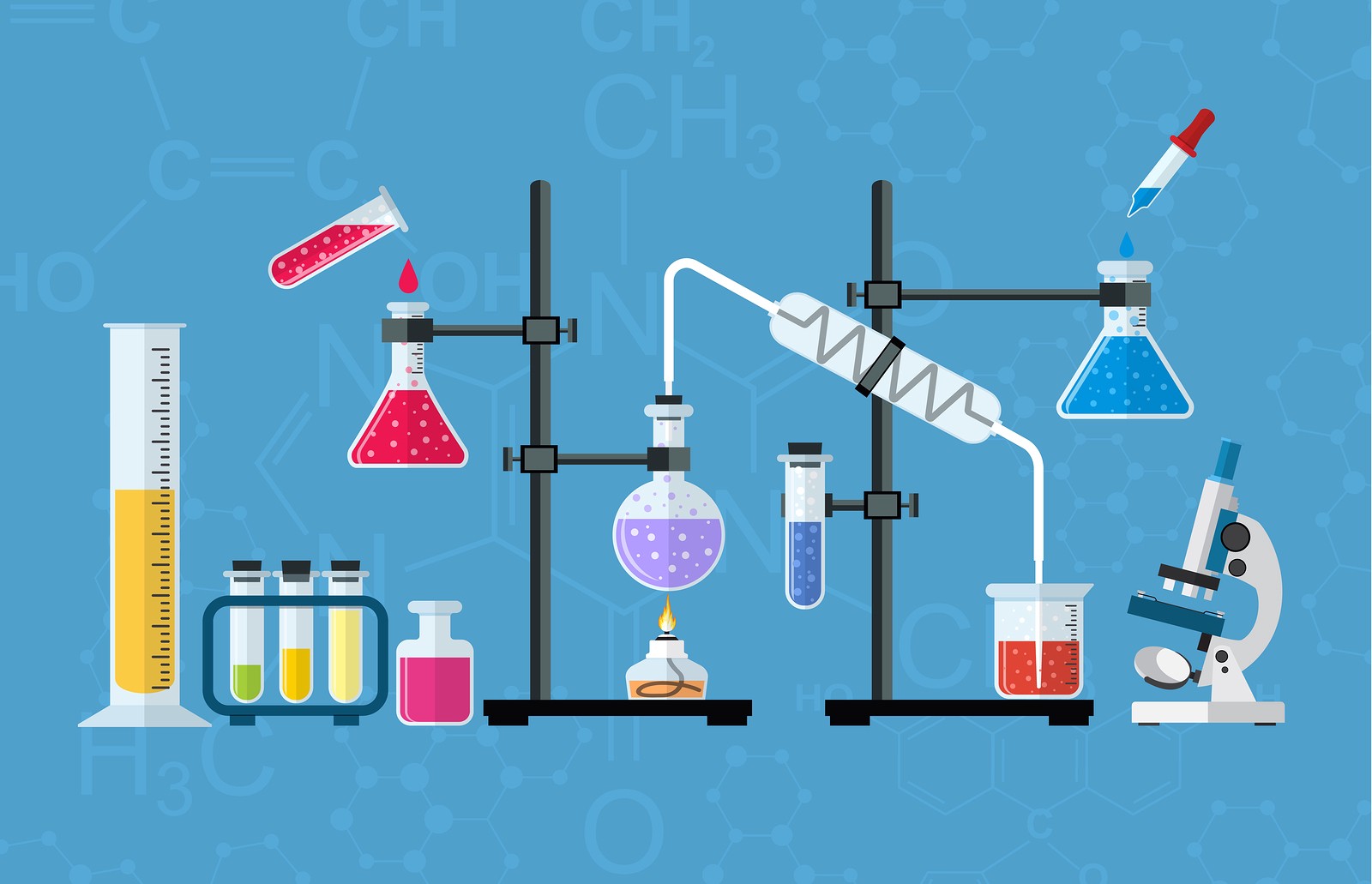
**A blockchain experiment with Apache Kafka**

November 10th 2017



Blockchain technology and Apache Kafka share characteristics which suggest a natural affinity. For instance, both share the concept of an ‘immutable append only log’. In the case of a Kafka partition:

*Each partition is an ordered, immutable sequence of records that is continually appended to — a structured commit log. The records in the partitions are each assigned a sequential id number called the offset that uniquely identifies each record within the partition [*[*Apache Kafka*](https://kafka.apache.org/intro)*]*

Whereas a blockchain can be described as:

*a continuously growing list of records, called blocks, which are linked and secured using cryptography. Each block typically contains a hash pointer as a link to a previous block, a timestamp and transaction data [*[*Wikipedia*](https://en.wikipedia.org/wiki/Blockchain)*]*

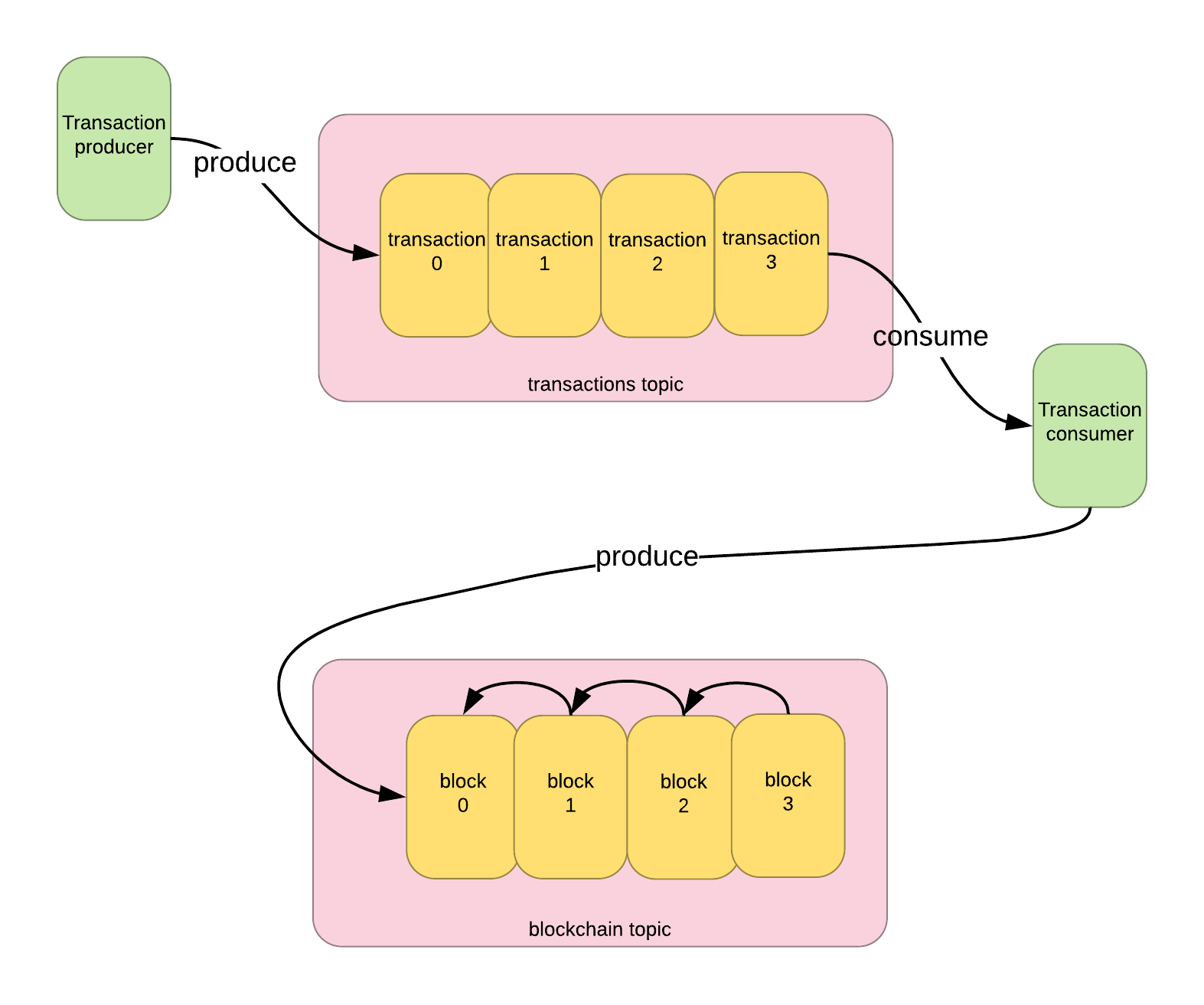
Clearly, these technologies share the parallel concepts of an immutable sequential structure, with Kafka being particularly optimized for high throughput and horizontal scalability, and blockchain excelling in guaranteeing the order and structure of a sequence.

By integrating these technologies, we can create a platform for experimenting with blockchain concepts.

Kafka provides a convenient framework for distributed peer to peer communication, with some characteristics particularly suitable for blockchain applications. While this approach may not be viable in a trustless public environment, there could be practical uses in a private or consortium network. See [Scaling Blockchains with Apache Kafka](https://blog.gridplus.io/scaling-blockchains-with-apache-kafka-814c85781c6) for further ideas on how this could be implemented.

Additionally, with some experimentation, we may be able to draw on concepts already implemented in Kafka (e.g. sharding by partition) to explore solutions to blockchain challenges in public networks (e.g. scalability problems).

The purpose of this experiment is therefore to take a simple blockchain implementation and port it to the Kafka platform; we’ll take Kafka’s concept of a sequential log and guarantee immutability by chaining the entries together with hashes. The blockchain topic on Kafka will become our distributed ledger. Graphically, it will look like this:



Visual of a Kafka blockchain

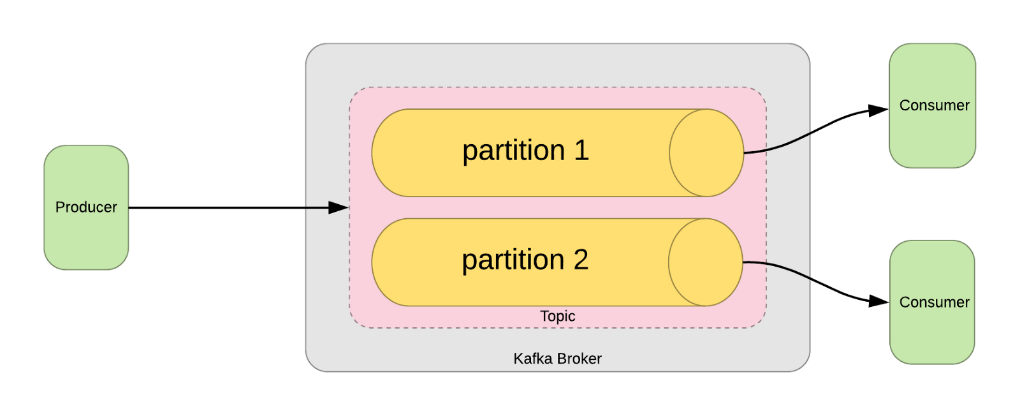
**Introduction to Kafka**



Kafka is a streaming platform designed for high-throughput, real-time messaging, i.e. it enables publication and subscription to streams of records. In this respect it is similar to a message queue or a traditional enterprise messaging system. Some of the characteristics are:

* High throughput: Kafka brokers can absorb gigabytes of data per second, translating into millions of messages per second. You can read more about the scalability characteristics in [Benchmarking Apache Kafka: 2 Million Writes Per Second](https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines).
* Competing consumers: Simultaneous delivery of messages to multiple consumers, typically expensive in traditional messaging systems, is no more complex than for a single consumer. This means we can design for [competing consumers](http://www.enterpriseintegrationpatterns.com/patterns/messaging/CompetingConsumers.html), guaranteeing that each consumer will receive only one of the messages and achieving a high degree of horizontal scalability.
* Fault tolerance: By replicating data across multiple nodes in a cluster, the impact of individual node failures is minimized.
* Message retention and replay: Kafka brokers maintain a record of consumer offsets — a consumer’s position in the stream of messages. Using this, consumers can rewind to a previous position in the stream even if the messages have already been delivered, allowing them to recreate the status of the system at a point in time. Brokers can be configured to retain messages indefinitely, which is necessary for blockchain applications.

In Kafka, each topic is split into partitions, where each partition is a sequence of records which is continually appended to. This is similar to a text log file, where new lines are appended to the end. The entries in the partition are each assigned a sequential id, called an offset, which uniquely identifies the record.



Kafka partitioning

The Kafka broker can be queried by offset, i.e. a consumer can reset its offset to some arbitrary point in the log to retrieve records from that point forward.

**Tutorial**

Full source code is [available here](https://github.com/lucrussell/kafka-blockchain).

**Prerequisites**

* Some understanding of blockchain concepts: The tutorial below is based on implementations from [Daniel van Flymen](https://hackernoon.com/learn-blockchains-by-building-one-117428612f46) and [Gerald Nash](https://medium.com/crypto-currently/lets-build-the-tiniest-blockchain-e70965a248b), both excellent practical introductions. The following tutorial builds heavily on these concepts, while using Kafka as the message transport. In effect, we’ll port a Python blockchain to Kafka, while maintaining most of the current implementation.
* Basic knowledge of Python: the code is written for Python 3.6.
* [Docker](https://www.docker.com/): docker-compose is used to run the Kafka broker.
* [kafkacat](https://github.com/edenhill/kafkacat): This is a useful tool for interacting with Kafka (e.g. publishing messages to topics)

On startup, our Kafka consumer will try to do three things: initialize a new blockchain if one has not yet been created; build an internal representation of the current state of the blockchain topic; then begin reading transactions in a loop:

Start

The initialization step looks like this:

Initialize the chain

First, we find the highest available offset on the blockchain topic. If nothing has ever been published to the topic, the blockchain is new, so we start by creating and publishing the genesis block:

The genesis block

In read\_and\_validate\_chain(), we’ll first create a consumer to read from the blockchain topic:

Create the blockchain consumer

Some notes on the parameters we’re creating this consumer with:

* Setting the consumer group to the blockchain group allows the broker to keep a reference of the offset the consumers have reached, for a given partition and topic
* auto\_offset\_reset=OffsetType.EARLIEST indicates that we’ll begin downloading messages from the start of the topic.
* auto\_commit\_enable=True periodically notifies the broker of the offset we’ve just consumed (as opposed to manually committing)
* reset\_offset\_on\_start=True is a switch which activates the auto\_offset\_reset for the consumer
* consumer\_timeout\_ms=5000 will trigger the consumer to return from the method after five seconds if no new messages are being read (we’ve reached the end of the chain)

Then we begin reading block messages from the blockchain topic:

Read blocks

For each message we receive:

1. If it’s the first block in the chain, skip validation and add to our internal copy (this is the genesis block)
2. Otherwise, check the block is valid with respect to the previous block, and append it to our copy
3. Keep a note of the offset of the block we just consumed

At the end of this process, we’ll have downloaded the whole chain, discarding any invalid blocks, and we’ll have a reference to the offset of the latest block.

At this point, we’re ready to create a consumer on the transactions topic:

Create the transaction consumer

Our example topic has been created with two partitions, to demonstrate how partitioning works in Kafka. The partitions are set up in the docker-compose.yml file, with this line:

KAFKA\_CREATE\_TOPICS=transactions:2:1,blockchain:1:1

transactions:2:1 specifies the number of partitions and the replication factor (i.e. how many brokers will maintain a copy of the data on this partition).

This time, our consumer will start from OffsetType.LATEST so we only get transactions published from the current time onwards.

By pinning the consumer to a specific partition of the transactions topic, we can increase the total throughput of all consumers on the topic. The Kafka broker will evenly distribute incoming messages across the two partitions of the transactions topic, unless we specify a partition when we publish to the topic. This means each consumer will be responsible for processing 50% of the messages, doubling the potential throughput of a single consumer.

Now we can begin consuming transactions:

Start consuming transactions

As transactions are received, we’ll add them to an internal list. Every three transactions, we’ll create a new block and call mine():

Mine a block

1. First, we’ll check if our blockchain is the longest one in the network; is our saved offset the latest, or have other nodes already published later blocks to the blockchain? This is our consensus step.
2. If new blocks have already been appended, we’ll make use of the read\_and\_validate\_chain from before, this time supplying our latest known offset to retrieve only the newer blocks.
3. At this point, we can attempt to calculate the proof of work, basing it on the proof from the latest block.
4. To reward ourselves for solving the proof of work, we can insert a transaction into the block, paying ourselves a small block reward.
5. Finally, we’ll publish our block onto the blockchain topic. The publish method looks like this:

Publish a block

**In Action**

1. First start the broker:

docker-compose up -d

2. Run a consumer on partition 0:

python kafka\_blockchain.py 0

3. Publish 3 transactions directly to partition 0:

4. Check the transactions were added to a block on the blockchain topic:

kafkacat -C -b kafka:9092 -t blockchain

You should see output like this:

To balance transactions across two consumers, start a second consumer on partition 1, and remove -p 0 from the publication script above.

**Conclusion**

Kafka can provide the foundation for a simple framework for blockchain experimentation. We can take advantage of features built into the platform, and associated tools like kafkacat, to experiment with distributed peer to peer transactions.

While scaling transactions in a public setting presents one set of issues, within a private network or consortium, where real-world trust is already established, transaction scaling might be achieved via an implementation which takes advantage of Kafka concepts.