NOKIA

Kafka versus RabbitMQ

A comparative study of two industry reference publish/subscribe implementations

Philippe Dobbelaere, Kyumars Sheykh Esmaili

Nokia Bell Labs Antwerp, Belgium

- **RabbitMQ and Kafka architecture**
- broker KPIs
- experimental validation
- use cases, determination table

Scope

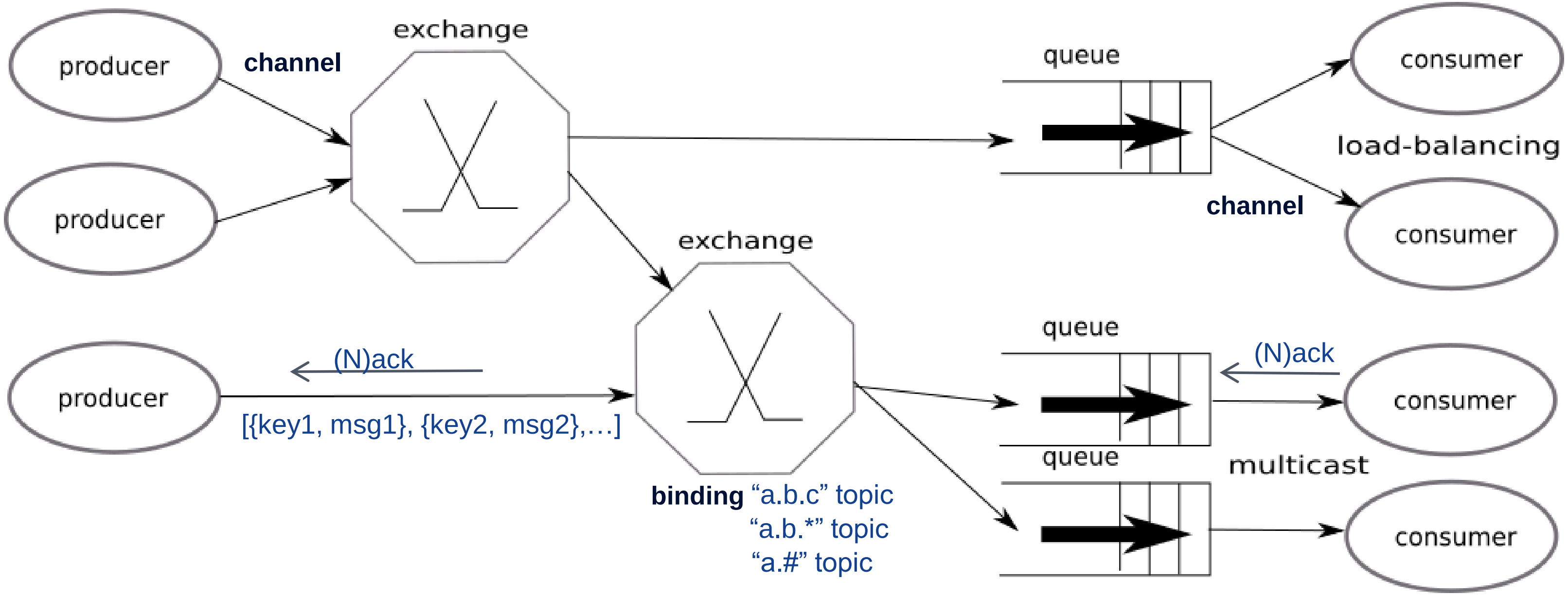
searching a broker function for our WorldWideStreams IoT platform

- pub/sub paradigm for scalability and loose coupling in distributed systems
  - **decoupling** of
    - **Entities**: publishers and consumers do not need to be aware of each other.
    - **Time**: interacting parties do not need to be actively participating in the interaction, or even stronger, switched on, at the same time.
    - **Synchronization**: asynchronous interaction between producers/consumers and broker, allowing maximum usage of processor resources at producers and consumers alike
  - **routing** logic (a.k.a subscription model)
    decides if and where a packet from a producer will end up at a consumer
- Kafka ecosystem extends beyond broker
  maybe the extra functionality fit's your platform requirements, maybe it doesn't.
- RabbitMQ is just the broker.

RabbitMQ
design principles

- AMQP - Advanced Message Queuing Protocol
  - a protocol for asynchronous pub/sub messaging
    stringent performance, scalability and reliability requirements from the finance community.
- RabbitMQ goes beyond AMQP
  - batching efficiency and transactional capabilities highly increased
    by more flexible publisher acknowledge
  - flow control mechanism to enhance system stability
  - queue insertion / extraction code is optimised for small queues in DRAM
    assumption is that consumers can follow the production rate
- RabbitMQ builds on top of Erlang/OpenTelecomPlatform
  exploiting the erlang actor model for IPC and OTP HA features

RabbitMQ architectural components

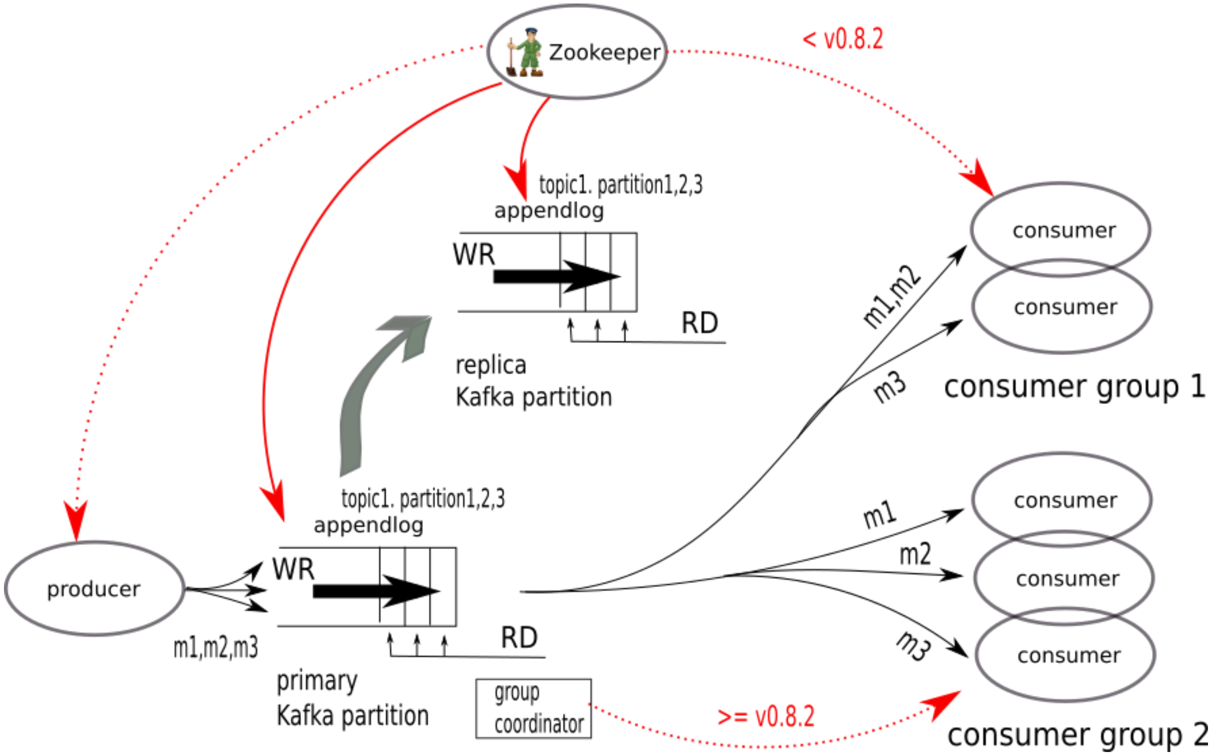


NOKIA

Kafka design principles

- Kafka tries to approximate a linearly accessed disk based append log
  - optimisations:
    - message batching
    - OS page cache, linear write disk access, cached read or mainly linear read disk access
- The Kafka design only covers half of what a pub/sub system typically covers - the other half is implemented in consumer libraries.
- Kafka does not remove messages on read but with a cleanup process. Consumers can easily replay messages based on the position pointer. Cleanup is triggered by time or log size.
- Kafka relies on Zookeeper for state management - later versions use Zookeeper only for less time critical tasks

Kafka architectural components



- RabbitMQ and Kafka architecture
- **broker Key Performance Indicators**
- experimental validation
- use cases, determination table

broker KPIs - correctness, availibility, transactions

▪correctness

	at most once	at least once
no order	-	single RabbitMQ node, fsync per message single Kafka node, fsync per message on demand at the expense of throughput. cluster of Kafka nodes can avoid fsync at the expense of quorum+network latency.
partitioned order	fastest mode for RabbitMQ (producer channel scope) and Kafka (partition scope)	RabbitMQ does reordering internally Kafka producers can only have a single produce request outstanding to conserve inter-batch ordering, which will impact throughput even more

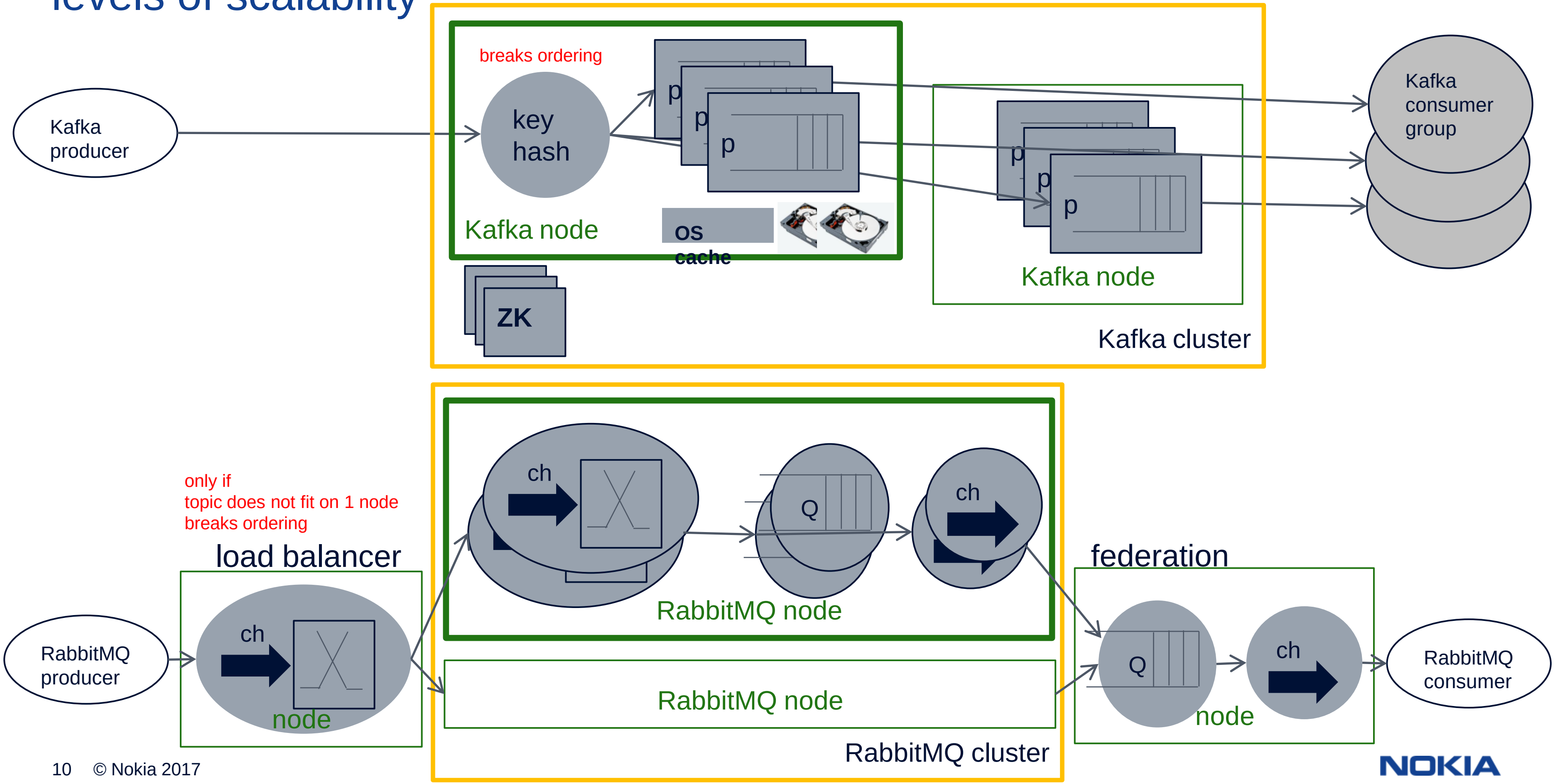
▪availibility

- RMQ = clusters to replicate configuration + mirrored queues to replicate messages
- Kafka = clusters + replication requirements on Zookeeper

▪transactions

- AMQP transactions are not very interesting from a performance point of view
- RabbitMQ has improved transactional behaviour (reject on batches)
  but atomicity is *not guaranteed* when a node crashes and restarts
- Kafka has transactions on the roadmap

levels of scalability



other features

- Kafka
  - long term message storage
    - but: no way to survive the timeout!
  - message replay
  - log compaction
    - for change feeds that are expressed as updates to keys,
      Kafka can retain only the last update ,
      for all the keys

- RabbitMQ
  - STOMP, MQTT
  - federated exchange, shovel
  - diskless use
  - time-to-live
  - builtin management and monitoring

- RabbitMQ and Kafka architecture
- broker KPIs
- **experimental validation**
- use cases, determination table

NOKIA

experimental setup

- Linux server
  - 24 cores (Intel Xeon X5660 @ 2.80GHz) and 12GB of DRAM running a 3.11 kernel.
  - hard disk was a WD1003FBYX-01Y7B0 running at 7200 rpm.
- tooling up
  - every single experiment logged
  - memory and cycle consumption recorded
  - warm-up time, then stats taken
  - column based DB (CSV)
    - easy "upgrade" of results when testbench gets more complex
  - homebrewn data browser
  - gnuplot backend

latency comparison

RabbitMQ latency results are optimal if the broker is allowed to have **outstanding unconfirmed publishes**

When RabbitMQ is running close to maximum load (an exceptional setting), the broker will start to write packets to disk to free up memory it needs for computation, effectively meaning the latency figures will rapidly deteriorate

In case of Kafka, when consumers are slow (here: 30%), packets will have to be transferred from disk to cache before a read completes, which will significantly impact latency

Kafka introduces more uncertainty (P99.9 but no max...)

	mean	max
with and without replication	1–4 ms	2–17 ms

(a) RabbitMQ

	50 percentile	99.9 percentile
without replication	1 ms	15 ms
with replication	1 ms	30 ms

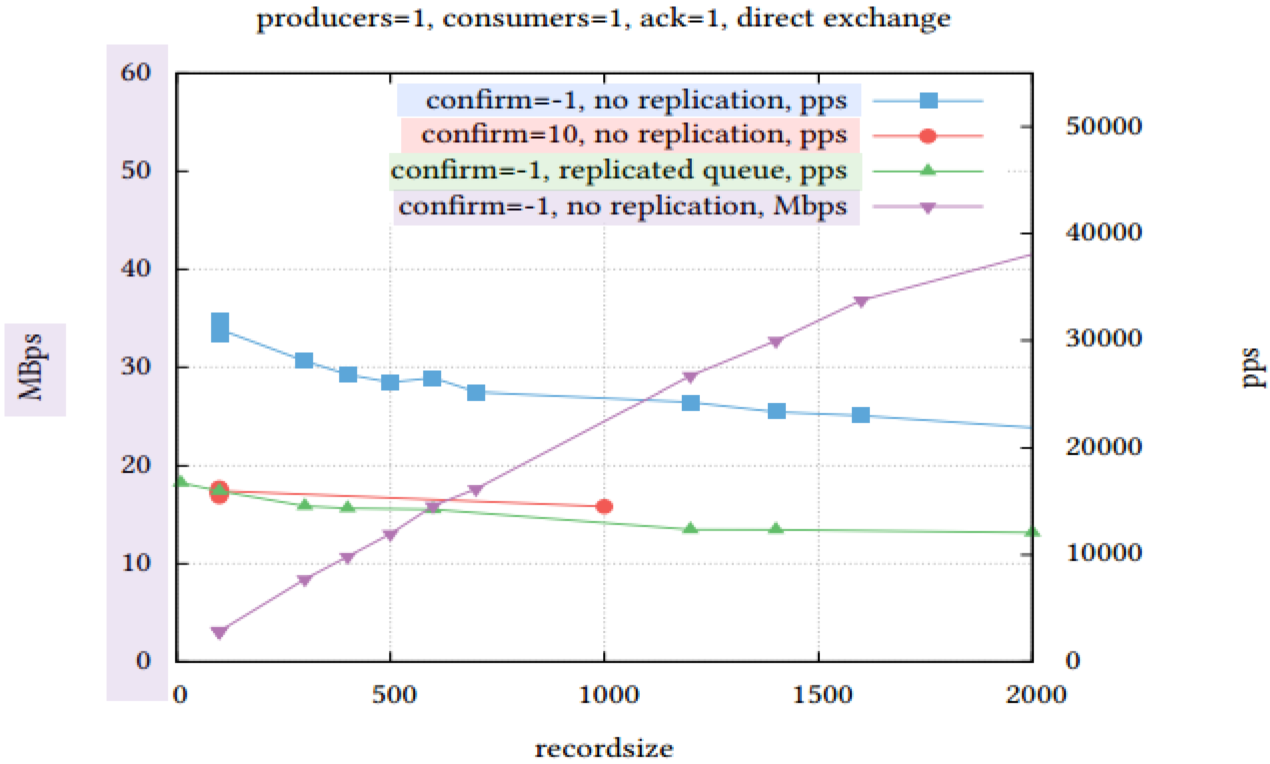
(b) Kafka

RabbitMQ throughput in MegaBytesPerSecond or PacketsPerSecond impact of configuration/message characteristics

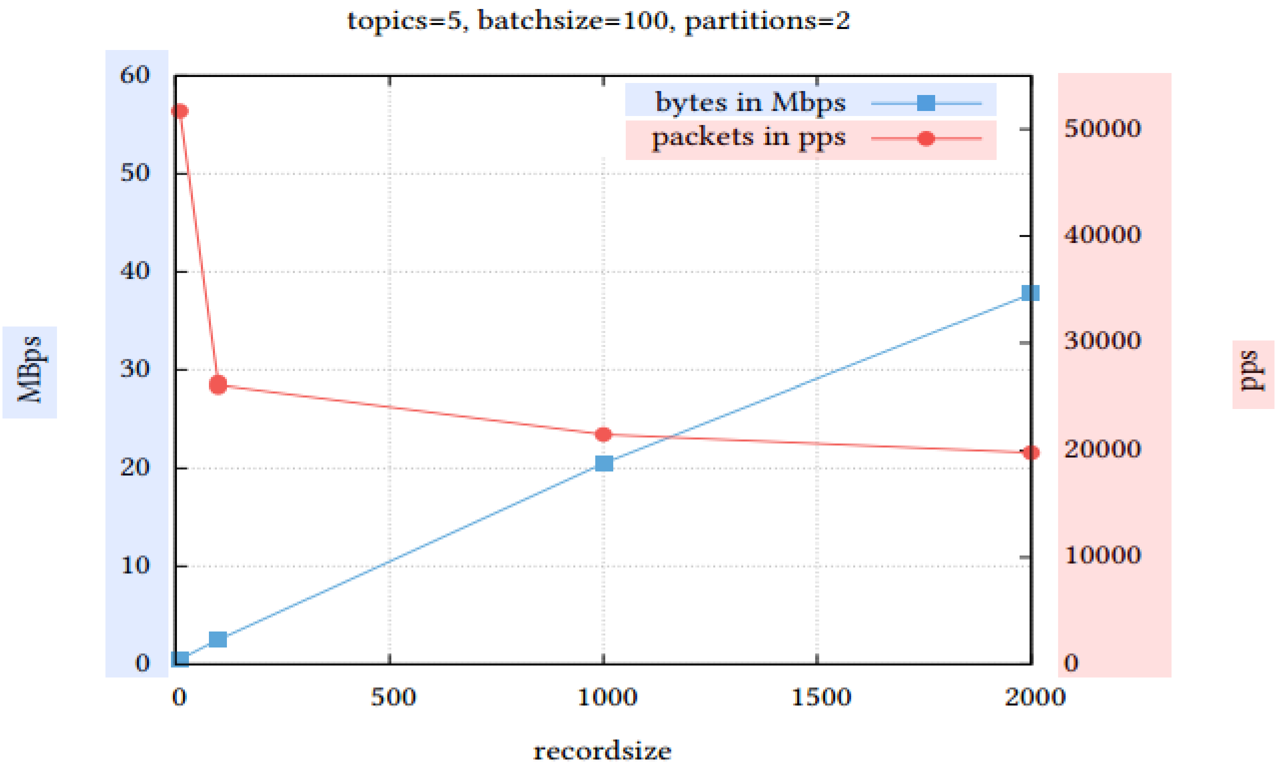
optimal if the broker is configured to allow an unlimited number of unconfirmed publishes (confirm == -1)

(confirm == 10) → 50% drop

replication → 50% drop



Kafka throughput in MegaBytesPerSecond or PacketsPerSecond impact of recordsize



NOKIA

Kafka throughput

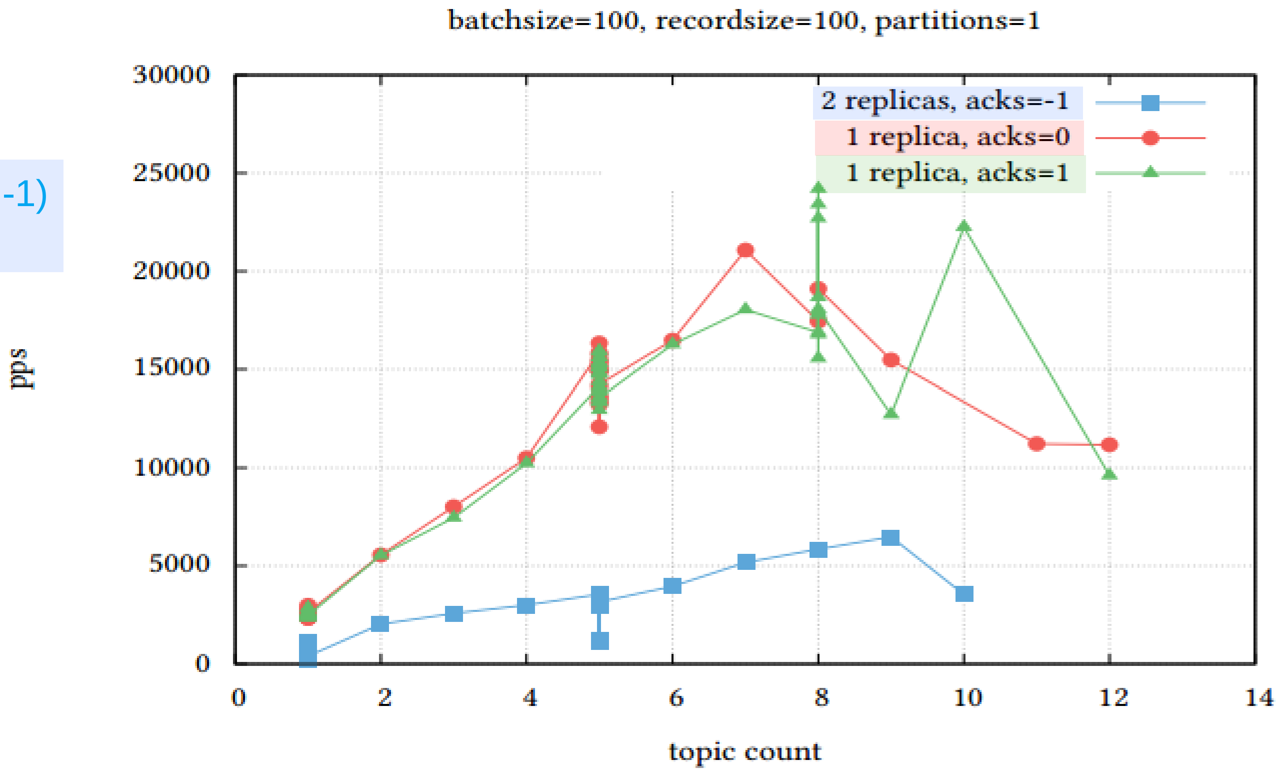
impact of topic count / replication per node (fixed recordsize)

at least once mode (acks == -1)
→ 50% to 75% drop

compared to the

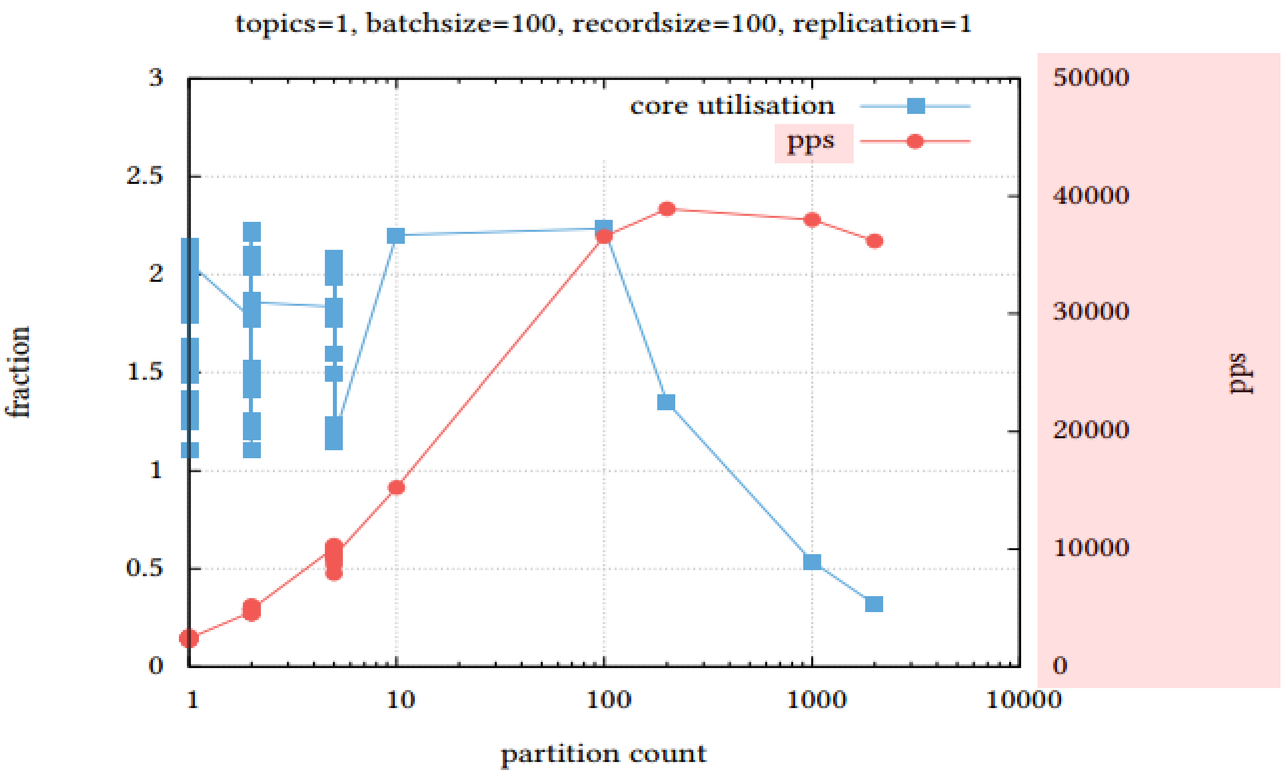
best effort scenario
(acks == 0).

optimal topic count per single node



Kafka throughput

impact of partition count per node (fixed recordsize)



optimal partition count per single node

NOKIA

throughput comparison
RabbitMQ versus Kafka - single node

RabbitMQ is mainly constrained by **routing complexity** (up till frame sizes of a few 1000 bytes, at which time packet copying becomes non-negligible)

$$\frac{|producers|}{U_{routing}+size*U_{byte}}$$

(a)

	$U_{routing}$	U_{byte}	Mean Error
no replication	$3.24e-5$	$7.64e-9$	3%
replicated queue	$6.52e-5$	$8.13e-9$	4.5%

erlang actors
are predictable

it is more appropriate to express Kafka throughput in **bytes**, since Ubyte is dominant even for small frames.

$$\frac{|producers|*|partitions|}{U_{routing}+|topics|*U_{topics}+effective\_size^{0.5}*U_{byte}}$$

(a)

	$U_{routing}$	U_{topics}	U_{byte}	Mean Error
acks = 0, rep. = 0	$3.8e-4$	$2.1e-7$	$4.9e-6$	30%
acks = 1, rep. = 0	$3.9e-4$	$9.1e-8$	$1.1e-6$	30%
acks = -1, rep. = 2	$9.4e-4$	$7.3e-5$	$2.9e-5$	45%

JVM GC?
OS cache

NOKIA

- RabbitMQ and Kafka architecture
- broker KPIs
- experimental validation
- **use cases, determination table**

NOKIA

use case overview

Kafka	RabbitMQ
pub/sub with XXL throughput per topic	pub/sub with complex routing
enterprise data layer infrastructure (batch and realtime)	
operational metrics tracking with offline processors	operational metrics tracking with complex filters on realtime streams
change feed dispatcher	
ingestion system for platforms such as Spark, Fink (Samza)	
	RPC dispatcher
	transport solution of an IoT PaaS offer

combinations?

- Kafka → RabbitMQ
  - global throughput XL, throughput per topic within RabbitMQ capabilities
- RabbitMQ → Kafka
  - adding long term storage to RabbitMQ solution
- Kafka || RabbitMQ
  - legacy integration

NOKIA

use case determination table - some highlights

Kafka Streams?

predictable latency?	complex routing?	long term storage?	very large throughput per topic?	packet order important?	dynamic elasticity behavior?	system throughput?	at least once?	high availability?	
N	N	*	*	N	N	XL	N	N	Kafka with multiple partitions
N	N	*	*	N	N	XL	Y	Y	Kafka with replication and multiple partitions
N	N	*	*	Y	N	L	N	N	single partition Kafka
N	N	*	*	Y	N	L	Y	Y	single partition Kafka with replication
*	*	N	N	*	*	L	*	N	RabbitMQ
*	*	N	N	*	*	L	*	Y	RabbitMQ with queue replication
*	*	Y	N	*	*	L	*	*	RabbitMQ with Kafka long term storage
N	Y	*	*	N	N	XL	N	*	Kafka with selected RabbitMQ routing

[1] Y - feature required, N - feature not required, * - wildcard, replaces two rows that are identical but in this feature, one with Y and one with N

[2] L(arge), (e)X(tra)L(arge), see 5.2 for some more quantitative throughput figures

NOKIA

NOKIA