





# Did you get my message?

Comparing three modern messaging  
frameworks



# Thank you! 🙏

- To the Stir Trek organizers and conference staff for their hard work. The logo for Stir Trek, featuring the words "STIR" and "TREK" in a stylized, italicized font, with a yellow starburst graphic to the right.
- To my employer Improving for encouraging us to participate in activities like this. The logo for Improving, featuring the word "improving" in a blue, lowercase font, with a blue starburst graphic to the right and the tagline "It's what we do.™" below it.
- To **you** 👊, the conference participants. Your interest makes events like this happen.

**Jack Bennett**  
Principal Consultant  
Improving, Columbus  
[jack.bennett@improving.com](mailto:jack.bennett@improving.com)



# Motivation for this talk

- A business story that had a few ups and downs and lessons learned...
- We needed a messaging framework with persistence.
- After a lot of adventures, we selected Apache Kafka.
- We learned some things to do and some things not to do.
- I share some of what we learned here.

# 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
- Have some fun! 🎉🌟

# Recommendations

- The more your requirements list increases, and the bigger your application(s) grow, the more likely an existing framework will be the best solution ("**there are no new problems**").
- Select your framework based on its **unique portfolio of features and limitations** that line up best with your business needs (need/nice-to-have/don't-care/don't-want/dealbreaker)
- 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)

# What do we mean by messaging?

- A working definition: **one software entity sends discrete bundles of data to another software entity.**
- **"Entity"**: object, thread, process, application, ???
- **"Sends"**: there exists a mechanism for transmitting and receiving data.
- **"Discrete"**: messages are bounded in size – they have a defined beginning and end.

Some more detailed  
definitions...



# How big is a message?

- **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 like SCP or SFTP
- **“Typically”** 10s–1000s of bytes in most applications.

# What protocol?

- Message brokers generally communicate over plain TCP/IP socket connections
- Could also run over WebSocket, or some other higher-level streaming protocol
- Some frameworks themselves implement higher-level message protocols like AMQP, STOMP, MQTT, industry-specific protocols like FIX, or proprietary protocols

# What is the message payload/content?

- Deliberately unspecified here: **payload is "whatever you want"** or "whatever your application needs".
- Serialized bytes is the most general case
  - it can decode to any data type or data structure (e.g. use protocol buffers or equivalent)

# Real-world example of message content: FIX 1

- FIX messages are used in electronic securities trading between brokers, dealers, clients, and other trading counterparties
- Business logic message types include:
  - Order placement
  - Acknowledgement
  - Order Fulfillment ("fill")
  - Order rejection, etc.
- High-volume connections transmit millions of messages per day or more

# Real-world example of message content: FIX 2

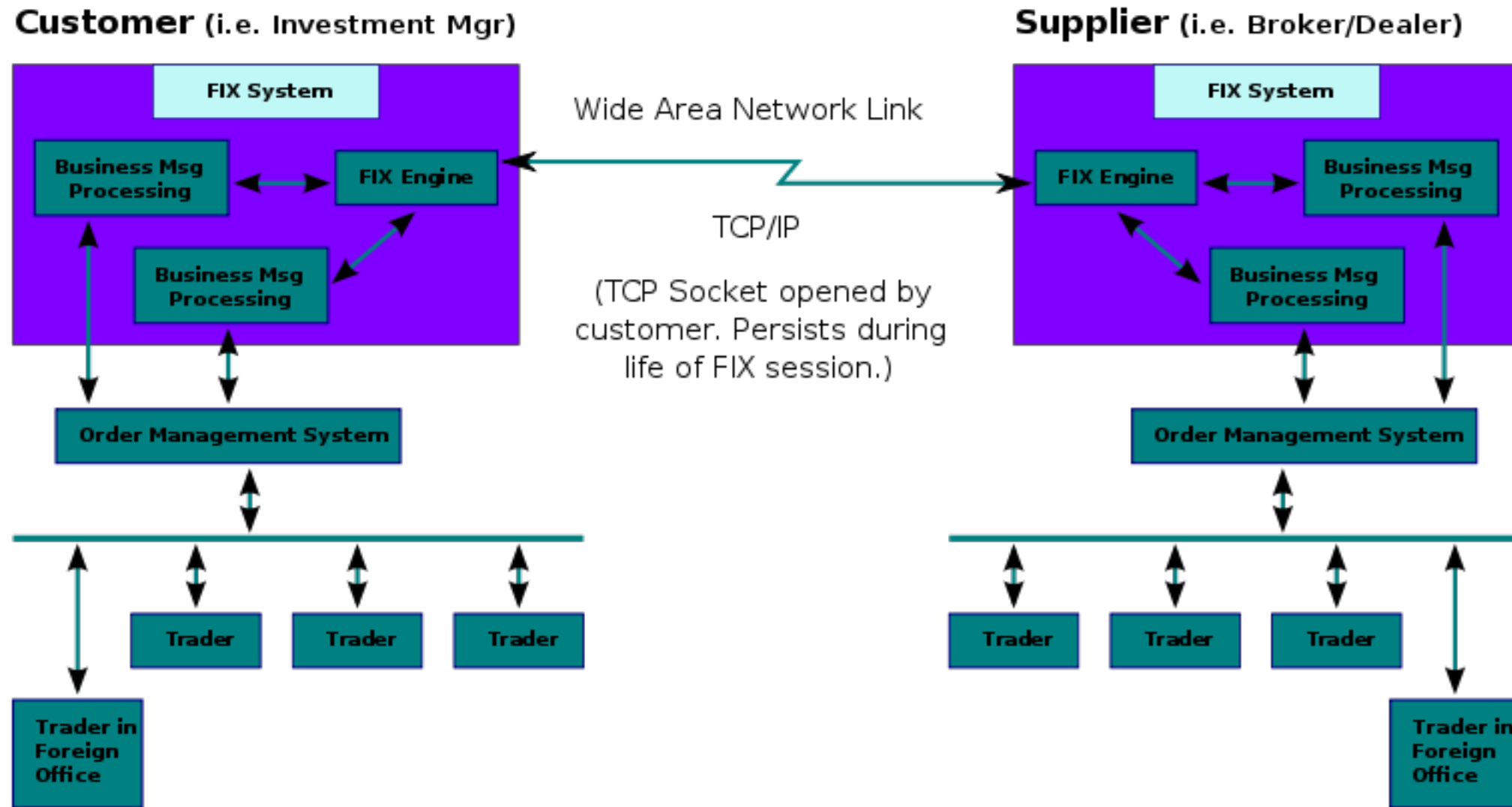
- Key features:
  - Message Header, Length field, Trailer (Checksum)
  - Set of **tag-value pairs**. Tag is **int**, value is **typed** (str, int, float, date, etc).
  - Delimited by ASCII 0x01 (SOH / ^A)
  - Usually 10s-100s of bytes in length; typically no more than a couple of kB.

# Real-world example of message content: FIX 3

```
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 |
```

"|" character (vertical pipe, ASCII 0x7c) is used here as a human readable placeholder for the 0x01 / SOH delimiters employed by the FIX protocol

# Real-world example of message content: FIX 4



[https://en.wikipedia.org/wiki/Financial\\_Information\\_eXchange](https://en.wikipedia.org/wiki/Financial_Information_eXchange)

# Why messaging? What are the alternatives?

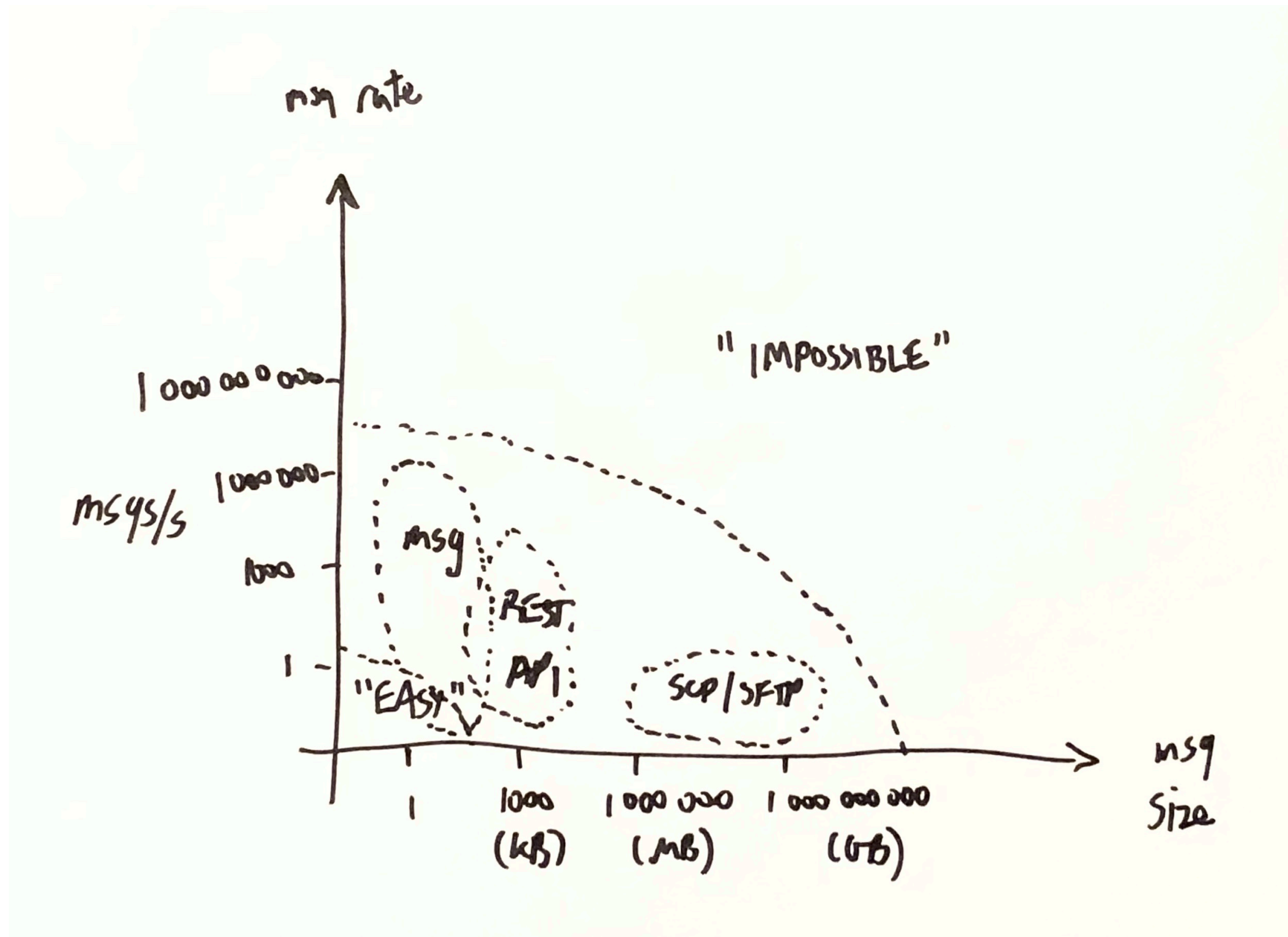
- According to the book *Enterprise Integration Patterns* (2003), there are four main alternatives for "integration"
- "Integration": **sharing data, state, etc. between applications**
  - (1) File Transfer
  - (2) Shared Database
  - (3) Remote Procedure Invocation
  - **(4) Messaging**
- What did they forget? ...
  - **REST API! (but that's not their fault - 2009)**



# Message broker vs REST API

- Messaging
  - persistent, long-lived connection
  - bidirectional data stream (peer-to-peer)
- REST API
  - intermittent / ad-hoc request-response model
  - stateless connection
  - sharp client/server distinction
- A lot of what I say about messaging frameworks (but not everything) could also apply to REST APIs

# Performance envelope



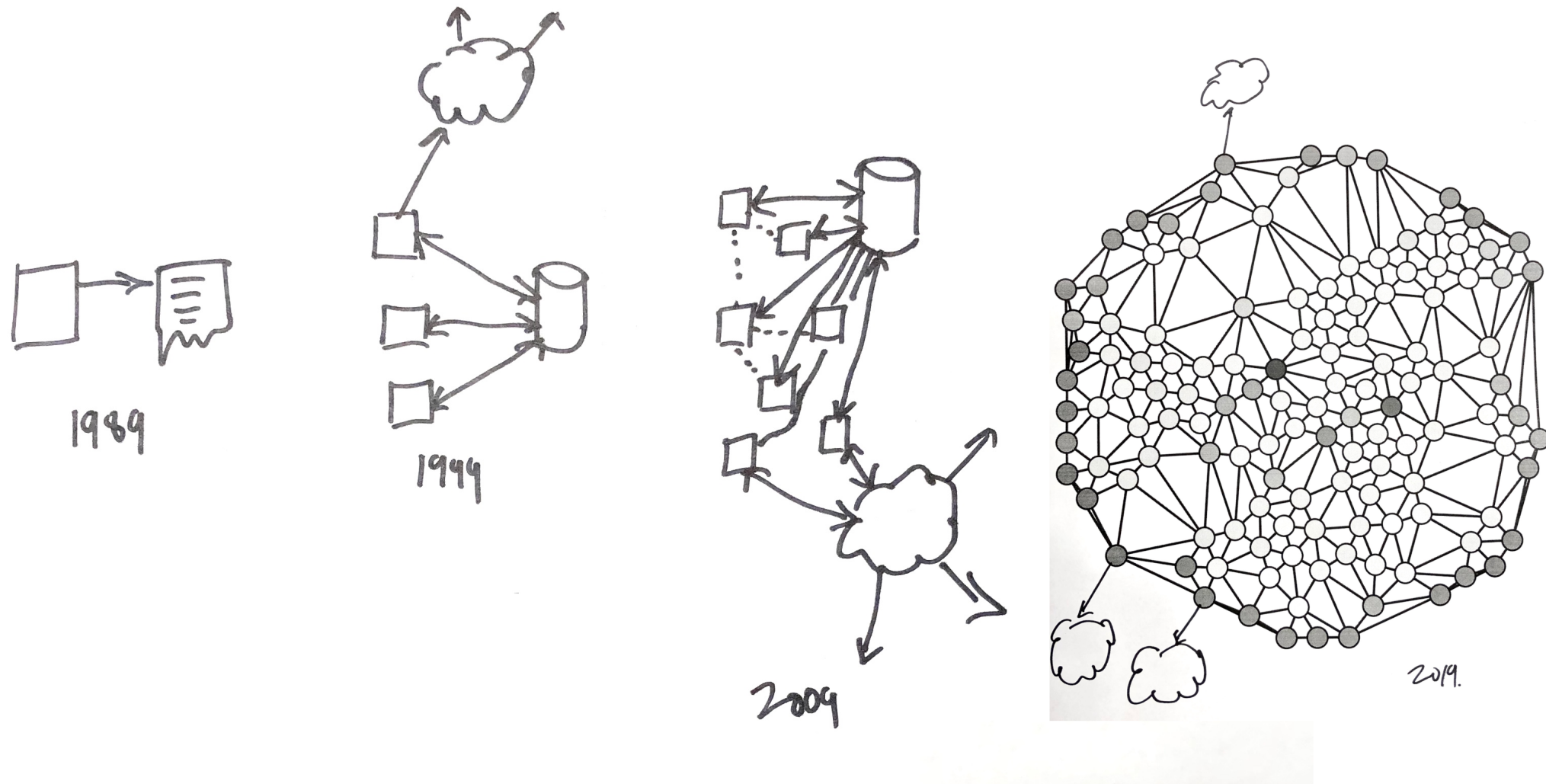
# Characteristics of messaging

- "Real-time" updates between software components (communication, synchronization)
- Relatively fine granularity of data (via small message size)
- Potentially high data rates (via high message rates)

# Typical use cases for messaging

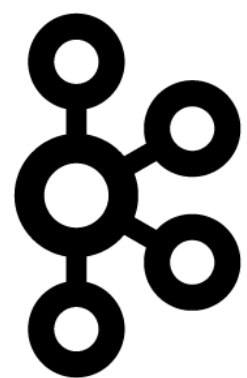
- Distributed applications with loose coupling for resilience and concurrency
- Versatile enabling technology for popular architectural paradigms like "cloud", "microservices", etc.
- Send and receive streaming data across boundaries (application, division, organization, cloud provider, etc)

# Application architectures: why is messaging so popular now?



**1989 vs 1999 vs 2009 vs 2019**

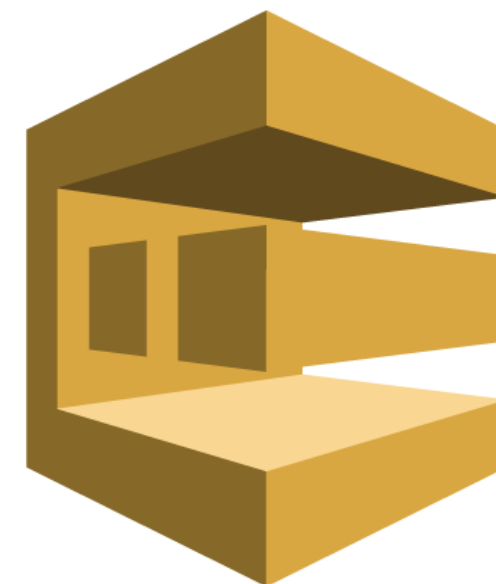
There are lots  
of choices...



kafka



Microsoft®  
BizTalk® Server



APACHE  
ACTIVE MQ™



RabbitMQ



redis

ØMQ

Opid

# Frameworks that we look at in this talk

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



# 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**

# Why these three in particular? 🤔

- They were the actual options that we considered in the scenario from a few years ago.
- High contrast between the options:
  - Kafka - streaming platform with a tightly integrated datastore
  - RabbitMQ - traditional "enterprise message broker"
  - ZeroMQ - lightweight messaging library to use directly from within applications

# Why choose A over B?

- All these frameworks could probably meet the needs of many different projects. 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 to isolate the framework and allow you to switch framework if needed

# Making a messaging framework decision

- **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 carefully at the default settings in the framework!** They tell you a lot about the intentions of the designers and programmers, and the expected applications.

# Test/Demo intro

- Demonstrations in a Vagrant VM
- Ubuntu 18.04 (Bionic Beaver)
- Install needed libraries in VM (globally)
- Run each demonstration (DIY, Kafka, RabbitMQ, ZeroMQ) in its own Python virtualenv
- Install required python libraries in each virtualenv
- I'll share demo code & config on GitHub

# Test plan: basic operation

## • Server

- listen on port for client connection
- when client sends data, send reply message

## • Client

- connect to server on known port
- send message
- receive reply message from server

# Test plan: performance

- **Server**

- listen on port for client connection
- when client sends data, send reply message

- **Client**

- connect to server on known port
- send message
- receive reply message from server
- Repeat  $10^6$  times and measure time elapsed

# 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!"*

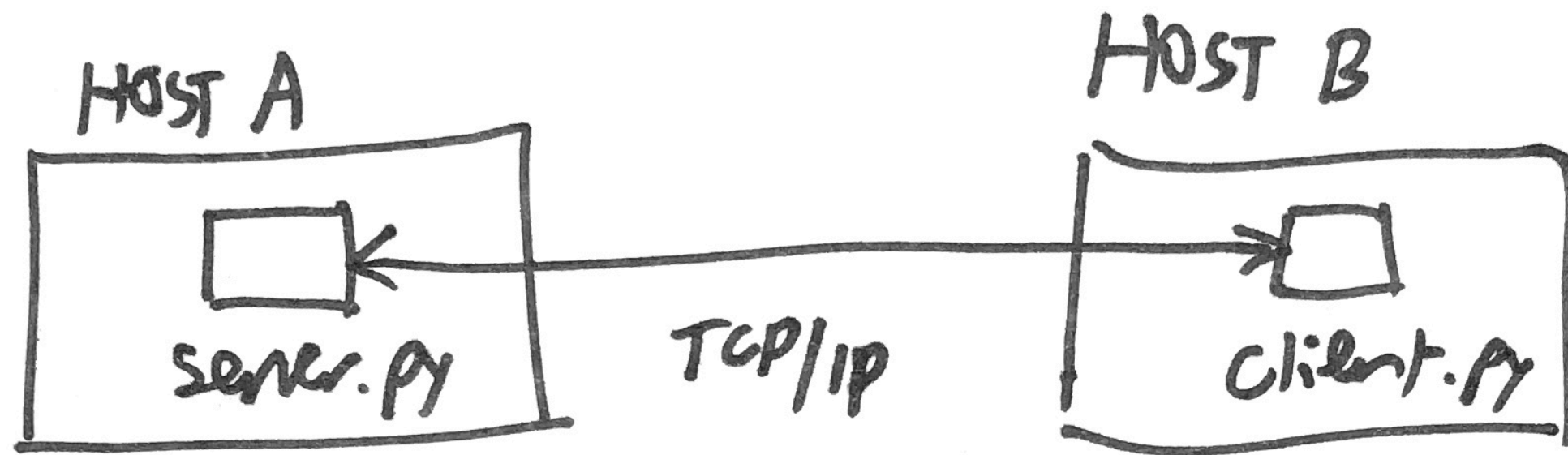


# Demo of DIY simple messaging



# DIY architecture

DIY



# DIY observations

- It's easy to prototype something very simple and specific
- However, things can get complicated very quickly!
- Be careful of sunk costs. Leaky or blurry abstractions can make sunk costs worse (i.e. increase technical debt 💰😱).

# Adding features?

- DIY can work well as a starting point or prototyping ... but what about:
  - Authentication & security
  - Persistence / database / logging / storage
  - Monitoring, metrics, dashboards
  - Failure recovery
  - Redundancy & replication
  - Scalability
  - Optimization of message rate, latency, throughput, and other metrics

# Apache Kafka: in their own words 1

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

# Apache Kafka: in their own words 2

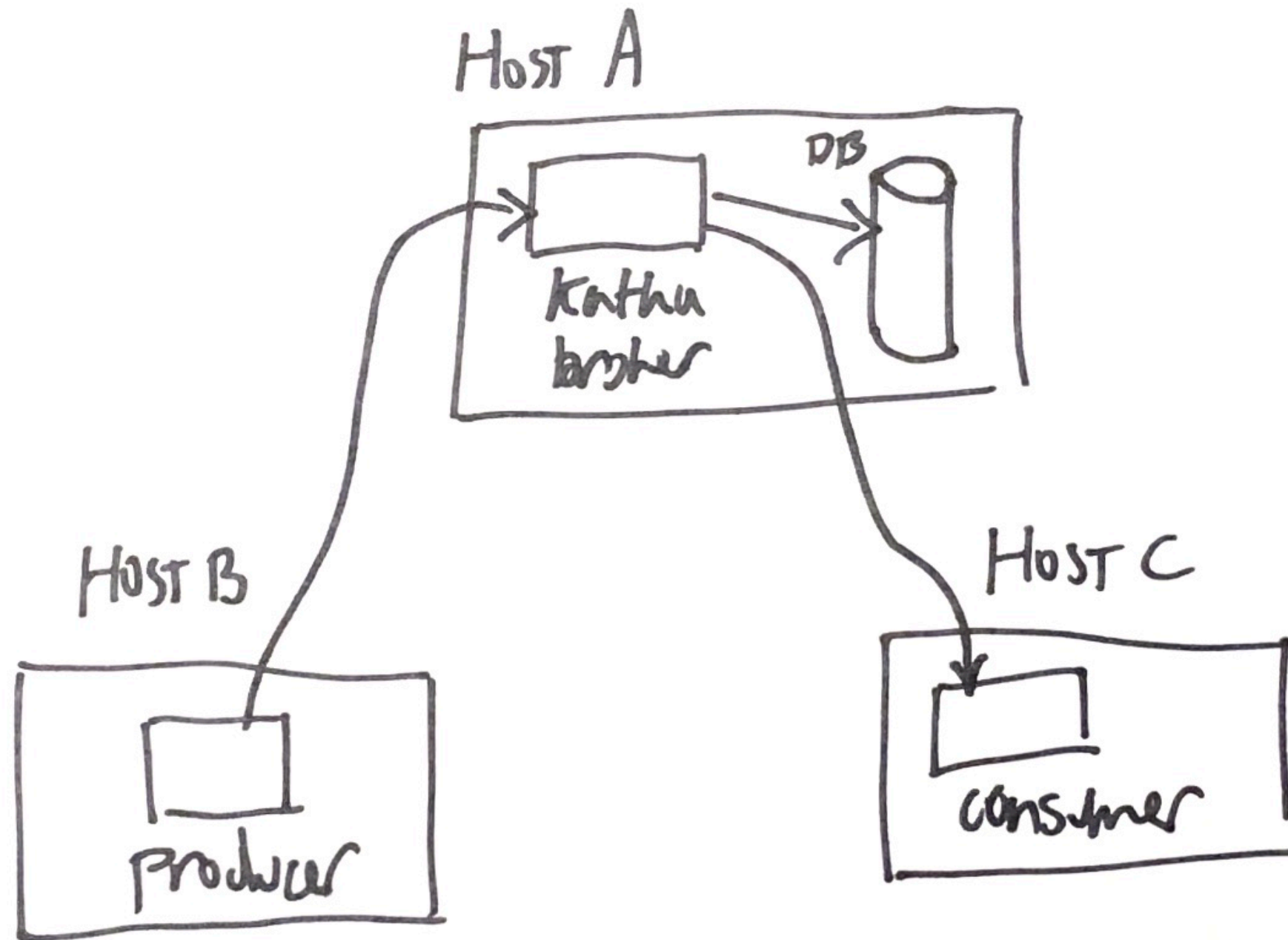
- Kafka is generally used for two broad classes of applications:
  - **Building real-time streaming data pipelines** that reliably get data between systems or applications
  - **Building real-time streaming applications** that transform or react to the streams of data
- <https://kafka.apache.org/intro>

# Demo of Kafka basics





# Kafka architecture





# Kafka observations 1

- Kafka is a messaging framework ... PLUS
  - Streaming data processing platform
  - Persistent storage system for records
- Oriented toward storing time-series data + metadata
  - datastore is a sequential list of Record objects
- JVM based; also depends on Apache Zookeeper.
- Needs "a lot" of RAM
  - more than your stock Vagrant VM
  - base config uses >1GB in java command line options

# Kafka observations 2

- Extensive configuration available
- Replication for redundancy and availability
- Kafka-specific lingo
  - Record
  - Topic
  - Partition
  - Group

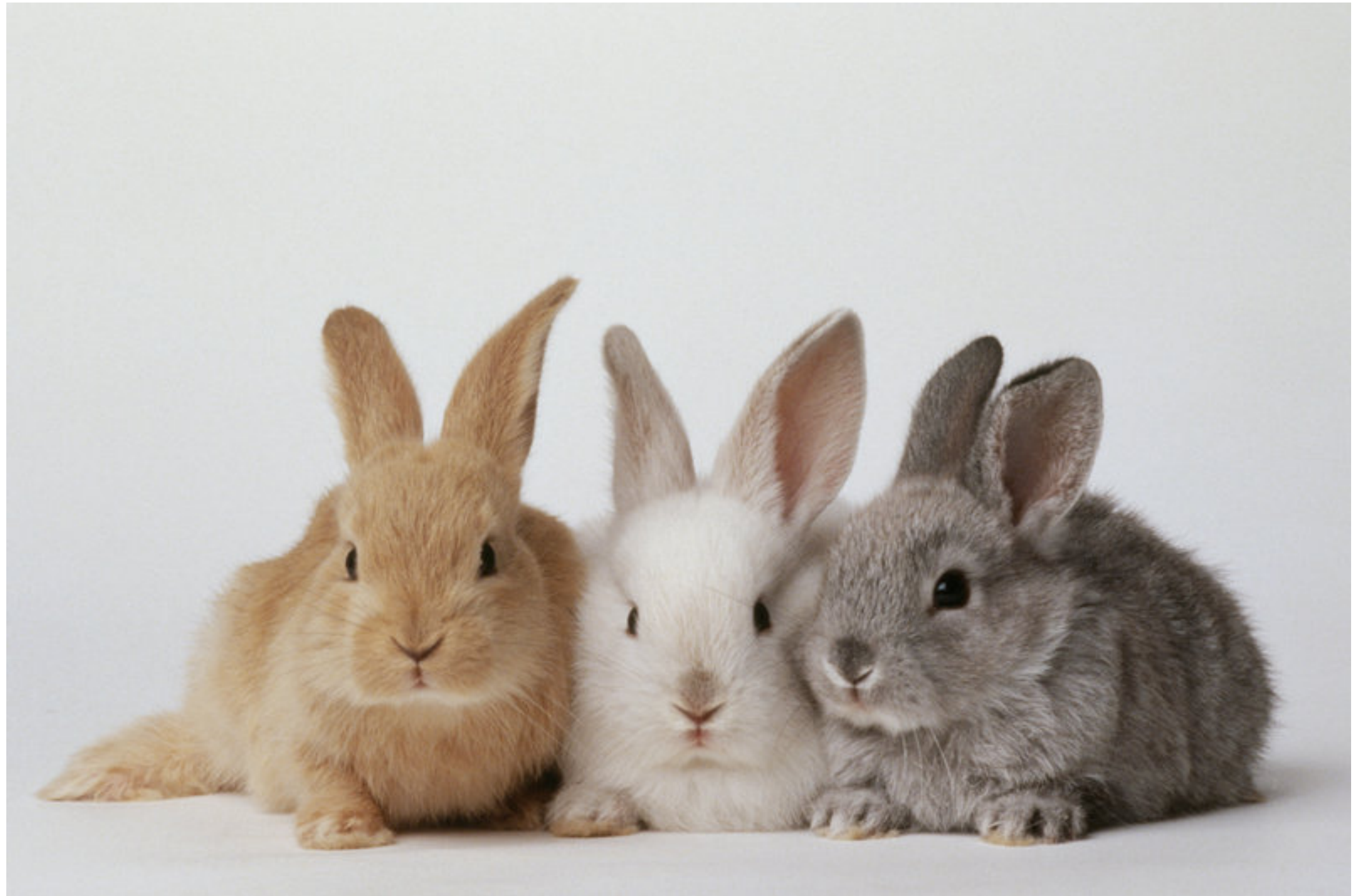
# Kafka terminology

- 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.

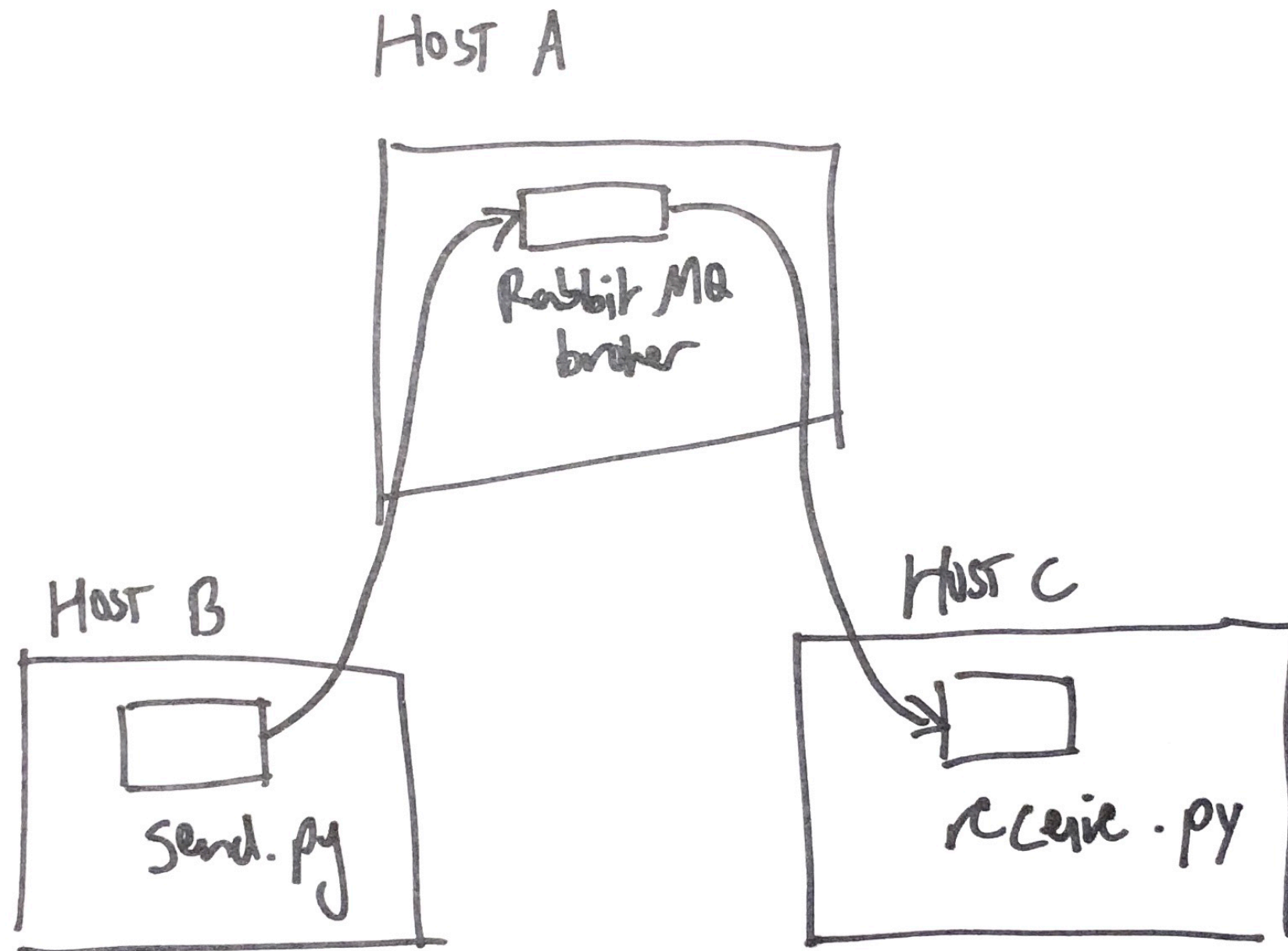
# RabbitMQ: in their own words

- **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

- Implemented 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.
- Nice built-in web UI / dashboard

# 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/>



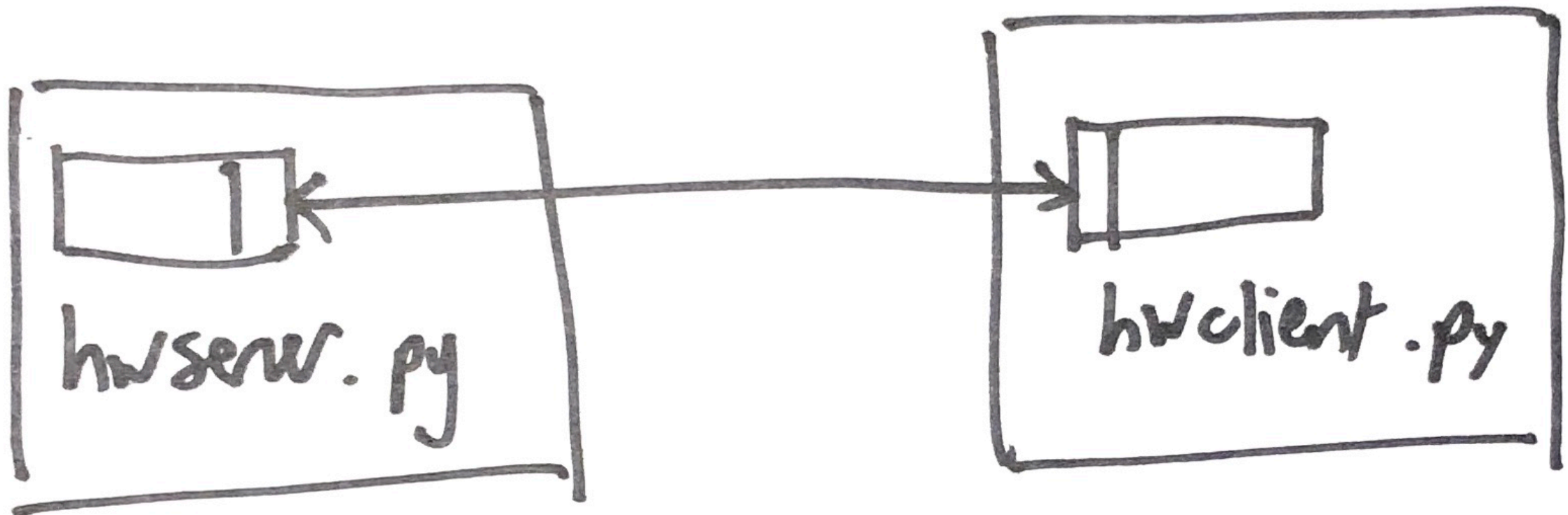
# Demo of ZeroMQ basics



# ZeroMQ architecture

Host A

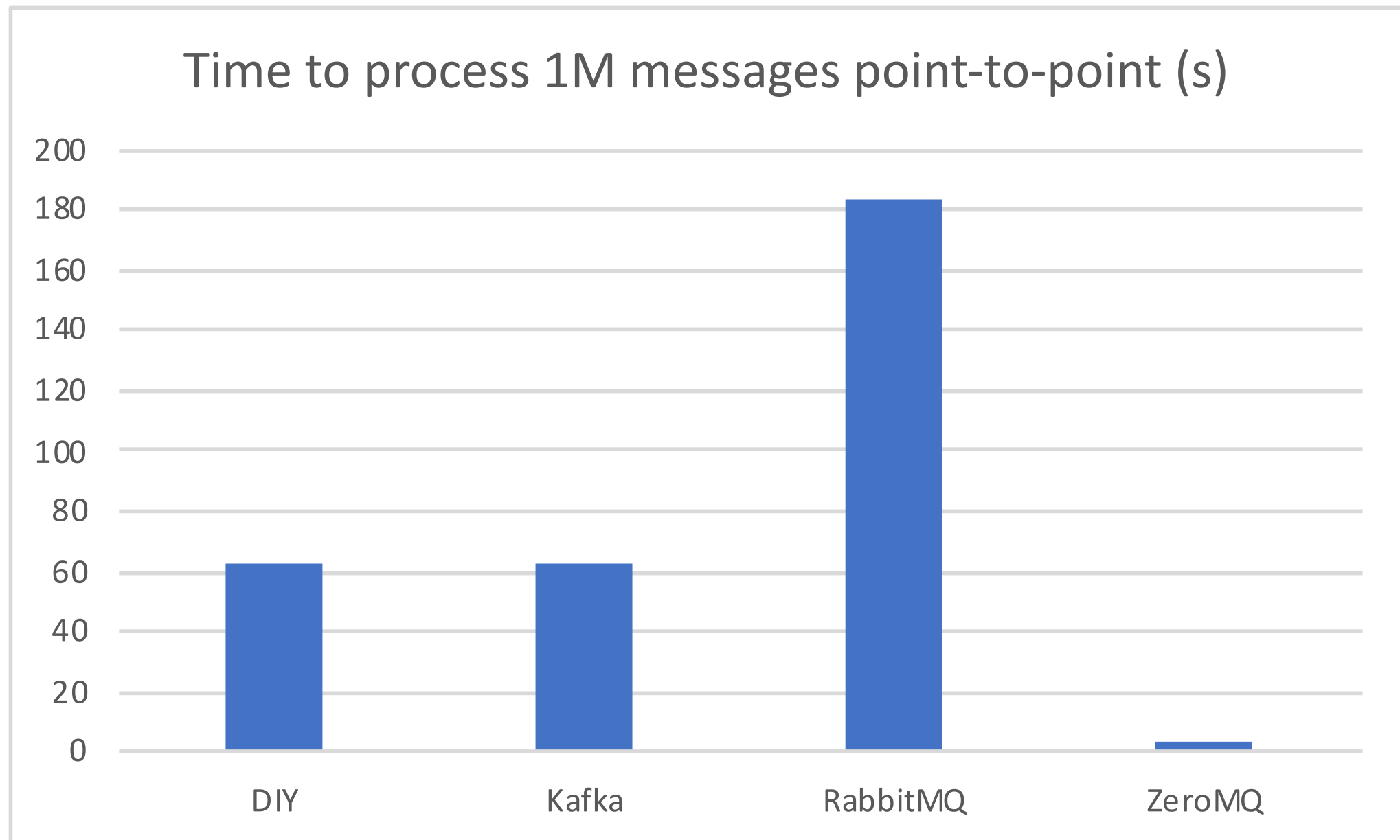
Host B



# 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.

# Message rates in testing



**Shorter bars are better**

# Message rates in testing

Framework	Message Rate (Kmsgs / s)
DIY	16.0
Kafka	15.9
RabbitMQ	5.46
ZeroMQ	253.4

# 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 (the "easy" region of the performance envelope).
- Reinventing / reimplementing messaging in a small application is not *necessarily* a bad idea. It is not necessarily a good idea either...

# Fundamental Laws 1

- A message handoff / hop always costs **something** - i.e. a broker isn't free, but can often be worth it
- More generally: TANSTAAFL. Know what tradeoffs you are making and why.
  - Persistence has a price
  - Delivery guarantees have a price
  - Recoverability/resilience has a price
  - Redundancy has a price
  - Etc.



# Fundamental Laws 2

- You can do anything, but know why you're doing it. Don't hold a beach ball underwater if you don't have to.
- Test everything, assume nothing. Example: ZeroMQ with REQ/REP vs PAIR.
- Language of framework implementation (probably...) doesn't matter: messaging is great for coupling together disparate software systems.

# Fundamental Laws 3

- Rule of the contractor:
  - customer can always request changes, and
  - changes always get more expensive as a project goes on (time, money, resources, pain, team morale)
- However, if you isolate your messaging system behind good abstractions, it may be less painful to change it.

# You can always change your mind...

## From Kafka to ZeroMQ for real-time log aggregation



Tomasz Janczuk on 04 Sep 2015



This post is about ditching Kafka in favor of ZeroMQ, based on a year-long experience of using Kafka for real-time log aggregation in a production setting of Auth0 Webtasks. What, why, and how - read on.

### What is Auth0 Webtasks?

Auth0 Webtasks is a hosting platform for Node.js microservices. You can author a snippet of Node.js code, and deploy it in a jiffy to the cloud without worrying about the things that most hosting platforms make you worry about. In other words, *all you need is code*.

<https://tomasz.janczuk.org/2015/09/from-kafka-to-zeromq-for-log-aggregation.html>

# Fundamental Laws 4

- Do pairwise simulations (point-to-point) before simulating more complicated topologies - you can learn a lot from simple tests
- Proverb: If your application is complicated when things are simple, it's not going to get any simpler when things are complicated

# Recommendations

- The more your requirements list increases, and the bigger your application(s) grow, the more likely an existing framework will be the best solution ("**there are no new problems**").
- Select your framework based on its **unique portfolio of features and limitations** that line up best with your business needs (need/nice-to-have/don't-care/don't-want/dealbreaker)
- 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)

Thank you! 🙏

