Luxun - A Persistent Messaging System Tailored for Big Data Collecting & Analytics

By William
http://bulldog2011.github.com

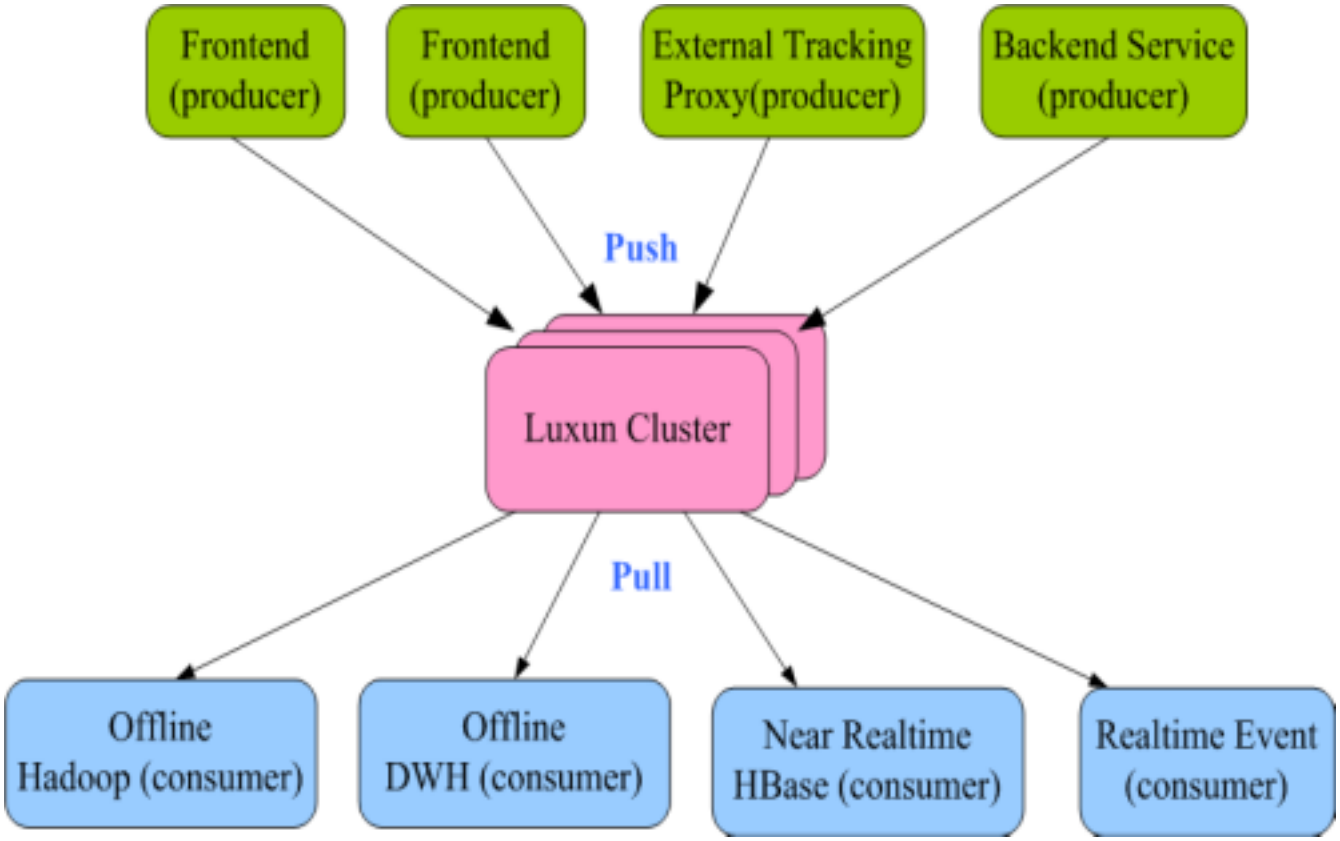
Performance Highlight

- On Single Server Grade Machine with Single Topic
  - Average producing throughput **> *100MBps***, peak **> *200MBps***
  - Average consuming throughput **> *100MBps***, peak **> *200MBps***
- In Networking Case,
  - Throughput only limited by network and disk IO bandwidth

Typical Big Data or Activity Stream

- Logs generated by frontend applications or backend services
- User behavior data
- Application or system performance trace
- Business, application or system metrics data
- Events that need immediate action

Unified Big Data Pipeline



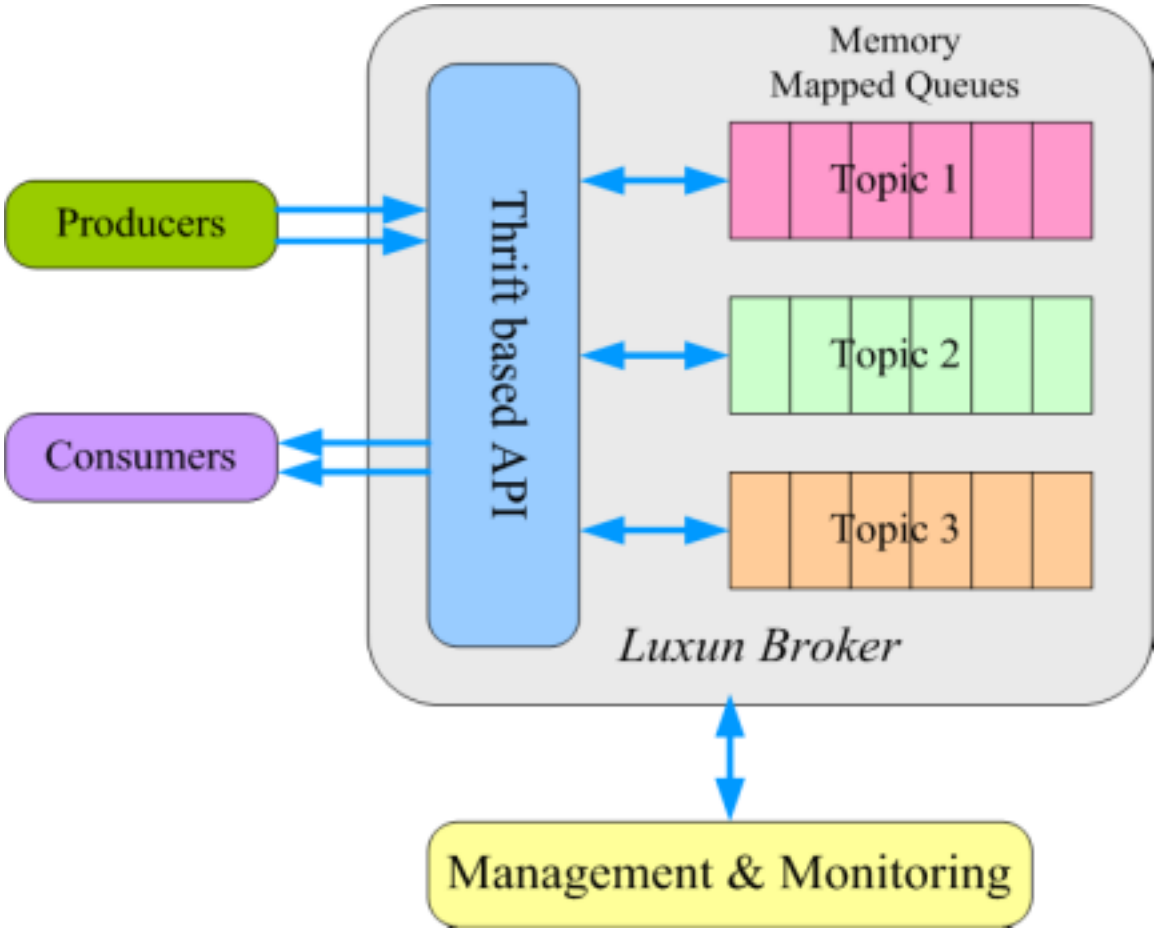
Luxun Design Objectives

- **Fast & High-Throughput**
  - Top priority, close to O(1) memory access
- **Persistent & Durable**
  - All data is persistent on disk and is crash resistant
- **Producer & Consumer Separation**
  - Each one can work without knowing the existence of the other
- **Realtime**
  - Produced message will be immediately visible to consumer
- **Distributed**
  - Horizontal scalable with commodity machines
- **Multiple Clients Support**
  - Easy integration with clients from different platforms, such as Java, C#, PHP, Ruby, Python, C++…
- **Flexible consuming semantics**
  - Consume once, fanout, can even consume by index
- **Light Weight**
  - Small footprint binary, no Zookeeper coordination

Basic Concepts

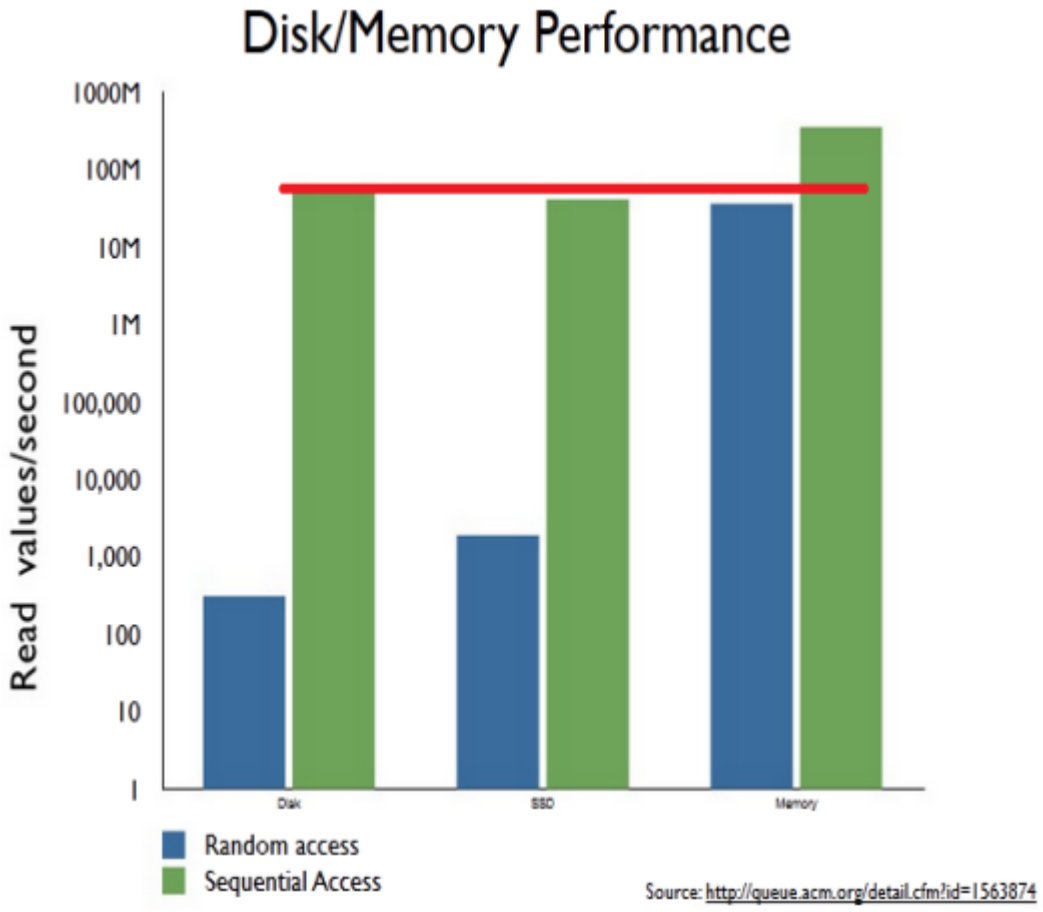
- **Topic**
  - Logically it's a named place to send messages to or to consume messages from, physically it's a persistent queue
- **Broker**
  - Aka Luxun server
- **Message**
  - Datum to produce or consume
- **Producer**
  - A role which will send messages to topics
- **Consumer**
  - A role which will consume messages from topics
- **Consumer Group**
  - A group of consumers that will receive only one copy of a message from a topic

Overall Architecture



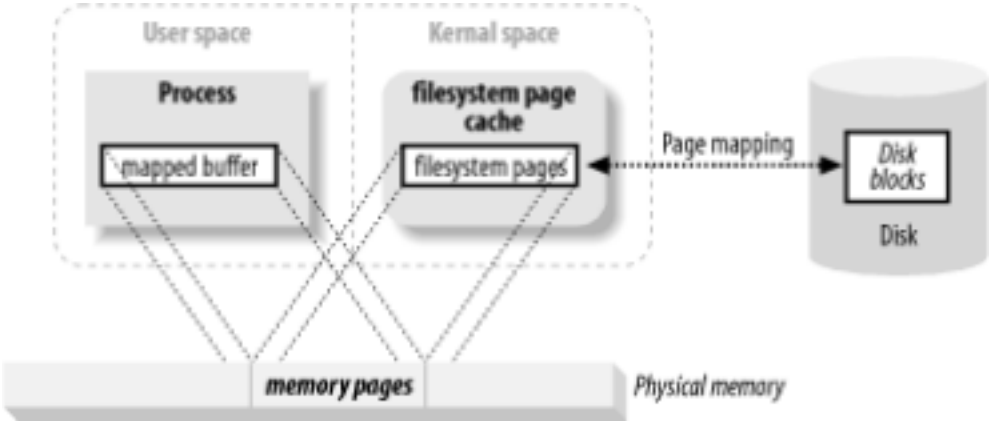
Core Principle

- Sequential disk read can be comparable to or even faster than random memory read



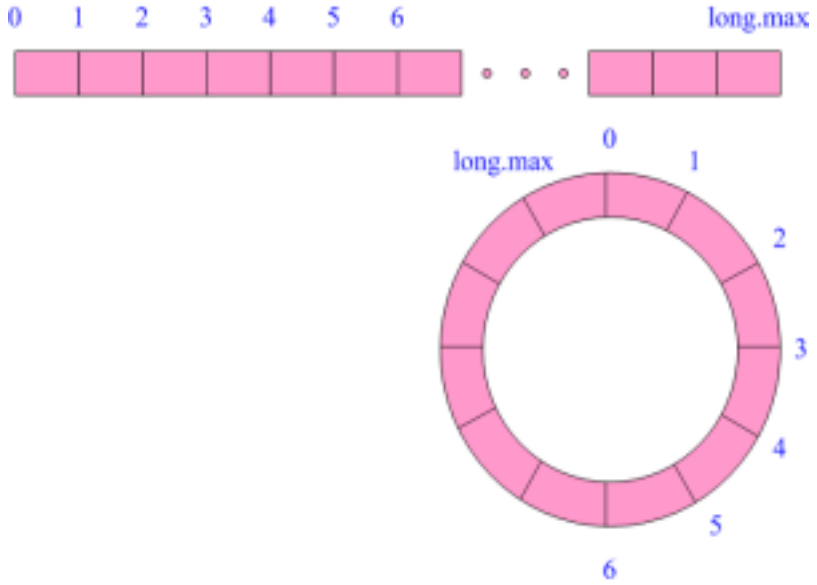
Core Technology – Memory Mapped File

- Map files into memory, persisted by OS
  - OS will be responsible to persist messages even the process crashes
- Can be shared between processes/threads
  - Produced message immediately visible to consumer threads
- Limited by the amount of disk space you have
  - Can scale very well when exceeding your main memory size
  - In Java implementation, does not use heap memory directly, GC impact limited

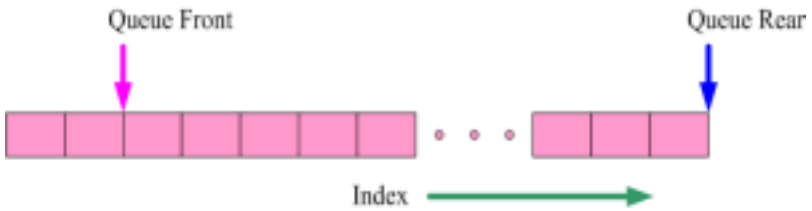


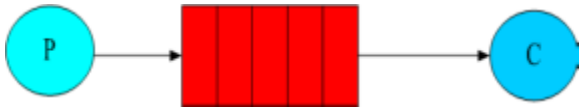
Persistent Queue – Logic View

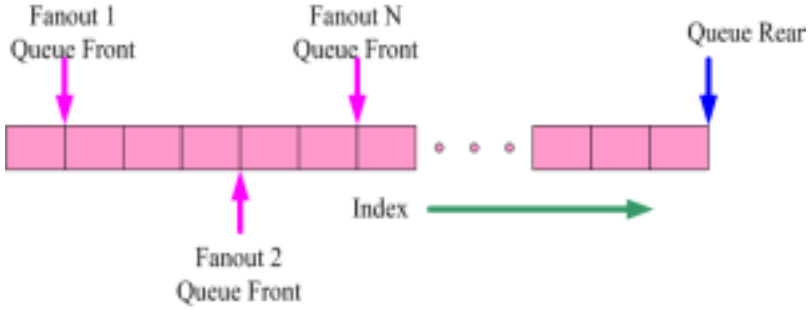
- Just like a big array or circular array, message appended/read by index

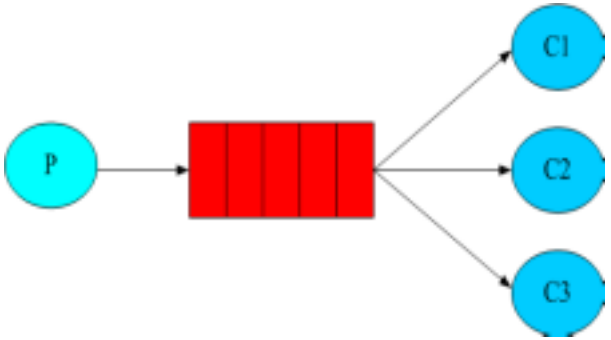


Persistent Queue – Consume Once & Fanout Queue



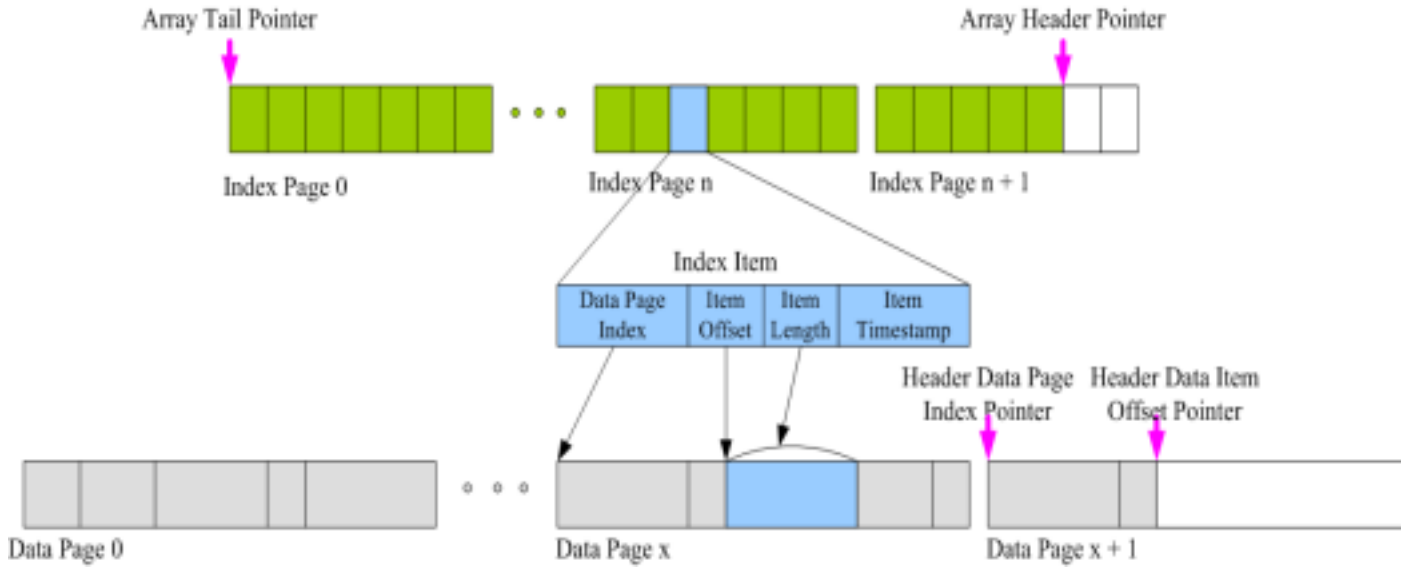






Persistent Queue – Physical View

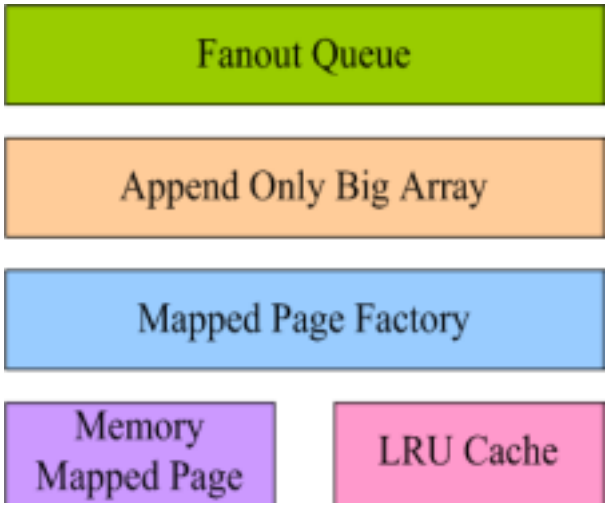
- Paged index file and data file



Persistent Queue - Concurrency

- Append operation is synchronized in queue implementation
- Read operation is already thread safe
- ***Array Header Index Pointer*** is a read/write barrier

Persistent Queue – Components View



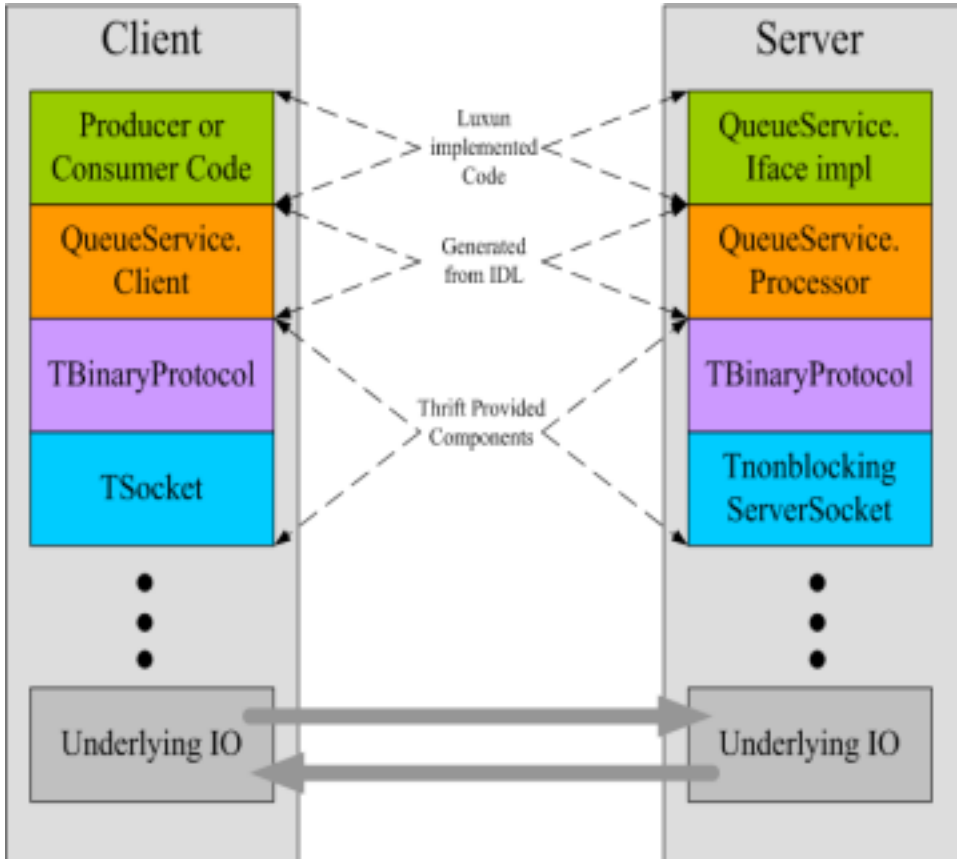
Persistent Queue – Dynamic View

- Memory Mapped Sliding Window
  - Leverage locality of rear append and front read access mode of queue

Communication Layer – Why Thrift

- **Stable & Mature**
  - Created by Facebook, used in Cassandra and HBase
- **High Performance**
  - Binary serialization protocol and non-blocking server model
- **Simple & Light-Weight**
  - IDL driven development, auto-generate client & server side proxy
- **Cross-Language**
  - Auto-generate clients for Java, C#, C++, PHP, Ruby, Python, …
- **Flexible & Pluggable Architecture**
  - Programming like playing with building blocks, components are replaceable as needed.

Communication Layer – Components View



Communication Layer – Luxun Thrift IDL

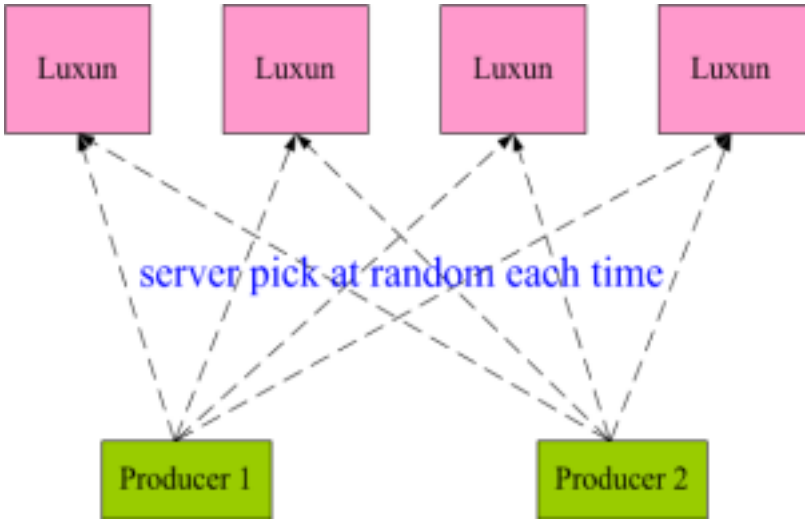
```
service QueueService {

    ProduceResponse produce(1: ProduceRequest produceRequest);

    oneway void asyncProduce(1: ProduceRequest produceRequest);

    ConsumeResponse consume(1: ConsumeRequest consumeRequest);

    FindClosestIndexByTimeResponse findClosestIndexByTime(1: FindClosestIndexByTimeRequest findClosestIndexByTimeRequest);

    DeleteTopicResponse deleteTopic(1: DeleteTopicRequest deleteTopicRequest);

    GetSizeResponse getSize(1: GetSizeRequest getSizeRequest);

}
```

Producer – The Interface

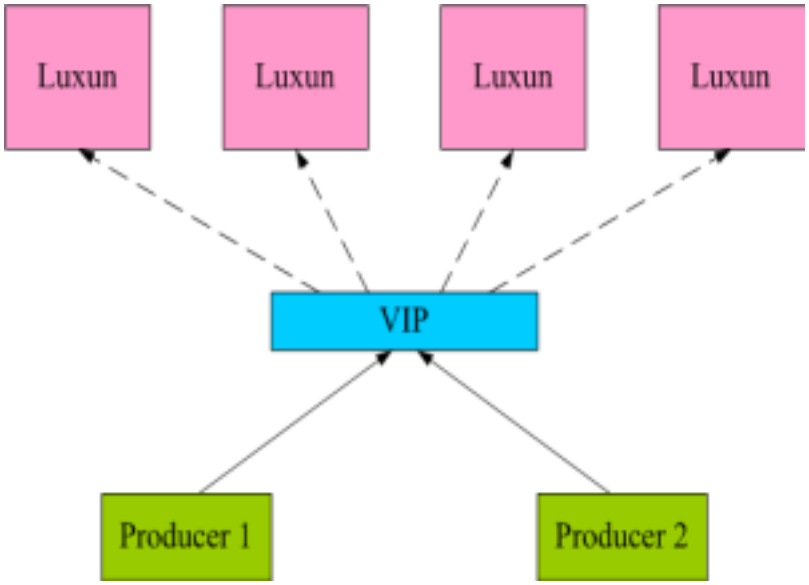
```java
/**
 * Producer interface
 *
 */
public interface IProducer<K, V> extends Closeable {

    void send(ProducerData<K, V> data) throws NoBrokersForTopicException;

}
```

```java
/**
 * Represents the data to be sent using the Producer send API
 *
 */
public class ProducerData<K, V> {

    /** the topic under which the message is to be published */
    private String topic;

    /** the key used by the partitioner to pick a broker */
    private K key;

    /** variable length data to be published as Luxun messages under topic */
    private List<V> data;

}
```

Producing Partitioning on Producer Side



Producing Partitioning through VIP

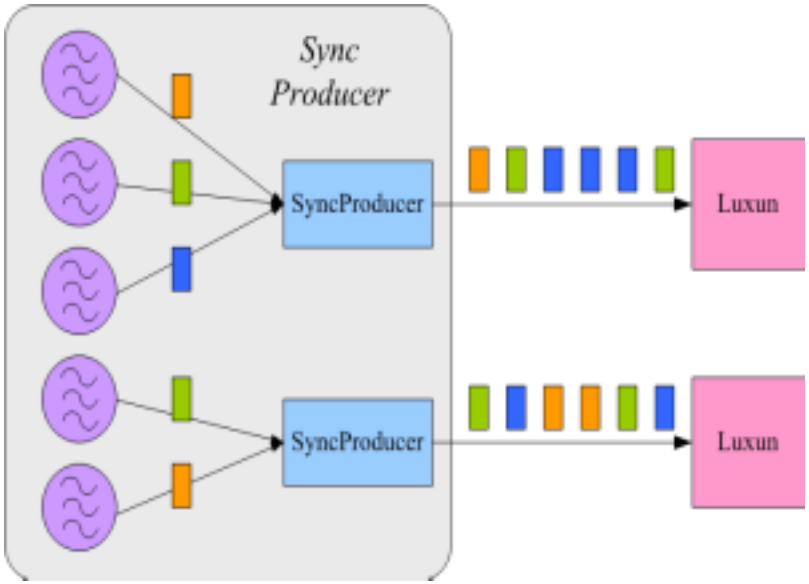


Producing Compression

- Current Support
  - 0 – No compression
  - 1 – GZip compression
  - 2 – Snappy compression
- Enable Compression for better utilization of
  - Network bandwidth
  - Disk space

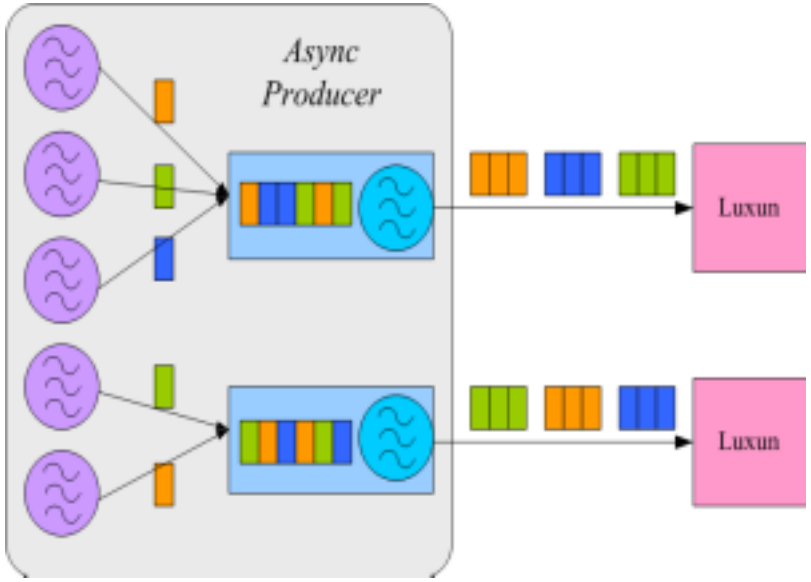
Sync Producing

- Better real-time, worse throughput
  - Use this mode only if real-time is the top priority



Async & Batch Producing

- Better Throughput, sacrifice a little real-time
  - Should be enabled whenever possible for higher throughput



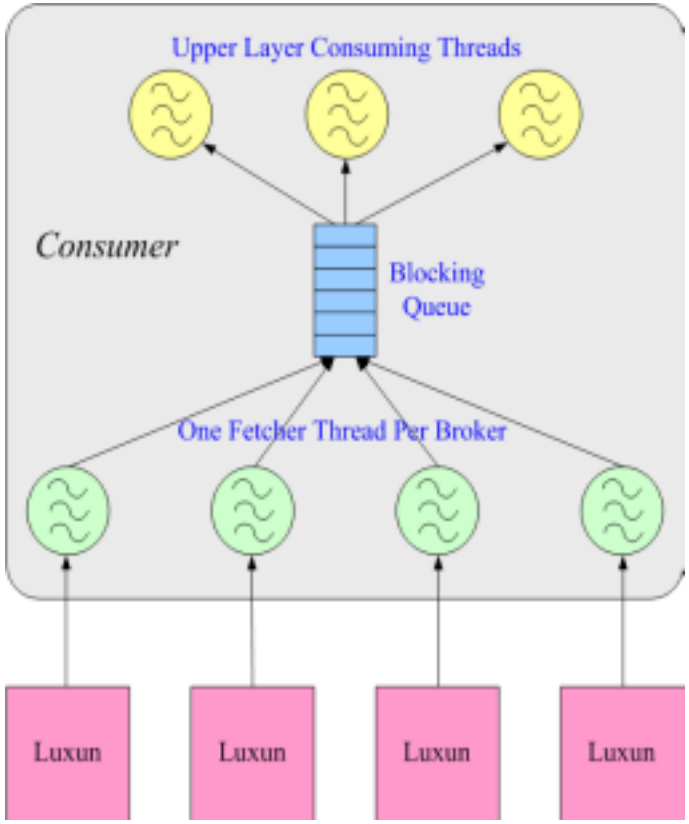
Simple Consumer

```java
/**
 * Consume by index
 *
 */
public List<MessageList> consume(String topic, long index, int fetchSize) throws TException



/**
 * Consume by fanoutId(aka consumer group name)
 *
 */
public List<MessageList> consume(String topic, String fanoutId, int fetchSize) throws TException
```

Advanced Stream Style Consumer

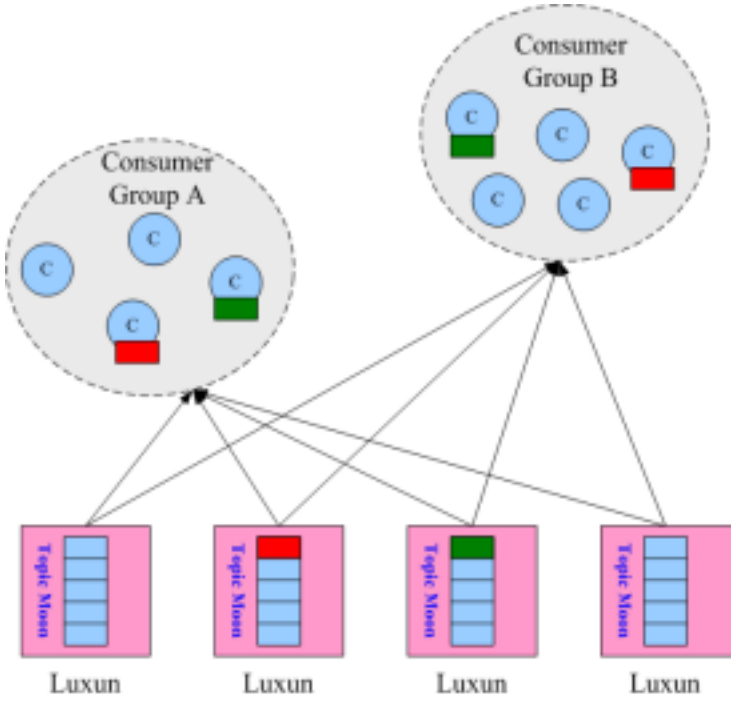
```java
/**
 * Factory interface for streaming consumer
 *
 * @author bulldog
 *
 */
public interface IStreamFactory extends Closeable {

    /**
     * Create a list of {@link MessageStream} for each topic
     *
     */
    <T> Map<String, List<MessageStream<T>>> createMessageStreams(//
            Map<String, Integer> topicThreadNumMap, Decoder<T> decoder);

    /**
     * Create a list of {@link MessageStream} for each topic with default Luxun message decoder
     *
     */
    <T> Map<String, List<MessageStream<Message>>> createMessageStreams(//
            Map<String, Integer> topicThreadNumMap);

    /**
     * Shut down the consumer
     */
    public void close() throws IOException;
}


public class MessageStream<T> implements Iterable<T>
```

Advanced Consumer Internals

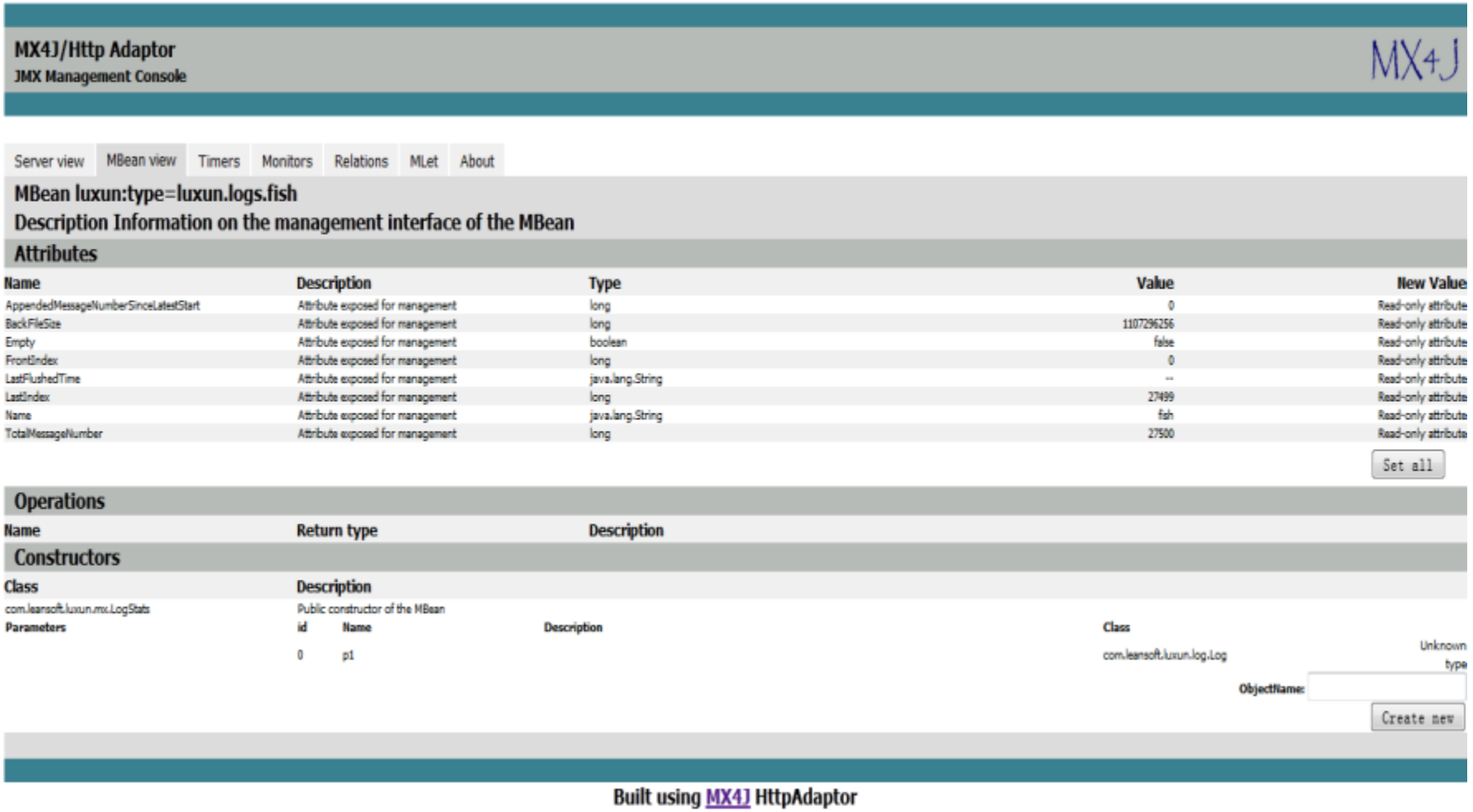


Consumer Group

- Within same group, consume once sematics
- Among different groups, fanout semantics



JMX Based Monitoring



Key Performance Test Observations

- On single machine, throughput is only limited by disk IO bandwidth
- In networking case, throughput is only limited by network bandwidth
  - 1Gbps network is ok, 10Gbps network is recommended
- **Not sensitive to** JVM heap setting,
  - Memory mapped file uses off-heap memory
  - 4GB is ok, >8GB is recommended
- Throughput > 50 MBps even on normal PC

Key Performance Test Observations Continue

- Performs good on both Windows and Linux platforms
- The throughput of async batch producing is ***order of magnitude better*** than sycn producing
- Flush on broker has negative impact on throughput, recommend to ***disable flush*** because of unique feature of memoy mapped file
- The throughput of one way not confirmed produce interface is ***3 times better*** than two way confirmed produce interface

Key Performance Test Observations Continue

- The overall performance will not change as number of topics increase, the throughput will be shared among different topics.
- Compression should be enabled for better network bandwidth and disk space utilization, Snappy has better efficiency than GZip.

Operation - Most Important Performance Configurations

- On broker
  - **Flush** has negative impact to performance, recommend to turn it off.
- On producer side
  - Compression
  - Sync vs async producing
  - Batch size
- On consumer side
  - Fetch size

Operation – Log Cleanup Configurations

- Expired log cleanup can be configured with:
  - ***log.retention.hours*** – old back log page files outside of the retention window will be deleted periodically.
  - ***log.retention.size*** – old back log page files outside of the retention size will be deleted periodically

Luxun vs Apache Kafka – the Main Difference

- Luxun is inspired by Kafka, however, they have following main differences:

	Luxun	**Kafka**
Persistent Queue	Memory Mapped File	Filesystem & OS page cache
Communcation layer	Thrift RPC	Custom NIO and messaging protocol
Message access mode	Index Based	Offset based
Distribution for scalability	Random distribution	Zookeeper for distributed coordination
Partitioning	Only on server level	Partition within a topic

Credits

- Luxun borrowed design ideas and adapted source from following open source projects:
  - ***Apache Kafka*** - http://kafka.apache.org/index.html
  - ***Jafka*** - https://github.com/adyliu/jafka
  - ***Java Chronicle*** - https://github.com/peter-lawrey/Java-Chronicle
  - ***Fqueue*** - http://code.google.com/p/fqueue/
  - ***Ashes-queue*** - http://code.google.com/p/ashes-queue/
  - ***Kestrel*** - https://github.com/robey/kestrel

Next Steps

- Add a sharding layer for distribution and replication
- More clients, C#, PHP, Ruby, Python, C++, etc
- Big data apps, such as centralized logging, tracing, metrics and events systems based on Luxun.

Origin of the Name

- In memorial of LuXun, a great Chinese writer

Source, Docs and Downloadable

https://github.com/bulldog2011/luxun