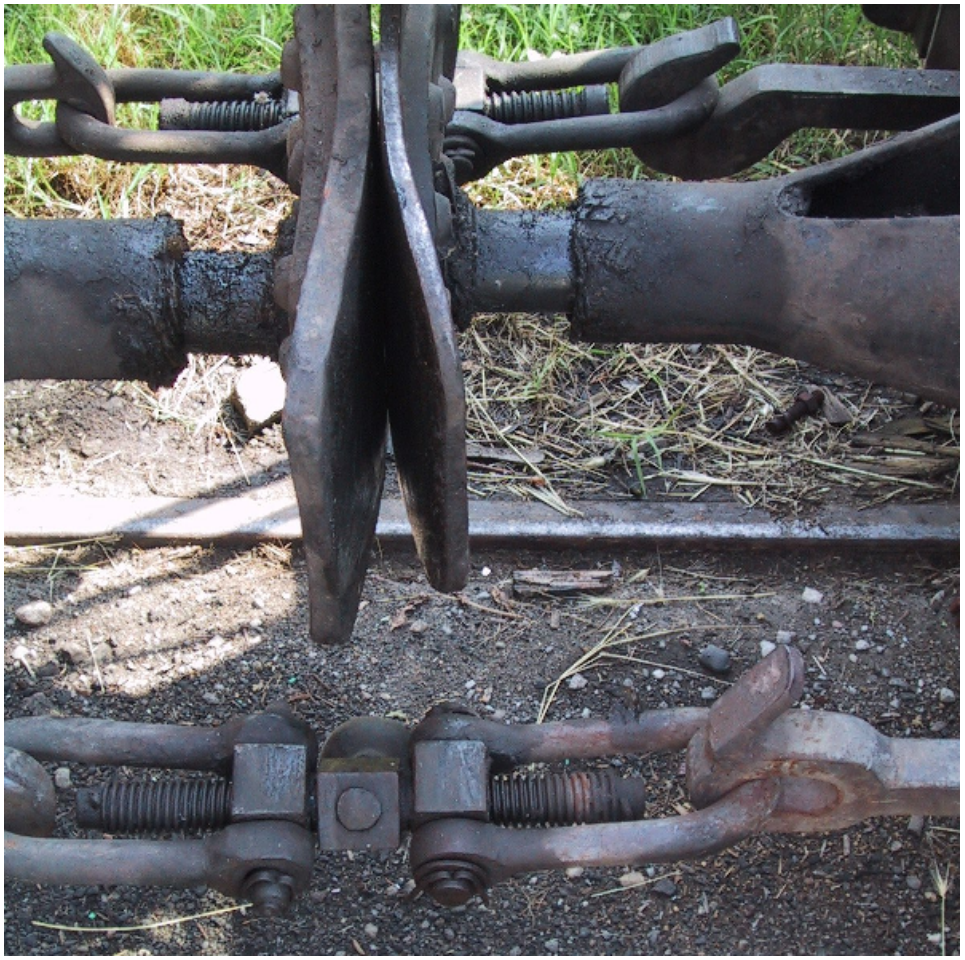
Event Based Approaches

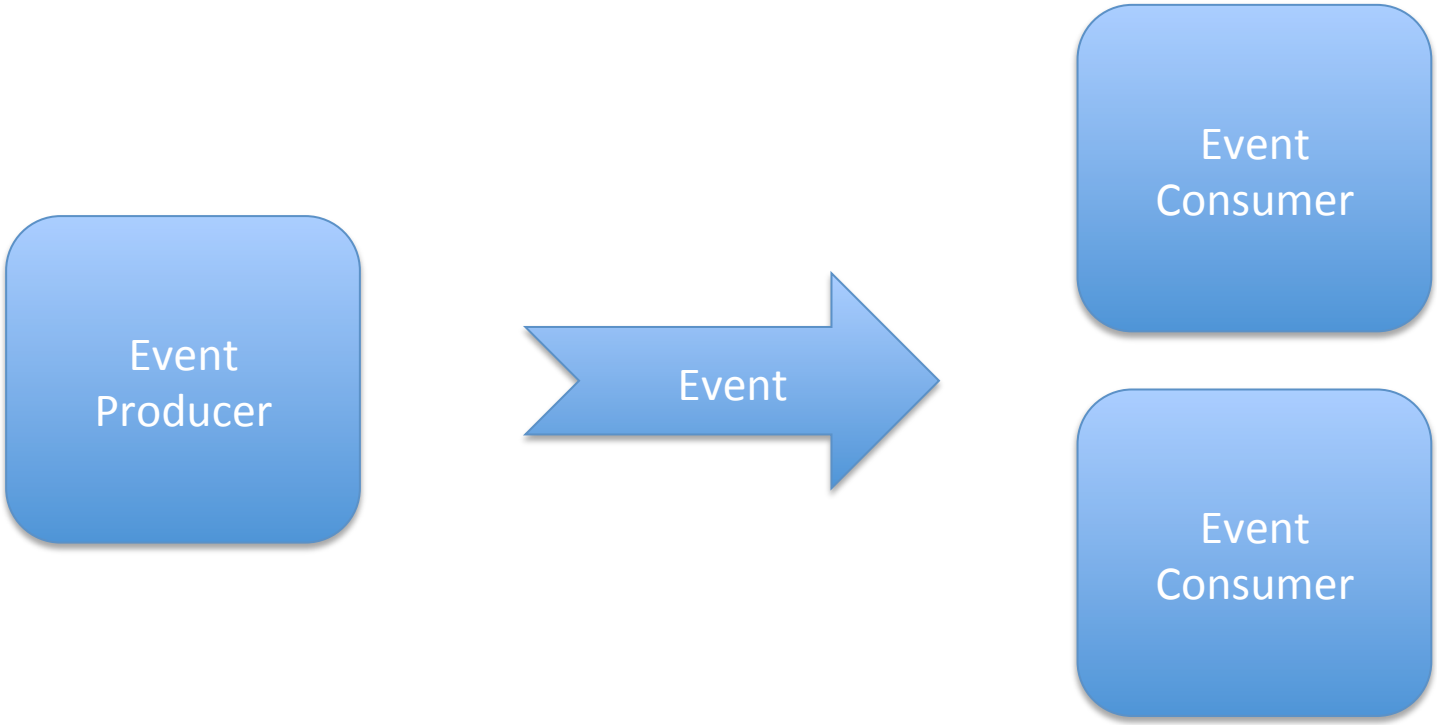
Oxford University
Software Engineering
Programme
December 2018

Why Asynchronous?

Loose coupling



Event Driven Architecture

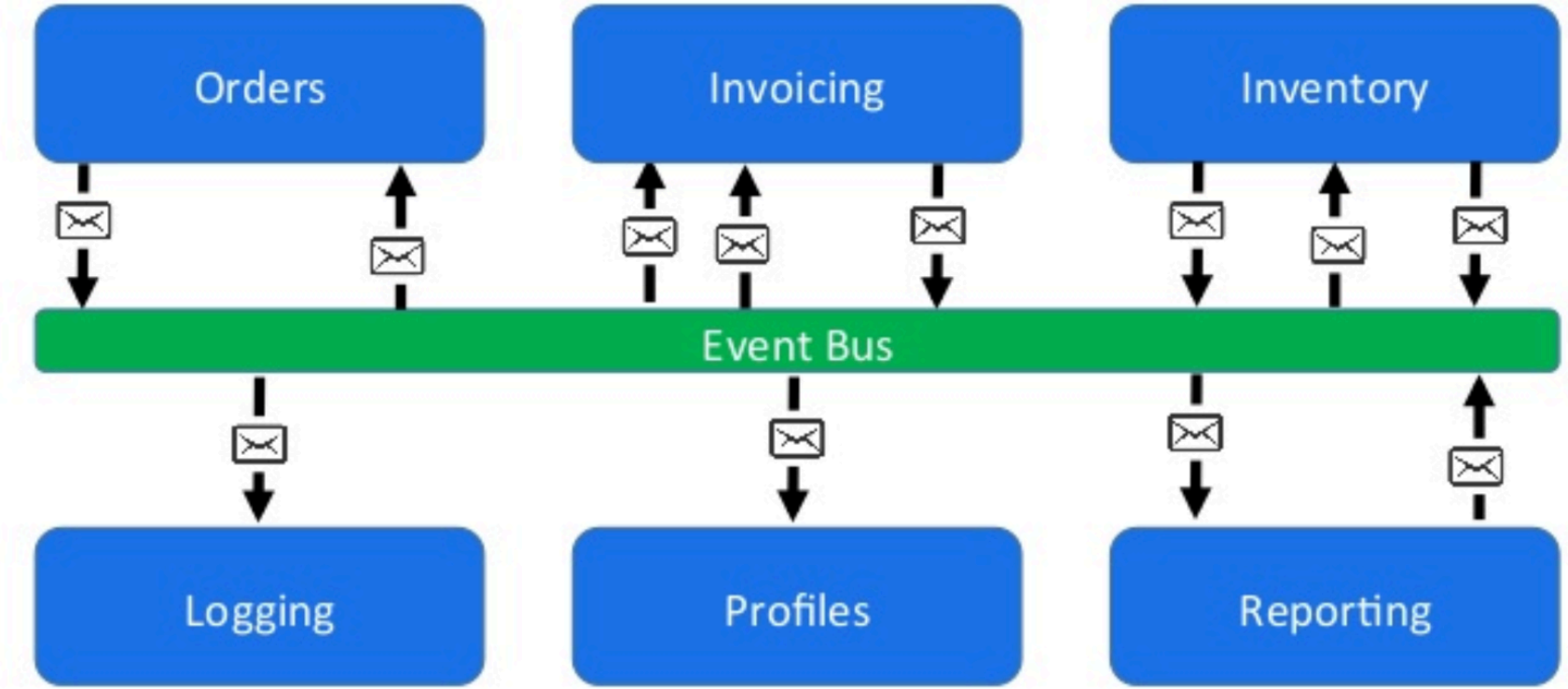


Loose coupling in EDA

- Location
  - Logical addresses
- Time
  - Asynchronous, Store/Forward, Replay
- Message
  - JSON, XML, ProtoBuf, etc
- Pattern
  - Pub/Sub, Queue, 1-1, 1-many, many-many, request-reply

EDA

Loose coupling via event bus



https://www.slideshare.net/CentricConsulting/eventdriven-architecture-57613466

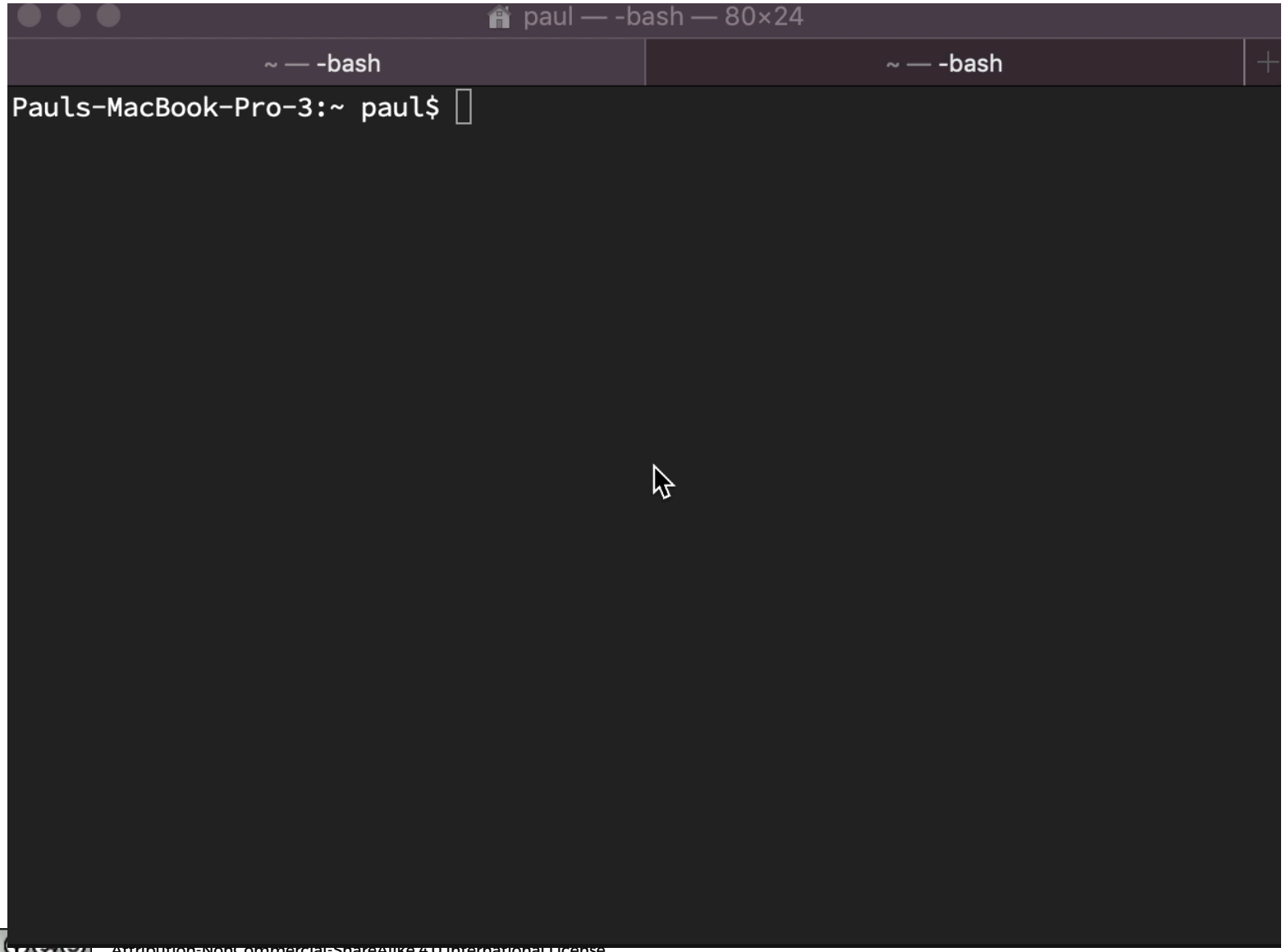
Message distribution systems

- NATS
- MQTT
- STOMP
- AMQP / RabbitMQ
- Kafka

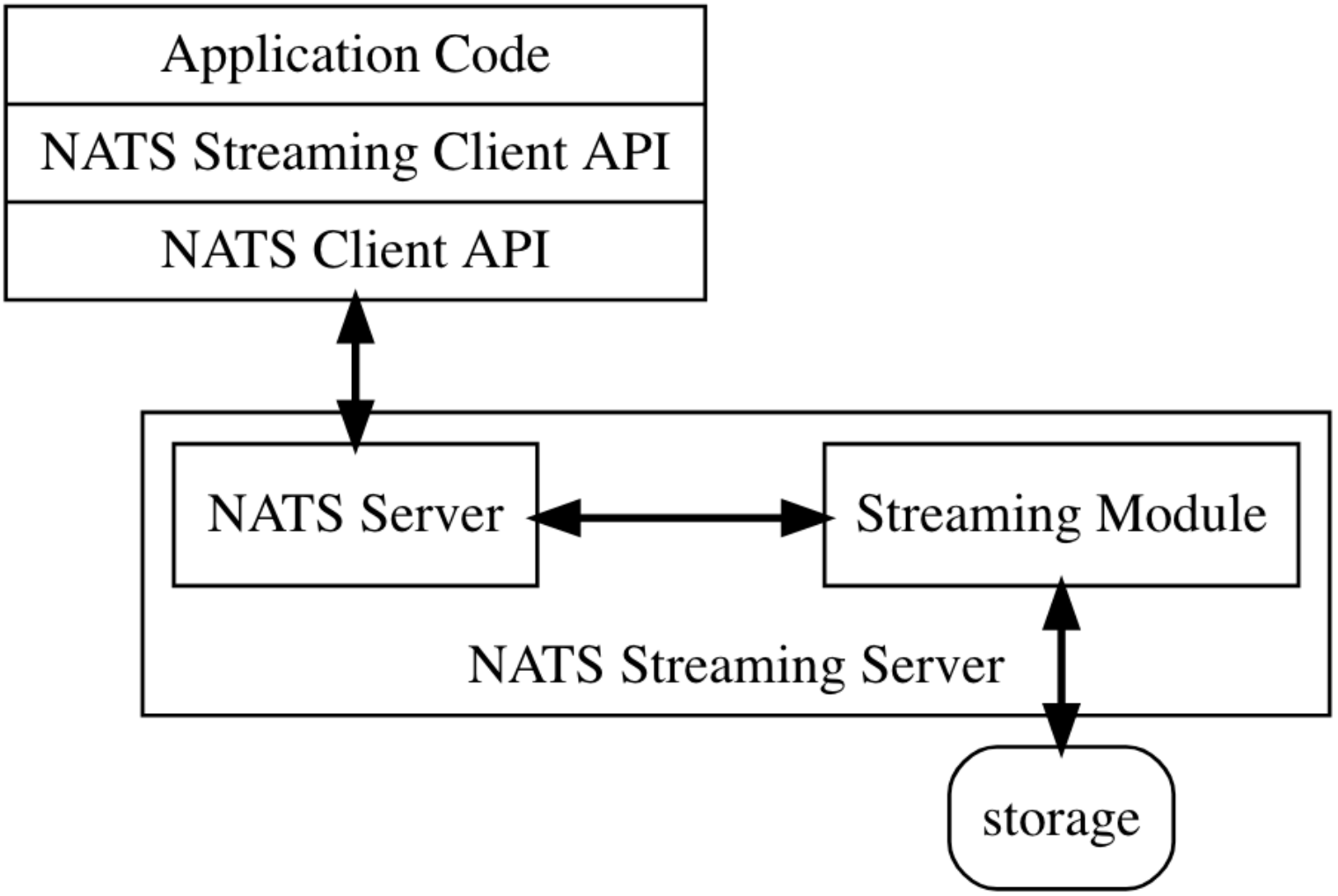
NATS

- Simple text based protocol
- Multiple patterns
  - Pure Pub/Sub
  - Request-Reply
  - Queuing
- Clustered servers
  - Distributed queue across clusters
  - Cluster aware clients

NATS simple demo



NATS Streaming



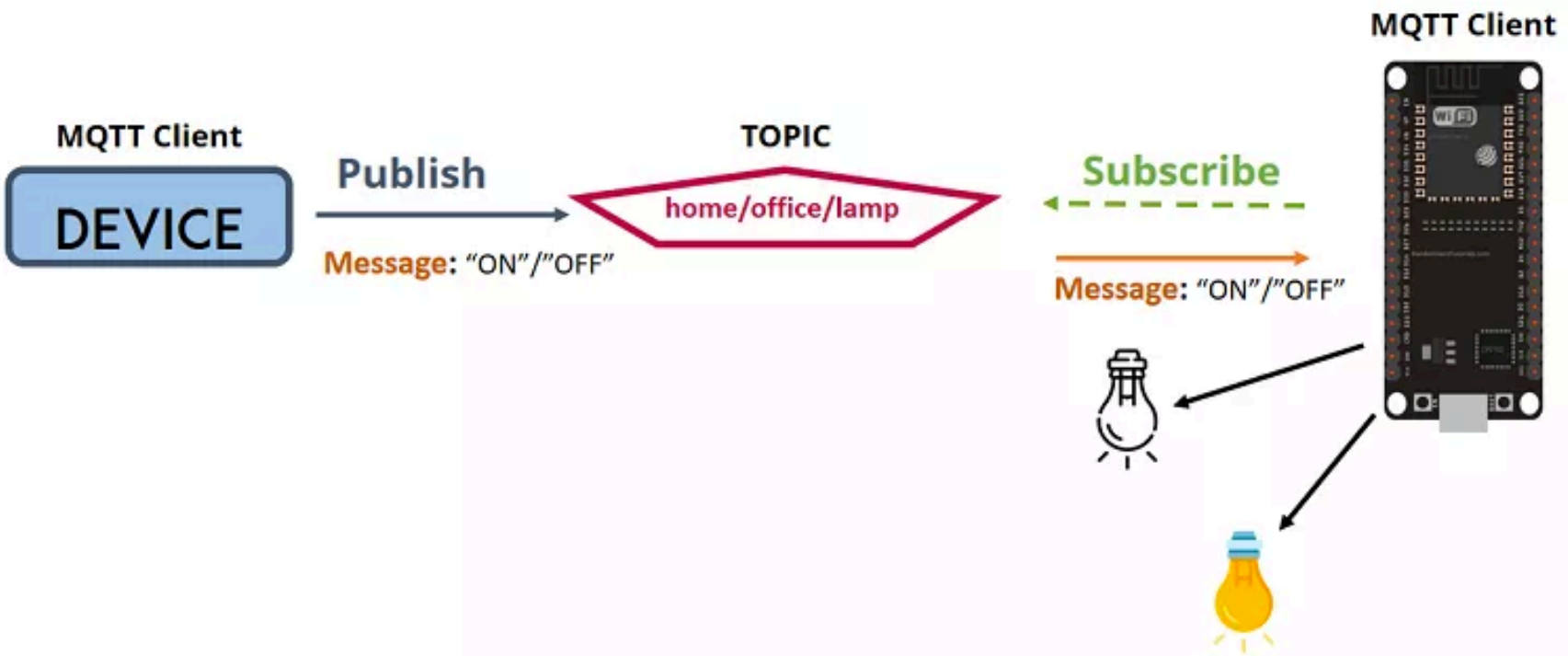
NATS Streaming

- At least once delivery
- Publisher rate limiting
- Subscriber rate limiting
- Message Replay
- Durable Subscriptions

MQTT

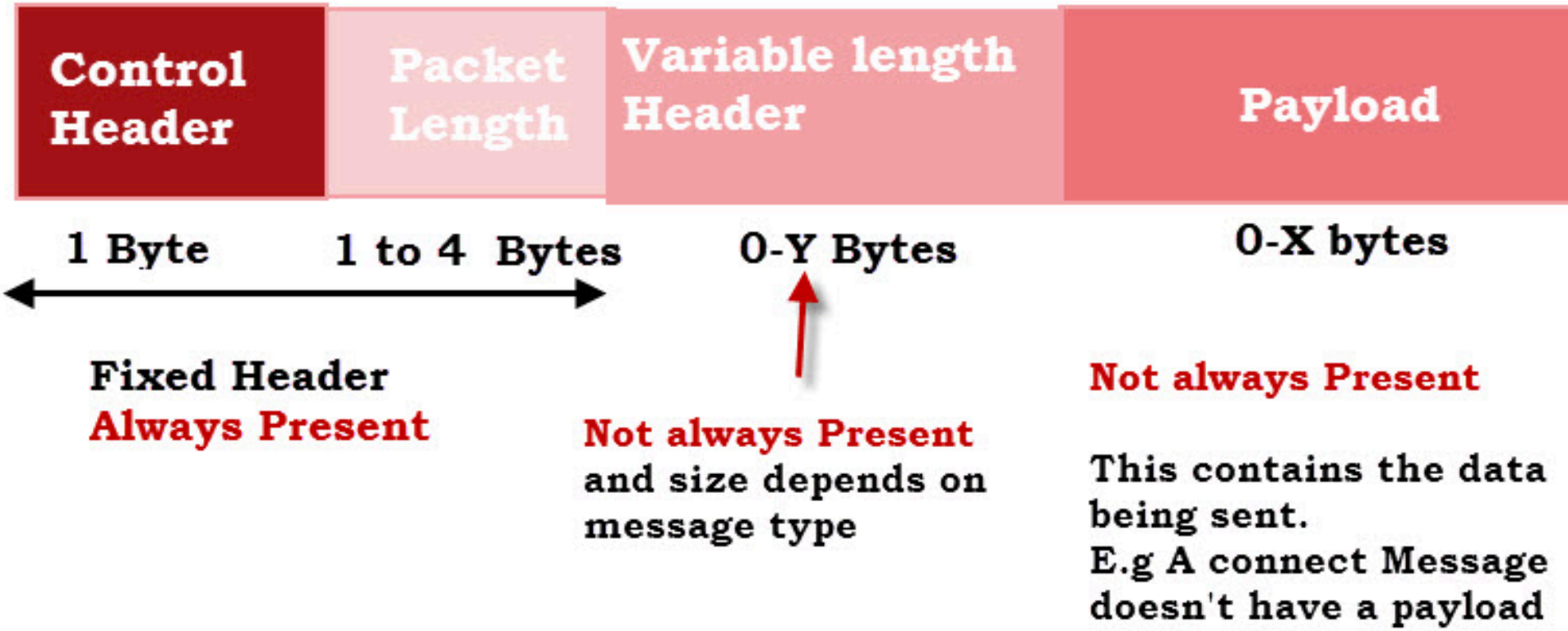
- Very lightweight binary protocol
  - 2-byte overhead
- Widely used in IoT scenarios
- Pub-sub only until MQTT5
- QoS levels
  - Fire and forget QoS0
  - At least once QoS1
  - Exactly once QoS2

MQTT



MQTT Packets

http://www.steves-internet-guide.com/mqtt-protocol-messages-overview/



MQTT Standard Packet Structure

MQTT Message Types and Hex Codes

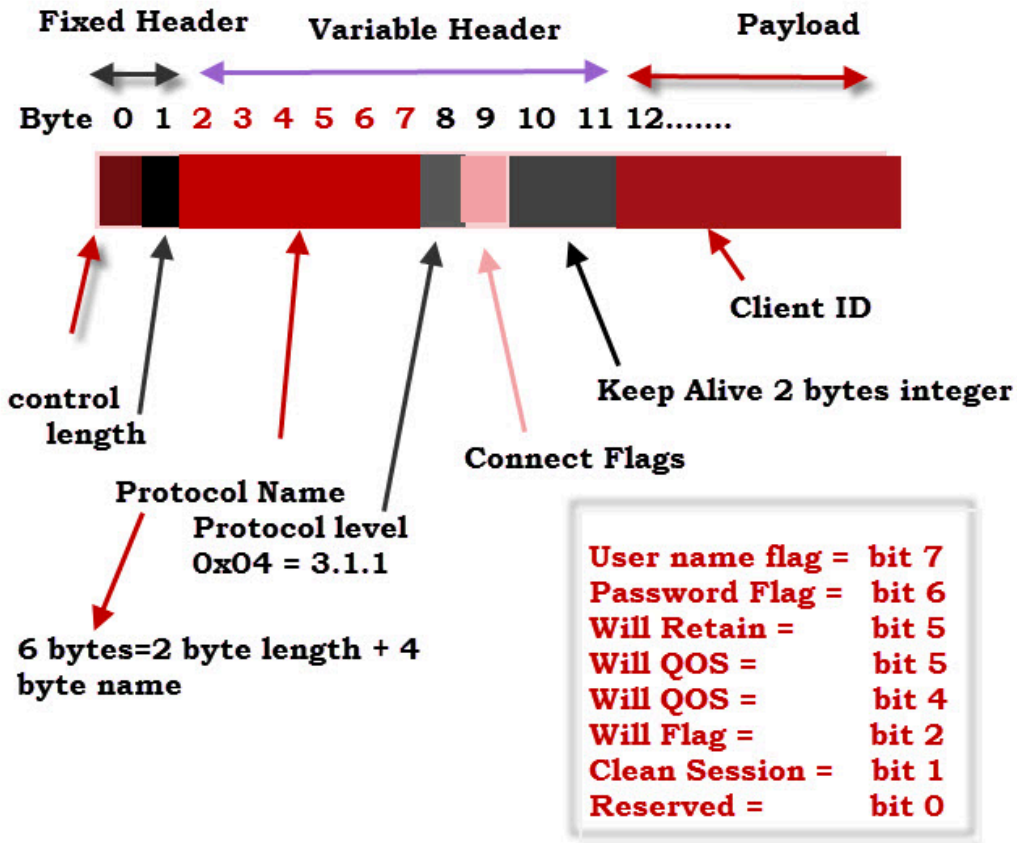
```
# Message types
CONNECT = 0x10
CONNACK = 0x20
PUBLISH = 0x30
PUBACK = 0x40
PUBREC = 0x50
PUBREL = 0x60
PUBCOMP = 0x70
SUBSCRIBE = 0x80
SUBACK = 0x90
UNSUBSCRIBE = 0xA0
UNSUBACK = 0xB0
PINGREQ = 0xC0
PINGRESP = 0xD0
DISCONNECT = 0xE0
```

=16 decimal

=128 decimal

=224 decimal

MQTT Connect Message Structure



python_test

Connection message code example

```
connecting client =,name, clean session = True
sending command  0x10  sending flags = 0
sending  bytearray(b'\x10\x17\x00\x04MQTT\x04\x02\x00<\x00\x0bpyth
on_test')
```

Total length =23 decimal

connect flags clean session is True

keep alive =60 seconds
Ascii < =60

length of client id 0xb = 11 decimal

STOMP

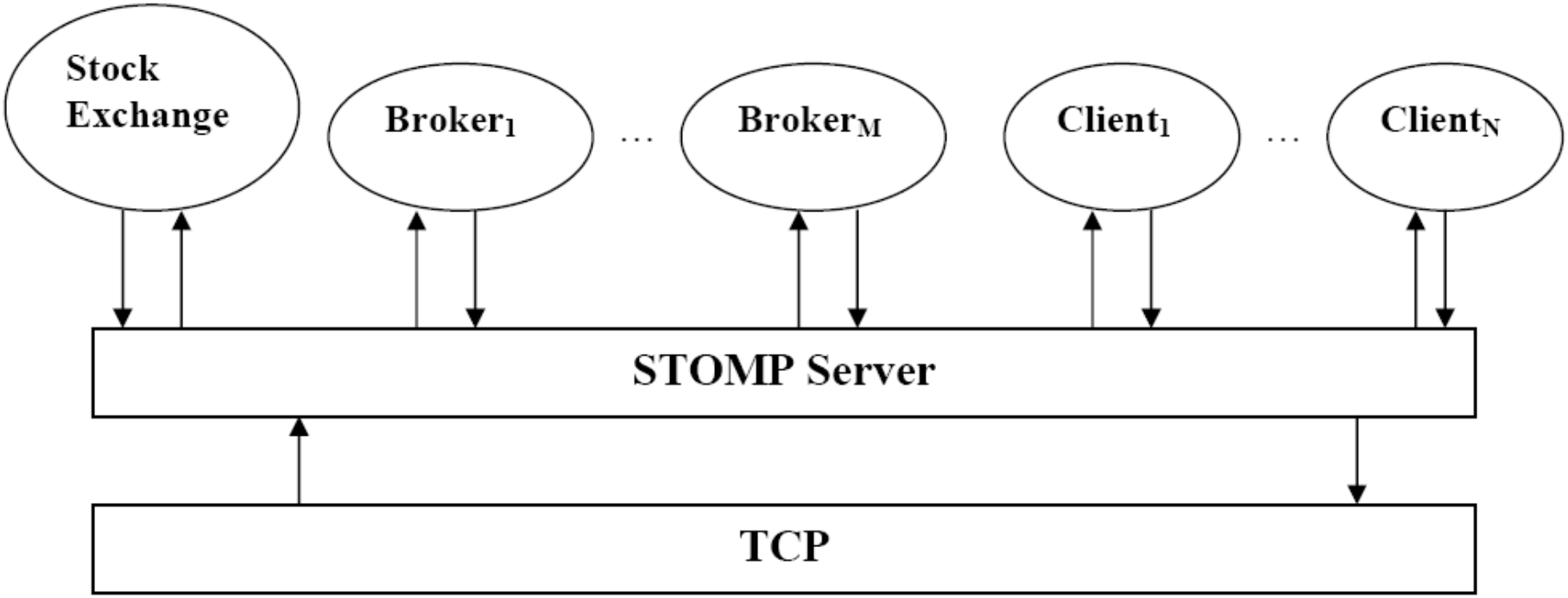


Figure 3: STOMP Server and Clients, over TCP layer

Stomp 

The Simple Text Oriented Messaging Protocol

What is it?

STOMP is the Simple (or Streaming) Text Orientated Messaging Protocol.

STOMP provides an interoperable wire format so that STOMP clients can communicate with any STOMP message broker to provide easy and widespread messaging interoperability among many languages, platforms and brokers.

Simple Design

STOMP is a very simple and easy to implement protocol, coming from the HTTP school of design; the server side may be hard to implement well, but it is very easy to write a client to get yourself connected. For example you can use Telnet to login to any STOMP broker and interact with it!

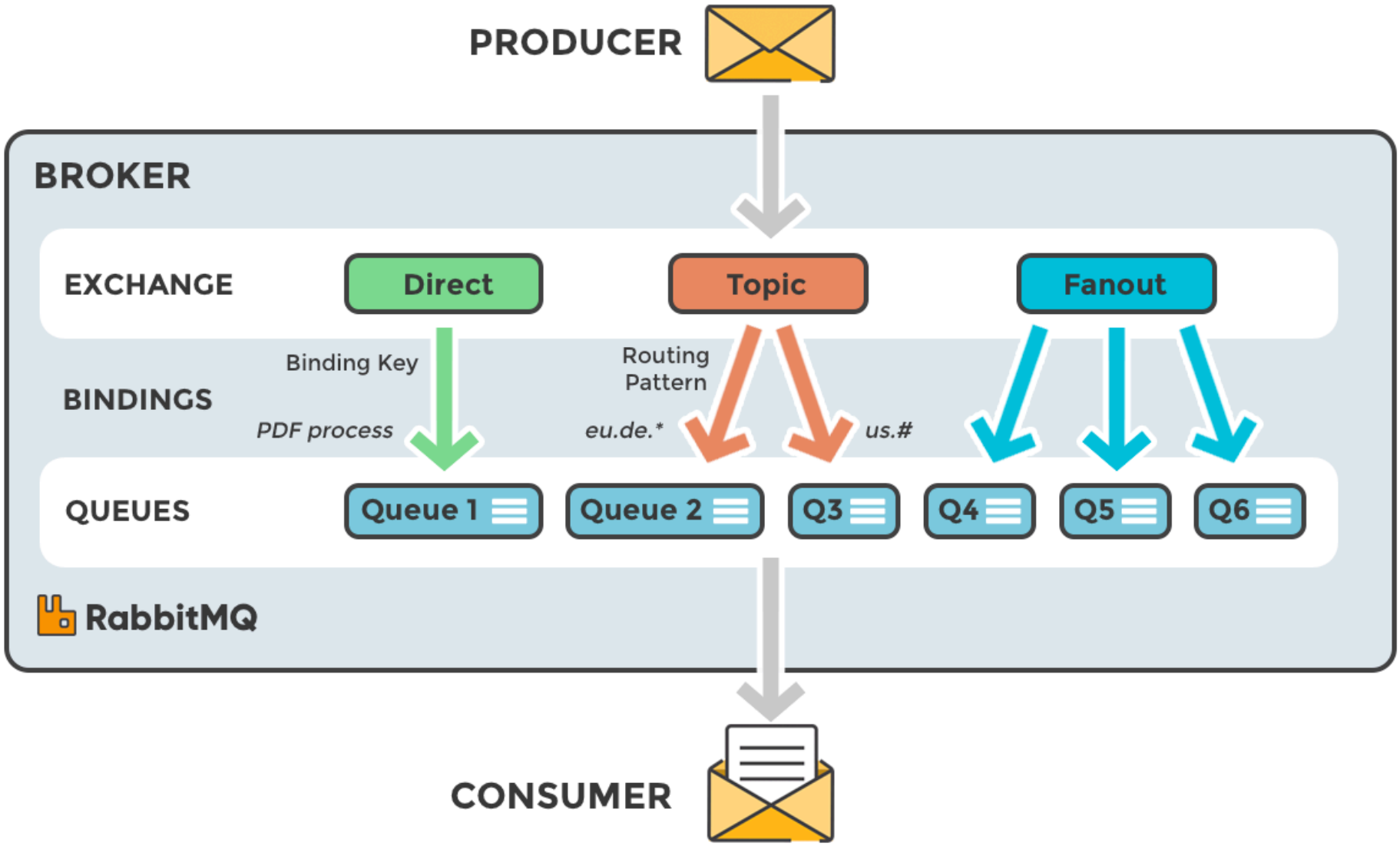
Many developers have told us that they have managed to write a STOMP client in a couple of *hours* to in their particular language, runtime or platform into the STOMP network. So if your favored language/runtime of choice does not offer a good enough STOMP client don't be afraid to write one.

STOMP “verbs”

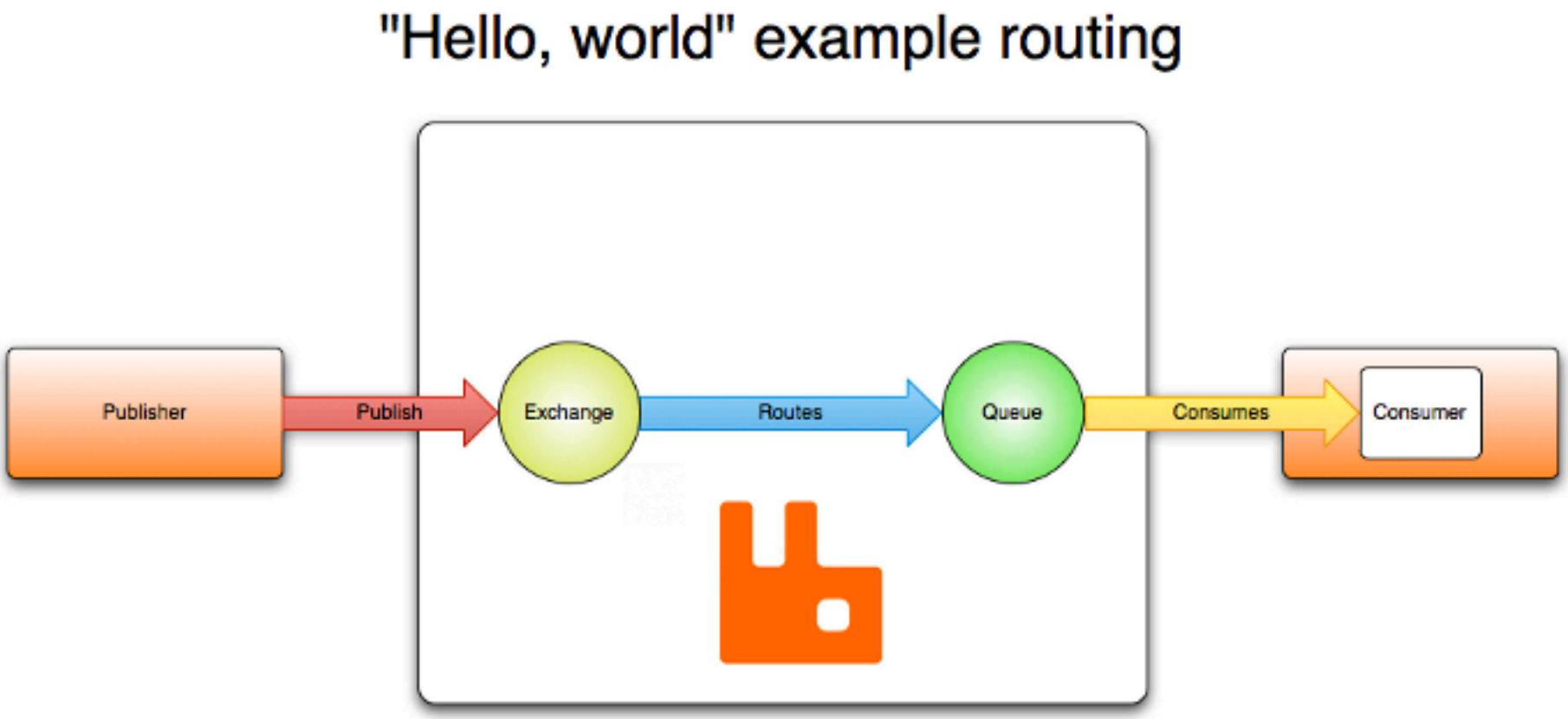
- CONNECT
- SEND
- SUBSCRIBE
- UNSUBSCRIBE
- BEGIN
- COMMIT
- ABORT
- ACK
- NACK
- DISCONNECT

AMQP / RabbitMQ

- AMQP is an advanced messaging protocol
  - Designed to meet more enterprise needs
  - Emerged from JP Morgan attempting to decouple from proprietary systems
- Standardised in OASIS
  - Although many implementations prefer 0-91 to 1-00



RabbitMQ

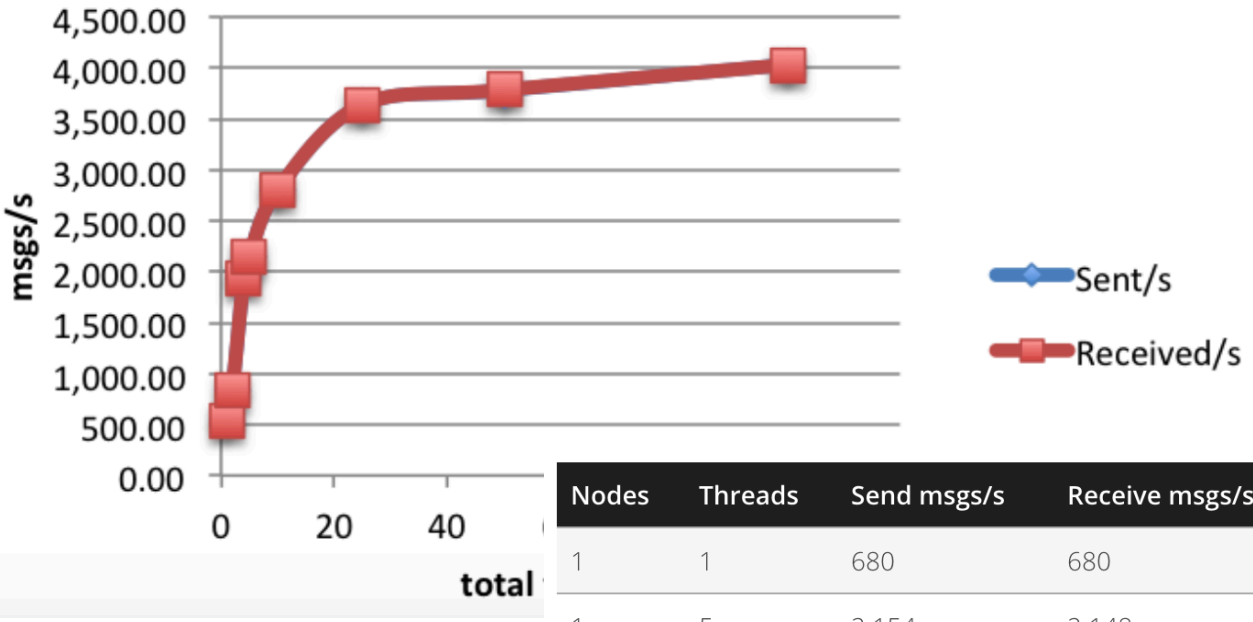


RabbitMQ

- Written in Erlang
- Designed to implement the AMQP 0-91 spec
  - Now extended to support STOMP, MQTT, AMQP 1-0 and HTTP
- Clusterable and high-performance
- Transactional

RabbitMQ performance

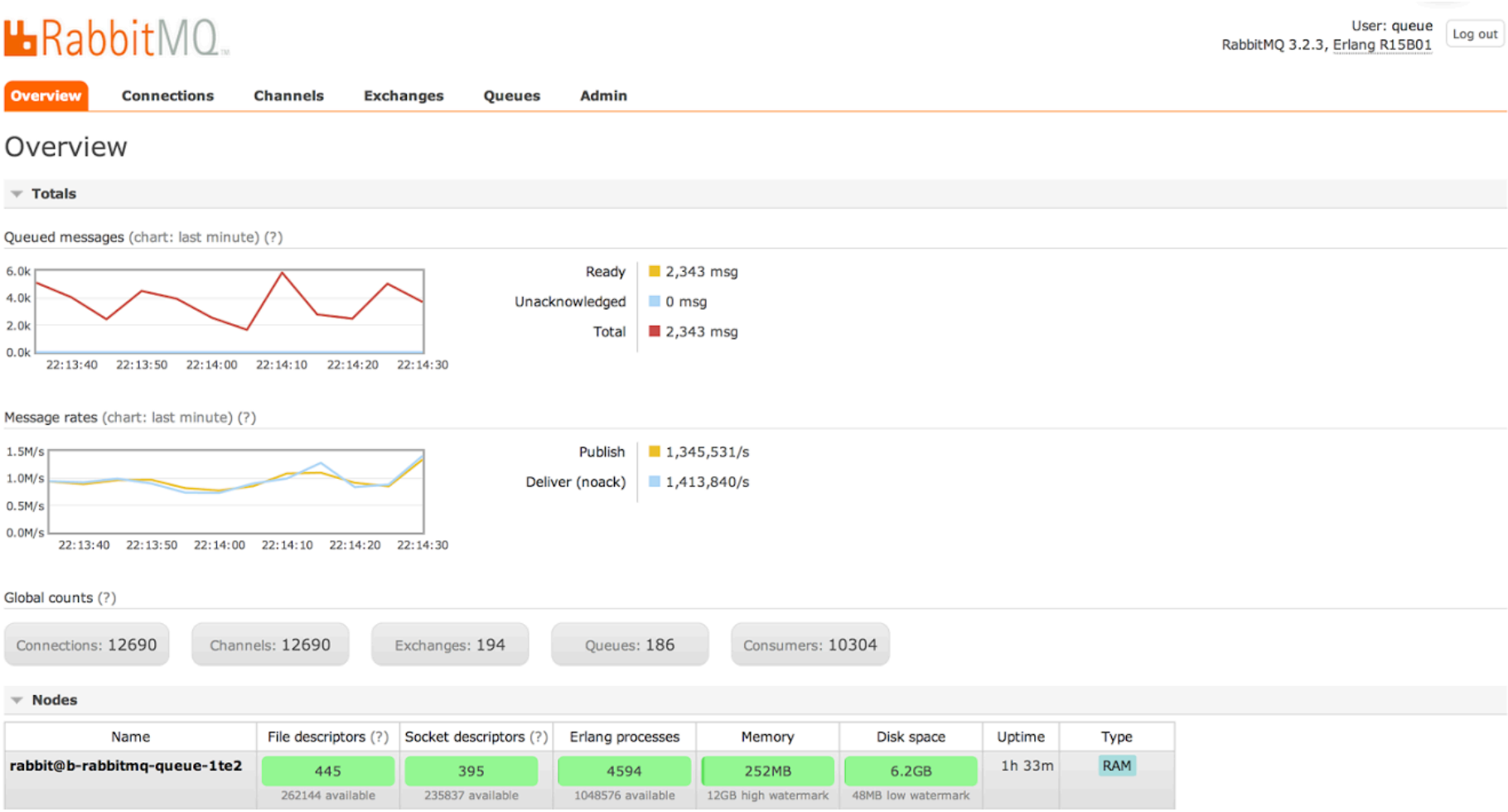
https://softwaremill.com/mqperf/



Nodes	Threads	Send msgs/s	Receive msgs/s	Processing latency	Send latency
1	1	680	680	**99**	48
1	5	2 154	2 148	107	**48**
1	25	3 844	3 844	122	66
2	1	844	843	109	
2	5	2 803	2 805	113	
2	25	3 780	3 784	141	
4	1	1 929	1 930	99	
4	5	3 674	3 673	126	
4	25	**4 331**	**4 330**	179	504

>1m msgs/sec

https://content.pivotal.io/blog/rabbitmq-hits-one-million-messages-per-second-on-google-compute-engine



ActiveMQ / Artemis

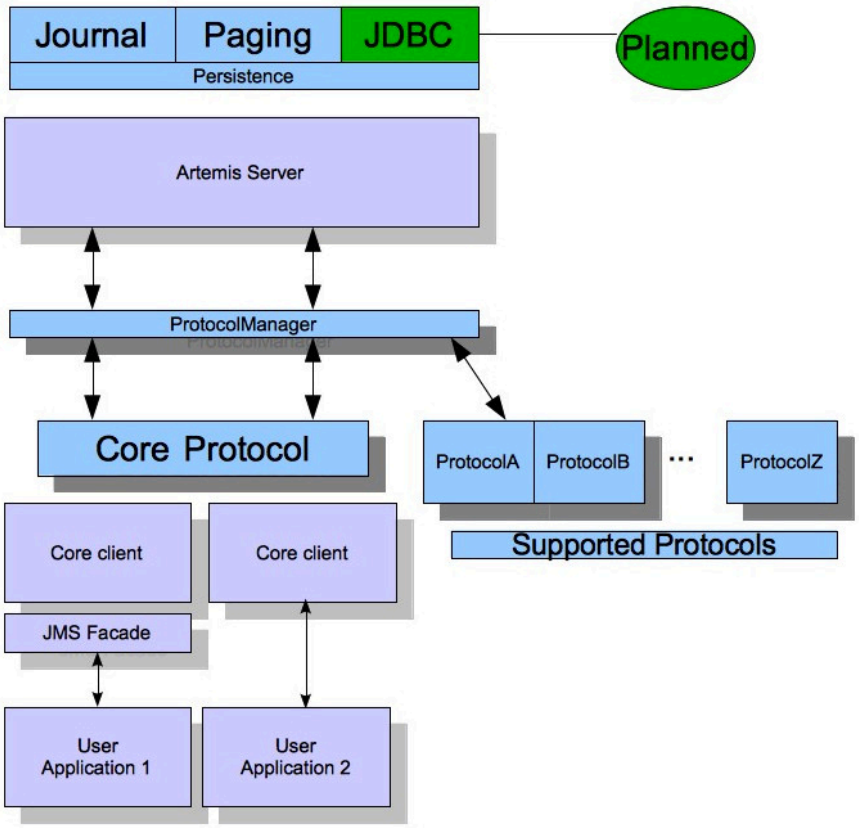


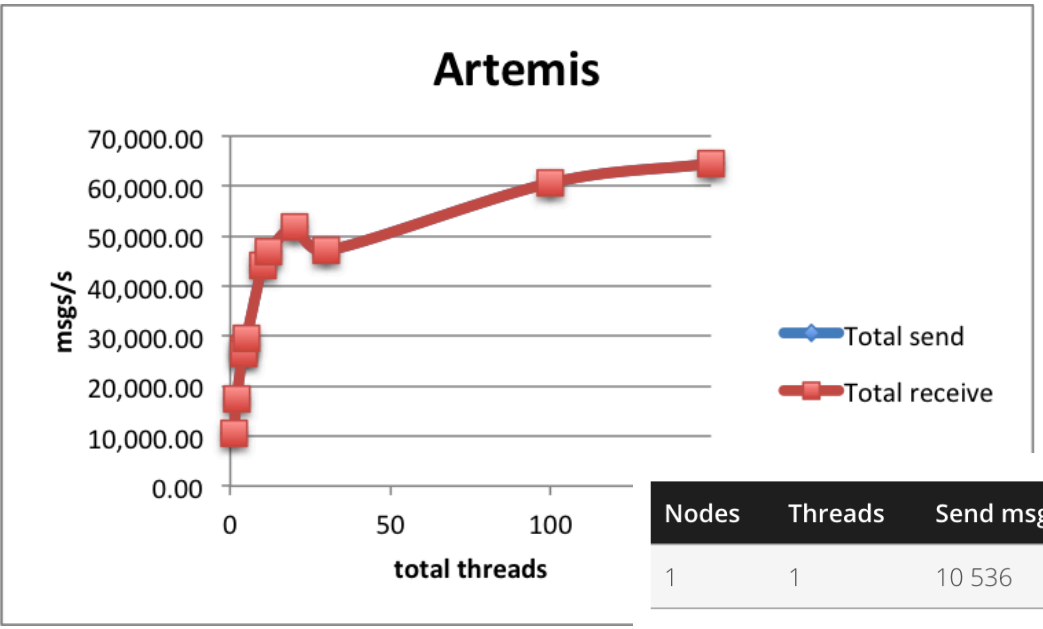
Figure 3.1 Artemis High Level Architecture

Artemis

- Supports multi-protocols:
  - "JMS", AMQP, STOMP, OpenWire, MQTT, REST
  - Highly available and clusterable
  - Written in Java

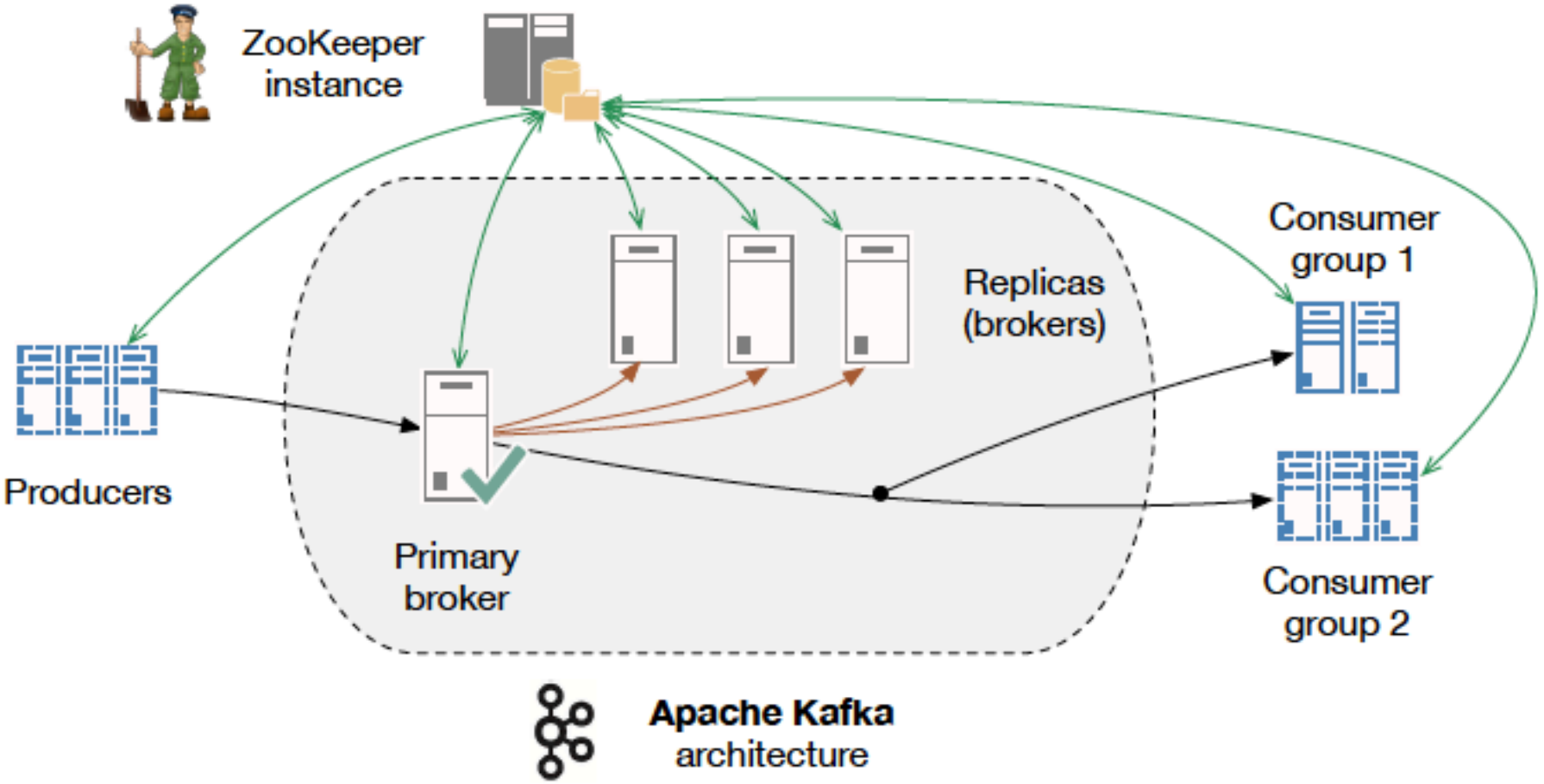
Artemis Performance

https://softwaremill.com/mqperf/

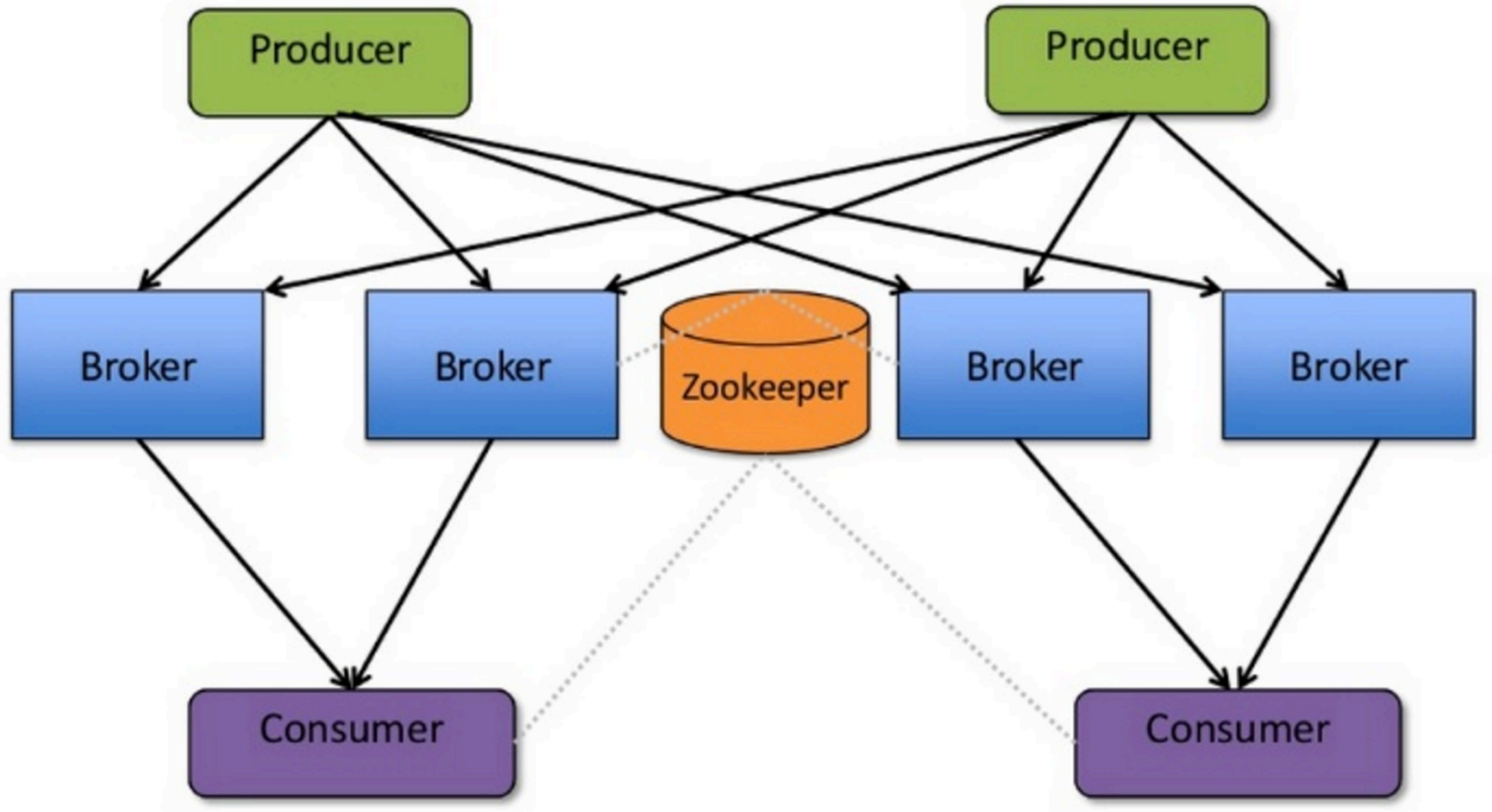


Nodes	Threads	Send msgs/s	Receive msgs/s	Processing latency	Send latency
1	1	10 536	10 536	48	45
1	5	29 476	29 476	48	47
2	1	17 515	17 515	46	46
2	5	44 003	44 003	46	47
4	1	27 197	27 197	47	47
4	5	51 724	51 720	46	47
4	25	60 619	60 619	62	48
6	5	47 078	47 082	**47**	48
6	25	**64 485**	**64 487**	122	**48**

Apache Kafka



Apache Kafka

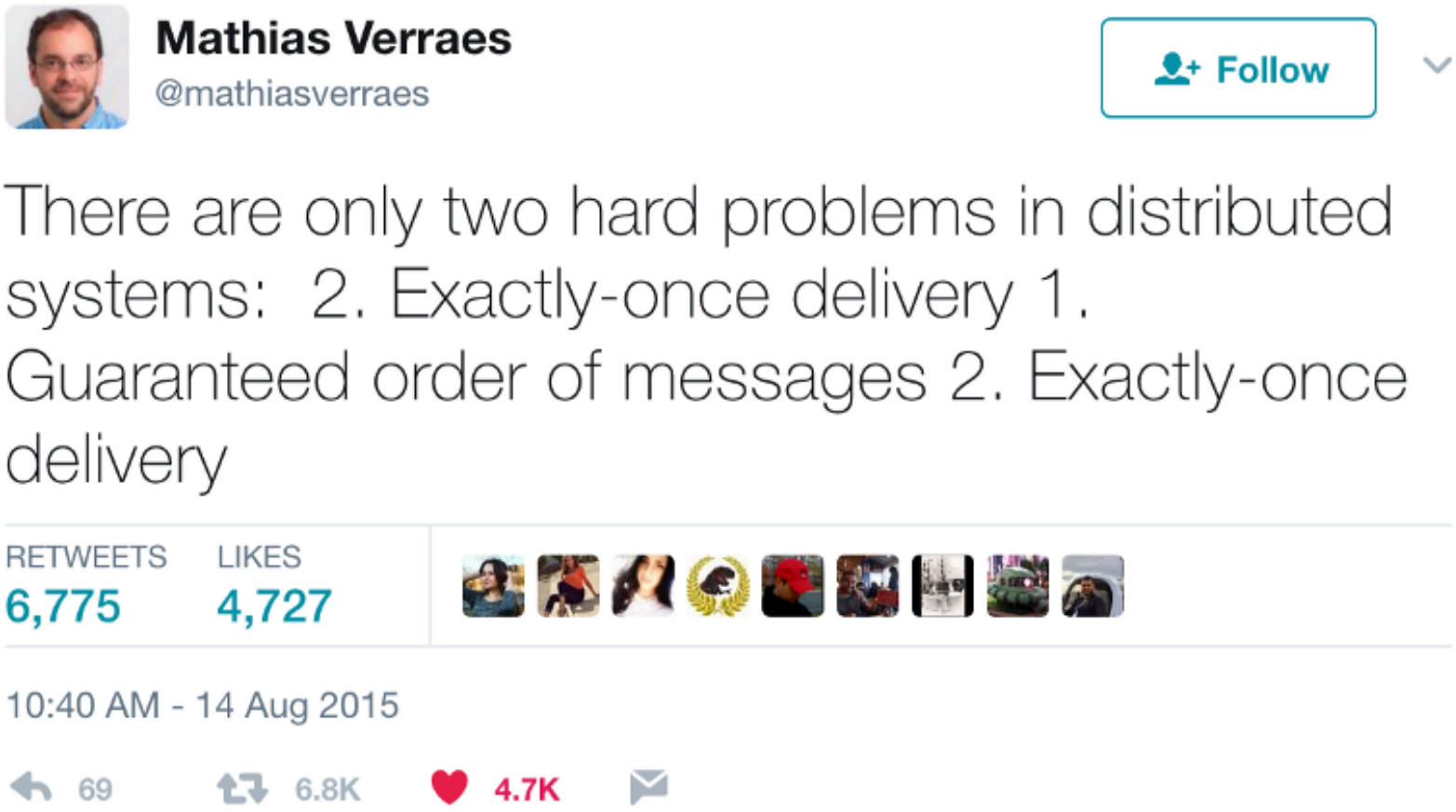


Source: http://www.slideshare.net/charmalloc/

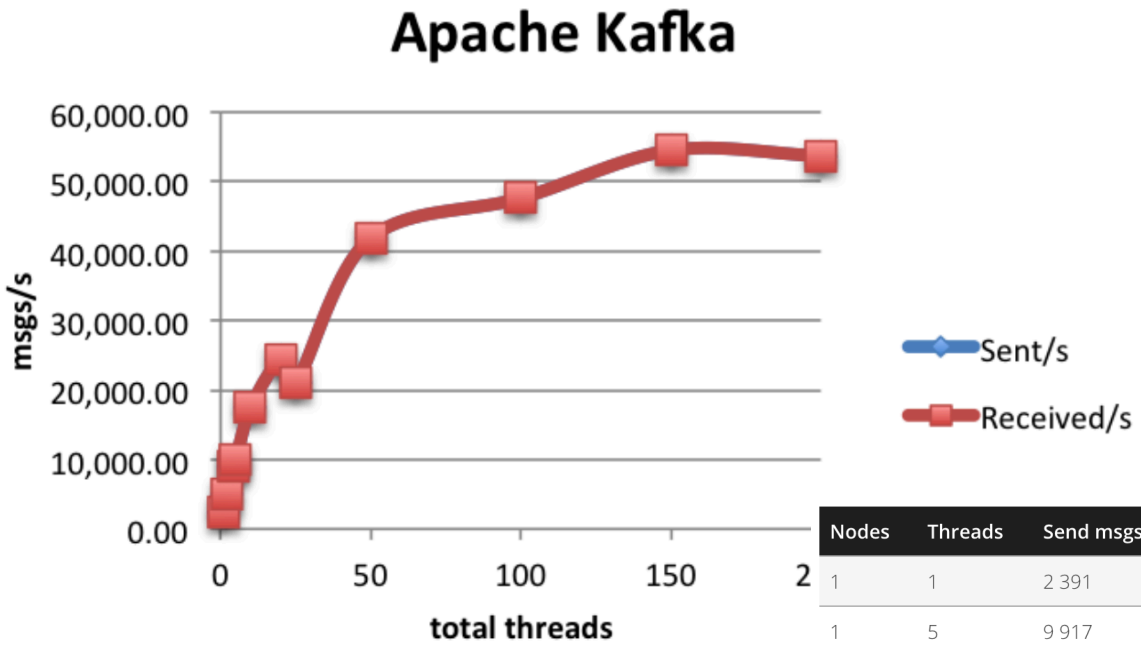
Kafka

- Applying “big data” approaches to messaging:
  - Partitioning
  - Multiple brokers
  - Elastically scalable
  - Supports clusters of co-ordinated consumers
  - Automatic re-election of leaders

Kafka exactly-once semantics

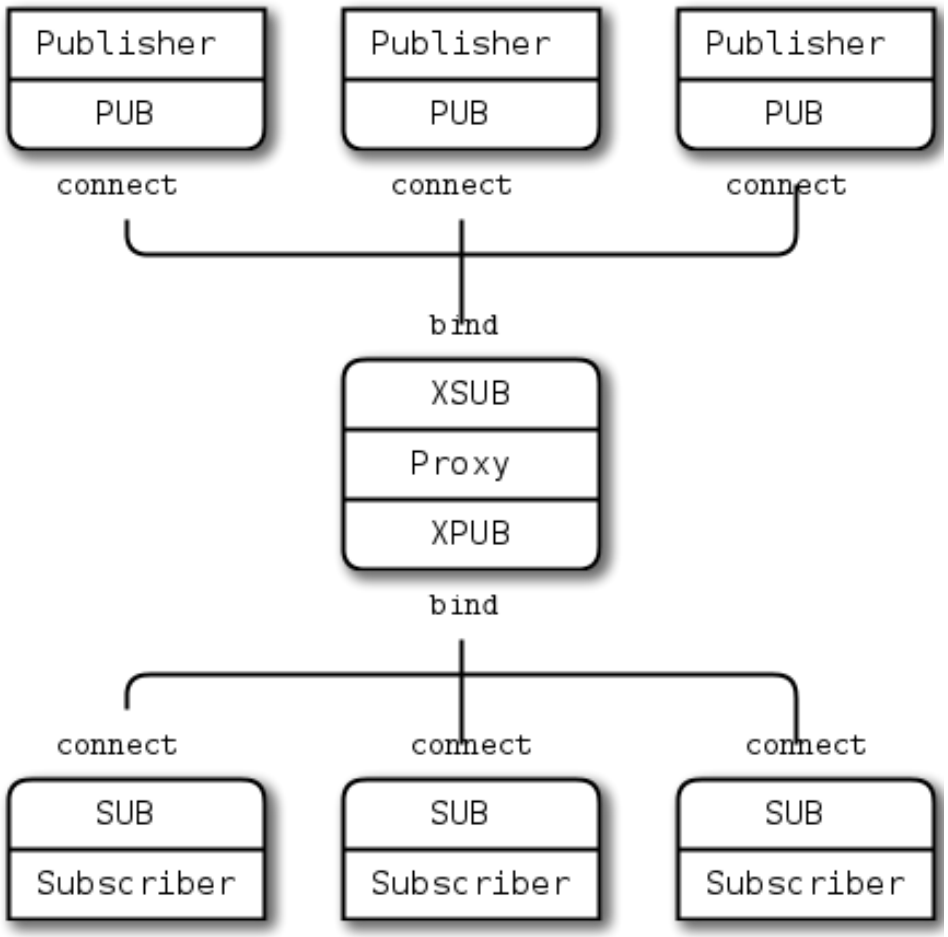


Kafka Performance



Nodes	Threads	Send msgs/s	Receive msgs/s	Processing latency	Send latency
1	1	2 391	2 391	48	48
1	5	9 917	9 917	48	48
1	25	20 982	20 982	46	48
2	1	4 957	4 957	47	
2	5	17 470	17 470	47	
2	25	41 902	41 901	45	48
4	1	9 149	9 149	47	
4	5	24 381	24 381	47	48
4	25	47 617	47 618	47	48
6	25	**54 494**	**54 494**	**47**	**48**
8	25	53 696	53 697	47	48

ZeroMQ



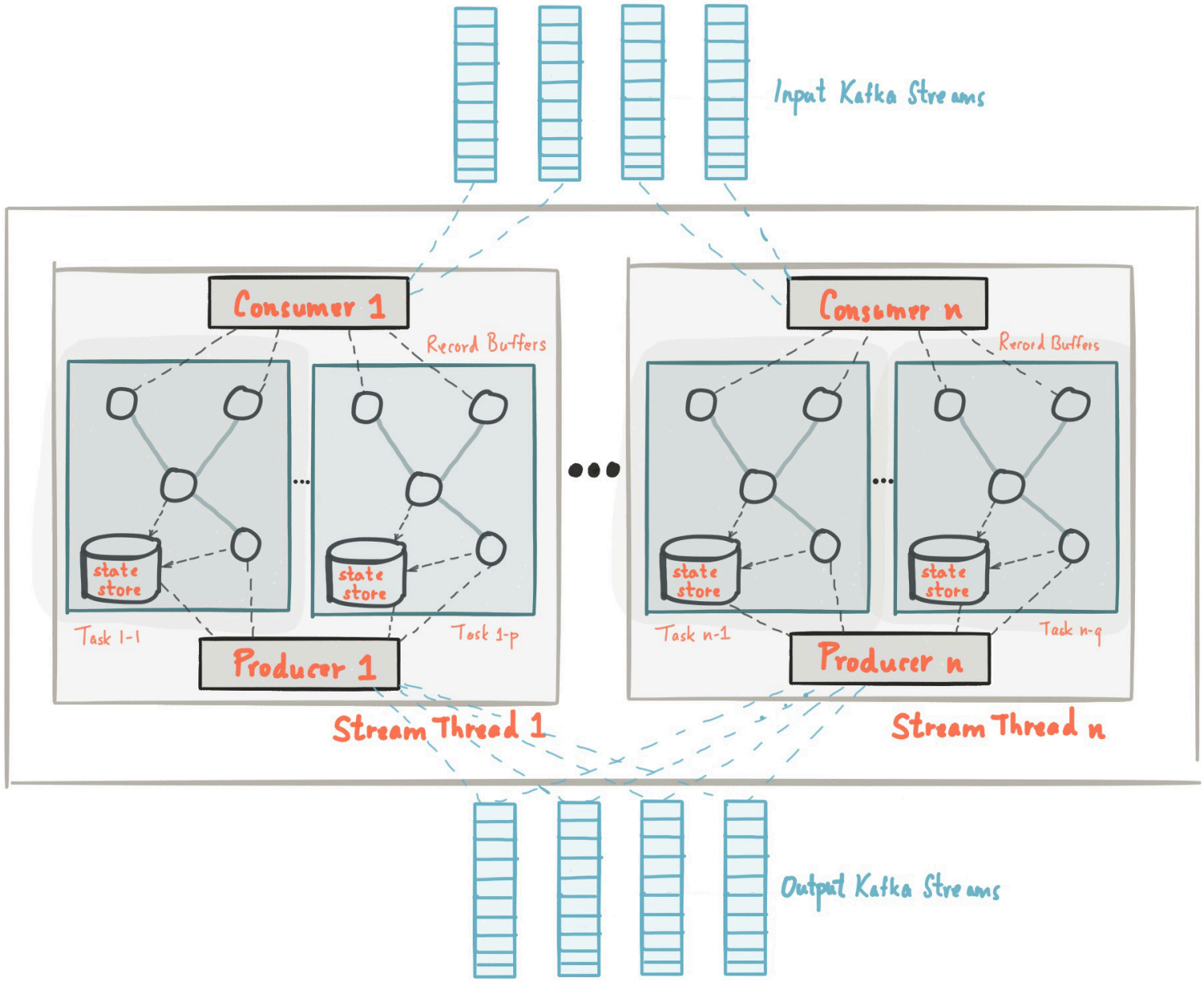
Streaming

- Continuous data flow
  - "Unbounded streams of data"
- Usually uses a message distribution system
  - JMS
  - Apache Kafka
  - MQTT
  - Etc
- An unbounded set of events with time
  - <t1, E1>, <t2, E2>, ..., <tn, En>, ...

Stream processing categorization

- Simple event processing
  - Working on an event at a time
    - e.g. filter out all events where the wind speed > 50 mph
- Event stream processing
  - Time-based processing of a single stream of events
    - Average wind speed over the last hour compared to the average over the last day
- Complex Event Processing
  - Correlation of events across different streams
    - Emergency calls correlated with wind speed in real time

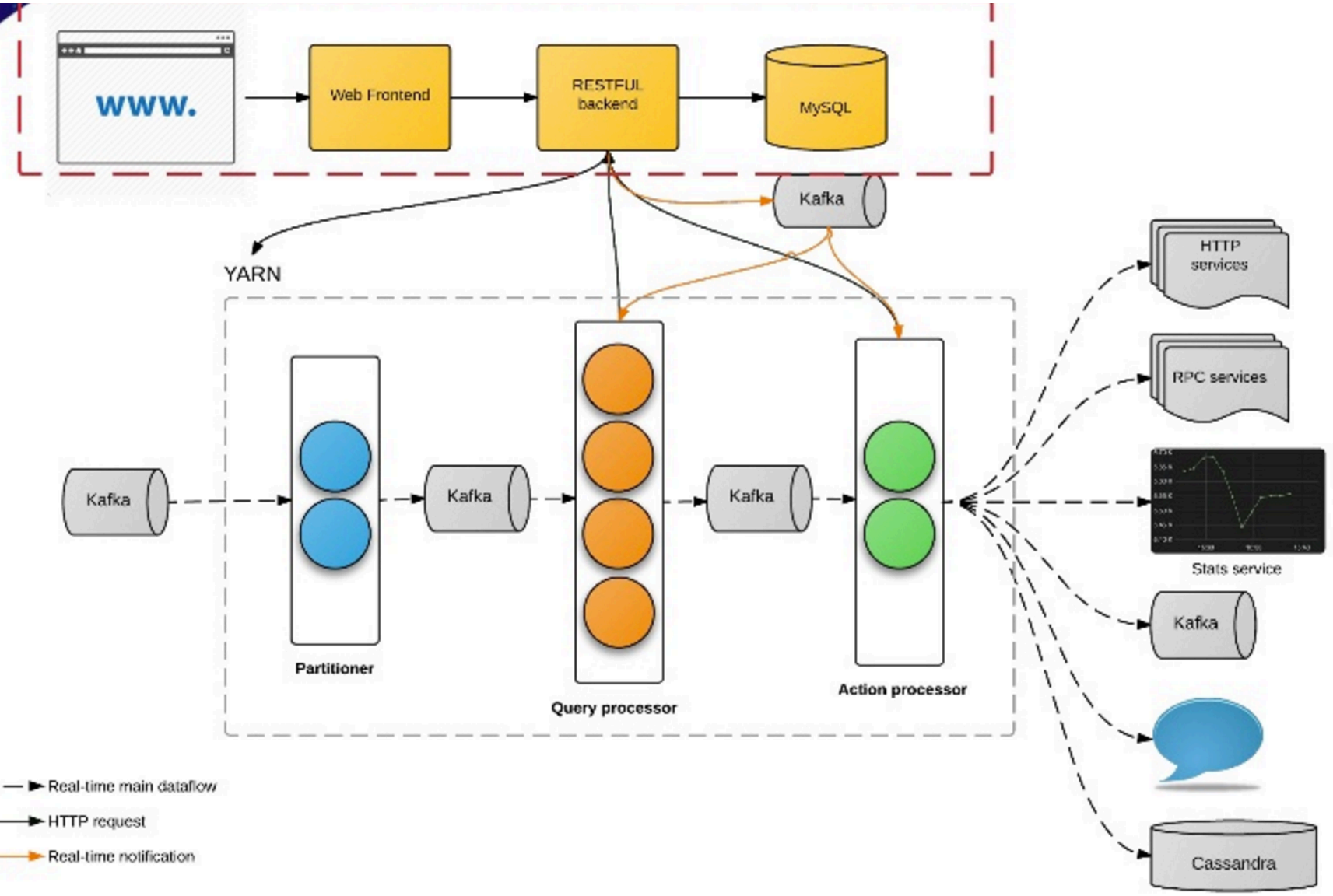
Kafka Streams



Kafka Streams

- Event-at-a-time processing (not microbatch) with millisecond latency
- Stateful processing including distributed joins and aggregations
- A convenient DSL
- Windowing with out-of-order data using a DataFlow-like model
- Distributed processing and fault-tolerance with fast failover
- Reprocessing capabilities so you can recalculate output when your code changes
- No-downtime rolling deployments

Siddhi at Uber



Siddhi at Uber

- 100+ production apps
- 30 billion messages / day
- Fraud, anomaly detection
- Marketing, promotion
- Monitoring, feedback
- Real time analytics and visualization

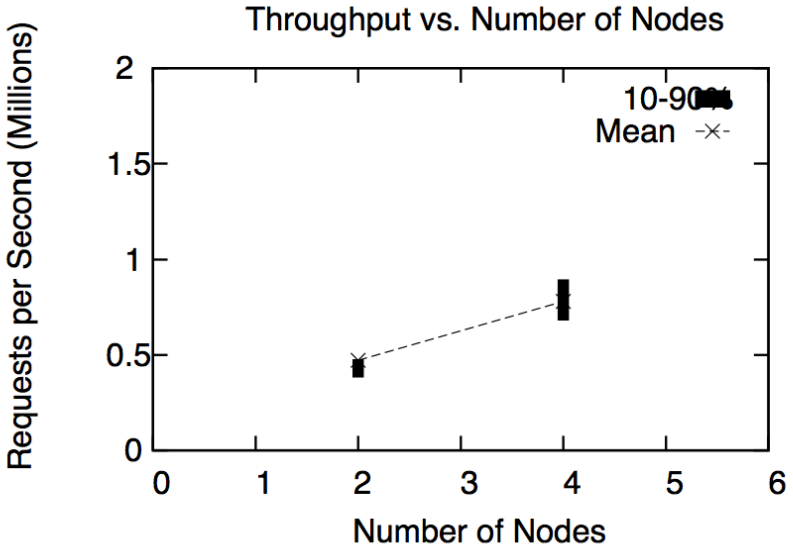
https://freo.me/siddhi-uber

Siddhi

- A stateful query model
- SQL-like language for querying streams of data
  - Extended with **windows**
    - Time, Event count, batches
  - Partitioned
    - Based on data in the events
  - Pattern matching
    - A then B then C within window

Siddhi

- Apache Licensed Open Source on Github
  - https://github.com/wso2/siddhi/
- Pluggable into Storm, Spark and Kafka Streams
- Supports millions of events/sec
- http://freo.me/DEBS_Siddhi



SiddhiQL

```
FROM login_stream#window.time(10 min)
SELECT ip,
       count(ip) as loginCount,
       cityId
GROUP BY ip
HAVING loginCount > 10
INSERT INTO login_attemp_repeatedly_stream;
```

Long-running aggregation

```
define aggregation SweetProductionAggregation
from SweetProductionStream
select name, sum(amount) as totalAmount
group by name
aggregate every hour...year
```

```
from GetTotalSweetProductionStream as b join SweetProductionAggregation as a
  on a.name == b.name
  within b.duration
  per b.interval
select a.AGG_TIMESTAMP, a.name, a.totalAmount
insert into HourlyProductionStream;
```

Streaming in Ballerina

```
function initRealtimeRequestCounter() {

    stream<RequestCount> requestCountStream;

    // Whenever the `requestCountStream` stream receives an event from the streaming rules defined in the `forever` block,
    // the `printRequestCount` function is invoked.
    requestCountStream.subscribe(printRequestCount);

    // Gather all the events coming in to the `requestStream` for five seconds, group them by the host, count the number
    // of requests per host, and check if the count is more than six. If yes, publish the output (host and the count) to
    // the `requestCountStream` stream as an alert. This `forever` block is executed once, when initializing the service.
    // The processing happens asynchronously each time the `requestStream` receives an event.
    forever {
        from requestStream
        window timeBatch(10000)
        select host, count(host) as count
        group by host
        having count > 6
        => (RequestCount[] counts) {
        // `counts` is the output of the streaming rules and is published to the `requestCountStream`.
        // The `select` clause should match the structure of the `RequestCount` record.
            requestCountStream.publish(counts);
        }
    }
}
```

Questions?