Mahidol University
Faculty of Information
and Communication Technology

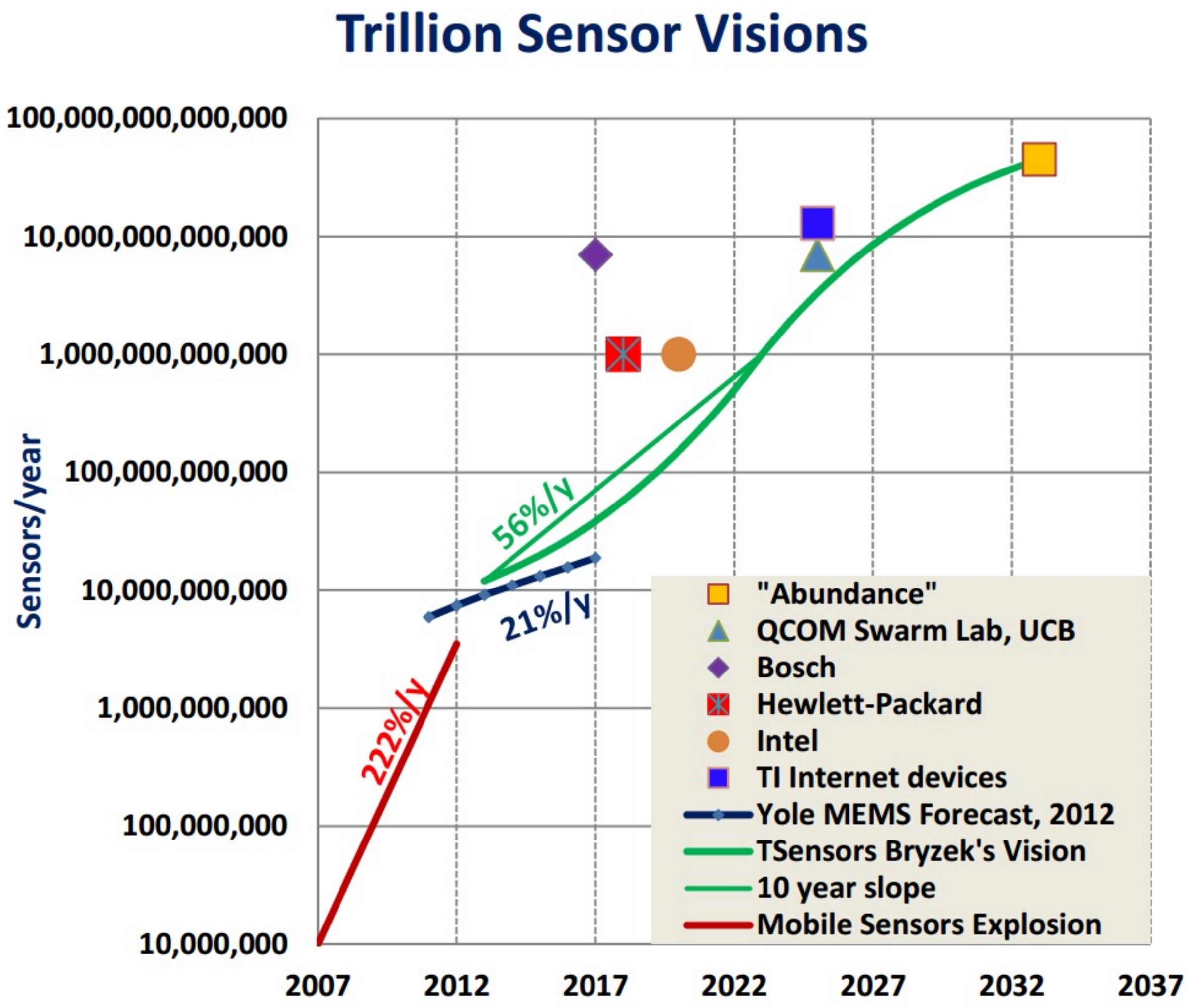
AIST
NATIONAL INSTITUTE OF
ADVANCED INDUSTRIAL SCIENCE AND TECHNOLOGY (AIST)

muMQ
A lightweight and scalable MQTT broker

Wiriyang Pipatsakulroj[1], Vasaka Visoottiviseth[1], Ryousei Takano[2]

[1]Mahidol University [2]AIST

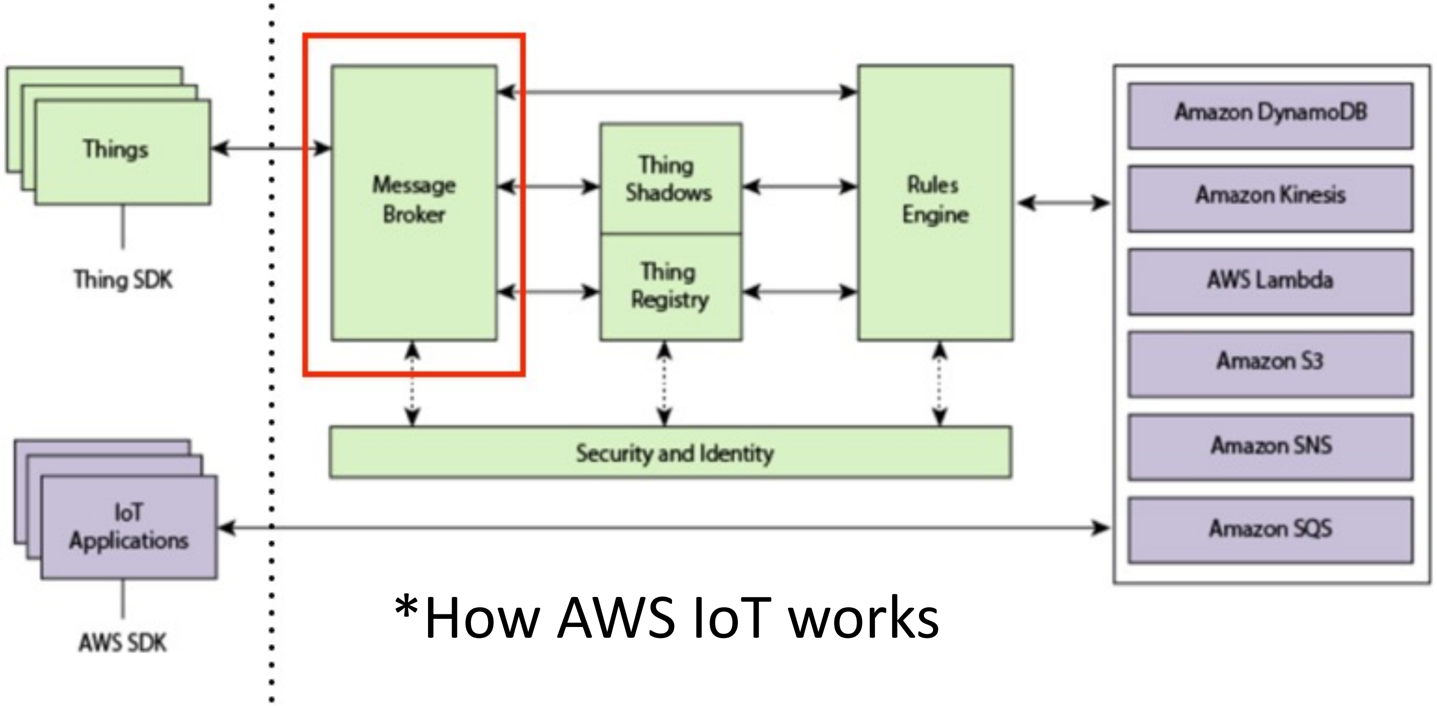
IoT growth



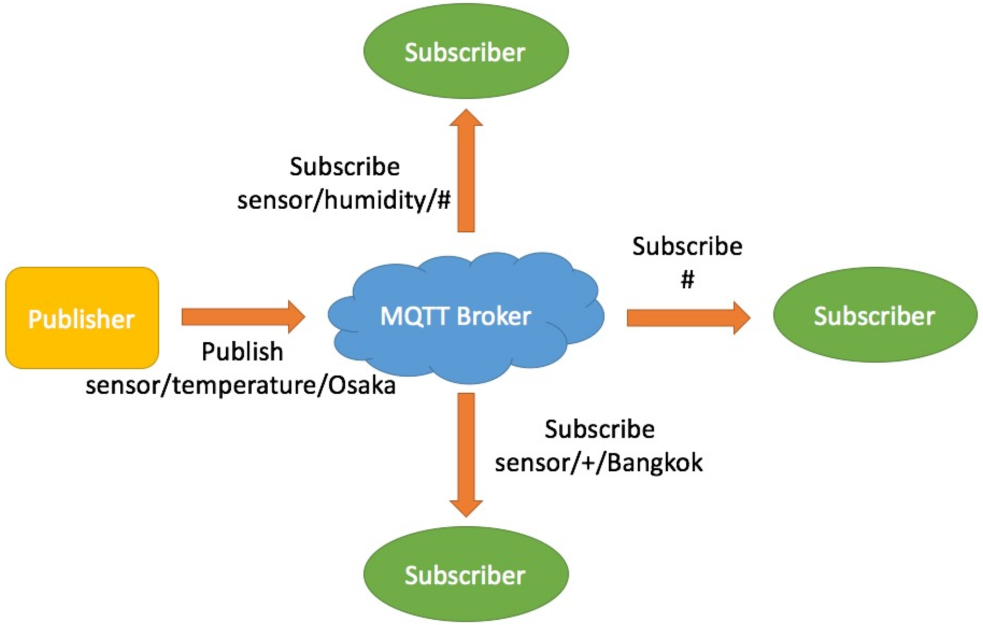
- The mobile sensor market grew **over 200%** per year (2007 – 2012).
- A growth of sensor devices continues to **trillion** within a decade
- How can IoT service providers handle such number of devices while using the number of machines effectively?

*J. Bryzek, “Roadmap for the trillion sensor universe,” Berkeley, CA

MQTT Broker in the IoT system



*How AWS IoT works

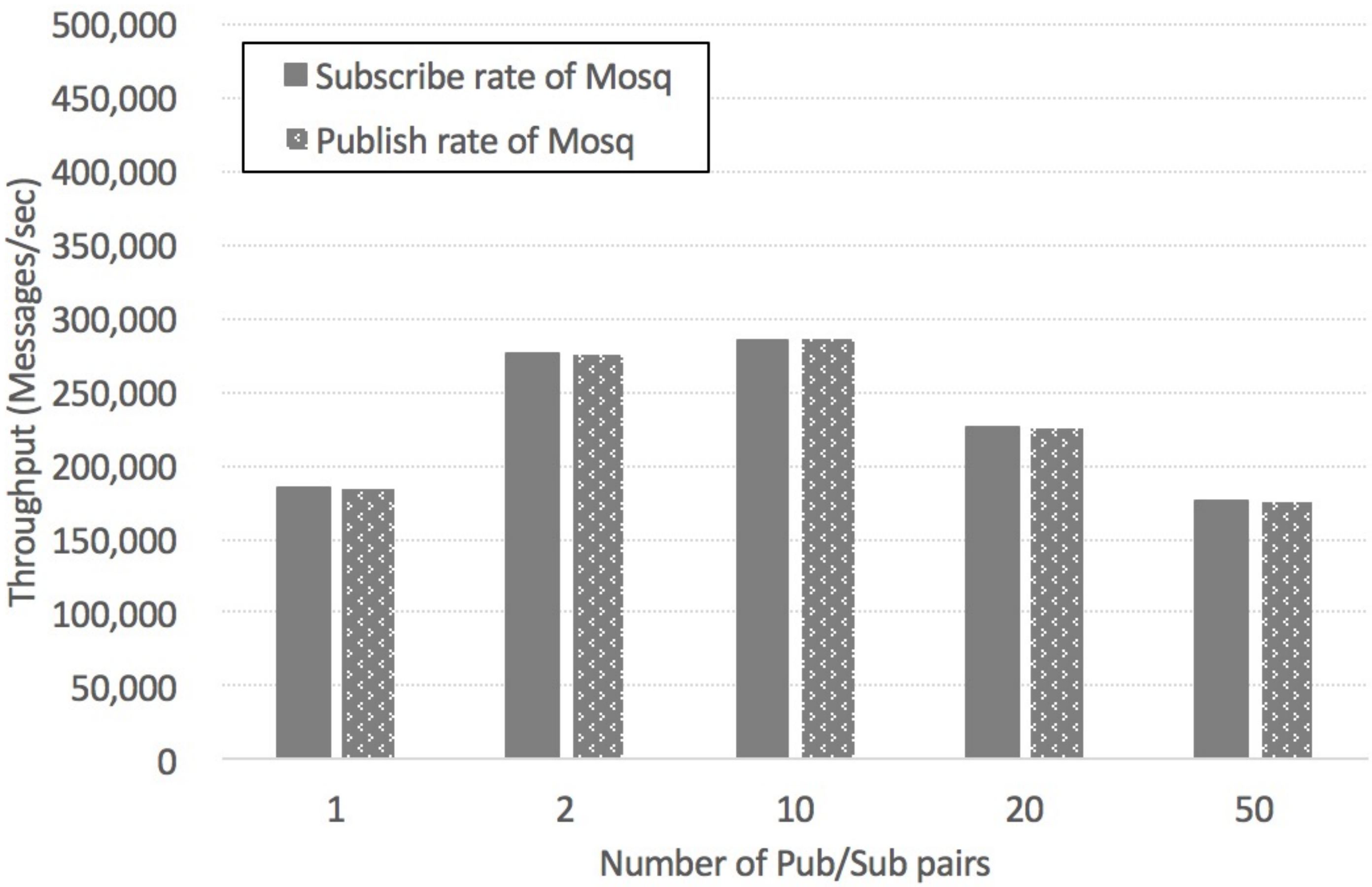


- Intermediate node
- Between devices and a platform
- Like a hub

- Pub/sub pattern
- MQTT
  - Topic filter with wildcard (#,+)
  - QoS support (3 levels)

*How AWS IoT Works: http://docs.aws.amazon.com/iot/latest/developerguide/aws-iot-how-it-works.html

Can current MQTT broker software handle such workload?

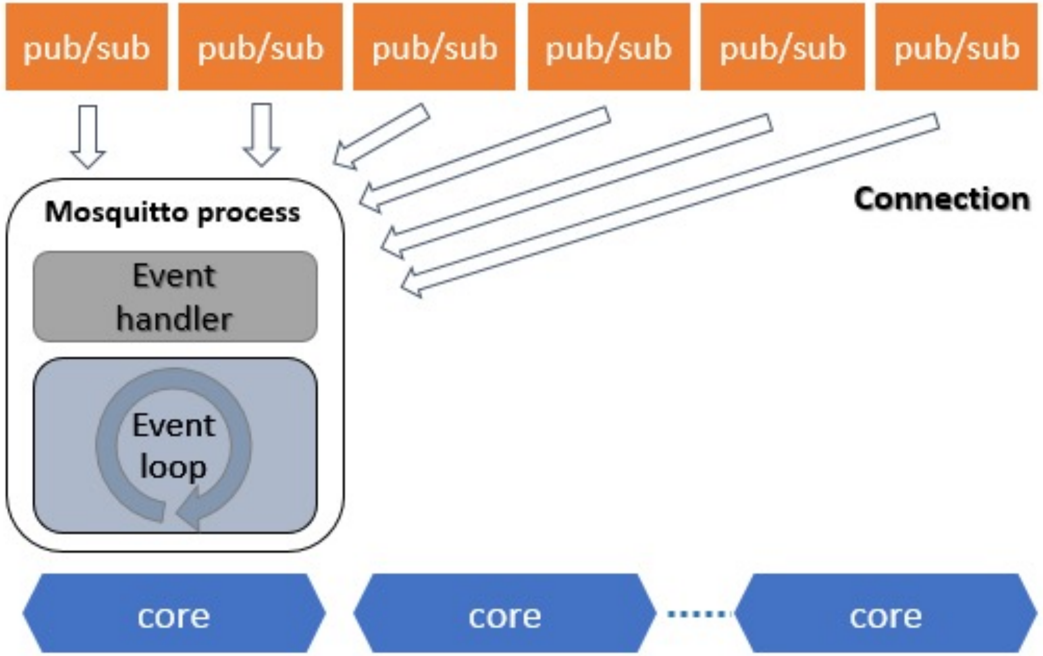


e.g. Mosquitto

- Well-known message broker
- Mainly support MQTT
- I/O multiplexing to handle multiple connections
- **Utilizing single core**
- **In-kernel TCP/IP stack**

Can vertical scalability help?

- ❑ Vertical scalability is an inefficient approach to increase the capacity of an MQTT broker running on a single machine because;
  - **Software cannot fully utilize all available CPU cores**



  - **In-kernel TCP/IP stack cannot process n/w packets efficiently.**

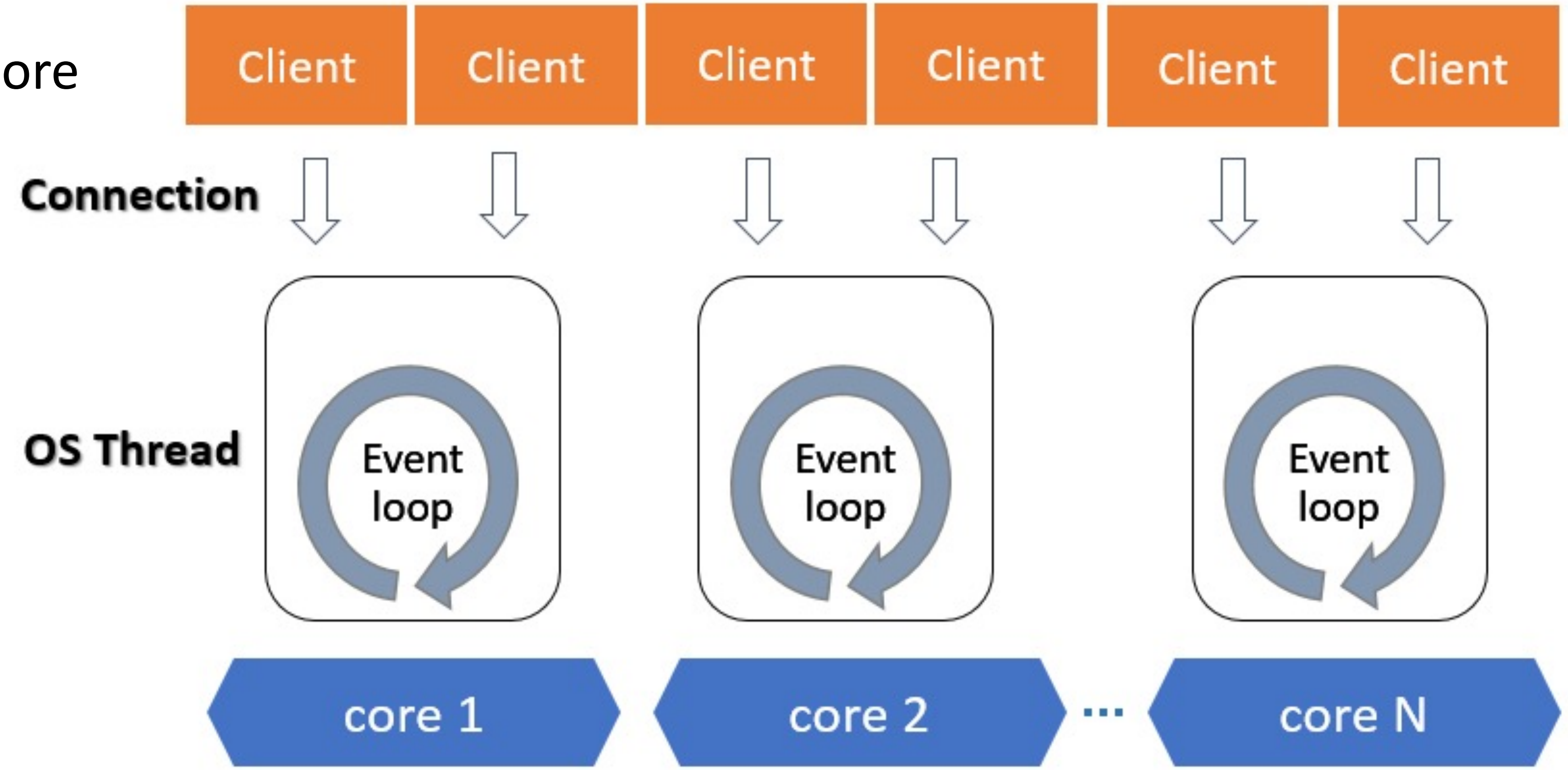
The C10M problem

- How one server machine handles 10 million concurrent connections simultaneously
- 3 main problems
  - **Multi-core scalability**
  - **No fast data path**
  - **Memory alignment**

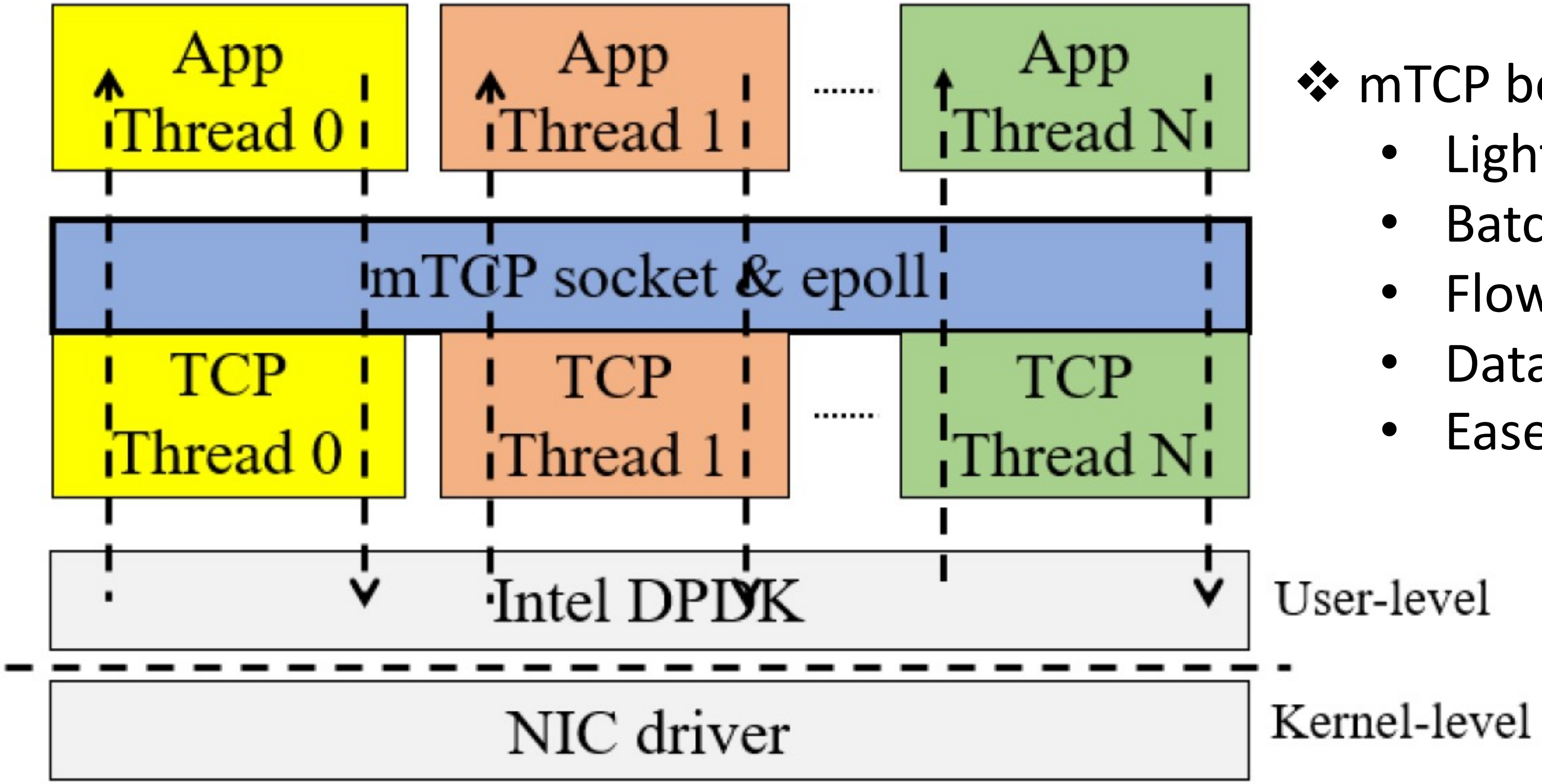
*The C10M problem: http://c10m.robertgraham.com/p/manifesto.html

Scalable I/O multiplexing

- Multiple requests per core
- Utilize many cores
- No context switching (1 thread per core)
- Data locality



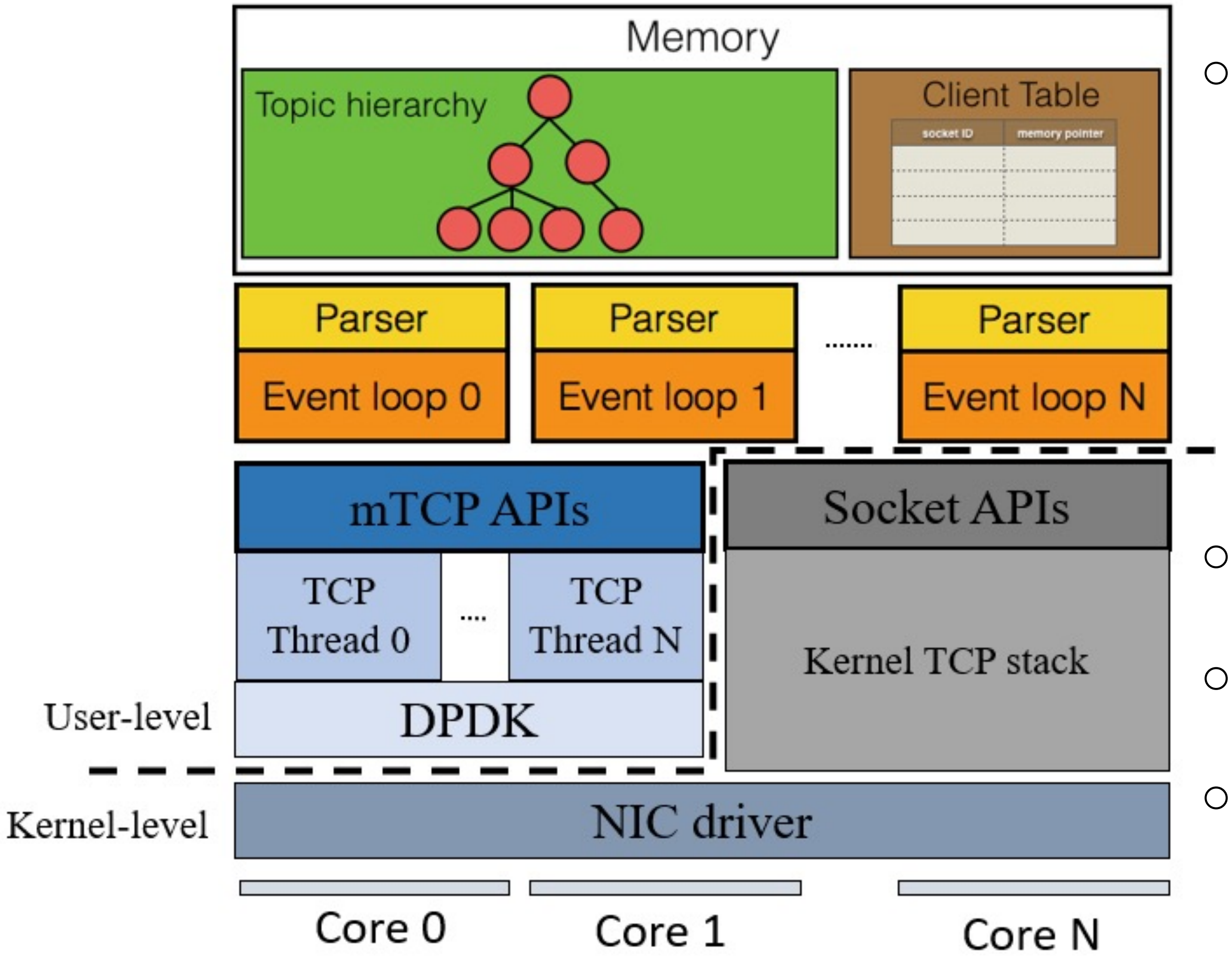
mTCP: User-space TCP/IP stack



- ❖ mTCP benefits
  - Lightweight connection setup
  - Batch event handling
  - Flow-level core affinity
  - Data locality
  - Ease of port

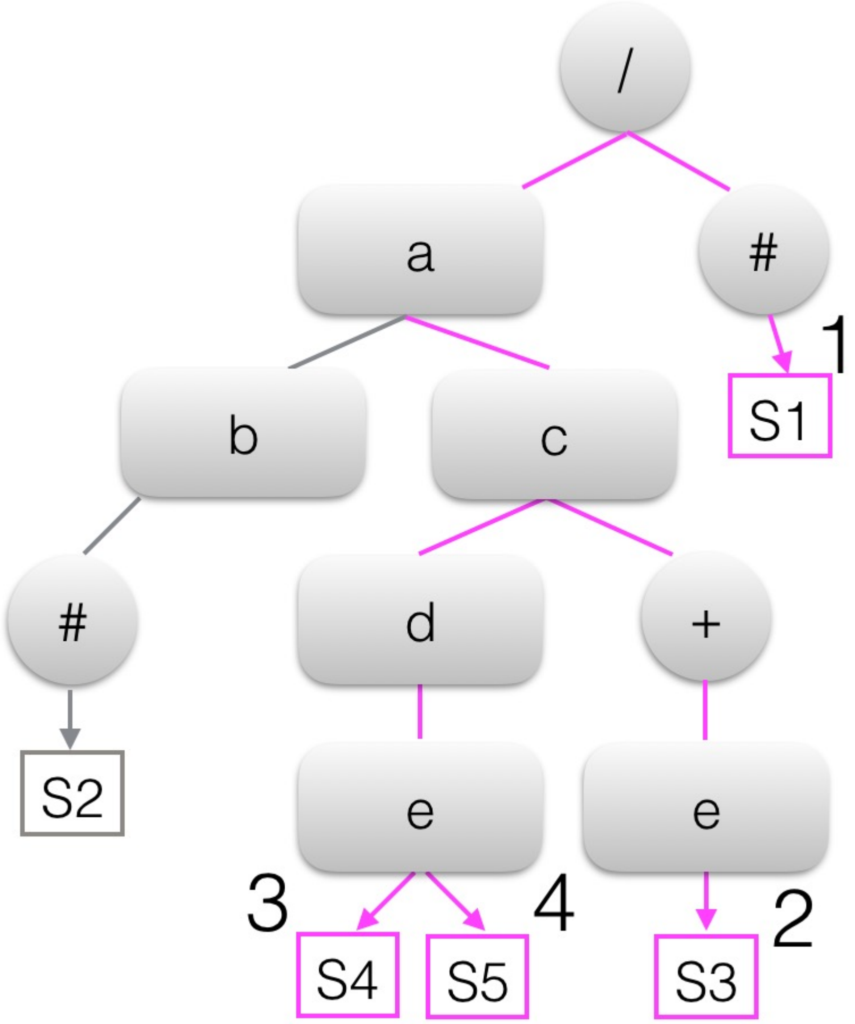
*mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems (USENIX NSDI 2014)

Overview of muMQ



- 3 main components at application level
  - Socket event loops
    - Monitoring socket events
    - Parsing and processing MQTT logic
  - Topic tree
    - Store subscription topics
  - In-memory table
    - Keep client info
- muMQ running either on in-kernel TCP/IP stack or on mTCP
- mTCP using Intel DPDK as a packet I/O engine
- TCP thread spawning an event loop (app thread), running on the same core

Subscription topic matching



Topic hierarchy

- Dynamic tree
- Constructed by subscription topics
- Topic nodes containing to subscribers
- Use traverse tree approach

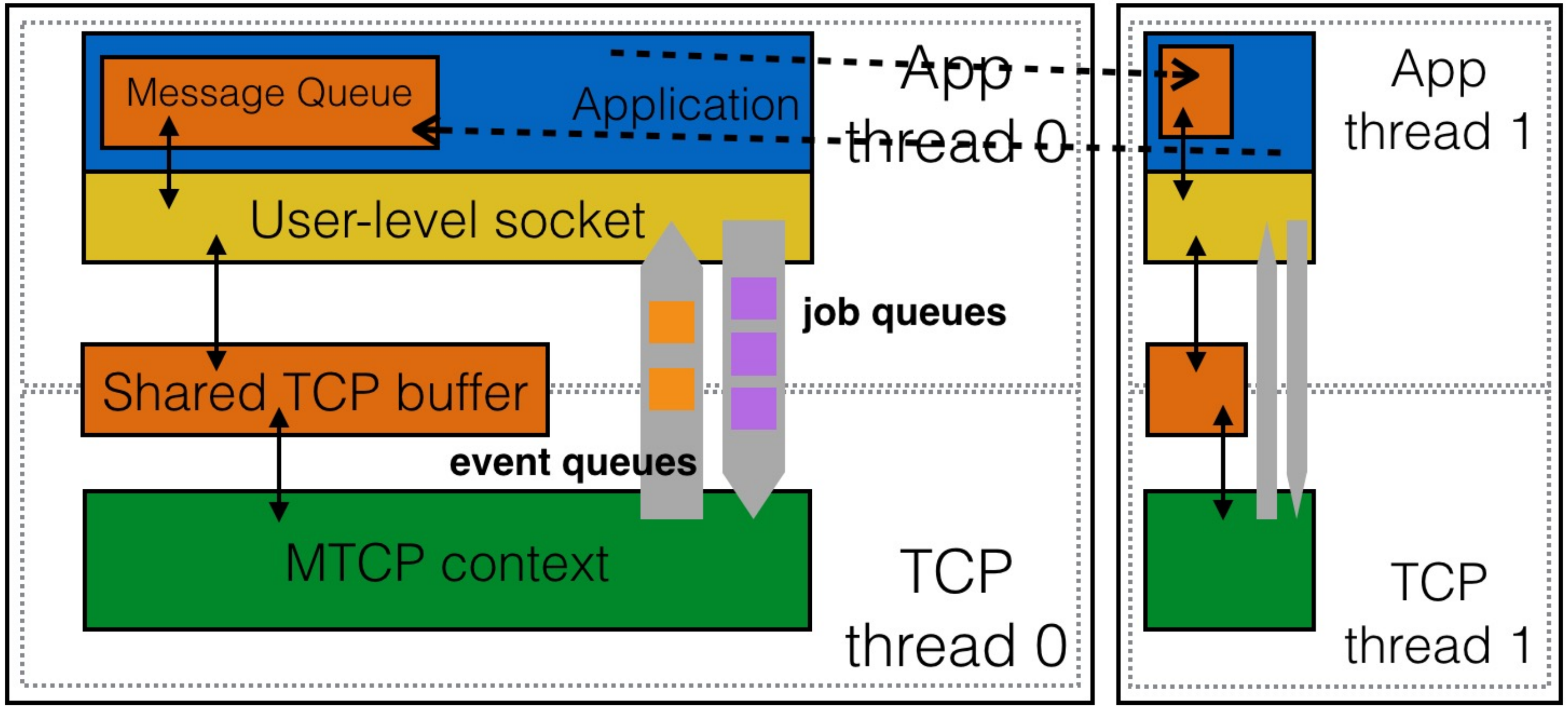
<u>Subscription topics</u>

- S1 -> ‘#’
- S2 -> ‘a/b/#’
- S3 -> ‘a/c/+/e’
- S4 -> ‘a/c/d/e’
- S5 -> ‘a/c/d/e’

<u>Publish topic</u>

- ‘a/c/d/e’

Thread model of muMQ



- 1:1 thread model between a TCP thread and an app thread.
- M:N for app threads, each of which can access others' message queue.
- Message queue is FIFO linked list.

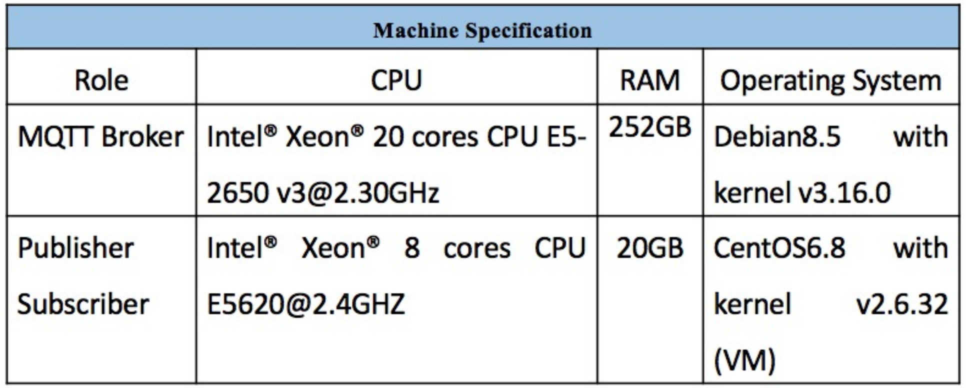
Implementation

- ❖ Event notification
  - epoll with a level-triggered mode
  - SO_REUSEPORT and pthread for scalable I/O multiplexing
- ❖ MQTT parser
  - Support basic and QoS0 commands
- ❖ Subscription matching logic
  - Emulate logic from Mosquitto
  - Wildcards support
- ❖ Port to mTCP
  - Code changed ~90 LoC

Evaluation

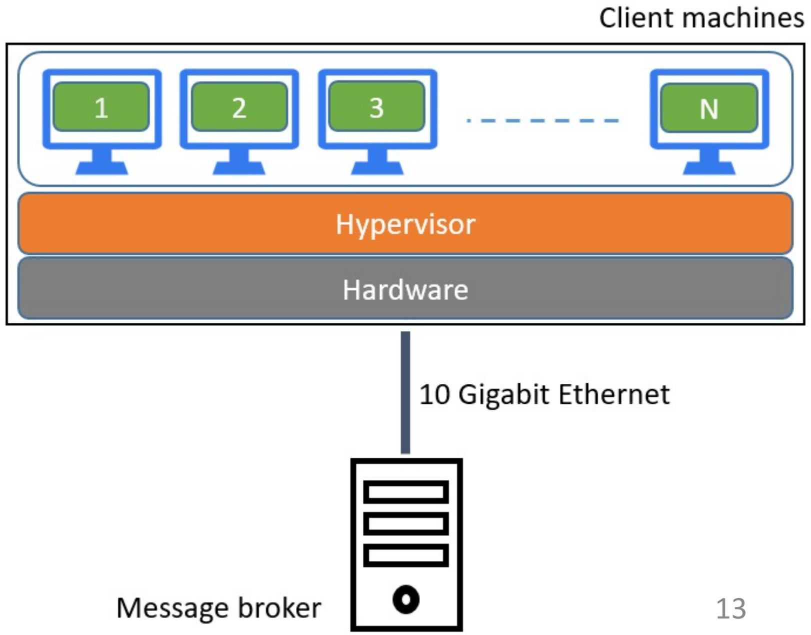
- End-to-end latency
  - Compare while handling large numbers of subscribers
- Throughput
  - Count the number of transactions under heavy workloads
- CPU utilization
  - Measure CPU usage during handling long-lived subscribers
- Compared systems
  - Mosquitto excluding persistent database and $SYS topic.
  - muMQ using Linux kernel TCP/IP stack (muMQ-k)
  - muMQ using mTCP (muMQ-m)

Note muMQ was configured to use 16 CPU cores

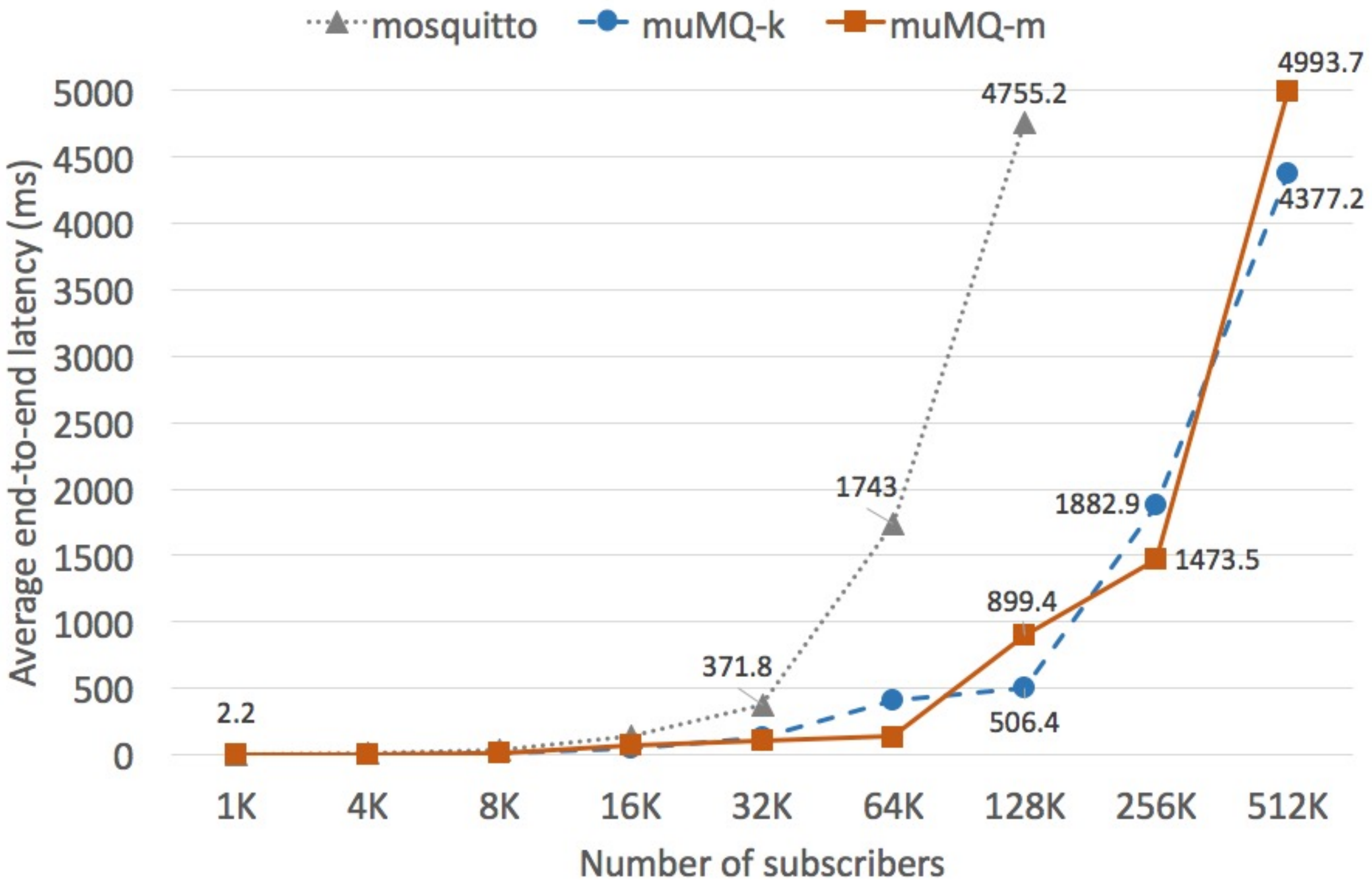


Machine Specification

Role	CPU	RAM	Operating System
MQTT Broker	Intel® Xeon® 20 cores CPU E5-2650 v3@2.30GHz	252GB	Debian8.5 with kernel v3.16.0
Publisher Subscriber	Intel® Xeon® 8 cores CPU E5620@2.4GHZ	20GB	CentOS6.8 with kernel v2.6.32 (VM)

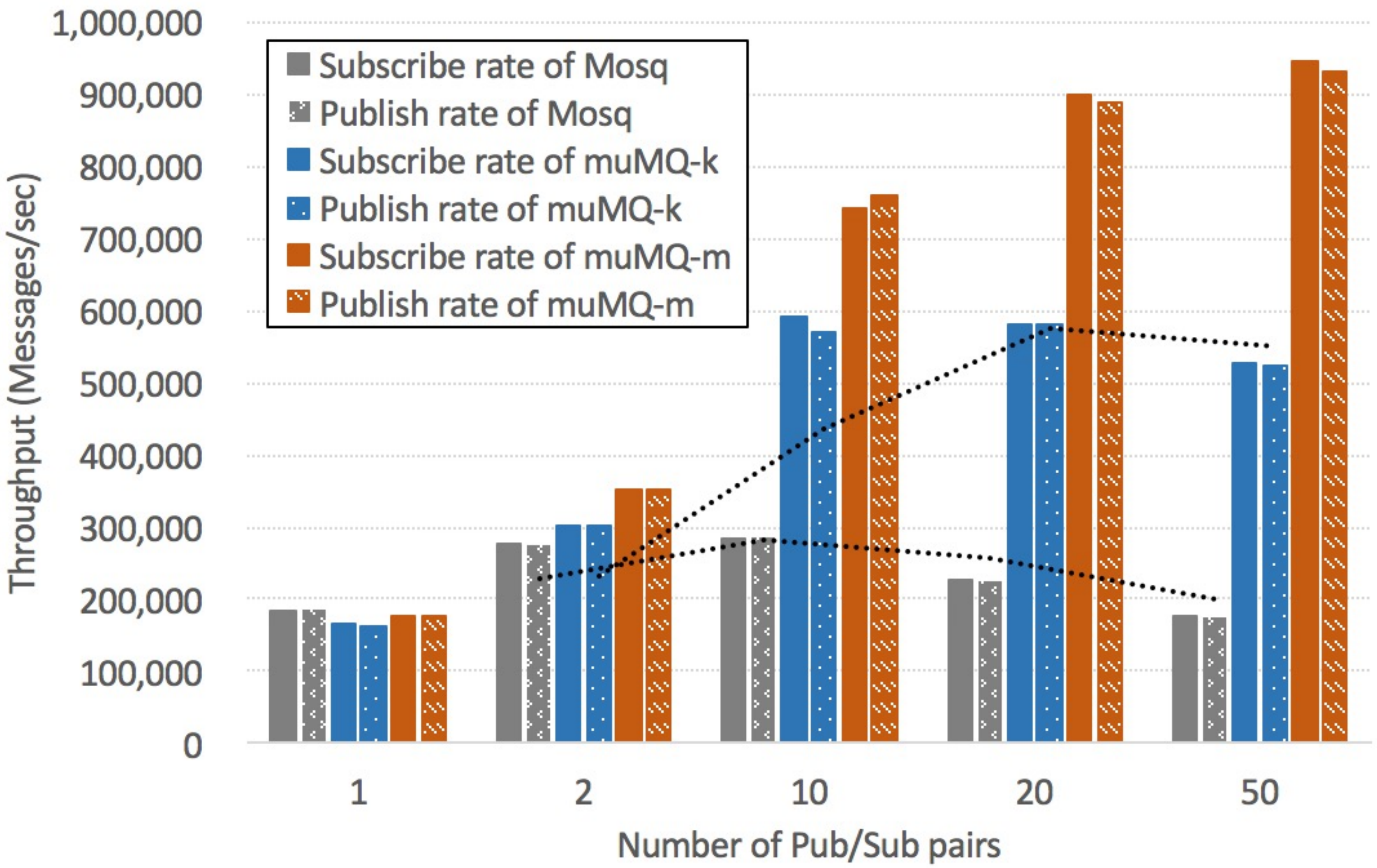


End-to-end latency



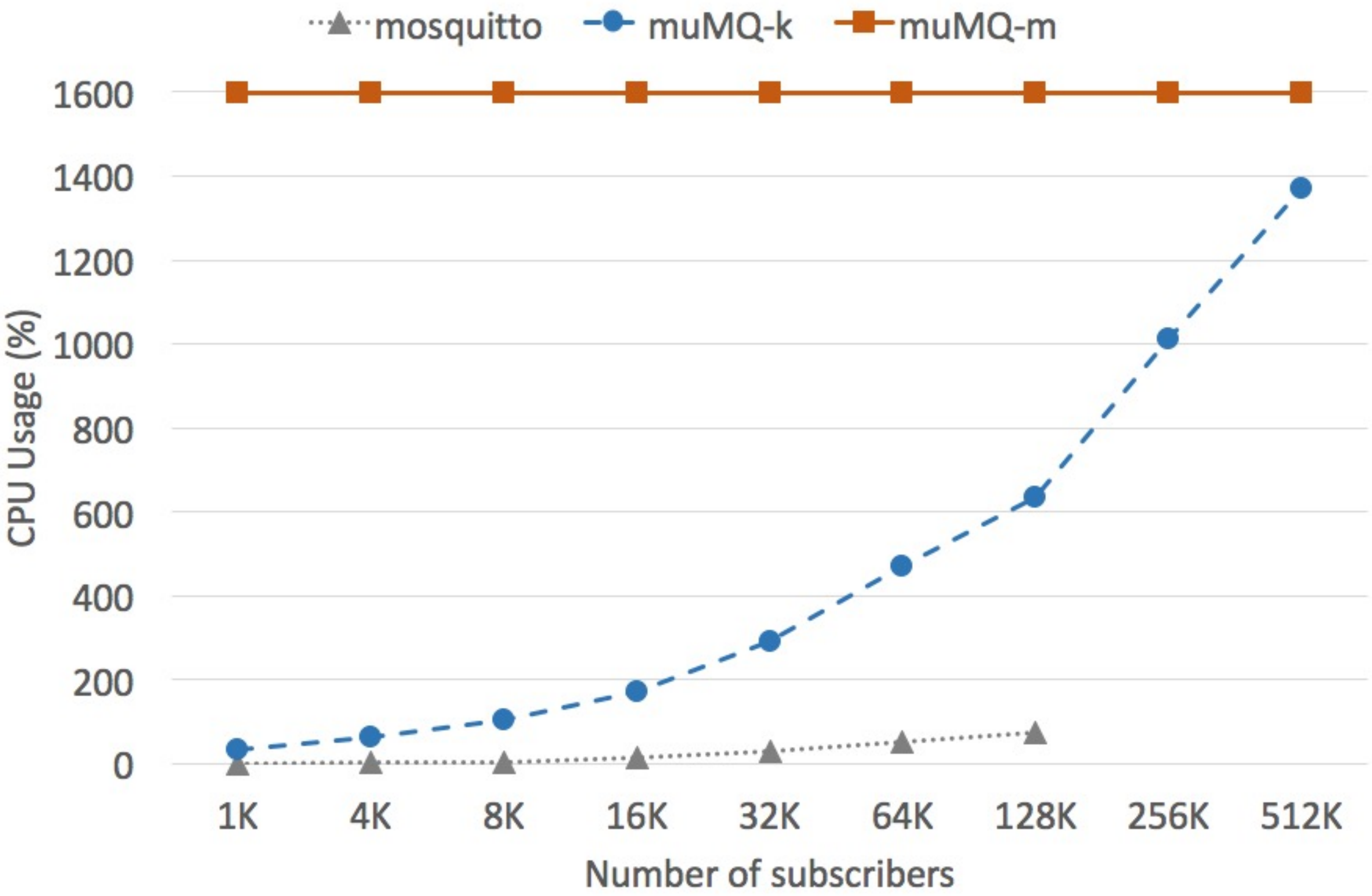
- 5 messages sent to subs
- 1:N topic relation
- Mosquitto latency 1,743 ms at 64K subs
- 4.3x and 13x higher than muMQ-k, -m
- muMQ > 512K subs

Throughput



- 64B Payload
- 1:1 topic relation
- Subs rate 944K msgs/s
- Pub rate 930K msgs/s
- 5.38x Mosquitto
- 1.8x muMQ-k
- Mosquitto’s, muMQ-k’s are saturated

CPU Utilization



- Mosquitto not over 100%
- muMQ-k increases as # subscribes
- muMQ-m always touch 1600%

Discussion

- Porting a message broker to mTCP is not a straightforward task.
  - Unable to make use of flow-level core affinity as well as per-core data structure due to cross-thread access.
  - Topic-based affinity can address; how can a commodity NIC support it?
- A multi-process architecture is another solution for utilizing multi-core facility.
  - Mosquitto can do that by using bridge mode to form a cluster of message brokers.
  - Nonetheless, the communication overhead between processes can penalize the performance.

Conclusion

- muMQ is a lightweight and scalable MQTT broker.
- Critical factor improving the performance is multi-core scalability.
- It overcomes the performance limitation of the traditional TCP/IP stack by exploiting mTCP.
- Throughput of muMQ using mTCP outperforms Mosquitto’s by up to 538%.
- Future work is to compare the performance of other well-known MQTT brokers supporting multi-threading.

Thank you

The source code is available (soon) from
https://bitbucket.org/aistmu/mumq

Related works

- MQTT brokers
  - Mosquitto uses the I/O multiplexing technique to handle multiple sockets by a sinble core.
  - Apache ActiveMQ creates a thread pool to handle a client connection per thread.
  - RabbitMQ uses an Erlang VM thread pool to perform I/O operation asynchronously.
- mTCP-based KVS
  - *Our previous study ports memcached and redis to use mTCP. But the work shows porting to mTCP is difficult because some monitored file descriptors are not TCP sockets.

*Toward Fast and Scalable Key-Value Stores Based on User Space TCP/IP Stack (AINTEC 2015)