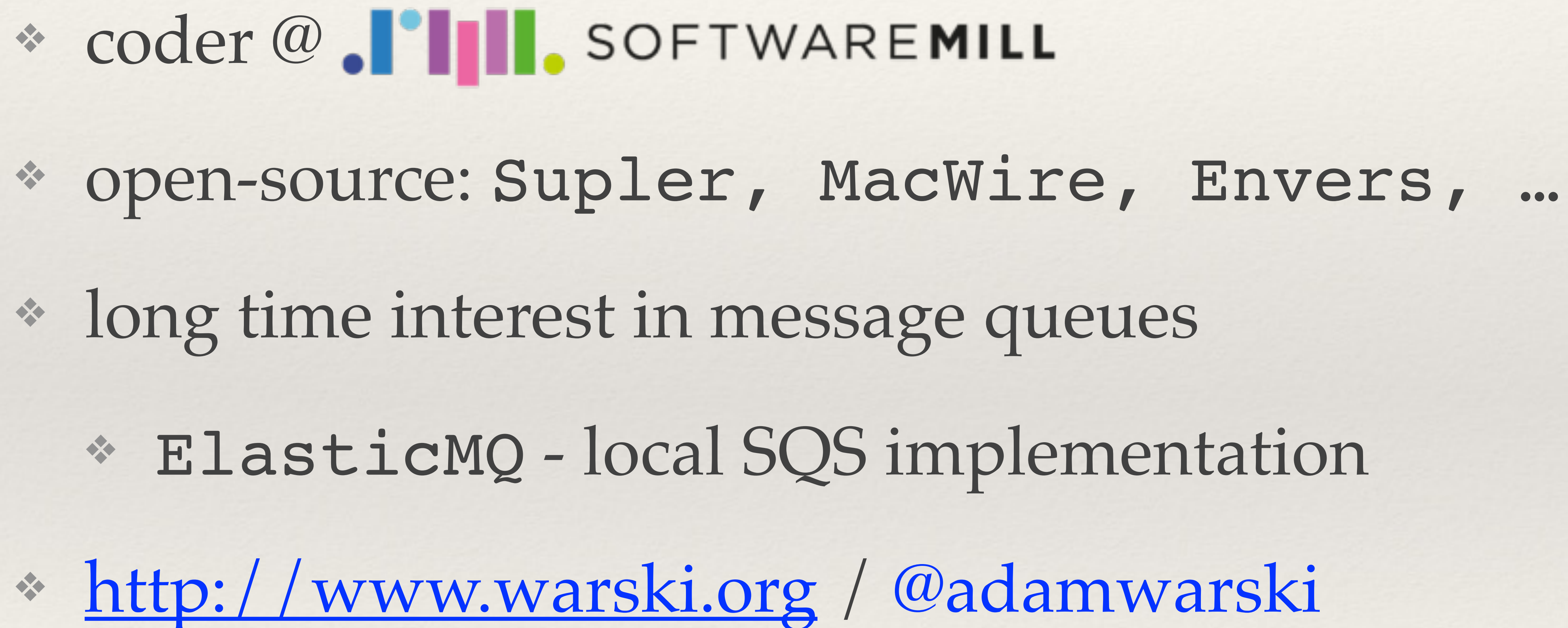
VOXXED DAYS VIENNA

Evaluating persistent, replicated message queues

Adam Warski
SoftwareMill

About me

- coder @ 

- open-source: `Supler, MacWire, Envers, …`
- long time interest in message queues
  - `ElasticMQ` - local SQS implementation
- http://www.warski.org / @adamwarski

Why message queues?

- Reactive Manifesto: message driven
- Microservices integration:
  - REST
  - MQ
- Any kind of asynchronous processing

reactivemanifesto.org

We Are Reactive

The Reactive Manifesto

Published on September 16 2014. (v2.0)

reactivemanifesto.org

Message Driven: Reactive Systems rely on asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation, location transparency, and provides the means to delegate errors as messages. Employing explicit message-passing enables load management, elasticity, and flow control by shaping and monitoring the message queues in the system and applying back-pressure when necessary. Location transparent messaging as a means of communication makes it possible for the management of failure to work with the same constructs and semantics across a cluster or within a single host. Non-blocking communication allows recipients to only consume resources while active, leading to less system overhead.

Responsive
Elastic
Resilient
Message Driven

Jobs? messages? tasks?

- Similar concepts:
  - message queue
  - job queue
  - asynchronous task

Exactly-once

- Everybody would like that
- Hard to achieve
  - needs distributed transactions
- Systems advertised as exactly-once are usually not

At-[least | most]-once

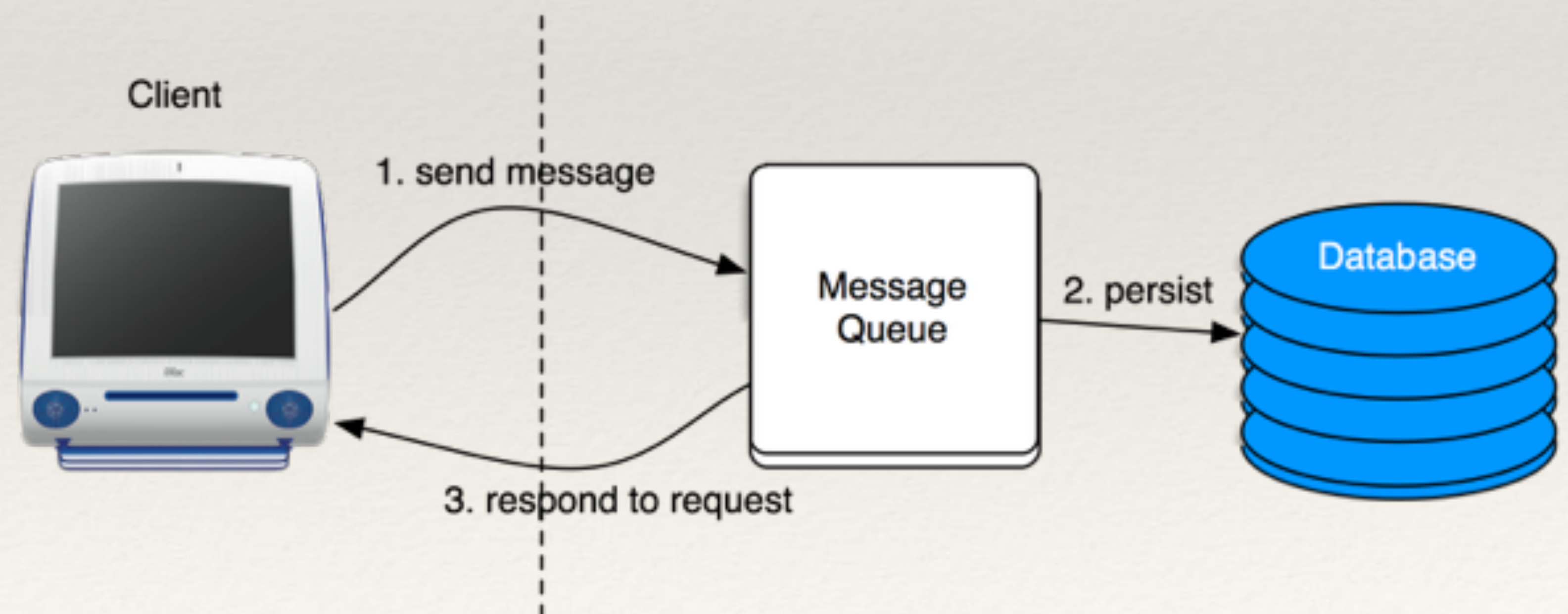
- "Almost exactly once"
- Least / most: tradeoffs
- Message acknowledgments
- Idempotent processing

Why persistent & replicated?

- Reactive manifesto: responsive, resilient
- We want to be sure no messages are lost
- Brings new problems
- But, “it depends”

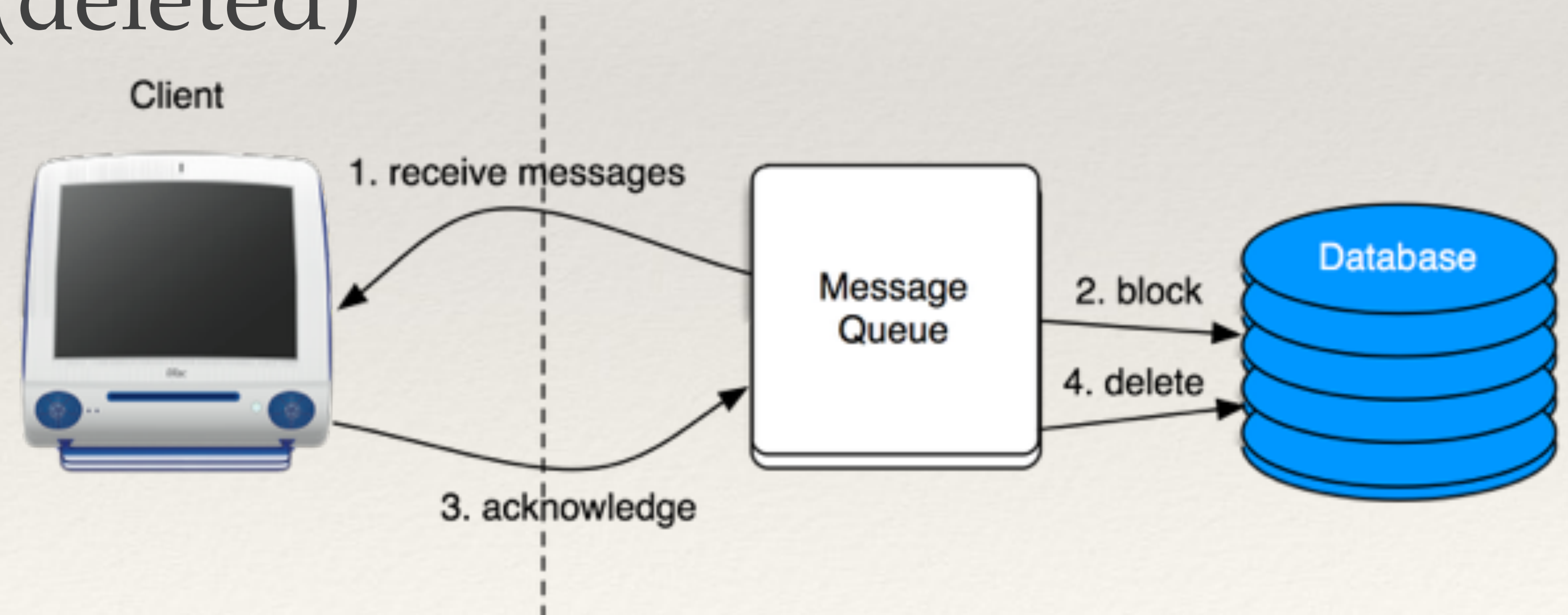
Scenario: send

- Client wants to send a message
- If the request completes, we want to *be sure* that the message will be eventually processed
- Making sure by:
  - writing to disk
  - replicating



Scenario: receive

- At-least-once-delivery
- Message is received from queue
- Processed
- And acknowledged (deleted)



Systems under test

- RabbitMQ
- HornetQ
- Kafka
- SQS
- MongoDB
- (EventStore)

What is measured

- **Number of messages per second** sent & received
- Msg size: **100 bytes**
- Other interesting metrics, not covered:
  - Send latency
  - Total msg processing time
  - Resource consumption at a given msg rate

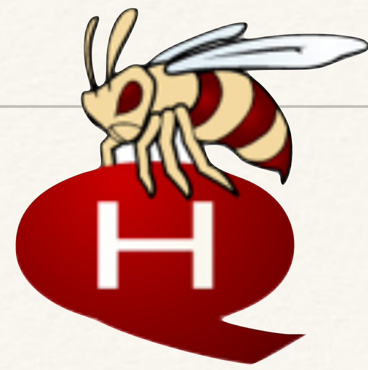
Testing methodology

- Message broker: **3** nodes
- **1-4** nodes sending, **1-4** nodes receiving
- Each sender / receiver node: **1-25** threads
- Each thread:
  - sending messages in batches, random size **1-10** (**1-100** / **1-1000**)
  - receiving messages in batches, acknowledging

@adamwarski

Servers

- Single EC2 availability zone
  - -> fast internal network
- m3.large
  - 2 CPUs
  - 7.5 GiB RAM
  - 32GB SSD storage



HornetQ

- RedHat/JBoss project
- multi-protocol, embeddable, high-performance, asynchronous messaging system
- JMS, STOMP, AMQP, native

HornetQ replication

- Live-backup pairs
- Data replicated to one node
- Fail-over:
  - manual, or
  - automatic, but: split-brain

HornetQ replication

- Once a transaction commits, it is written to the primary node’s journal
- Replication is asynchronous

HornetQ operations

- Send: transactions
- Receive:
  - one message at a time
  - blocking confirmations turned off

@adamwarski

HornetQ results

Threads	Nodes	Send msgs/s	Receive msgs/s
1	1	1 108	1 106
25	1	12 791	12 802
1	4	3 768	3 627
25	**4**	**17 402**	**16 160**

@adamwarski

HornetQ notes

- Poor documentation of replication guarantees
- Poor documentation on network failure behaviours
- Very high load: primary node considered dead even though working

- Leading open-source messaging system
- AMQP
- Very rich messaging options

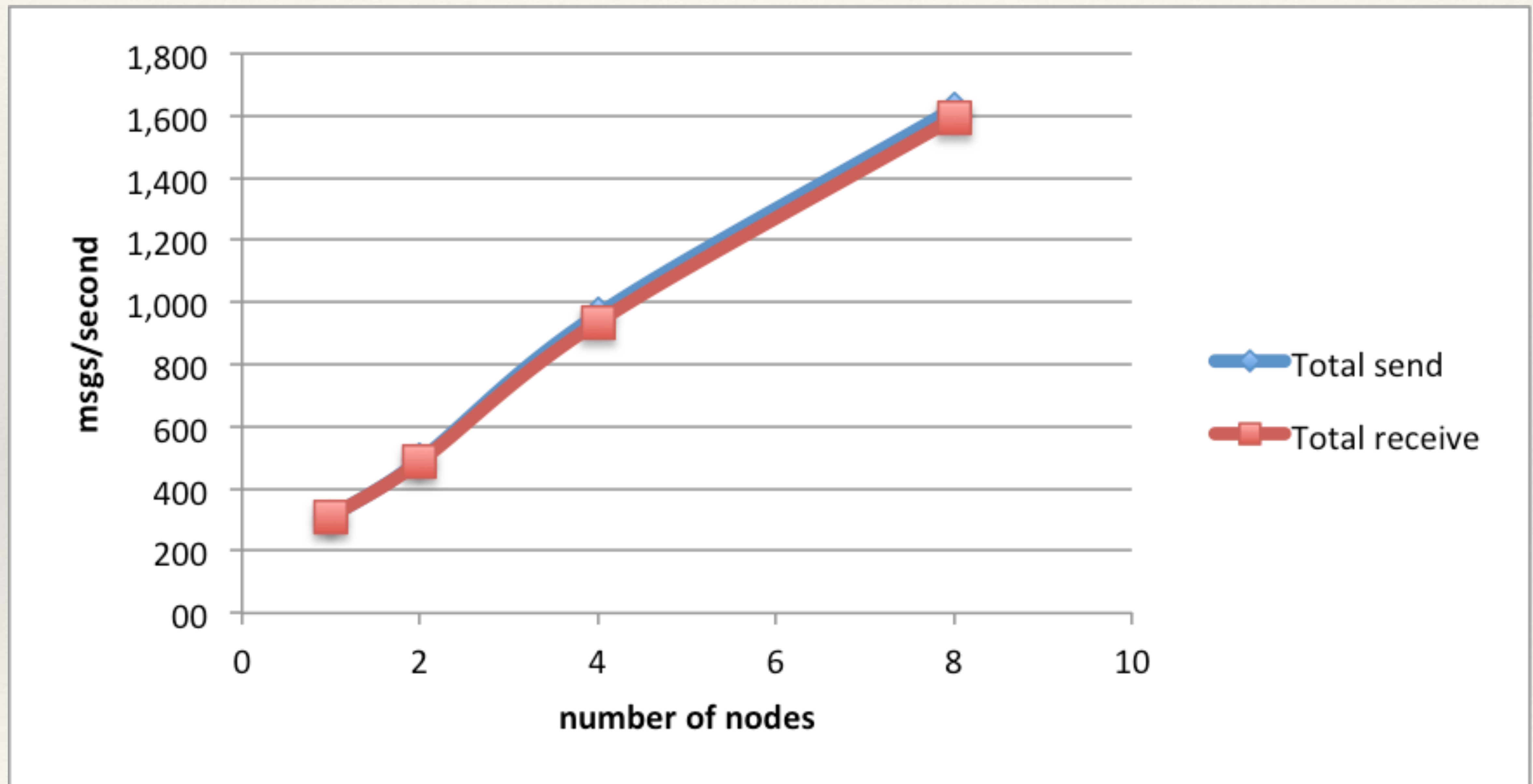
RabbitMQ replication

- 3 nodes
- Using publisher acknowledgments
  - AMQP extension
  - cluster-wide
- Does not cope well with network partitions
  - documented!

RabbitMQ operations

- Sending a batch, waiting for confirmations
- Receiving batch, acknowledging one-by-one
- Redelivery: connection broken

RabbitMQ results



@adamwarski

RabbitMQ results

Threads	Nodes	Send msgs/s	Receive msgs/s
1	1	1 829	1 811
1	4	**3 158**	**3 124**

Batch 100

Threads	Nodes	Send msgs/s	Receive msgs/s
1	1	3 181	2 549
1	4	**3 566**	**3 533**

Batch 1000

RabbitMQ notes

- Publisher confirms seems to be killing it
- Documented network partition behaviour
- Shovel/Federation plugins

SQS

- As-a-service
- Part of Amazon's Web Services
- Simple interface
- Priced basing on load
- Easy to set up

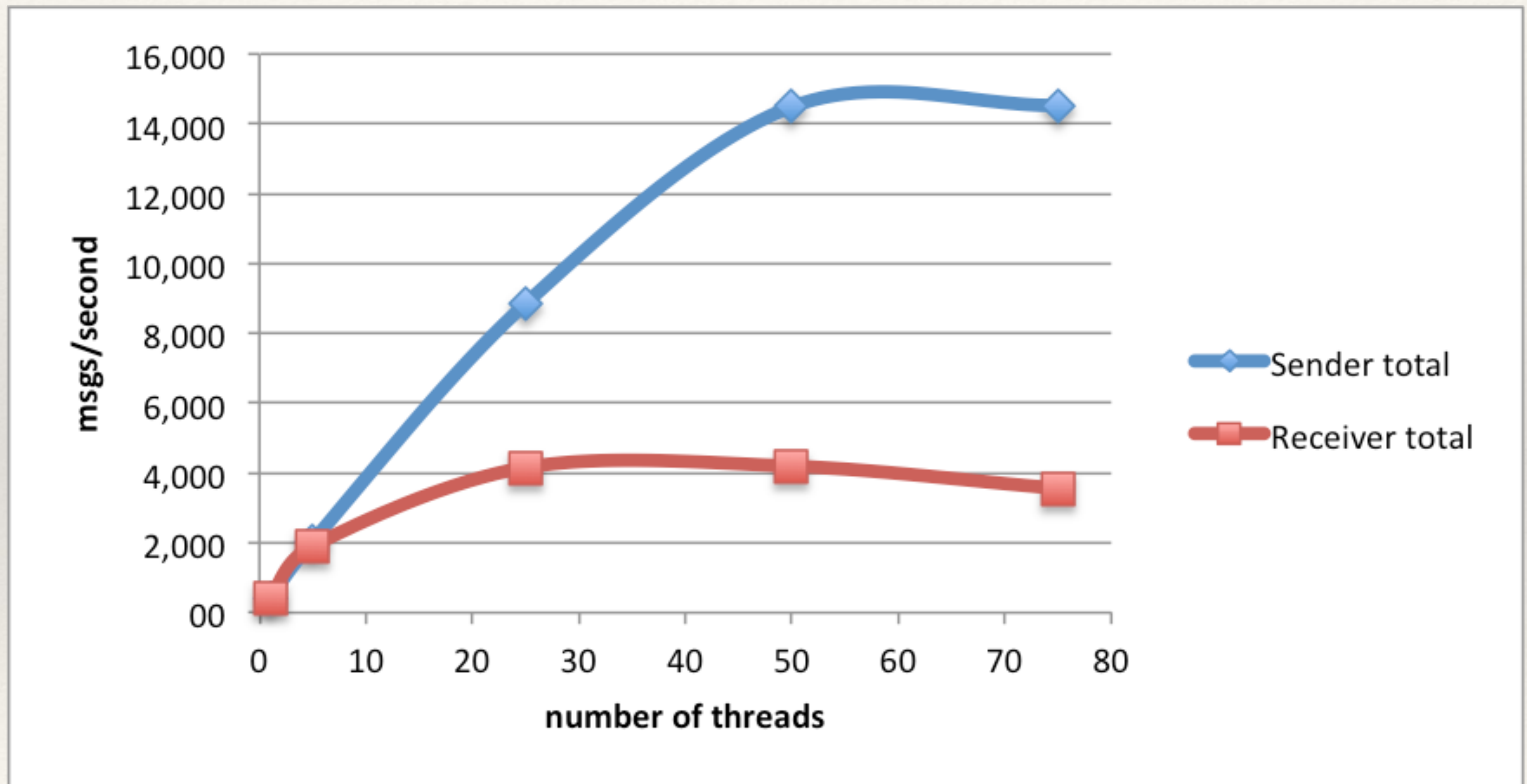
SQS replication

- We don’t really know ;)
- If a send completes, the message is replicated to multiple nodes
- Unfair competition: might use multiple replicated clusters with routing/load-balancing clients

SQS operations

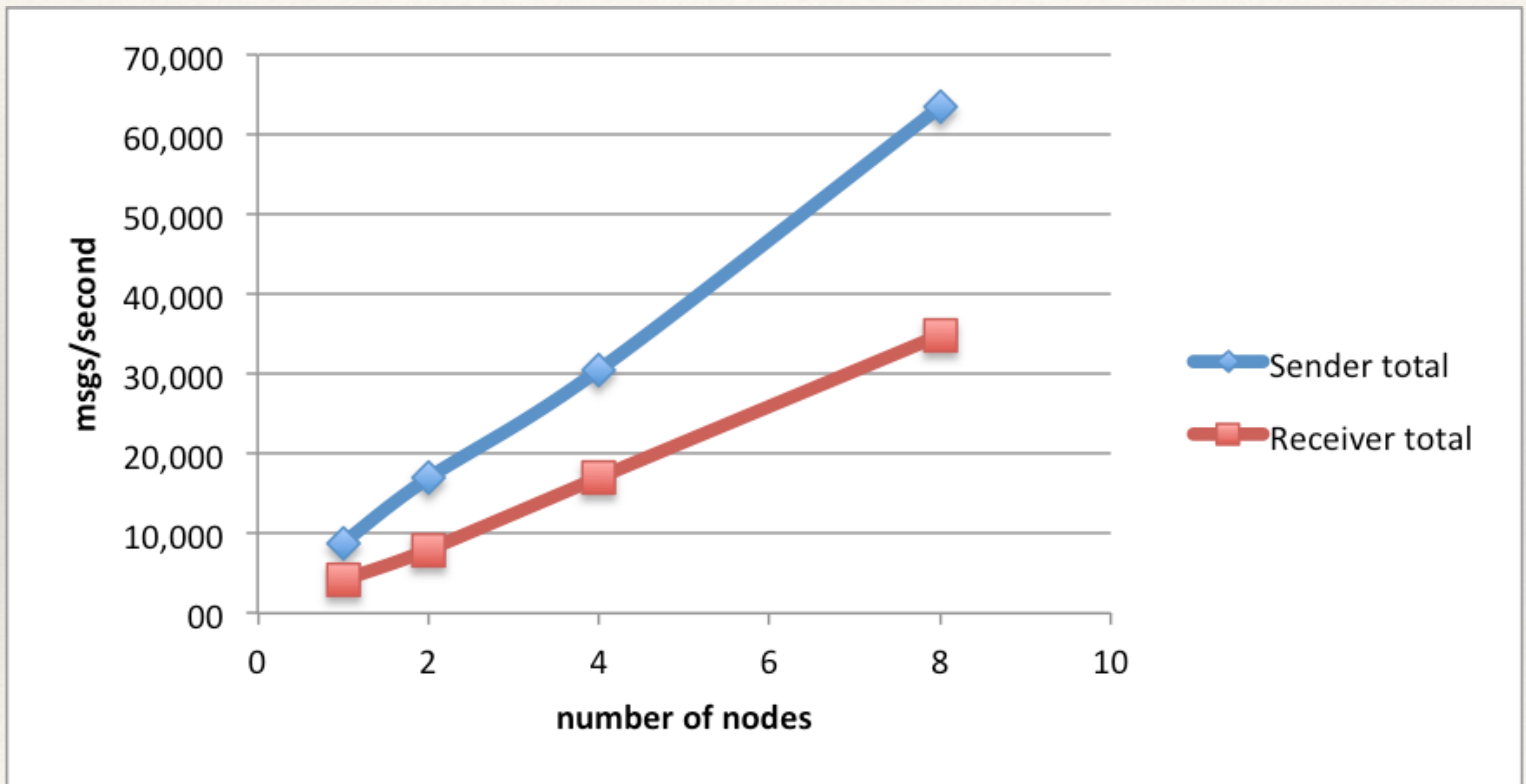
- Sending messages in batches
- Receiving messages in batches (long polling).
- Redelivery: after timeout (message blocked for some time)
- Deleting (acknowledging) in batches

SQS results



@adamwarski

SQS results



SQS notes

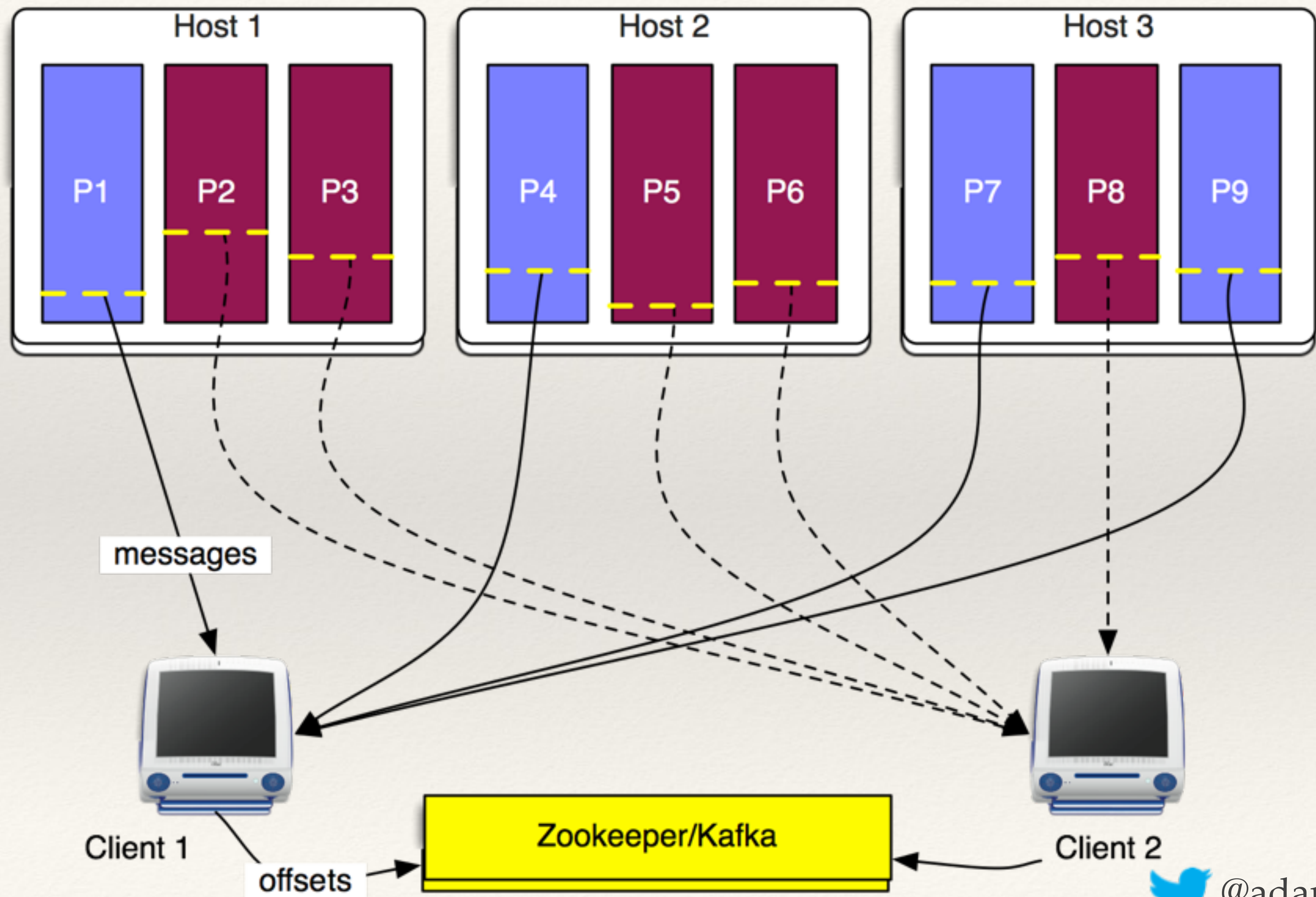
- Can re-deliver even if no failure in the client
  - failure in SQS

- Different approach to messaging
- Streaming publish-subscribe system
- Topics with multiple partitions
    - more partitions -> more concurrency

Point-to-point messaging in Kafka

- Messages in each partition are processed in-order
- Consumers should consume at the same speed
- Messages can't be selectively acknowledged, only "up to offset"
- No "advanced" messaging options

Point-to-point messaging in Kafka



@adamwarski

Kafka replication

- Multiple nodes (here: 3)
- Replication factor (here: 3)
- Uses Zookeeper for coordination

Kafka operations

- Send: blocks until accepted by partition leader, no guarantees for replication
- Consumer offsets: committed every 10 seconds manually; during that time, message receiving is blocked
- Redelivery: starting from last known stream position

@adamwarski

Kafka results

Threads	Nodes	Send msgs/s	Receive msgs/s
1	1	2 558	2 561
25	1	29 691	27 093
25	4	**33 587**	**31 891**

@adamwarski

Kafka notes

- Scaling potential:
  - adding more nodes
  - increasing number of partitions

mongoDB

- Not really a queue - I know ;)
- Very simple replication setup
- Document-level atomic operations: find-and-modify

Mongo replication

- 3 nodes
- Controllable guarantees:
  - WriteConcern.ACKNOWLEDGED
  - WriteConcern.REPLICA_ACKNOWLEDGED (majority)

Mongo operations

- Sending: in batches, waiting until the DB write completes
- Receiving: find-and-modify, one-by-one
- Redelivery: after timeout (message blocked for some time)
- Deleting: in batches, DB delete

Mongo results

Threads	Nodes	Send msgs/s	Receive msgs/s
1	1	7 968	1 914
25	1	**10 903**	**3 266**

"Safe"

Threads	Nodes	Send msgs/s	Receive msgs/s
1	1	1 489	1 483
25	2	**6 550**	**2 841**

"Replica safe"

@adamwarski

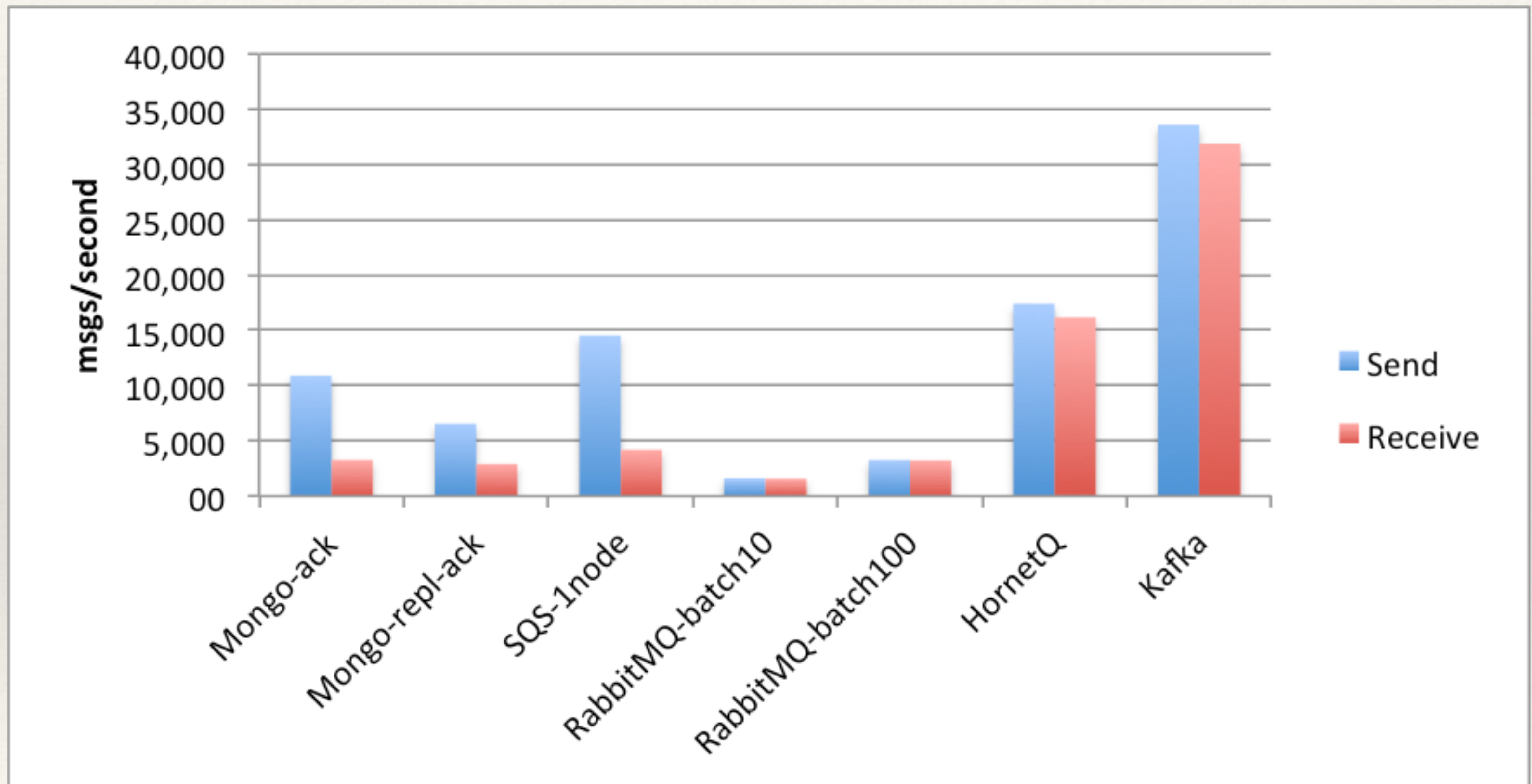
EVENT STORE

- Primary use-case: event sourcing
- Competing consumers: servers keeps track
- Hybrid acknowledgment model:
  - selective
  - with checkpoints
- Message time-outs

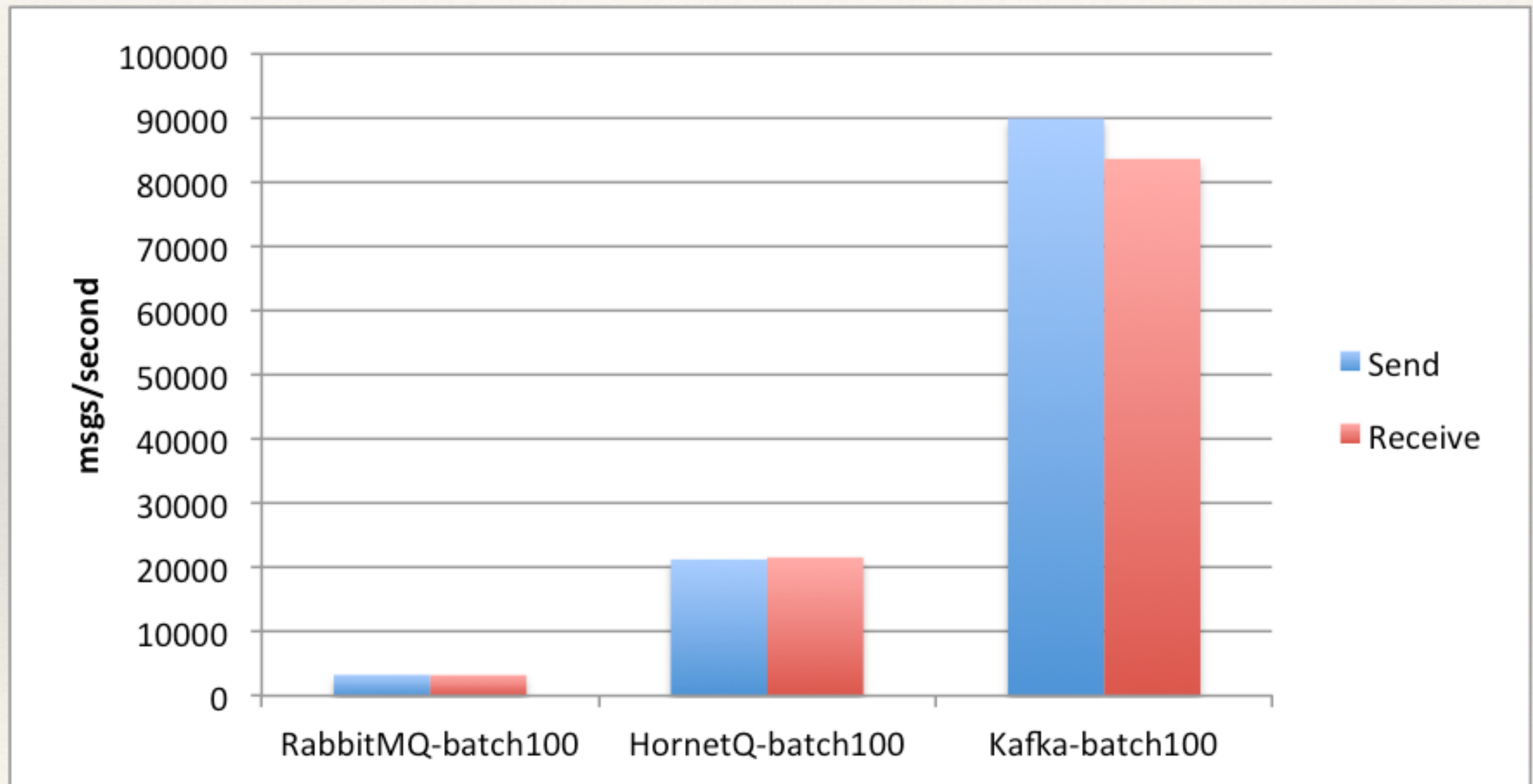
Summing up

- **SQS**: good performance, easy setup
- **Mongo**: no need to maintain separate system
- **RabbitMQ**: rich messaging options, good persistence
- **HornetQ**: good performance, many interfaces
- **Kafka**: best performance and scalability

Summary - batch 10



Summary - batch 100



@adamwarski

Thanks!

- **Questions?**

Scalar
11/04/2015

@adamwarski