FaBRiQ: Fast, Balanced and Reliable Queue

Iman Sadooghi
Outline

• Kafka
• SQS
• Fabriq
  – Motivation
  – Design
  – features
  – Communication cost analysis
• Performance evaluation
Kafka
Motivation

• Log Data
  – Orders of Magnitude larger than the actual data
  – Facebook 5TB daily
  – China Mobile 5-8 TB daily

• Many types of events
  • user activity events: impression, search, ads, etc
  • operational events: call stack, service metrics, etc

• High volume: billions of events per day

• Both online and offline use case
  • reporting, batch analysis
  • security, news feeds, performance dashboard
Solution!

• Traditional Messaging Systems
  – JMS
    • Acknowledge after msg consumption
    • Weak distributed support
    • No batching
  – IBM WebSphere MQ
    • Provides transactional support!
  – ActiveMQ

• Assuming msgs should be consumed real quick
• No offline support
• No focus on throughput
Log Aggregators

• Collect data and load into DWH or Hadoop
  – Facebook Scribe
    • Periodically dumps bunch to HDFS
  – Cloudera Flume
    • Uses push approach
  – Yahoo Data Highway

• Problem
  – All for offline data consumption
  – No online consumption support
Message queues
• ActiveMQ
• TIBCO
Log aggregators
• Flume
• Scribe

KAFKA

• Low throughput
• Secondary indexes
• Tuned for low latency

• Focus on HDFS
• Push model
• No rewindable consumption
Kafka

- Collect and deliver high volumes of large data
- Low latency
- Scalable
Design

- Problem: point to point pipelines!!
Pub-Sub

Producer

publish(topic, msg)

Topic 1
Topic 2
Topic 3

Publish subscribe system

Consumer

subscribe

msg

Consumer

msg

subscribe

msg
Design

- Producer, Consumer
- Broker
- Topic, Partition
- Load balance
Efficiency

• No message IDs
  – Logical offset

• Consumer consumes msgs from a Partition sequentially (starting on an offset)

• Single partition in a topic
  – Only used by single consumer

• No Master
  – Consumers and brokers coordinate via ZooKeeper
Stateless Broker

- Broker doesn’t keep track of consumption
- Each consumer maintains its own state
- Message deletion driven by retention policy, not by tracking consumption
  - rewindable consumer
ZooKeeper

- Create a path
- Set value to path
- Read value on path
- Delete path
- Get notifications on a path
- Provides replication
Auto Consumer Load Balance

- brokers and consumers register in zookeeper
- consumers listen to broker and consumer changes
- each change triggers consumer rebalancing
Guarantees

• At least once delivery
• In order delivery, inside a single partition
• No guarantee on order from diff partitions
• No support for duplicated messages
• Persistence
  – If broker goes down
    • msgs temp unavailable
  – If broker disk damaged
    • Msgs lost permanently
Usage in LinkedIn
Hadoop Data Load for Kafka

Live data center

Real time consumers

Kafka

Offline data center

Kafka

Dev Hadoop

PROD Hadoop
Performance Evaluation

- 2 Linux boxes
  - 16 2.0 GHz cores
  - 6 7200 rpm SATA drive RAID 10
  - 24GB memory
  - 1Gb network link
- 200 byte messages
- 10 million msgs in total
- Batch size:
  - 1: 50K msgs/sec
  - 50: 400K msgs/sec
Producer performance
Consumer performance

![Graph showing consumer performance](image)
Scalability

(10 topics, broker flush interval 100K)
SQS

• Amazon Simple Queue Service (SQS)
  – Distributed message delivery queue
    • Highly scalable
    • Messages sent and read simultaneously
    • Reliable
      – Guarantees message delivery
        » At least once delivery
Outline

• Kafka

• SQS

• **Fabriq**
  – Motivation
  – Design
  – features
  – Communication cost analysis

• Performance evaluation
Motivation

• More than 2.5 exabytes of data is generated every day
  – more than %70 of it is unstructured
• not possible for the traditional data processing systems to handle needs in Big Data processing.
  – There is a need to reinvent the wheel instead of using the traditional systems
• Traditional data processing middleware being replaced:
  – SQL databases by No-SQL datastores
  – file system by key-value storage systems
Motivation

• A Distributed message queue
• useful in various data movement and communication scenarios
  • monitoring,
  • workflow applications,
  • big data analytics,
  • log processing
• Companies started using queues:
  – Linkedin, Facebook, Cloudera and Yahoo have developed similar queuing solutions
    • to handle gathering and processing of terabytes of log data on their servers
  – Kafka feeds hundreds of gigabytes of data into Hadoop clusters and other servers every day
• Queues can play an important role in Many Task Computing (MTC) and High Performance Computing (HPC)
  – handle data movement on HPC and MTC workloads in larger scales without adding significant overhead to the execution process
  – CloudKon
Challenges

• Traditional queue services
  – usually have centralized architecture
  – cannot scale well to handle today’s big data requirements
  – Providing transactional support
  – Providing consumption acknowledgement
  – Persistence: Many are in memory queues.
  – Delivery guarantee!
Introducing FaBRiQ

• FaBRiQ (Fast, Balanced and Reliable Queue):
  – a persistent message queue that aims to achieve high throughput and
  – low latency
  – while keeping the near perfect load balance and high utilization on large scales
  – Uses ZHT as its building block
    • Communications
    • Storing data
• Distributing queues among servers
FaBRiQ Server

Diagram showing Metadata Lists and ZHT Server:
- Metadata Lists:
  - Q1
    - server-i
  - Qn
    - server-i
- ZHT Server:
  - Key
  - Value
  - mld-1
    - value1
  - mld-2
    - value2
  - mld-i
    - value i
  - ...
  - ...
  - mld-j
    - ...
  - ...

Operations

• CreateQueue
• Push
• Pop
• Remove
Push

a) Message ID Queue exists
   1) push(Qx,"msg-contents")
   2) push(Qx,mld-k,"msg")

Client
↓
Server-s

Metadata Lists
Qx
<table>
<thead>
<tr>
<th>server-1</th>
<th>...</th>
<th>...</th>
<th>...</th>
</tr>
</thead>
</table>

Message Queue
Qx
<table>
<thead>
<tr>
<th>mld-j</th>
<th>mld-k</th>
<th>...</th>
<th>...</th>
</tr>
</thead>
</table>

(3) push message ID
(4) put(mld-k,"msg") to ZHT

ZHT Server
Key | Value
--- | ---
mld-1 | value1

b) Message ID Queue does not exist
   1) push(Qx,"msg-contents")
   2) push(Qx,mld-k,"msg")

Client
↓
Server-s

Metadata Lists
Qx
<table>
<thead>
<tr>
<th>server-1</th>
<th>...</th>
<th>...</th>
<th>...</th>
</tr>
</thead>
</table>

Message Queue
Qx
<table>
<thead>
<tr>
<th>mld-j</th>
<th>mld-k</th>
<th>...</th>
<th>...</th>
</tr>
</thead>
</table>

(3) create mld queue Qx
(4) push message ID
(5) put(mld-k,"msg") to ZHT
(6) updateMetaList(Qx,server-s)
(6) updateMetaList(Qx,server-s)

ZHT Server
Key | Value
--- | ---
mld-1 | value1

(7) update metadata list Qx
Qx
<table>
<thead>
<tr>
<th>server-1</th>
<th>server-s</th>
<th>...</th>
<th>...</th>
</tr>
</thead>
</table>

Message Queue
Qx
| mld-j | ... | ... | ...
|-------|-----|-----|-----|

:::
pop

• Keep the latency low
• steps to follow
  – Local Access
  – Random server
  – Last known server
  – Metadata list owner
    • Redirect to a message keeper
Pop

Client

(1) pop(Qx)

(2) dequeue message Id
(3) get(mld-k) from ZHT

Server

Metadata Lists

Q1
server i ... ...

Messageld Queues

Qx
mld-j mld-k ...

ZHT Server

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>mld-1</td>
<td>value1</td>
</tr>
</tbody>
</table>

(4) ”msg”
Load Balancing

- Load Balancing
  - Using a uniformly distributed hash function.
Features

• Order of messages
• Message delivery guarantee
• Persistence
• consistency and fault tolerance
• Multithreading
Communication cost analysis

- Push: 1 hop
- Pop: 0 – 3 hops
## Comparison

<table>
<thead>
<tr>
<th>Feature</th>
<th>Fabriq</th>
<th>Kafka</th>
<th>SQS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Persistence</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Delivery Guarantee</td>
<td>Exactly Once</td>
<td>At least Once</td>
<td>At least Once</td>
</tr>
<tr>
<td>Message Order</td>
<td>Inside Node</td>
<td>Inside Node</td>
<td>-</td>
</tr>
<tr>
<td>Replication</td>
<td>Customizable</td>
<td>Mirroring</td>
<td>3x</td>
</tr>
<tr>
<td>Shared Pool</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Batching</td>
<td>No (Future work)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Performance Evaluation

• FaBRiQ, Kafka, SQS
  – Latency
  – Throughput

• Measuring:
  – Push / produce
  – Pop / consume
Latency (CDF)
Throughput (short messages)

- Fabriq-push
- Fabriq-pop
- SQS-push
- SQS-pop
- Kafka-push
- Kafka-pop

Throughput (msgs/sec) vs. # Instances
Throughput (large messages)
## Duplicate messages

<table>
<thead>
<tr>
<th>Scale - #msgs</th>
<th>Kafka</th>
<th>SQS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 - 1000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2 - 2000</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4 - 4000</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>8 - 8000</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>16 - 16000</td>
<td>2</td>
<td>8-11</td>
</tr>
<tr>
<td>32 - 32000</td>
<td>4</td>
<td>8-15</td>
</tr>
<tr>
<td>64 - 64000</td>
<td>4</td>
<td>20-25</td>
</tr>
</tbody>
</table>
Future work

• Batching
Introduction

• Large Scale Task Execution
  – Run on distributed resources

  – Workloads
    • Tasks
      – More in number
      – Shorter in length

  – Requirements for high performance
    • Concurrency
    • Load Balance
    • System Utilization
State-of-the-art job schedulers

- **Centralized Master/Slaves architecture**
  - Scalability issues at petascale and beyond
  - Single point of failure
  - Example: SLURM, CONDOR, Falkon
State-of-the-art job schedulers

- Distributed Architectures
  - Hierarchical
    - several dispatchers in a tree-based topology
    - Example: Distributed Falkon, Dremel
  - Fully distributed
    - each computing node maintains its own job execution
    - Example: Sparrow, Omega, MATRIIX
- Common challenges
  - Complex Design and Implementatation
  - Load balancing
  - System utilization
Idea: Scheduling with Message Queues

• Idea: leverage Distributed Message Queues!
  – Mapping between Job Schedulers and Message Queues
Amazon AWS Cloud

• Amazon EC2
  • IaaS Cloud Service
    • Launch VMs and access remotely
    • Ability to launch more than 1000 instances

• Amazon Simple Queue Service (SQS)
  • Distributed message delivery queue
    • Highly scalable
    • Messages sent and read simultaneously
    • Reliable
      • Guarantees message delivery
        • At least once delivery
Proposed Work

• Use SQS as a task delivery component / task pool
• Decouple Clients and Workers
• Pushing vs. Pulling approach
  • Pushing
    • Local/global manager node needs to predict/decide
      • About the address of worker nodes
      • Underlying network topology
  • Pulling
    • No need to know about workers
    • Workers decide for themselves
• Load balancing
• System Utilization
CloudKon Architecture

- **MTC**
  - General format, running MTC tasks
  - Benefits:
    - Dynamic workforce
    - Non-blocking task submission

- **HPC**
  - Running HPC jobs with multiple tasks
Task consistency

- SQS only guarantees at least once delivery
- Some workloads require exactly once execution of tasks!
- Use DynamoDB to verify
- Use conditional write
  - Write if the task does not exist
  - Throw exception if exists
- Atomic operation
- Using a single operation, the checking is done
  - Minimize the communication overhead
Communication Cost

- Communication overhead is high on Cloud
  - Need to minimize the communication
- Message batching
  - Bundle tasks together to send
- Number of communications
  - Minimum possible number
Throughput (MTC)

- 1 to 1024 instances, 16K to 16.38M tasks
- MATRIX and Sparrow crashing on 256 instances
  - Too many sockets open on TCP connection
• Stable latency on different scales
  • Ranging from 90 ms to 104 ms
Efficiency – Homogenous Tasks

- MATRIX achieved better efficiency on shorter tasks
- Efficiency of MATRIX drops lower than CloudKon on 64 instances
- CloudKon is stable and scalable
Efficiency – Heterogenous Tasks

- Trace of a real MTC workload
- 2.07M tasks at the largest scale
- 1 milliseconds to 1 second tasks

CloudKon is more stable and scalable compared to the other two.
Efficiency of MATRIX and Sparrow drop lower than CloudKon on 64 instances.
Conclusion

- Design and implement simple yet effective distributed task execution framework
  - Using cloud services like SQS, DynamoDB
- Run on Public Cloud environment as an alternate resource
  - Optimum usage of cloud resources
- Outperforming other state of the art systems on larger scales
  - Sparrow 2013
  - MATRIX 2013