

Fish and Chips and Apache Kafka®

By Tibs (they / he)

Slides and accompanying material at
<https://github.com/tibs/fish-and-chips-and-kafka-talk>

tony.ibbs@aiven.io / @much_of_a

What we'll cover

- Me and messaging and Apache Kafka®
- Fish and chips
 - How to talk to Kafka
 - Start with a simple model and work up
 - There's a demo you can play with afterwards
 - Some ideas for things you can do to extend the demos

tony.ibbs@aiven.io / @much_of_a

Some message problems I've cared about

- between components on a Set Top Box
- configuration between microservices
- to / from Internet of Things devices, and their support systems

Kafka is a very good fit for the IoT cases, maybe less so for the others

tony.ibbs@aiven.io / @much_of_a

What I want from messaging

- multiple producers *and* multiple consumers
- single delivery
- guaranteed delivery
- resumes safely if system crashes
- no back pressure handling (queue does not fill up)

tony.ibbs@aiven.io / @much_of_a

Enter, Apache Kafka®



tony.ibbs@aiven.io / @much_of_a

Kafka terms

Messages are *Events*

Producers send messages, *Consumers* read them.

Can have multiple Producers and Consumers

A Producer send a message to a named *Topic*, each Consumer reads from a single Topic

Partitions can be used to "spread the load" within a Topic

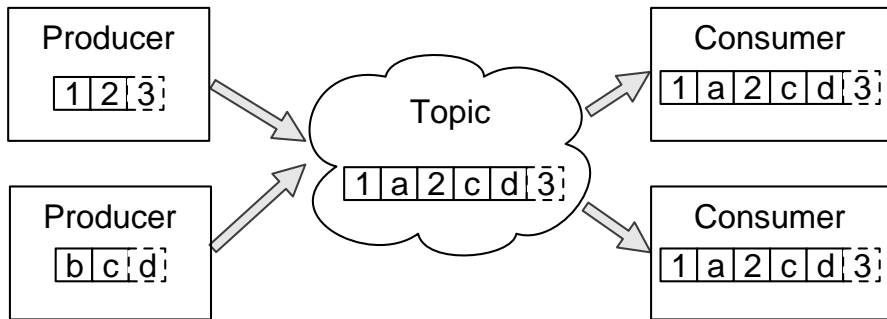
Producers, topics, consumers



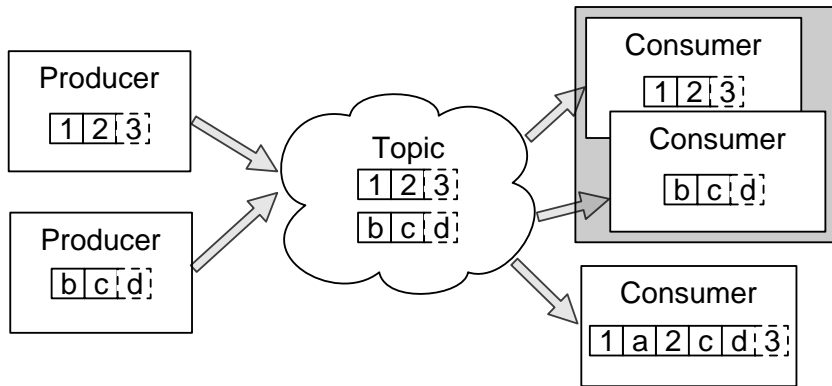
Events



Multiple produces, multiple consumers



Multiple partitions, consumer groups



Let's model a fish-and-chip shop

We start with a shop that

- just handles cod and chips
- which are always ready to be served

Glossary

- **Cod:** the traditional white fish for english fish-and-chip shops
- **Chips:** fatter, possibly soggier, french fries
- **Plaice:** a flat fish
- **Till:** a cash register

Serving a customer



An order

```
{  
  "order": 271,  
  "parts": [  
    ["cod", "chips"],  
    ["chips", "chips"],  
  ]  
}
```

Show first demo

1 till, 1 food preparer

tony.ibbs@aiven.io / @much_of_a

Libraries

kafka-python: <https://github.com/dpkp/kafka-python>

aiokafka: <https://github.com/aio-libs/aiokafka>

Textual: <https://github.com/Textualize/textual>

tony.ibbs@aiven.io / @much_of_a

Code: Producer

```
from kafka import KafkaProducer

producer = kafka.KafkaProducer(
    bootstrap_servers=f"{HOST}:{SSL_PORT}",
    security_protocol="SSL",
    ssl_cacfile=f'{certs_dir}/ca.pem',
    ssl_certfile=f'{certs_dir}/service.cert',
    ssl_keyfile=f'{certs_dir}/service.key',
    value_serializer=lambda v: json.dumps(v).encode('ascii'),

while SHOP_IS_OPEN:
    # get order from CUSTOMER
    producer.send('ORDER', order)
```

Code: Consumer

```
from kafka import KafkaConsumer

consumer = KafkaConsumer(
    "ORDER",
    bootstrap_servers=f"{HOST}:{SSL_PORT}",
    security_protocol="SSL",
    ssl_cafile="ca.pem",
    ssl_certfile="service.cert",
    ssl_keyfile="service.key",
    value_deserializer = lambda v: json.loads(v.decode('ascii')),
)

for msg in consumer:
    print(f'Message {msg.value}')
```

Code: Asynchronous - needs SSL context

```
import aiokafka.helpers

context = aiokafka.helpers.create_ssl_context(
    cafile=CERTS_DIR / "ca.pem",
    certfile=CERTS_DIR / "service.cert",
    keyfile=CERTS_DIR / "service.key",
)
```

Code: Asynchronous Producer

```
from aiokafka import AIOKafkaProducer

producer = aiokafka.AIOKafkaProducer(
    bootstrap_servers=f"{HOST}:{SSL_PORT}",
    security_protocol="SSL",
    ssl_context=context,
    value_serializer=lambda v: json.dumps(v).encode('ascii'),
)

await producer.start()

while SHOP_IS_OPEN:
    # get order from CUSTOMER
    await producer.send_and_wait('ORDERS', message)
```

Code: Asynchronous Consumer

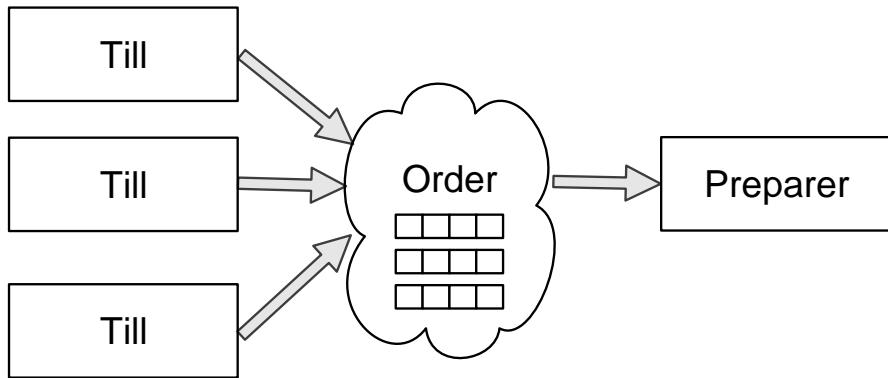
```
consumer = aiokafka.AIOKafkaConsumer(  
    'ORDERS',  
    bootstrap_servers=f"{HOST}:{SSL_PORT}",  
    security_protocol="SSL",  
    ssl_context=context,  
    value_deserializer = lambda v: json.loads(v.decode('ascii')),  
)  
  
await consumer.start()  
  
async for message in consumer:  
    print(f'Received {message.value}')
```

More customers - add more TILLs

Customers now queue at multiple TILLs, each TILL is a Producer.

Use the *till number* as the key to split the events up into partitions

Three tills



An order with multiple TILLs

```
{  
  "order": 271,  
  "till": 3,  
  "parts": [  
    ["cod", "chips"],  
    ["chips", "chips"],  
  ]  
}
```


How we alter the code

When creating the topic for the demo, request 3 partitions:

```
NewTopic(  
    name= 'DEMO2-ORDERS' ,  
    num_partitions=3,  
    replication_factor=1,  
)
```

Create 3 Till producers instead of 1

tony.ibbs@aiven.io / @much_of_a

Show demo: multiple TILLs

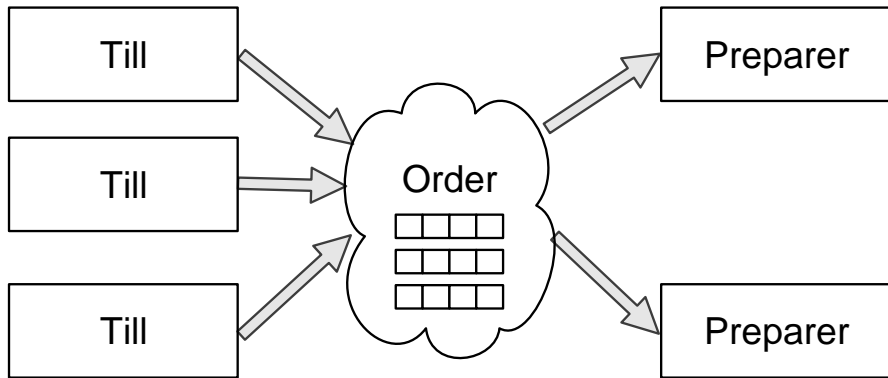
Three tills, 3 partitions, 1 food preparer

tony.ibbs@aiven.io / @much_of_a

But now the FOOD-PREPARER is too busy

tony.ibbs@aiven.io / @much_of_a

Add multiple *consumers*



How we alter the code

Send to different partitions

```
await producer.send(TOPIC_NAME, value=order, partition=self.instance_number-1)
```

Consumers need to be in same *consumer group*

```
consumer = aiokafka.AIOKafkaConsumer(  
    ...  
    group_id=CONSUMER_GROUP,  
    ...
```

Start consuming from a specific offset

If I run a demo more than once, there's a chance that a consumer might receive events from the previous demo. So we want to make sure that doesn't happen.

Various solutions - simplest for this case is to do:

```
await consumer.seek_to_end()
```

Show demo: multiple TILLs and PREPARERS

Three tills, 3 partitions, 2 food preparers

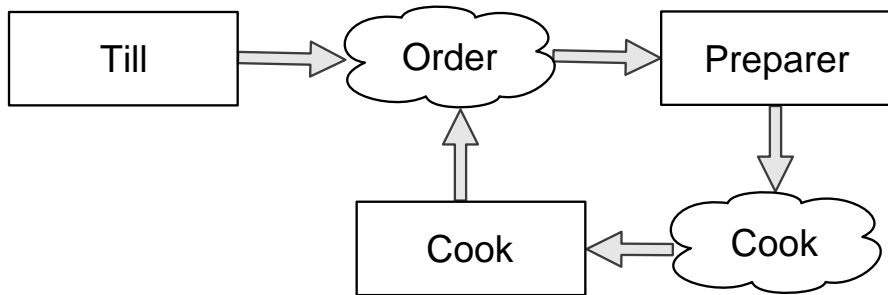
tony.ibbs@aiven.io / @much_of_a

Cod or plaice

Plaice needs to be cooked

So we need a COOK to cook it

Participant changes - add COOK



An order with plaice

```
{  
  "order": 271,  
  "till": 3,  
  "parts": [  
    ["cod", "chips"],  
    ["chips", "chips"],  
    ["plaice", "chips"],  
  ]  
}
```

Gets turned into...

```
{  
  "order": 271,  
  "till": 3,  
  "parts": [  
    ["cod", "chips"],  
    ["chips", "chips"],  
    ["plaice", "chips"],  
  ],  
  "ready": <boolean>  
}
```

tony.ibbs@aiven.io / @much_of_a

Code changes to add COOK

```
def all_order_available(self, order):  
    if 'ready' not in order:  
        all_items = itertools.chain(*order['order'])  
        order['ready'] = 'plaiice' not in all_items
```

```
order_available = self.all_order_available(order)  
if order_available:  
    await asyncio.sleep(random.uniform(PREP_FREQ_MIN, PREP_FREQ_MAX))  
    # Say order is finished  
else:  
    await self.producer.send(TOPIC_NAME_COOK, order)
```

In the Cook

```
async for message in consumer:
    ...
    # "Cook" the (plaice in the) order
    await asyncio.sleep(random.uniform(COOK_FREQ_MIN, COOK_FREQ_MAX))
    # It's important to remember to mark the order as ready now!
    # (forgetting to do that means the order will keep going round the loop)
    order['ready'] = True
    await self.producer.send(TOPIC_NAME_ORDERS, order)
```

Demo with COOK

1 till, 1 food preparer, 1 COOK

tony.ibbs@aiven.io / @much_of_a

Summary so far

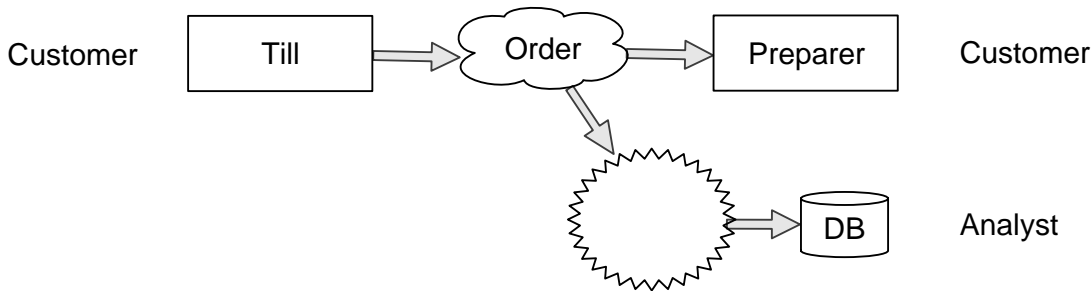
We know how to model the ordering and serving of our cod and chips

We know how to scale with multiple Producers and Consumers

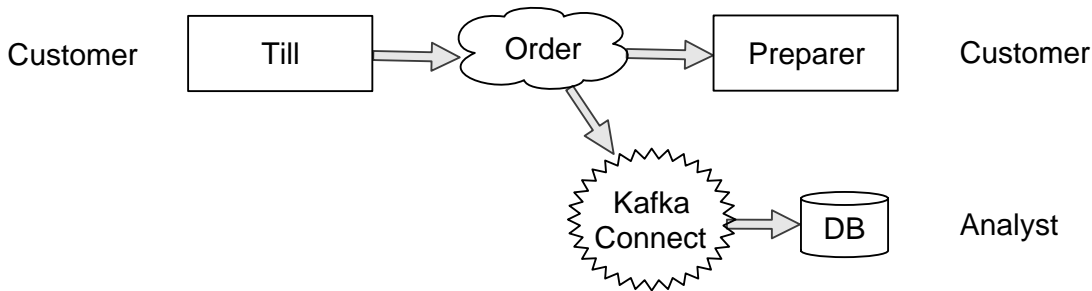
We made a simple model for orders with plaice

tony.ibbs@aiven.io / @much_of_a

Homework 1: Adding the ANALYST



Using Kafka Connect



Apache Kafka Connectors

These make it easier to connect Kafka to databases, OpenSearch, etc., without needing to write Python (or whatever) code.

tony.ibbs@aiven.io / @much_of_a

How I would do it

The Aiven developer documentation has instructions on how to do this at <https://docs.aiven.io/docs/products/kafka/kafka-connect/howto/jdbc-sink.html>

- Create an appropriate PostgreSQL database and table
- Make sure that the Kafka service has Kafka Connect enabled
- Use the Aiven web console to setup the new connector

And then add code to the Python demo to query PostgreSQL and make some sort of report over time.

tony.ibbs@aiven.io / @much_of_a

Homework 2: Model cooking the fish and chips

Use a Redis cache to simulate contents of the hot cabinet

Redis has entries for the hot cabinet content, keyed by `cod`, (portions of) `chips` and `plai`. We start with 0 for all of them.

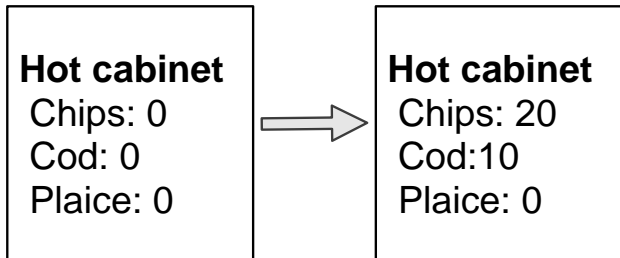
Using the cache

PREPARER compares the order to the counts in the cache. If there's enough "stuff" to make the order up, decrements the cache appropriately, and that's done

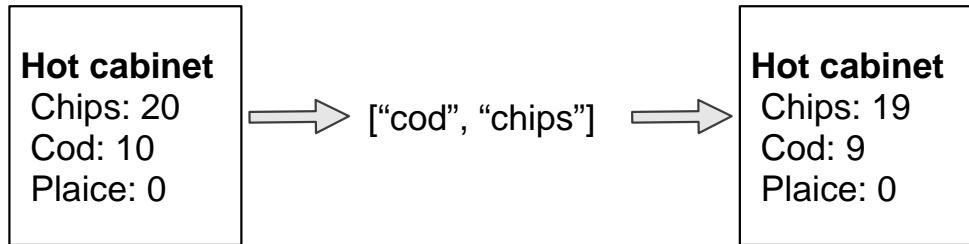
If not, sends the order to the COOK

COOK updates the cache - for `place`, adds as many as are needed, for the others, if they go below a threshold, adds a standard quantity back in ("cooking in batches"). Then sends the order back to the [ORDER] topic

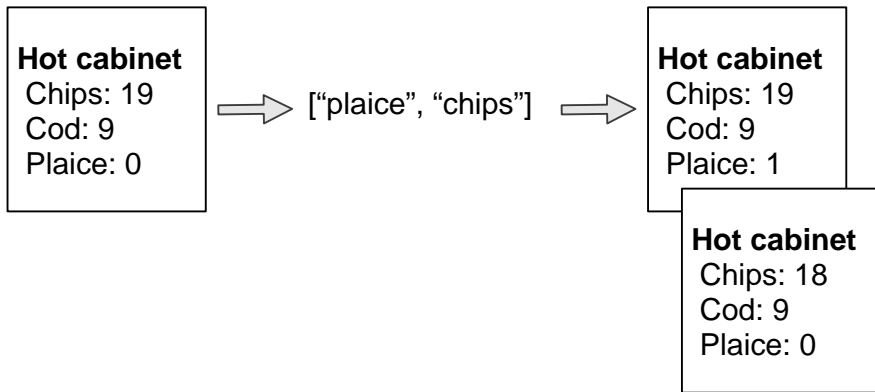
Start of day



Cod and chips



Plaice and chips



Final summary

We know how to model the ordering and serving of our cod and chips

We know how to scale with multiple Producers and Consumers

We made a simple model for orders with plaice

We talked briefly about using Kafka Connectors to share data with other data users

We talked briefly about how one might model the hot cabinet in more detail

tony.ibbs@aiven.io / @much_of_a

Acknowledgements

Apache, Apache Kafka, Kafka, and the Kafka logo are either registered trademarks or trademarks of the Apache Software Foundation in the United States and/or other countries

Postgres and PostgreSQL are trademarks or registered trademarks of the PostgreSQL Community Association of Canada, and used with their permission

Redis is a registered trademark of Redis Ltd. Any rights therein are reserved to Redis Ltd.

tony.ibbs@aiven.io / @much_of_a

Fin

Get a free trial of Aiven services at
<https://console.aiven.io/signup/email>

Also, we're hiring! See <https://aiven.io/careers>

Written in [reStructuredText](#), converted to PDF using
[rst2pdf](#)

Slides and accompanying material  at
<https://github.com/tibs/fish-and-chips-and-kafka-talk>



tony.ibbs@aiven.io / @much_of_a