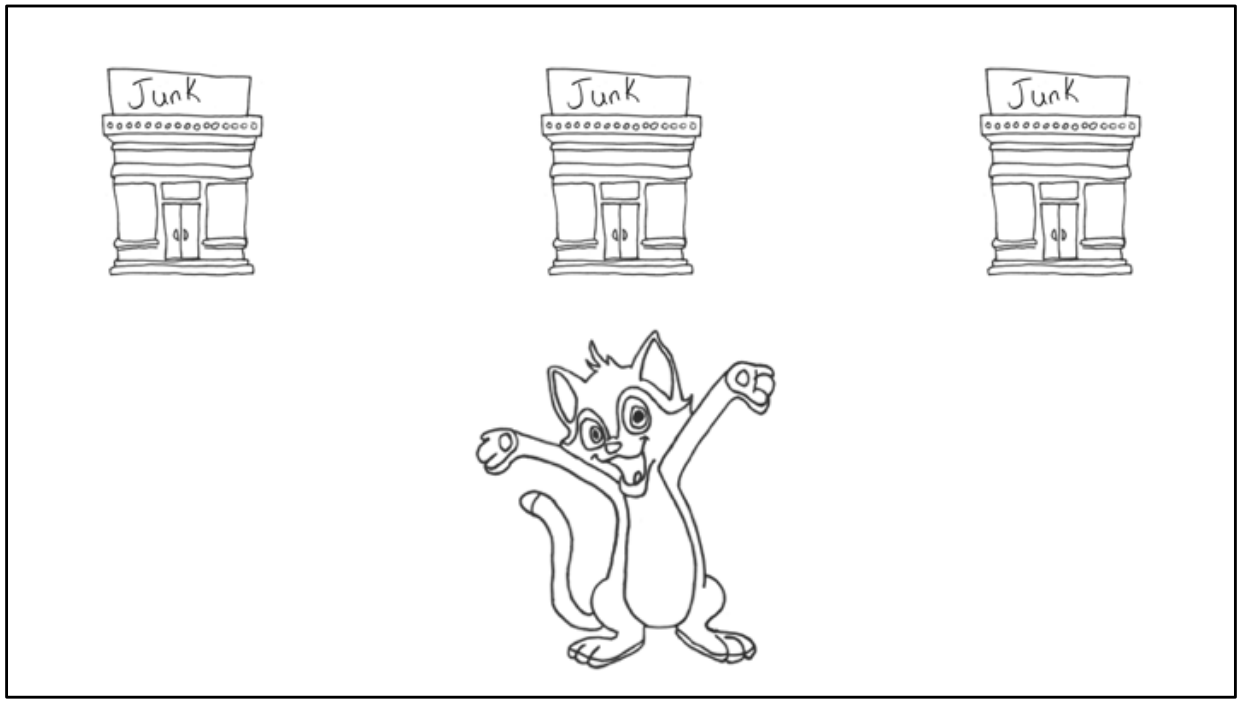
After the messages come out of the buffer, Kitty quickly gets to work reading the message, converting the post into an image, printing the merchandise and then gift wrapping it for her customers. All the while Kitty is busy working those stockroom workers are hard at work refilling the buffer with messages often faster than Kitty can print the tshirts.

```
consumer.commitSync();
```

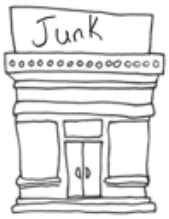
Once Kitty has successfully pulled a batch of messages, printed them on worthless items, and shipped them off, she can track her progress by telling Kafka to move her offset past the end of the last pulled message. If Kitty had any problems processing a message she cannot commit her offsets as she might cause her to mark message as done that she didn't process. Committing the offsets will ensure that the next time she opens the store she can skip all of the messages she has already seen.


```
consumer.close();
```

At the end of the day, the stock room workers tirelessly toiled and Kitty kinda worked, Kitty was after really strong sales.



Kitty kept the system running for several months and continued to have resounding success as people bought silly tshirts of fancy coffees, celebrity breakfast food, and selfies. Kitty was so successful that she eventually needed to open up multiple shops to keep up with all of Dilly's posts. Kitty now had the challenge of figuring out how to coordinate the work between the various stores.



partition 0

```
TopicPartition topicPartition = new
                            TopicPartition("livin_teh_duck_life", 0);
consumer.assign(Collections.singleton(topicPartition));
```

Kitty found out that she could manually manage the coordination between the various stores. Since she was still only dealing with messages from Dilly, she could easily assign a single partition to each store.

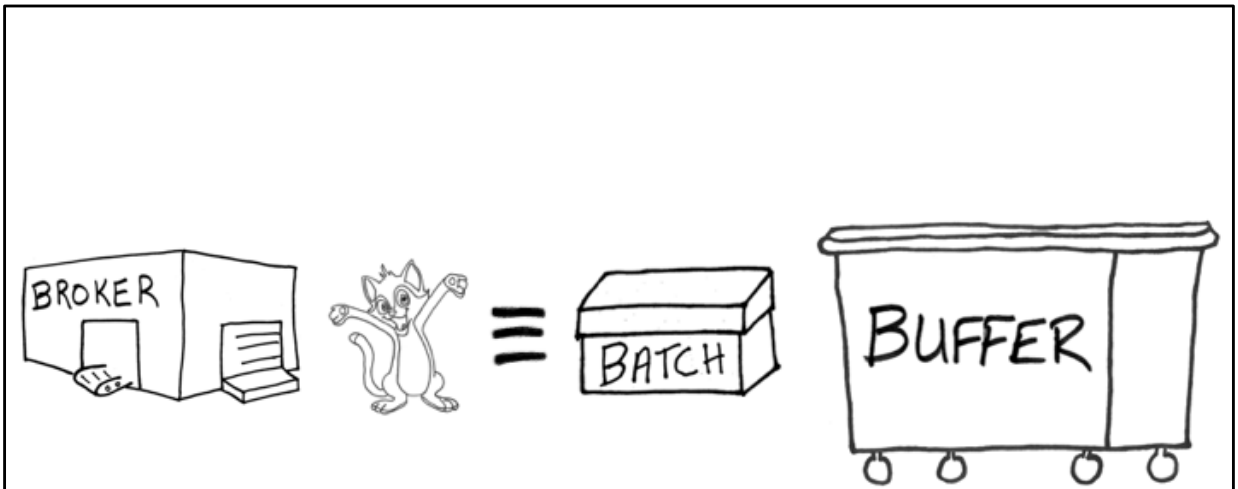

```
TopicPartition topicPartition = new
                          TopicPartition("livin_teh_duck_life", 1);
consumer.assign(Collections.singleton(topicPartition));
```

partition 0

partition 1

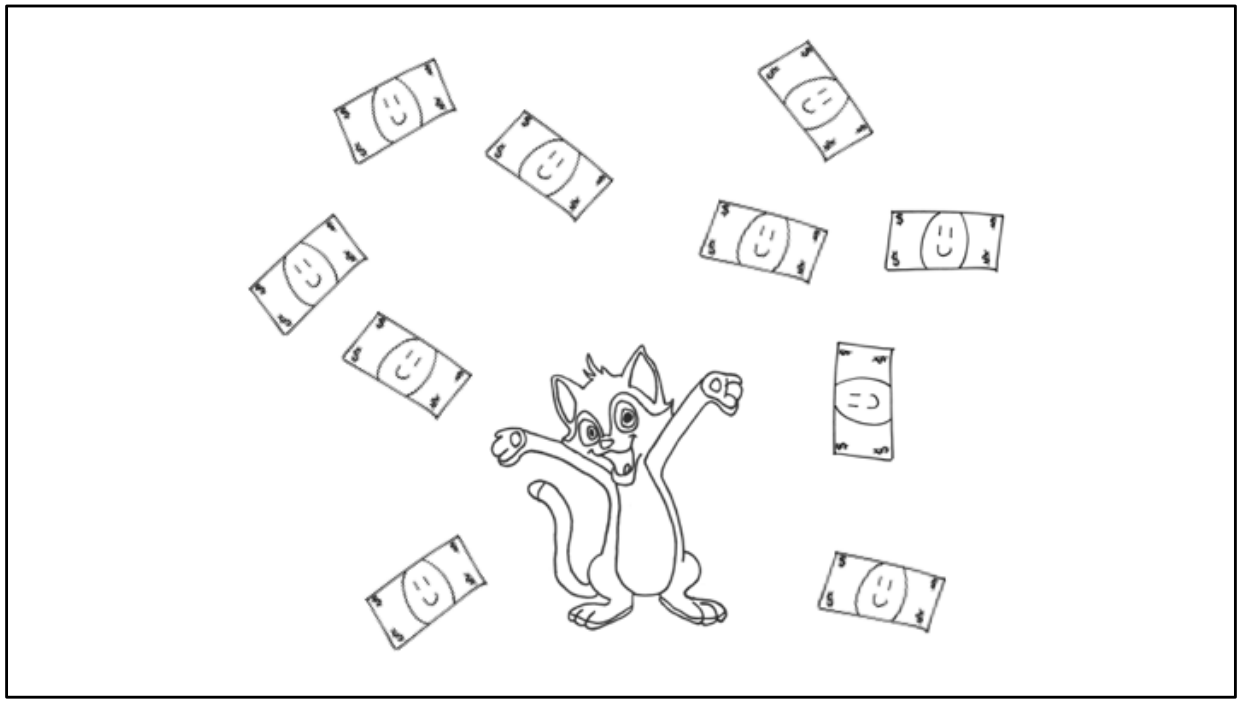
partition 2

```
TopicPartition topicPartition = new
                          TopicPartition("livin_teh_duck_life", 2);
consumer.assign(Collections.singleton(topicPartition));
```

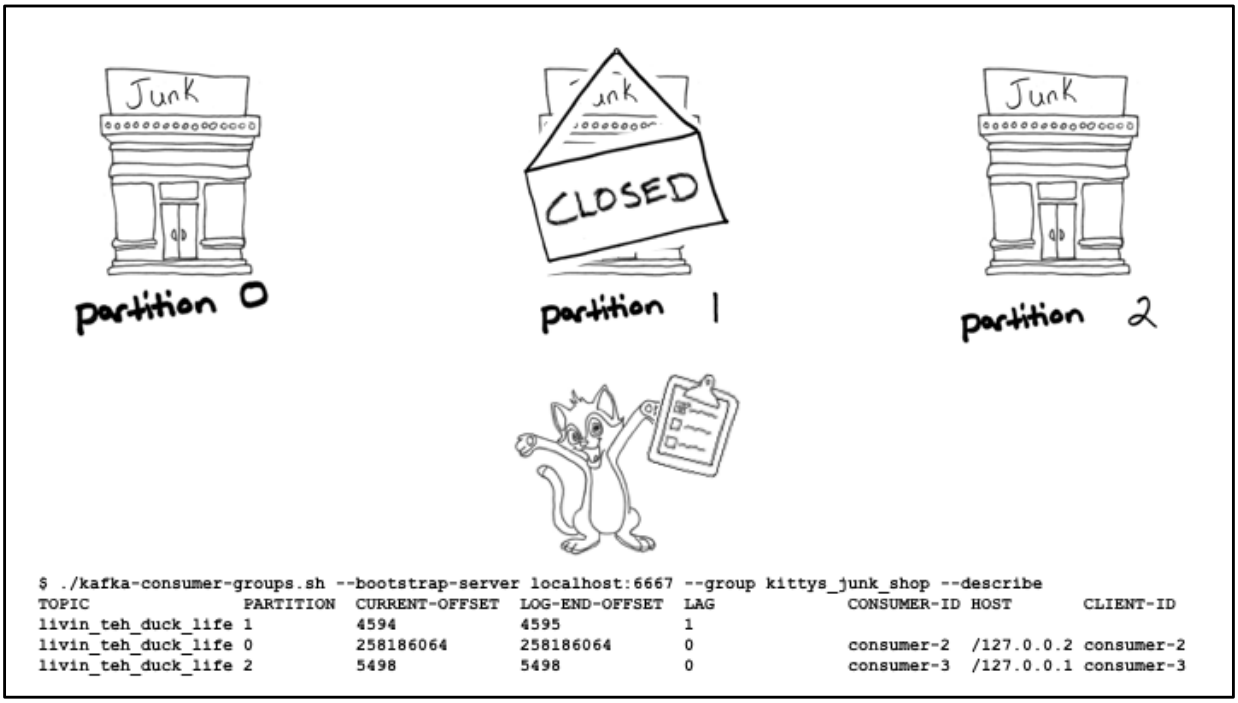



```
ConsumerRecords<SelfieHashTag, Selfie> records =
                                    consumer.poll(Duration.ofMillis(5000L));
```

Each store now only had to employ a single stockroom worker as it only needed to process a single partition. This meant that each partition got processed more fairly as it had no competition for the filling the buffer.

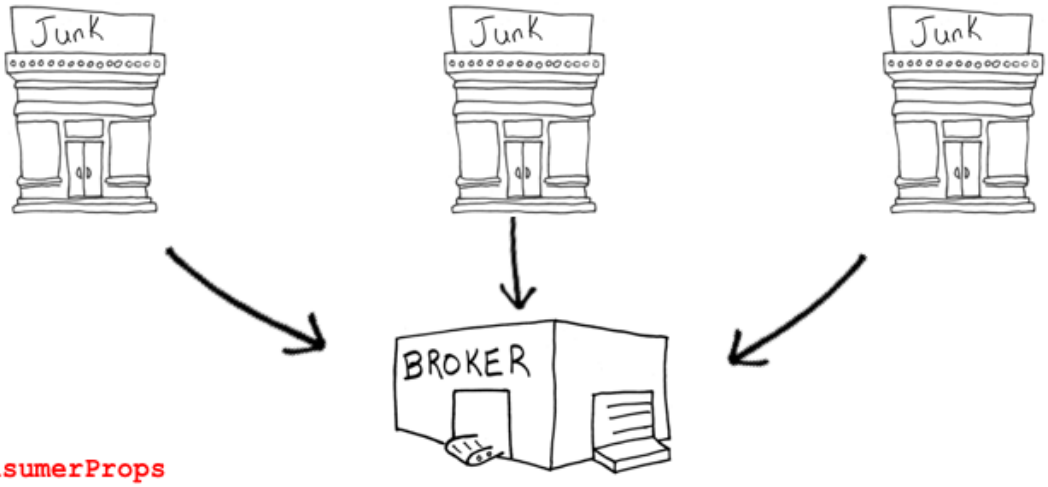


Kitty loved what she was seeing as the money started rolling in even faster as all three stores were able to keep up with the large influx of posts from Dilly.



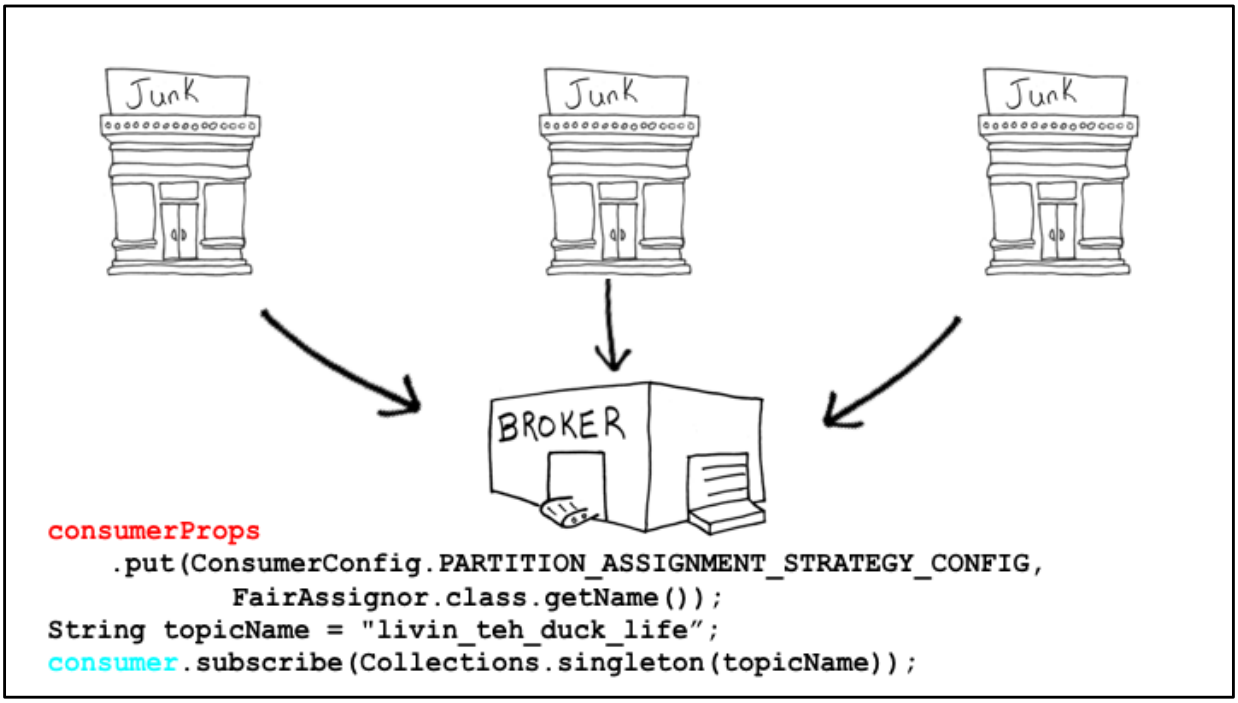

```
$ ./kafka-consumer-groups.sh --bootstrap-server localhost:6667 --group kittys_junk_shop --describe
TOPIC              PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG             CONSUMER-ID HOST       CLIENT-ID
livin_teh_duck_life 1          4594            4595            1
livin_teh_duck_life 0          258186064       258186064       0               consumer-2  /127.0.0.2 consumer-2
livin_teh_duck_life 2          5498            5498            0               consumer-3  /127.0.0.1 consumer-3
```

Kitty however hit a problem when one of her stores had an infestation of lice. It was going to be weeks before Kitty was going to be able to get into the store to determine what messages it was currently processing and which messages it had successfully shipped. The only thing Kitty knew was that no more messages for that partition were being processed and she was leaving money on the table. Kitty thought about it and knew she could reassign the partition to a different store but Kitty was worried about what might happen when she had even more partitions to manage and more stores. Kitty also didn't want to deal with micromanaging this situation and was instead hoping to sit by the pool counting her fat dough.



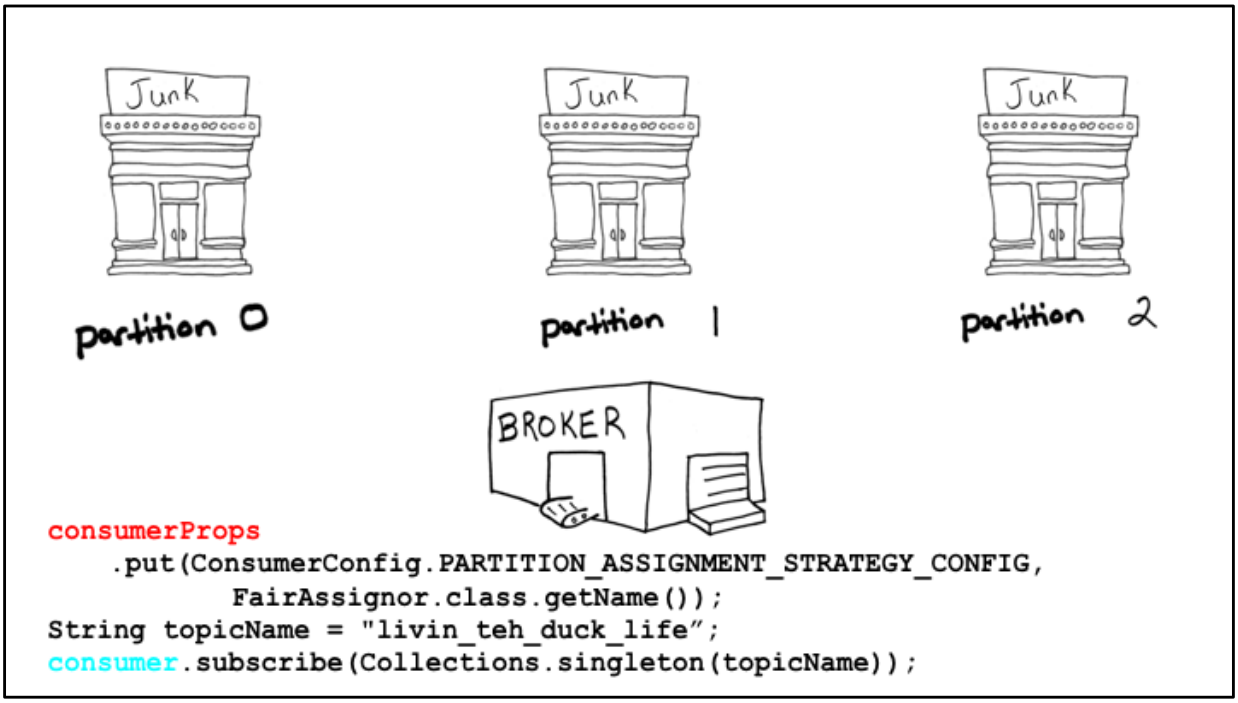

```
consumerProps
    .put(ConsumerConfig.PARTITION_ASSIGNMENT_STRATEGY_CONFIG,
            FairAssignor.class.getName());
String topicName = "livin_teh_duck_life";
consumer.subscribe(Collections.singleton(topicName));
```

Kitty discovered there was a premium feature provided by the Kafka cluster which was known as group coordination. If all of her stores subscribed to Kafka with the same group id, a Kafka broker would be elected coordinator for that group and be responsible for telling each store which partition it was responsible for consuming. Kitty could specify the rules for assigning if she wished as well. The best part was this strategy would work no matter what for when Kitty started to subscribe to more topics and as she opened more stores. The only restriction was Kitty could not have more stores than partitions because then her store would be bored.




```
consumerProps
    .put(ConsumerConfig.PARTITION_ASSIGNMENT_STRATEGY_CONFIG,
            FairAssignor.class.getName());
String topicName = "livin_teh_duck_life";
consumer.subscribe(Collections.singleton(topicName));
```

So as each store opens and connects to the broker, Kafka determines the size of the group and the number of partitions and starts to assign out the work.




```
consumerProps
    .put(ConsumerConfig.PARTITION_ASSIGNMENT_STRATEGY_CONFIG,
            FairAssignor.class.getName());
String topicName = "livin_teh_duck_life";
consumer.subscribe(Collections.singleton(topicName));
```

Each store gets assigned its partition and can start to begin working without any need of oversight from Kitty.


```
consumerProps.put(ConsumerConfig.HEARTBEAT_INTERVAL_MS_CONFIG, "3000");
consumerProps.put(ConsumerConfig.MAX_POLL_INTERVAL_MS_CONFIG, "300000");
consumerProps.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, "10000");
```

Now when a store needs to close due to an infestation, Kafka can detect that the store is closed and reassign the work to a different member in the group. The broker determines the store is open by ensuring the store is regularly communicating and asking for new messages. If the broker doesn't hear from the store it determines it is closed.


```
consumerProps.put(ConsumerConfig.HEARTBEAT_INTERVAL_MS_CONFIG, "3000");
consumerProps.put(ConsumerConfig.MAX_POLL_INTERVAL_MS_CONFIG, "300000");
consumerProps.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, "10000");
```

This solution even works when Kitty improperly trains her employees on how to safely handle fireworks at the grand opening and the store catches fire.

partition 1

partition 0

partition 2

```
consumerProps.put(ConsumerConfig.HEARTBEAT_INTERVAL_MS_CONFIG, "3000");
consumerProps.put(ConsumerConfig.MAX_POLL_INTERVAL_MS_CONFIG, "300000");
consumerProps.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, "10000");
```

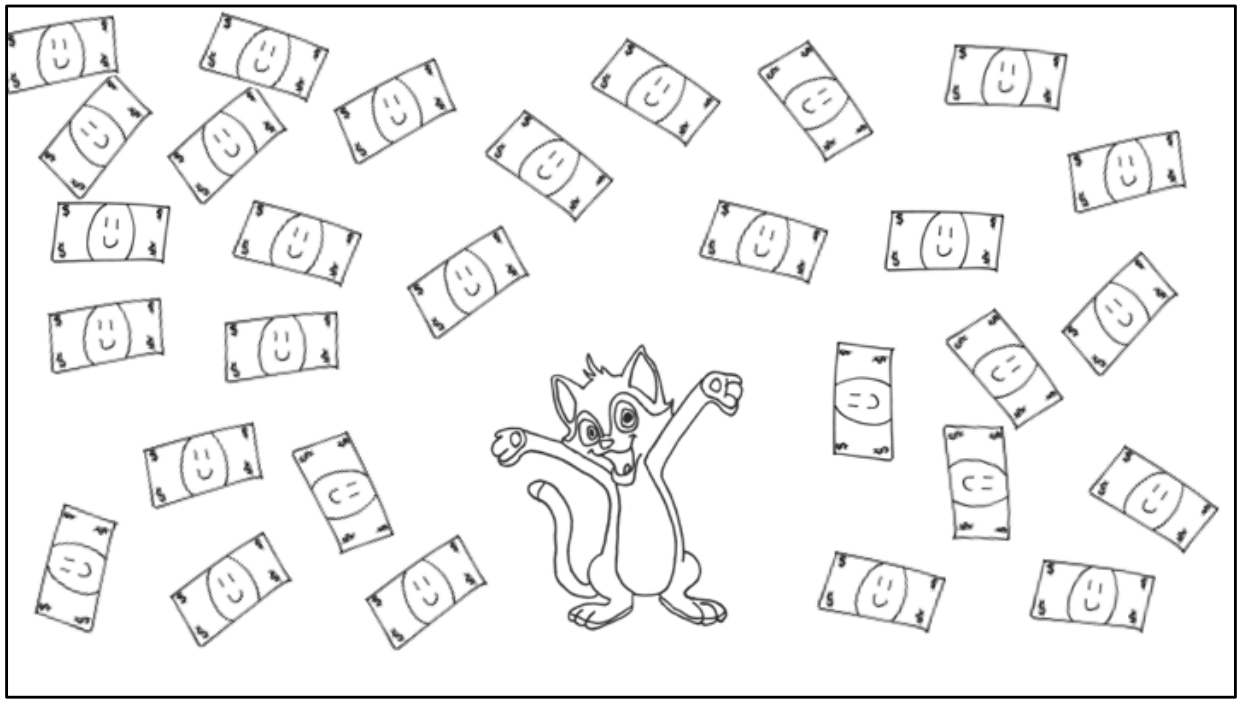
The partition assignment will fall back to the last consumer in the group ensuring continuity of business without Kitty or consumers ever knowing. Kitty realized there might be some duplicate tshirts printed as some messages might be dual processed but she was ok with that as she could recoup the profits at a summer clearance sale.


```
consumerProps.put(ConsumerConfig.HEARTBEAT_INTERVAL_MS_CONFIG, "3000");
consumerProps.put(ConsumerConfig.MAX_POLL_INTERVAL_MS_CONFIG, "300000");
consumerProps.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, "10000");
```

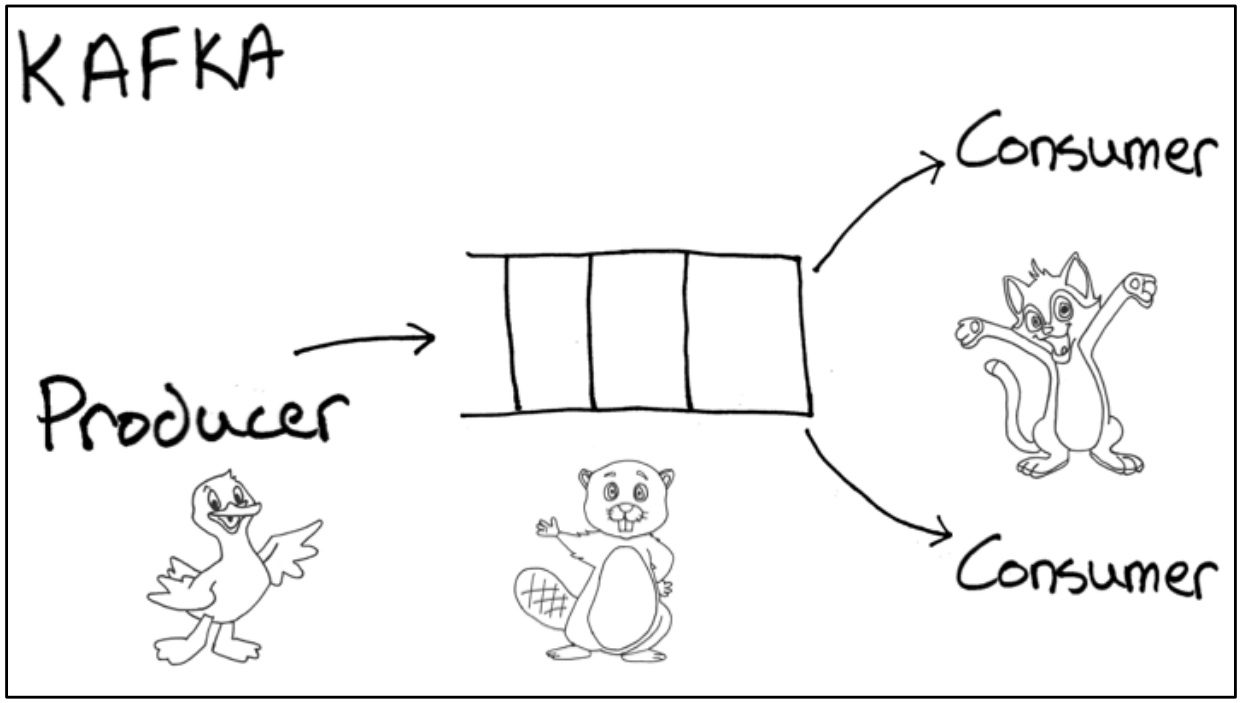
Then when the lice are cleaned up and the store reopens the work will be reassigned back to the store.

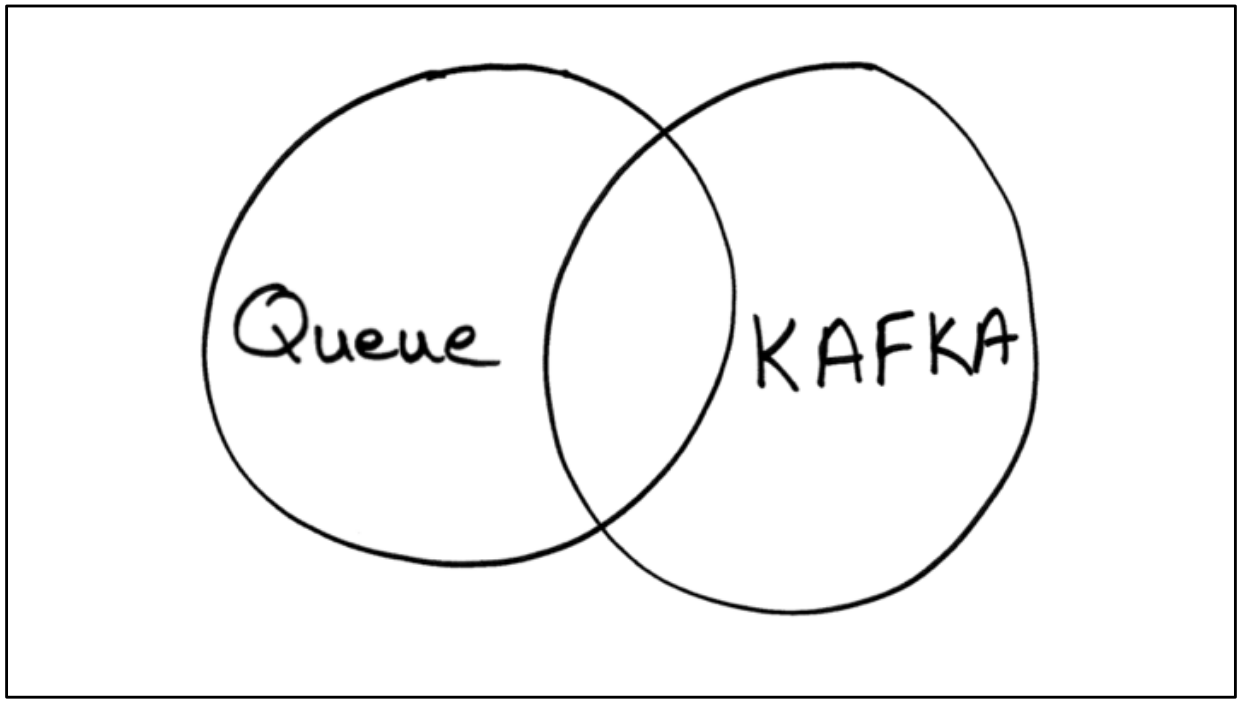

```
consumerProps.put(ConsumerConfig.HEARTBEAT_INTERVAL_MS_CONFIG, "3000");
consumerProps.put(ConsumerConfig.MAX_POLL_INTERVAL_MS_CONFIG, "300000");
consumerProps.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, "10000");
```

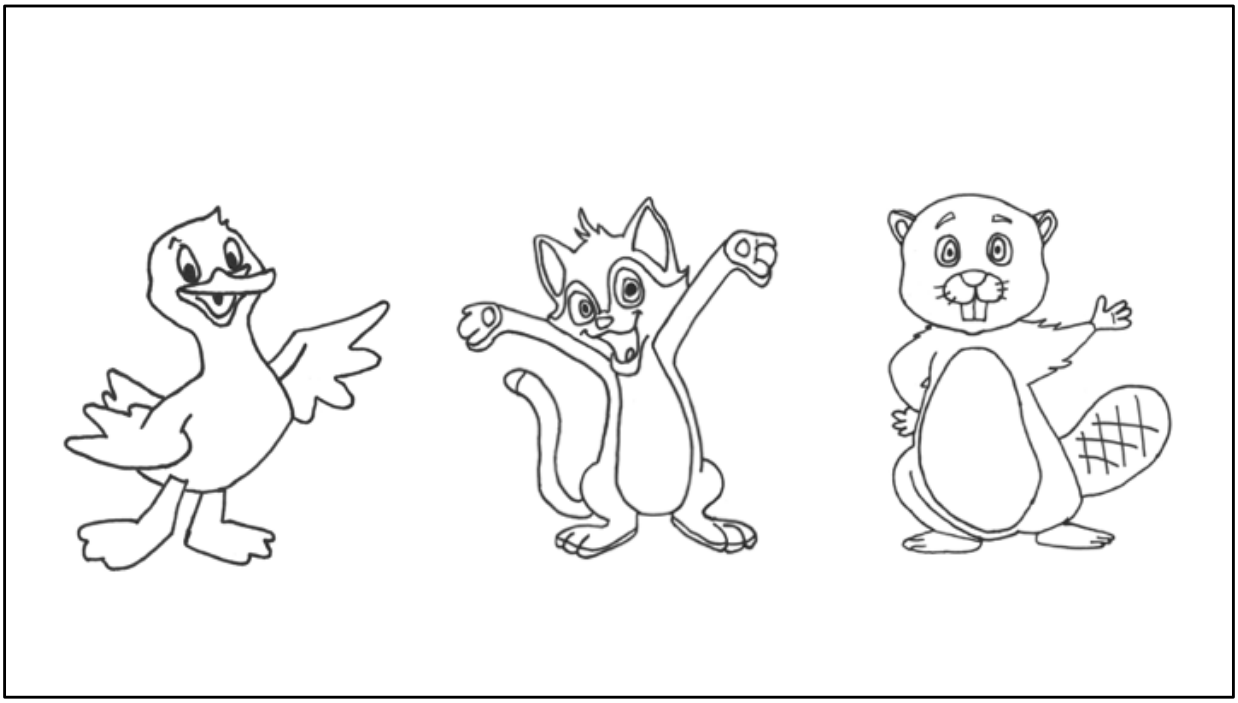
Also when the store was rebuilt from the fire damage that work will be reassigned as well and everything returns to normal and peak operating efficiency.



Kitty is now super excited. She's been able to use her education to start a successful business and is ready to start expanding to more stores and printing more content. Nothing can stand in her way now except for those pesky copyright laws she's been ignoring.


KAFKA
Consumer
Producer
Consumer


Queue
KAFKA

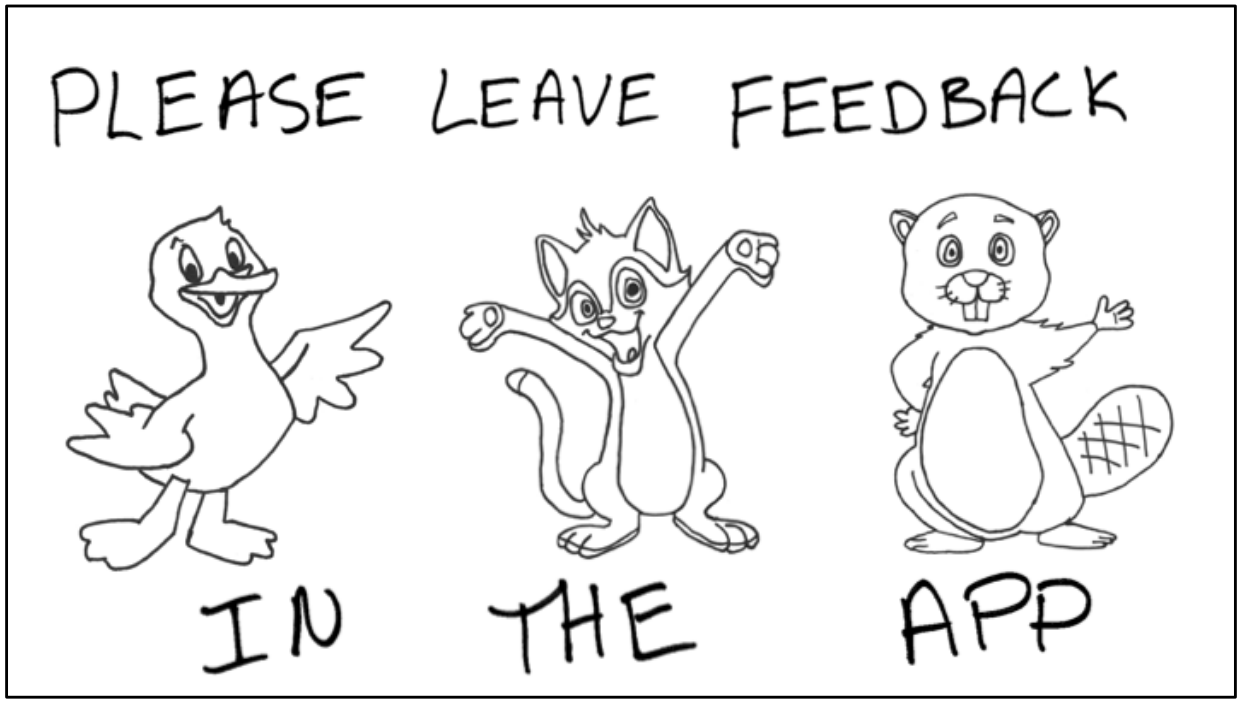


Note all of these books will be available for sale at your typical booksellers. Also look for additional books soon like:

- Sparky The Eel Distributes Processing
- Flink The Squirrel floats down stream
- Ozzy the Orca learns NoSQL

References

- https://kafka.apache.org/documentation/#maximizingefficiency
- https://medium.com/@durgaswaroop/a-practical-introduction-to-kafka-storage-internals-d5b544f6925f
- http://kafka.apache.org/documentation.html#replication
- https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Detailed+Consumer+Coordinator+Design
- https://cwiki.apache.org/confluence/display/KAFKA/Offset+Management
- https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying
- https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Improvement+Proposals#KafkaImprovementProposals-AdoptedKIPs


PLEASE LEAVE FEEDBACK
IN THE APP