Did you get my message?

A comparison of three modern messaging frameworks

CodeMash 2019

Jack Bennett, Improving (Columbus)
jack.bennett@improving.com

Motivation for this talk

- A business story that had some ups and downs and lessons learned...

Intention for this talk

- Quick walk-through of the process of selecting a messaging framework by comparing three different "options"
- Demo of some techniques that make this testing process a little easier
- Conversation about the alternatives, trade-offs, decision criteria

What do we mean by messaging?

- A working definition: **one software entity sends discrete bundles of data to another software entity.**

- **"Entity"**: object, process, thread, application, ???

- **"Sends"**: there is a mechanism for transmitting and receiving data.

- **"Discrete"**: a message is bounded in size.

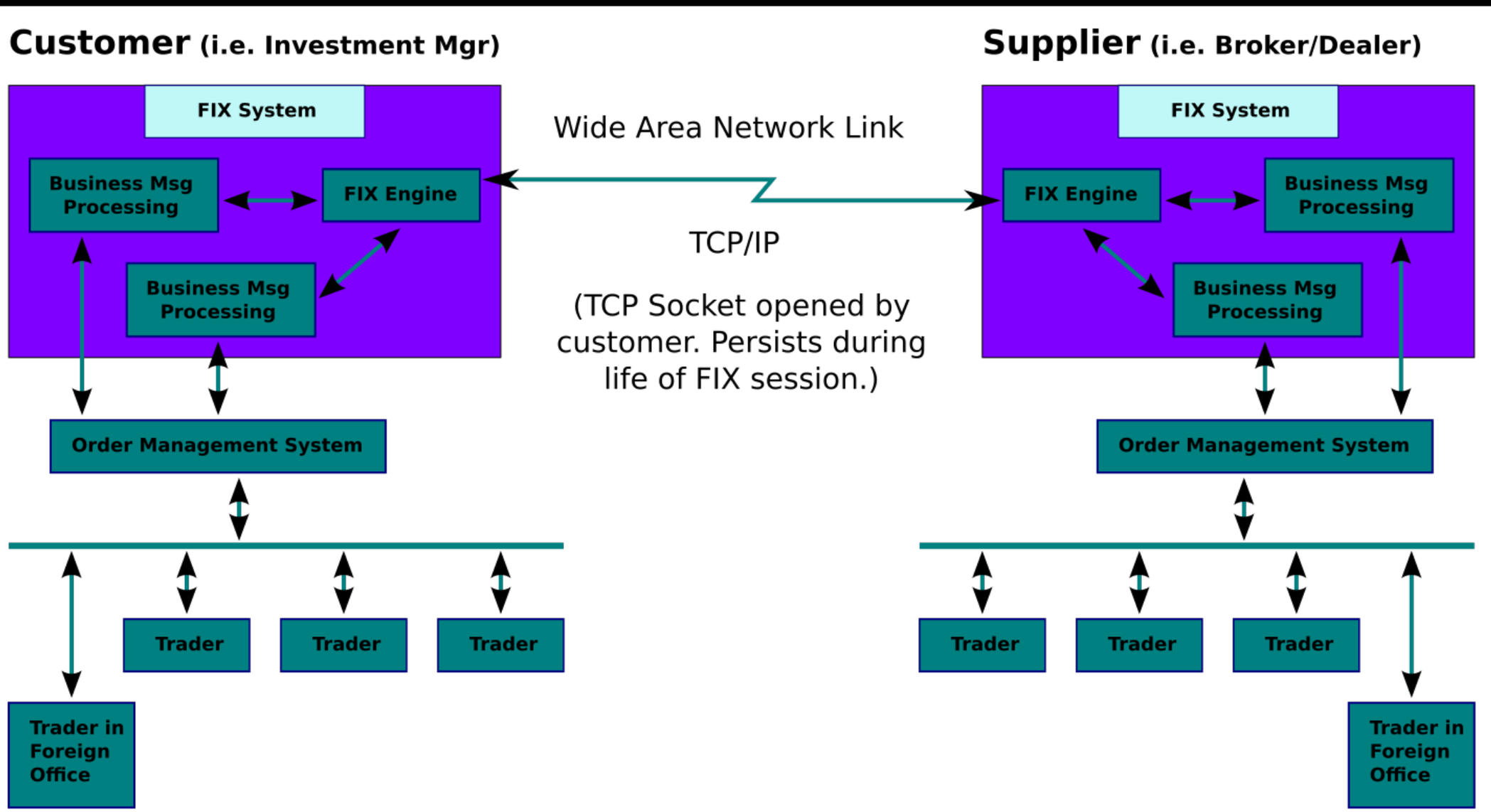
A more detailed definition

- **Message size?**
  - **Can be** as small as a few bytes (plus protocol overhead). Example: updating the value of a boolean flag
  - **Probably** no bigger than a few kB (i.e. not a bulk file transfer / SFTP)
  - **“Typically”** 10s-1000s of bytes in most applications.
- **Protocol?**
  - Probably runs over plain TCP socket connection
  - Could also run over HTTP, WebSocket, ???
  - Some frameworks implement higher-level message protocols like AMQP, STOMP, MQTT, or industry-specific protocols like FIX
- **Message content?**
  - Deliberately unspecified: payload is "whatever you want"
  - Serialized bytes is the most general case, can decode to anything

What does a message look like?

- Illustrative example - FIX message in equities trading.
- Features:
  - Message Header, Length field, Trailer (Checksum)
  - Set of tag-value pairs. Tag is **int**, value is **typed** (str, int, float, date, etc).
  - Delimited by ASCII 0x01 (SOH / ^A)
  - Usually 10s-100s of bytes in length; typically no more than a couple of kB.

```
8=FIX.4.2 | 9=176 | 35=8 | 49=XYZA | 56=BRKD |
52=20171123-05:30:00.000 | 11=ATOMNOCCC9990900 | 20=3 | 150=E |
39=E | 55=AAPL | 167=CS | 54=1 | 38=15 | 40=2 | 44=15 | 58=AAPL
EQUITY TESTING | 59=0 | 47=C | 32=0 | 31=0 | 151=15 | 14=0 | 6=0
| 10=128 |
```



Source: https://en.wikipedia.org/wiki/Financial_Information_eXchange

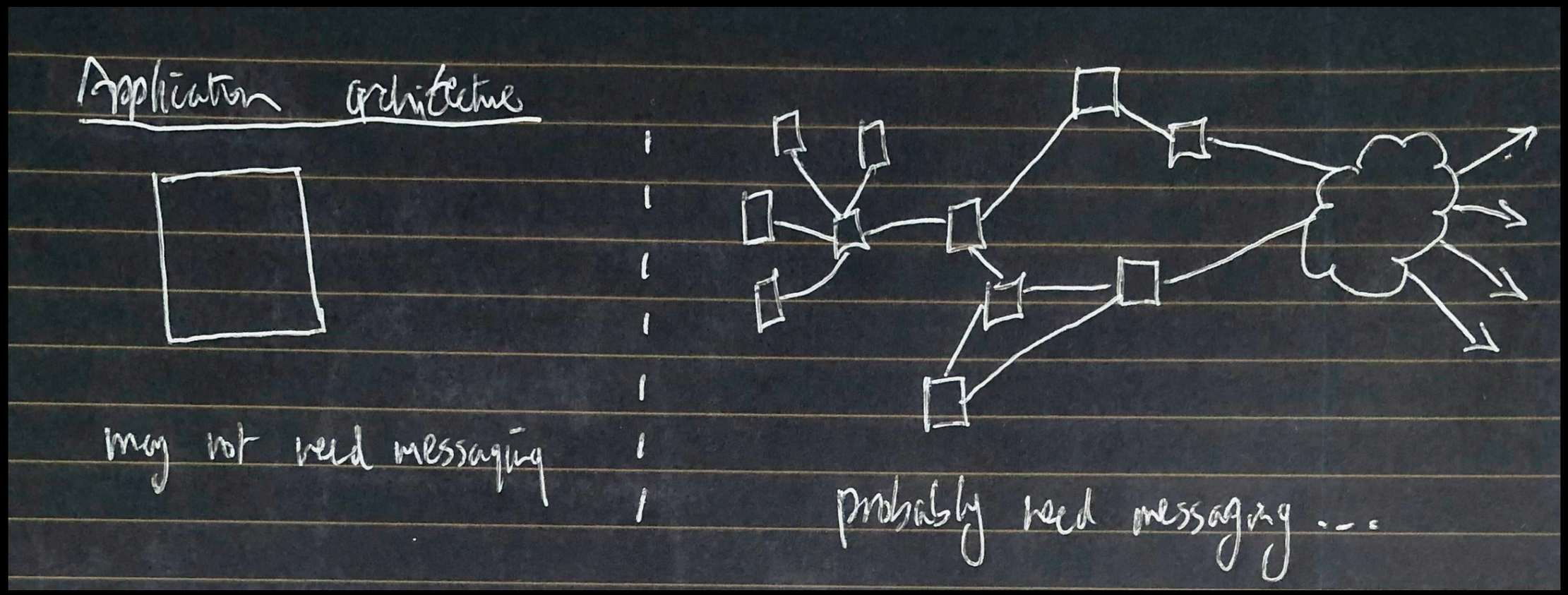
Why messaging? What are the alternatives?

- According to the book *Enterprise Integration Patterns* (2003), there exist four main alternatives for "integration":
  - (1) File Transfer
  - (2) Shared Database
  - (3) Remote Procedure Invocation
  - **(4) Messaging**
- **"Integration"**: sharing data, state, etc. between applications

Key solution areas for messaging

- "Real-time" updates between components
- Relatively fine granularity of data
- Potentially high data rates (message rates)
- Distributed applications with loose-coupling for resilience
- Versatile enabling technology for popular architectural paradigms like "cloud", "microservices", etc.

Application architectures



Frameworks that we look at in this talk

- **Kafka**
- **RabbitMQ**
- **ZeroMQ** (also styled ØMQ)

- A very incomplete list of other messaging frameworks
  - SQS (AWS)
  - Qpid
  - ActiveMQ
  - Messaging implementations embedded in specific applications or frameworks (e.g. Django, Meteor)

Common features of these three frameworks

- They are all **mature** (at least 6+ years development)
- They can all handle **simple (and complex) messaging patterns**
- They are all **fast enough** (for some definition of "fast")
- They all provide **libraries / APIs / bindings** for many languages
- They are all **active, open-source projects**
- They all have **high quality documentation**
- So... they can probably all meet your needs. Why choose one over another?
- General guidelines
    - Select a framework based on your specific business needs and application logic.
    - Test extensively before making a decision
    - Use abstractions that enable you to switch framework if needed

Making the decision about a framework

- Write down, in one sentence, what your system needs to do for you.
- Use numerical specifics for performance wherever possible (e.g. "10,000 msgs/s" instead of "fast").
- If you don't have estimates or experimental data for the item above, figure out what you need to do to get them (order of magnitude is fine).
- Look at what the default settings tell you!

Test/Demo intro

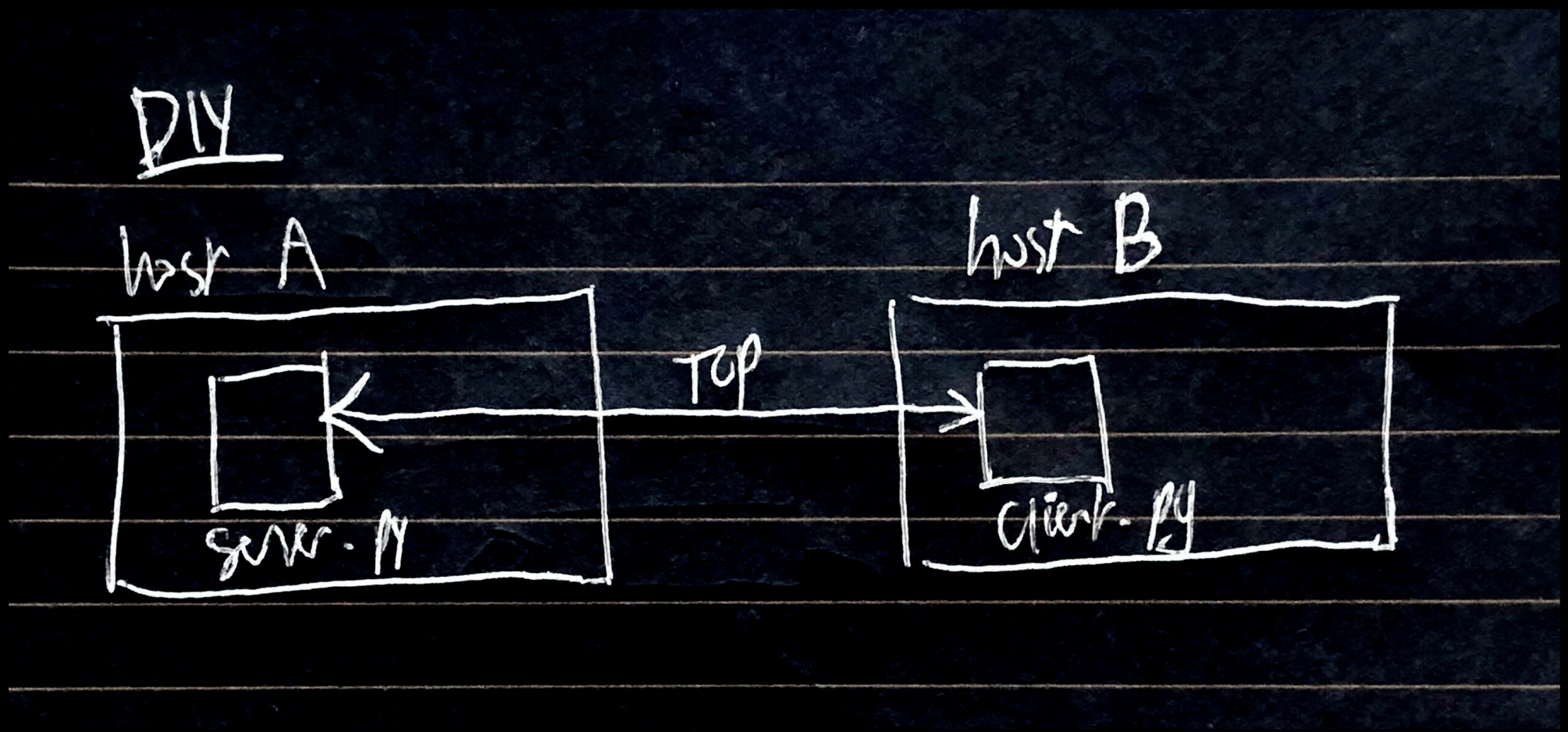
- Demonstrations in a vagrant VM
- Ubuntu 18.04 (Bionic Beaver)
- Install needed libraries in VM (globally)
- Each demonstration (DIY, Kafka, RabbitMQ, ZeroMQ) is in a separate Python virtualenv
- Install required python libraries in each virtualenv

Why not implement DIY messaging?

- It’s straightforward
- You have full control over all parts of the application
- "*How hard can it really be? You’re just sending some bytes over a socket! Easy!*"

Demo of DIY simple messaging

DIY architecture



DIY notes and observations

- It’s easy to do something very simple and specific
- However, things can get complicated very quickly
- Be careful of sunk costs! Leaky or blurry abstractions make sunk costs worse.

- As a starting point, DIY is good … but what about:
  - Authentication / security
  - Persistence / database
  - Monitoring and metrics
  - Failure recovery and reconnection
  - Redundancy and replication
  - Scalability
  - Optimization of speed, latency, throughput, and other metrics
  - ???

Apache Kafka: in their own words

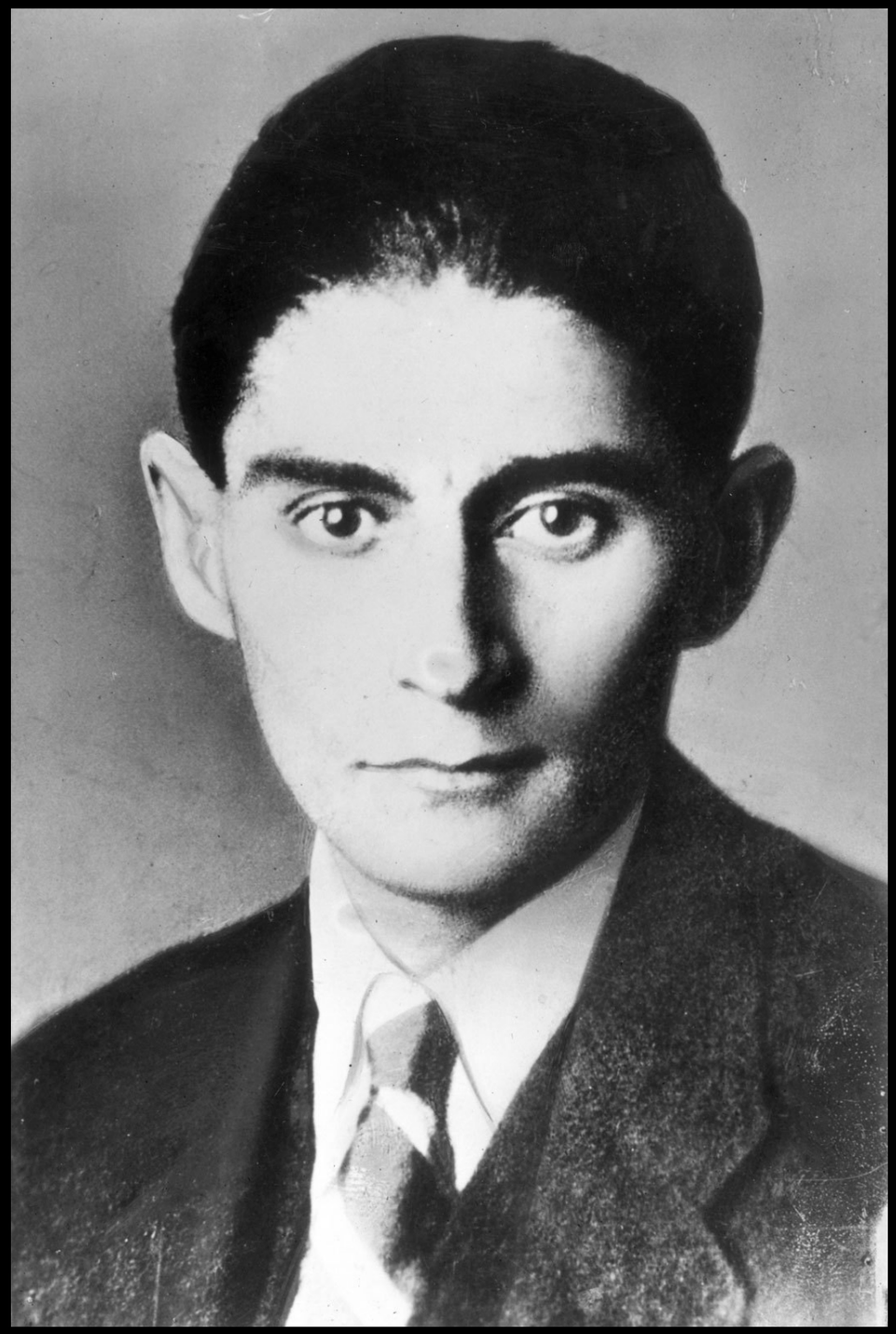
- **Apache Kafka® is *a distributed streaming platform*. What exactly does that mean?**

- A streaming platform has **three key capabilities**:
  - **Publish** and subscribe to streams of records, similar to a message queue or enterprise messaging system.
  - **Store** streams of records in a fault-tolerant durable way.
  - **Process** streams of records as they occur.

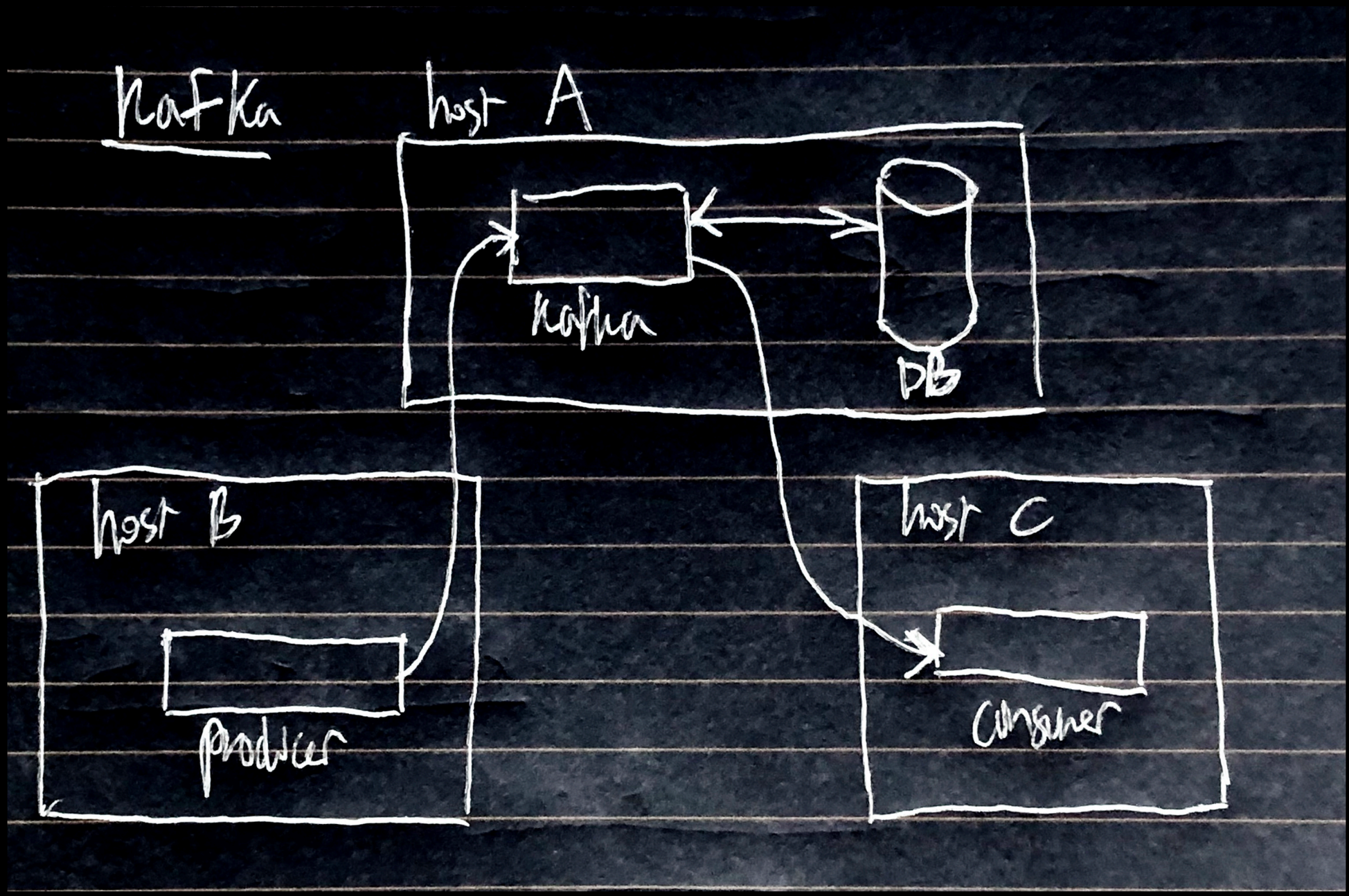
- Kafka is generally used for **two broad classes of applications**:
  - Building **real-time streaming data pipelines** that reliably get data between systems or applications
  - Building **real-time streaming applications** that transform or react to the streams of data

- https://kafka.apache.org/intro

Demo of Kafka basics



Kafka architecture



Kafka notes and observations

- Messaging system ... PLUS
  - Streaming data processing platform
  - Persistent storage system for records

- Oriented toward time-series data + metadata (list of Record objects)

- JVM based: also depends on Apache Zookeeper.

- Needs "a lot" of RAM (more than your stock Vagrant VM - samples use >1GB in java command line options)

- Extensive configuration available

- Replication for redundancy and availability

- Kafka-specific lingo (Record, Topic, Partition, Group, etc.)

Kafka terminology

- A **topic** is a category or feed name to which **records** are published. Topics in Kafka are always multi-subscriber; that is, a topic can have zero, one, or many consumers that subscribe to the data written to it.

- Each **partition** is an ordered, immutable sequence of records that is continually appended to - a structured commit log.

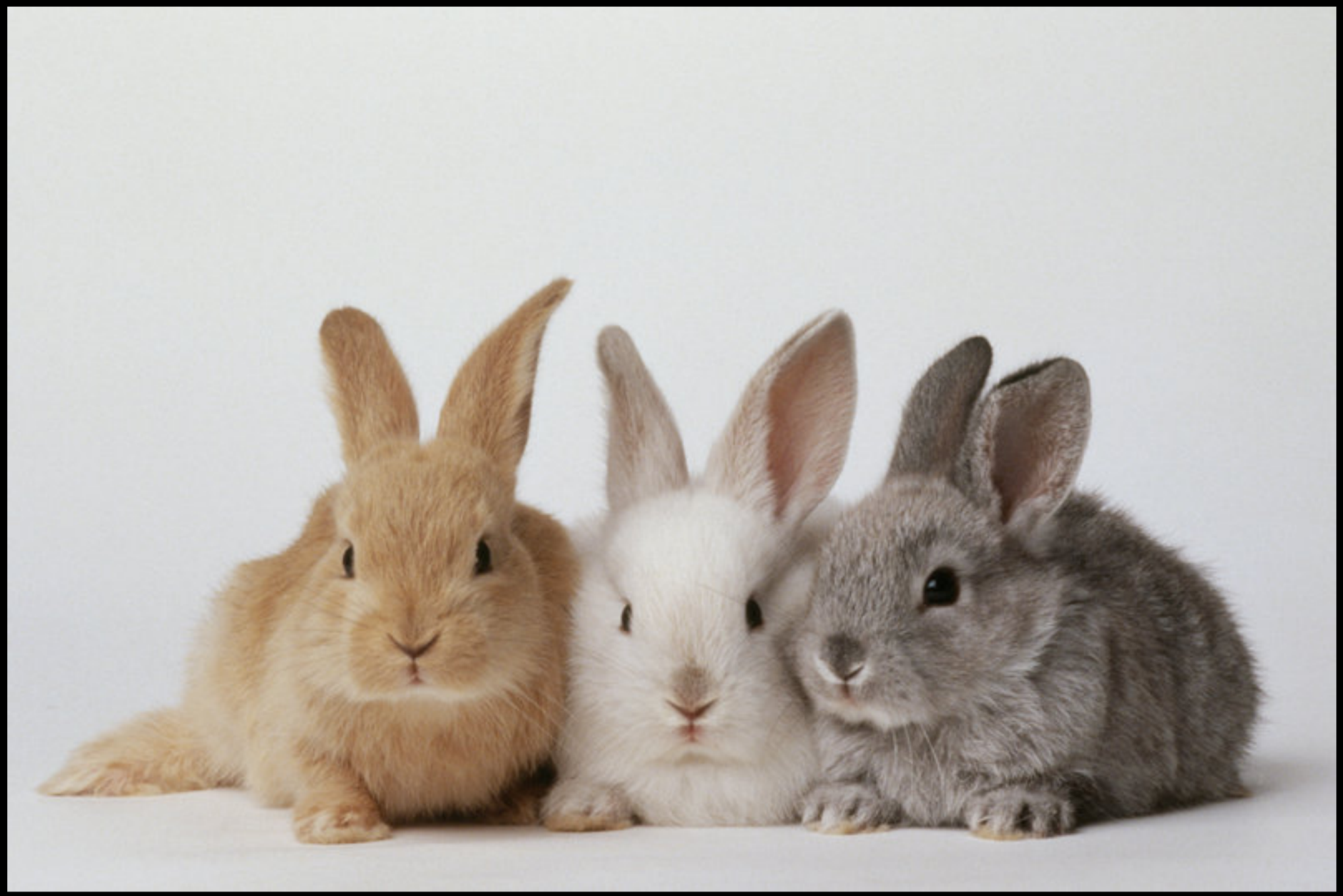
- The records in the partitions are each assigned a sequential id number called the **offset** that uniquely identifies each record within the partition.

- The Kafka cluster **durably persists all published records** - whether or not they have been consumed - using a configurable retention period. For example, if the retention policy is set to two days, then for the two days after a record is published, it is available for consumption, after which it will be discarded to free up space. Kafka's performance is effectively constant with respect to data size so storing data for a long time is not a problem.

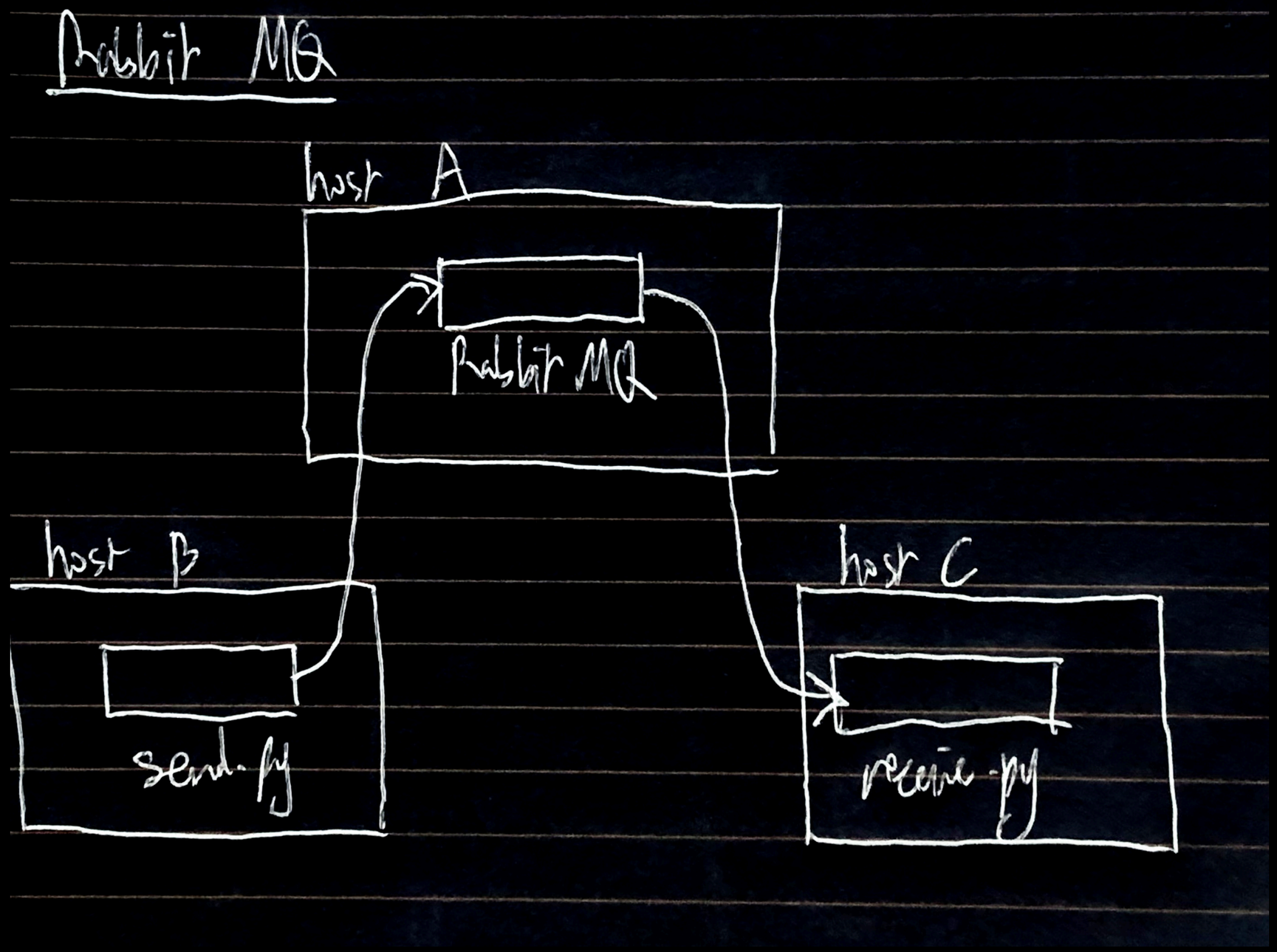
RabbitMQ: in their own words

- More than 35,000 production deployments of RabbitMQ world-wide at small startups and large enterprises
- RabbitMQ is the most popular open source message broker.
- RabbitMQ is lightweight and easy to deploy on premises and in the cloud. It supports multiple messaging protocols. RabbitMQ can be deployed in distributed and federated configurations to meet high-scale, high-availability requirements.
- RabbitMQ runs on many operating systems and cloud environments, and provides a wide range of developer tools for most popular languages.
- https://www.rabbitmq.com/

Demo of RabbitMQ basics



RabbitMQ architecture



RabbitMQ notes and observations

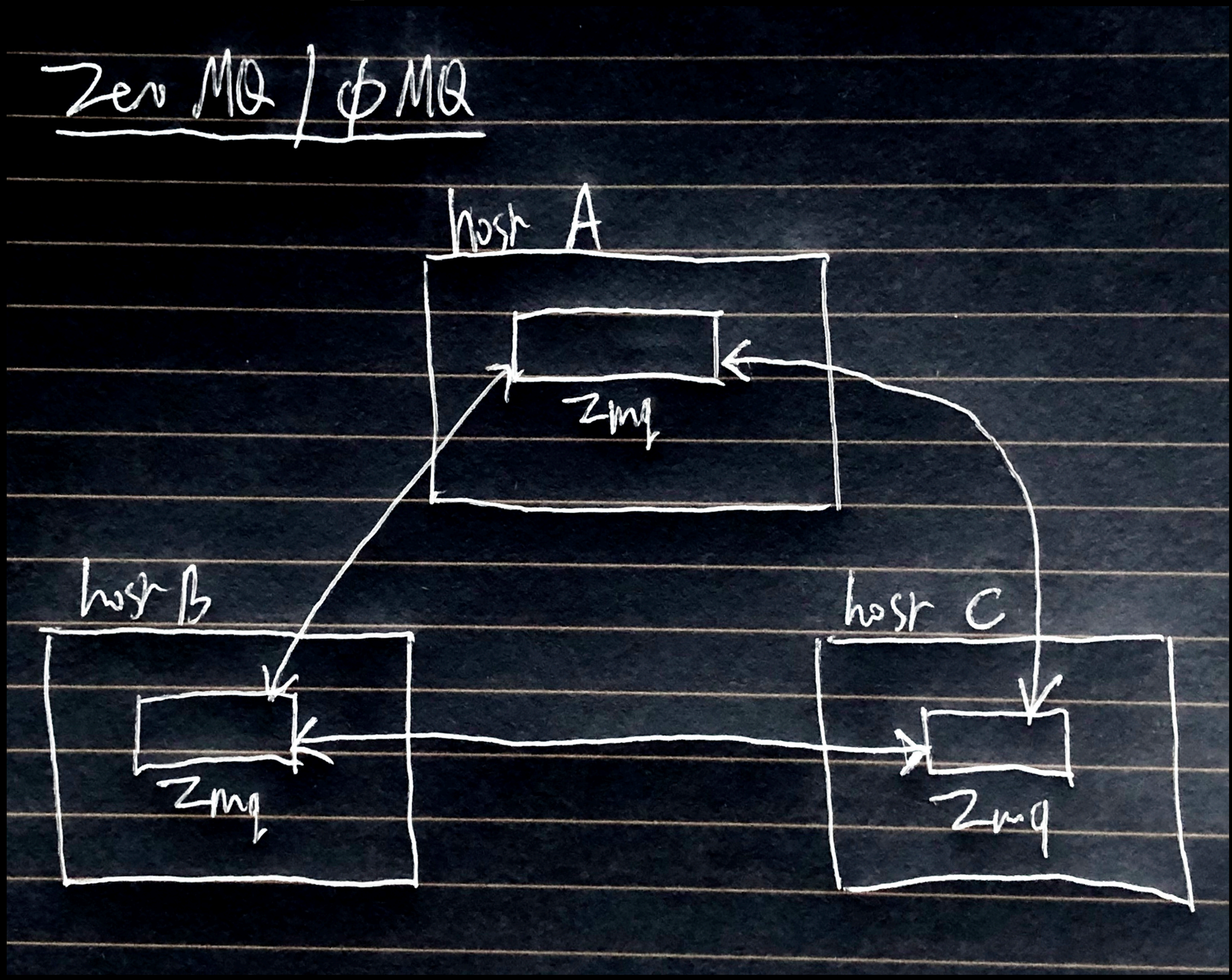
- Written in Erlang
- In contrast with Kafka, RabbitMQ is **not** designed to operate as a time-series database (keep the queues clear!)
- Implements AMQP (Advanced Message Queuing Protocol - message data format), among other protocols.

ZeroMQ: in their own words

- Connect your code in any language, on any platform.
- Carries messages across inproc, IPC, TCP, TIPC, multicast.
- Smart patterns like pub-sub, push-pull, and router-dealer.
- High-speed asynchronous I/O engines, in a tiny library.
- Backed by a large and active open source community.
- Supports every modern language and platform.
- Build any architecture: centralized, distributed, small, or large.
- http://zeromq.org/

Demo of ZeroMQ basics

ZeroMQ architecture



ZeroMQ notes and observations

- Originally implemented in C++. A pure Java version is also available ("JeroMQ").
- ZMQ broadens the TCP socket abstraction to support (e.g.) inter-thread and inter-process messaging, pub-sub, and other, more complex messaging patterns.
- ZMQ is not a message broker or server, but a messaging library that provides a range of messaging functionality and patterns based on the (expanded) socket abstraction. The library enables you to write clients, servers, peers, etc.

Comparison of three* messaging frameworks

	DIY	Kafka	RabbitMQ	ZeroMQ
Separate Server Process	?	✅	✅	
Web Admin UI	?		✅	
Replication	?	✅	✅	
Users/groups	?	✅	✅	
Persistence	?	✅		
Documentation and Tutorials	?	✅	✅	✅
Year of Origin	?	2011	2007	2007

General observations

- We have barely scratched the surface...
- All of these frameworks and approaches - *including DIY* - can do “Hello World” (and more!) with minimal effort or stress.
- Reinventing / reimplementing messaging in a small application is not *necessarily* a bad idea. (It is not necessarily a good idea either...)

Recommendations

- The more your requirements list increases, and the bigger your application(s) grow, the more likely an existing FOSS framework will be the best solution (**"there are no new problems"**).

- Select your framework based on its **portfolio of special features** that line up best with your business needs (need/nice-to-have/don't care)

- Architect and implement your applications so that it is **feasible to switch your messaging framework** if your needs change (place behind an abstraction barrier where possible)