Messaging Systems

How to make the right choice?

Adina MIHAILESCU
@acm1107

source: quickmeme.com

Messaging Systems

Messaging makes applications **loosely coupled** by communicating **asynchronously**, which also makes the communication more **reliable** because the two applications do not have to be running at the same time.

Why Messaging?

Performance

improve response times by doing some tasks asynchronously

Why Messaging?

Decoupling

reduce complexity by decoupling and isolating applications

Why Messaging?

Scalability

build smaller apps that are **easier to develop**, debug, test, and scale

distribute tasks across machines based on load

Why Messaging?

High-quality, cost-effective

build multiple apps, using the most **suitable language or framework**
for each, versus one big monolithic app

Why Messaging?

High availability

get **robustness and reliability** through message queue persistence

potentially get **zero-downtime** redeploys

Messaging Patterns

Routing schemes: multicast, broadcast, PointToPoint

Message **store** (persistent, transient)

Message **transformation**

Message-Oriented Middleware

infrastructure supporting **sending and receiving messages** between distributed systems

message transfer agent = **broker**

standard **protocols** and **API**s

Java Message Service

standard messaging **API** for **Java** platform
robust, flexible

cross platforms **interoperability** is possible, but ...
is **restrictive**, **limited**, and forces vendor lock-in.

Implementations: **ActiveMQ**, **HornetQ**, **Apollo**

JMS API

Producer example

JAVA

```
import javax.jms.Connection;
import org.apache.activemq.spring.ActiveMQConnectionFactory;

ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory();
connectionFactory.setBrokerURL("tcp://localhost:61616");
connection = connectionFactory.createConnection();
connection.start();
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
Destination destination = session.createTopic("Testtopic");
MessageProducer producer = session.createProducer(destination);
producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
String text = "Message String";
TextMessage message = session.createTextMessage(text);
producer.send(message);
session.close();
```

JMS API

Consumer example

JAVA

```
import javax.jms.Connection;
import org.apache.activemq.spring.ActiveMQConnectionFactory;

ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory();
connectionFactory.setBrokerURL("tcp://localhost:61616");
Connection connection = connectionFactory.createConnection();
connection.start();
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
MessageConsumer consumer = session.createConsumer(session.createTopic("Testtopic"));
consumer.setMessageListener(new MessageListener());
```

Why AMQP?

cross platforms **interoperability**

open standard protocol, less vendor lock in

efficient binary wire protocol

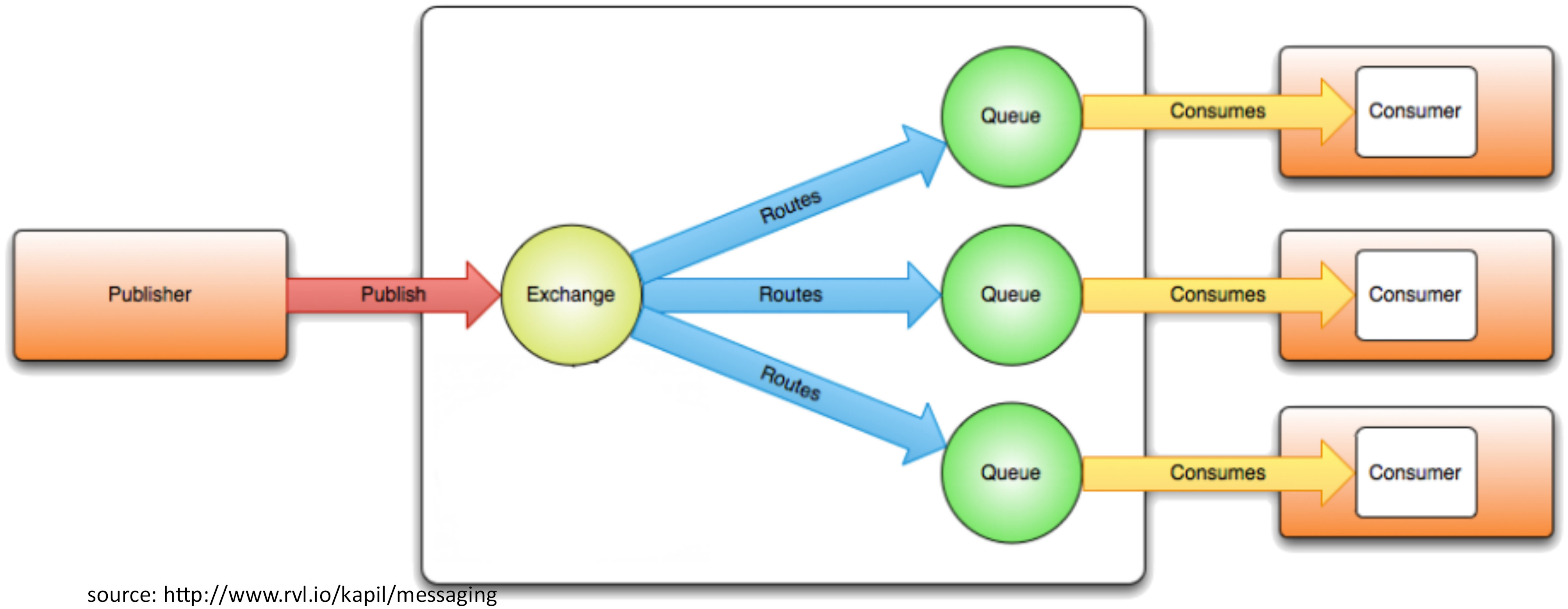
support in **various languages, on most platforms**, but ...

scope is too large ...

no interoperability between implementations (0.8, 0.9.1, 0.10, 1.0)

Implementations: **Apache Qpid, RabbitMQ, ActiveMQ (>5.8), Apollo**

AMQP: message flow



source: http://www.rvl.io/kapil/messaging

AMQP API

Ruby client - enqueue

RUBY

```
require 'amqp'

AMQP.start(:host => 'localhost') do |connection|
  AMQP::Channel.new(connection) do |channel, open_ok|
    channel.on_error do |channel, channel_close|
      puts "Error #{channel_close.reply_code}: #{channel_close.reply_text}"
    end
    channel.queue('/queue/test_queue', :durable => true) do |queue, declare_ok|
      channel.default_exchange.publish 'Message String',
        :routing_key => queue.name, :persistent => true
      EventMachine.add_timer(0.5) { connection.disconnect { EventMachine.stop } }
    end
  end
end
```

AMQP API

Ruby client - dequeue

RUBY

```
require 'amqp'

AMQP.start(:host => 'localhost') do |connection|
  AMQP::Channel.new(connection) do |channel, open_ok|
    channel.on_error(&method(:handle_channel_exception))
    channel.queue('/queue/test_queue', :durable => true) do |queue, declare_ok|
      queue.subscribe(:ack => true) do |header, message|
        puts "Received #{message}"
        header.ack
        if (message == 'QUIT')
          queue.unsubscribe
          connection.disconnect { EventMachine.stop }
        end
      end
    end
  end
end
```

Why not STOMP?

Simple/Streaming **T**ext **O**riented **M**essaging **P**rotocol

simple and **lightweight** (although verbose on the wire)

wide range of language **bindings**

widely-interoperable, but ...

- some features are implementation specific
- not straightforward to port code between brokers

Implementations: ActiveMQ, HornetQ, RabbitMQ

STOMP API

Ruby client - enqueue

RUBY

```ruby
require 'stomp'

client = Stomp::Client.new(:host => 'localhost')
client.publish '/queue/test_queue', 'Message String', :persistent => true
client.close
```

#MQ 19/37

STOMP API

Ruby client - dequeue

RUBY

```
require 'stomp'

client = Stomp::Client.new(:host => 'localhost')
client.subscribe '/queue/test_queue', :ack => :client do |message|
  puts "Received #{message}"
  client.acknowledge(message)
  client.close if (message == 'QUIT')
end
client.join
```

#MQ

Why not brokerless?

ZeroMQ

high performance and throughput oriented messaging middleware

a socket library

transport agnostic sockets: in-process, IPC, multicast, TCP

message oriented

topology aware

30+ languages bindings

ZeroMQ API

Ruby client - push

RUBY

```
require 'ffi-rzmq'

context = ZMQ::Context.new 1
sender = context.socket ZMQ::PUSH
sender.connect('tcp://127.0.0.1:5555')
sender.send_string('Message String')
sender.close
```

ZeroMQ API

Ruby client - pull

RUBY

```ruby
require 'ffi-rzmq'

context = ZMQ::Context.new 1
receiver = context.socket ZMQ::PULL
receiver.bind('tcp://127.0.0.1:5555')
message = ''
receiver.recv_string(message)
puts "Received #{message}"
receiver.close
```

U MOM in the Cloud!

Message **Q**ueuing **S**ervice
Message-**O**riented **M**iddleware deployed in a cloud using **S**oftware **a**s **a** **S**ervice model.

SaaS benefits

eliminates the traditional **overhead** associated with operating in-house messaging infrastructures

provides an **internet scale** messaging platform

simplifies **access** to messaging resources

facilitates **integration**

reduces cost

Distributed solutions in the Cloud (SaaS)

Amazon Simple Queue Service - ASQQ
RoQ - Open Source project backed up by EURA NOVA
CloudAMQP - RabbitMQ as a Service
StormMQ as a Service - AMQP in the Cloud

One size doesn’t fit all

Broker vs Brokerless

Transient vs. Persistent messages

Atomic transactions

Fault-tolerance

Testing framework

brokers: ActiveMQ, Apollo, QPid, RabbitMQ, HornetQ, ZMQ broker

protocols: AMQP, STOMP, socket

different **message size**: 32b, 1kb, 32kb

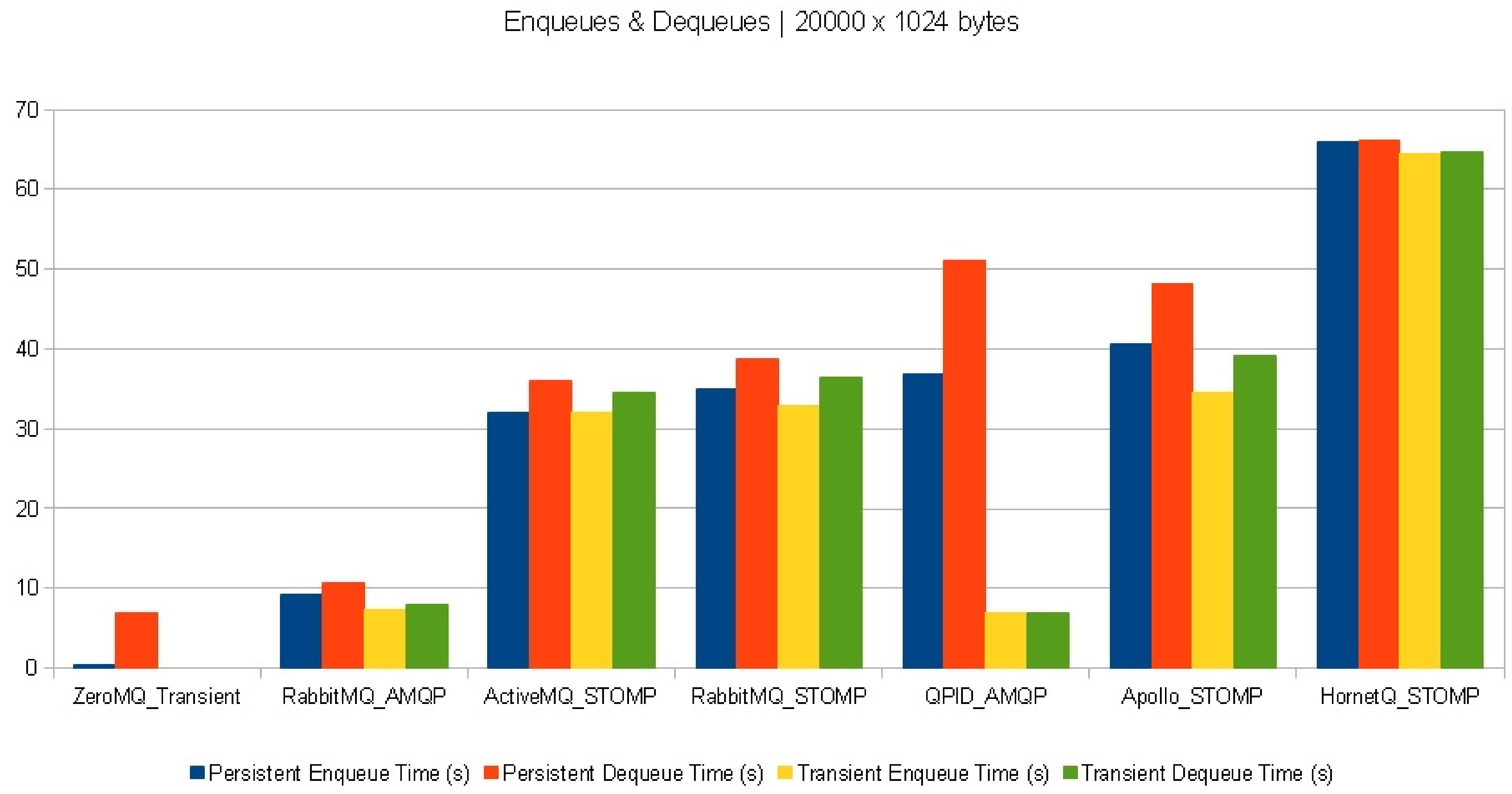
in-memory vs. **persistent** queues (except for ZMQ)

References

Code available on github

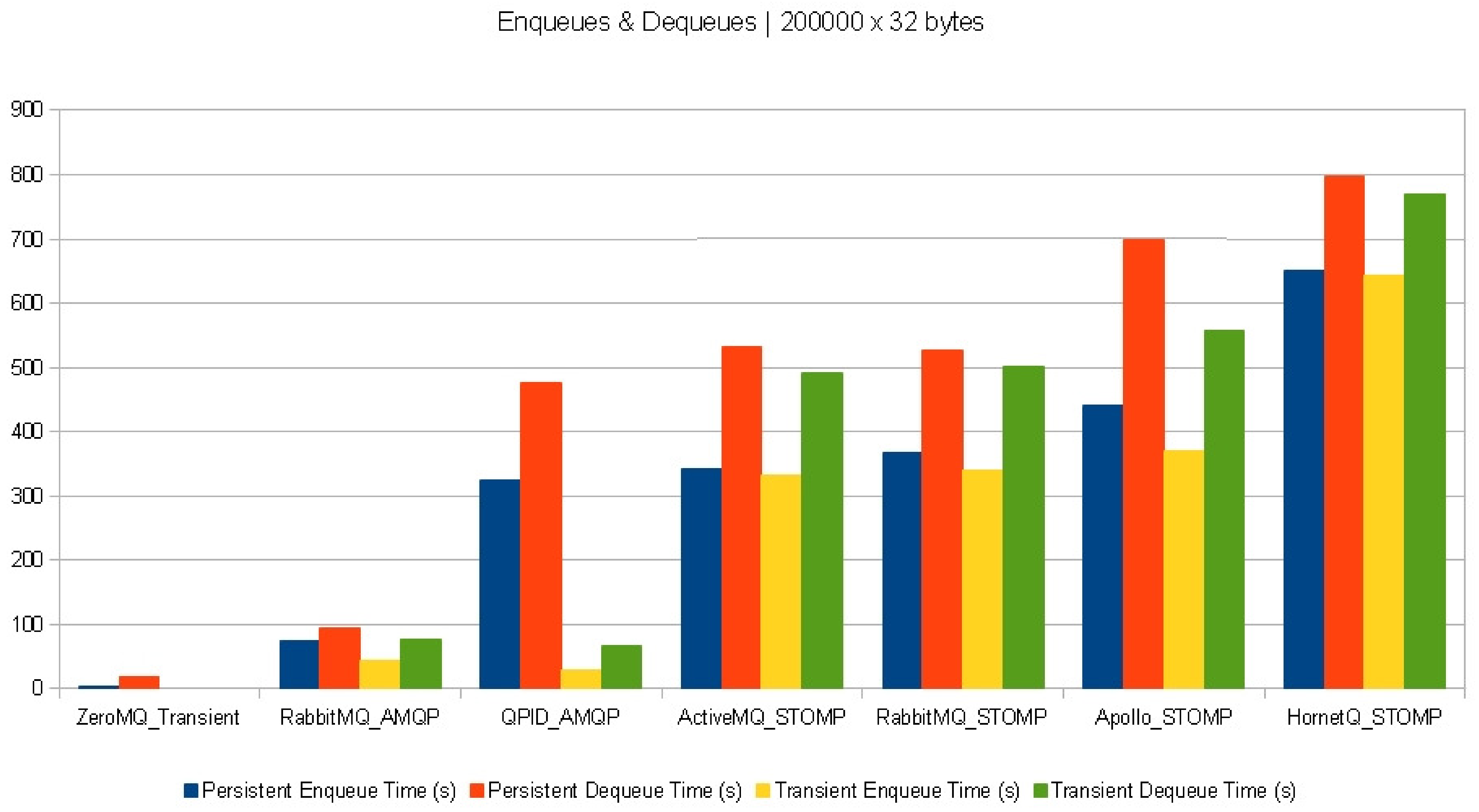
More details on Muriel's Tech Blog

Benchmark (i)

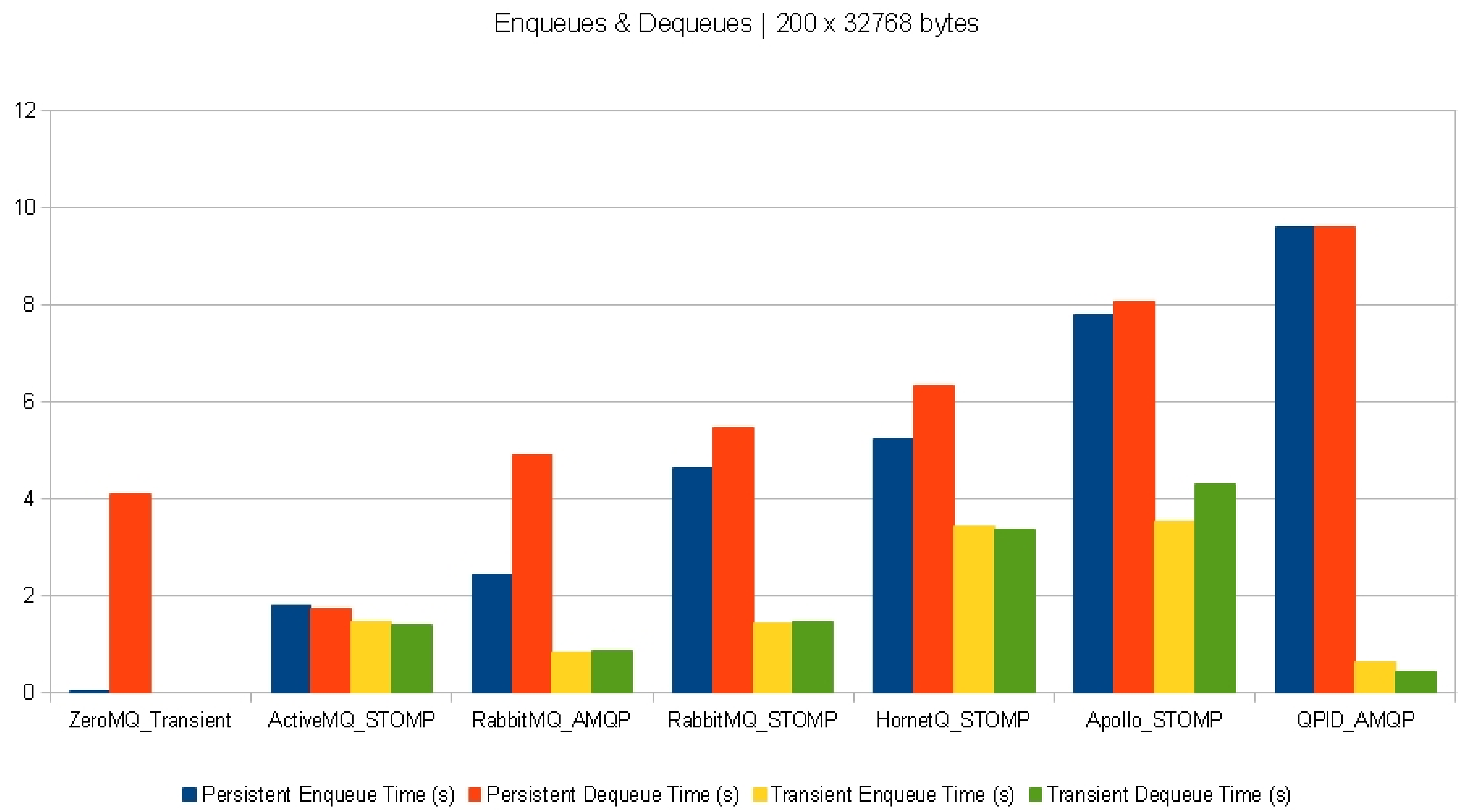


#MQ 29/37

Benchmark (ii)



Benchmark (iii)



#MQ 31/37

Functional requirements

sequencing: FIFO mode, total ordering

item expiration

LIFO filtering by content

dequeue throttling: items/s, bandwith(kb/s), item window size

priority message properties: size, max throughput, max items

atomic transactions: one, or multiple operations, XA support

queuing delay

retry policy: with regular, or exponential delay

asynchronous acknowledgement

Operational requirements

persistent queues

high availability for queuing

administrative features

Wrap Up

Why Messaging?

Messaging Patterns

Message-Oriented Middleware

JMS

Why AMQP?

Why not STOMP?

U MOM in the Cloud

One size doesn’t fit all

Testing framework - quick benchmark

Match MQ features to requirements

References: Ruby gems, test framework

AMQP
STOMP
ZeroMQ
MQ test framework

References: brokers, frameworks, protocols

Apache ActiveMQ
Apache Apollo
Apache Qpid
RabbitMQ
JBoss HornetQ
ZeroMQ

Enterprise Integration Patterns
AMQP 0.9.1 specs
AMQP OASIS 1.0 specs
STOMP specs

<Thank You!>

@acm1107