Building a Scalable Messaging Fabric with JRuby and Storm

R. Tyler Croy - Ian Smith

Lookout, Inc.

August 1, 2014

Your hosts

R. Tyler Croy

@agentdero - github.com/rtyler

Ian Smith

@metaforgotten - github.com/ismith

Lookout

@lookouteng - github.com/lookout

What/Why is Storm

Traditional Message Infrastructures

Redis-based

Resque - Sidekiq - BLPOP/RPUSH

“Enterprise Message Queues”

ActiveMQ - RabbitMQ - HornetQ

Traditional Workers

```
loop { work(consume()) }
```

incredibly complex

Messaging Requirements

The must-haves

- Reliable message delivery
- One-to-many message delivery
- Scalability

Kafka

tl;dr

more gooder

Storm Basics

tuples

the currency of Storm

spouts

your input

bolts

basic unit of operation

topology

a directed graph of plumbing metaphors

The Storm Cluster

zookeeper

discovery and configuration

nimbus nodes

coordinate it

worker nodes

doing things with input

doing the work

worker process - executors - tasks

What/Why is Storm

Message Design

Not everybody will be Ruby

message definition should be cross-platform

Consistency is important

leave your JSON at home

Protocol Buffers

Thrift

Avro

Home-grown

Working with everything else

Metron

```
package metron;

message Event {
  required string channel  = 1;
  required bytes data      = 2;
  optional string tstamp   = 3;
  optional string uuid     = 4;
  optional string event_id = 5;
}
```

Storm and your applications

Who owns the data store?

perhaps the most important question

Topologies talking to data stores

feasible but requires some footwork

Topologies making RPC calls

better!

Phew

Developing Storm Topologies with JRuby

You could use ShellBolts - a bolt and a script in a non-JVM language (pipes & JSON)

But! Ruby *is* a JVM language, via JRuby

RedStorm
https://github.com/colinsurprenant/redstorm

Native, Trident & DSL

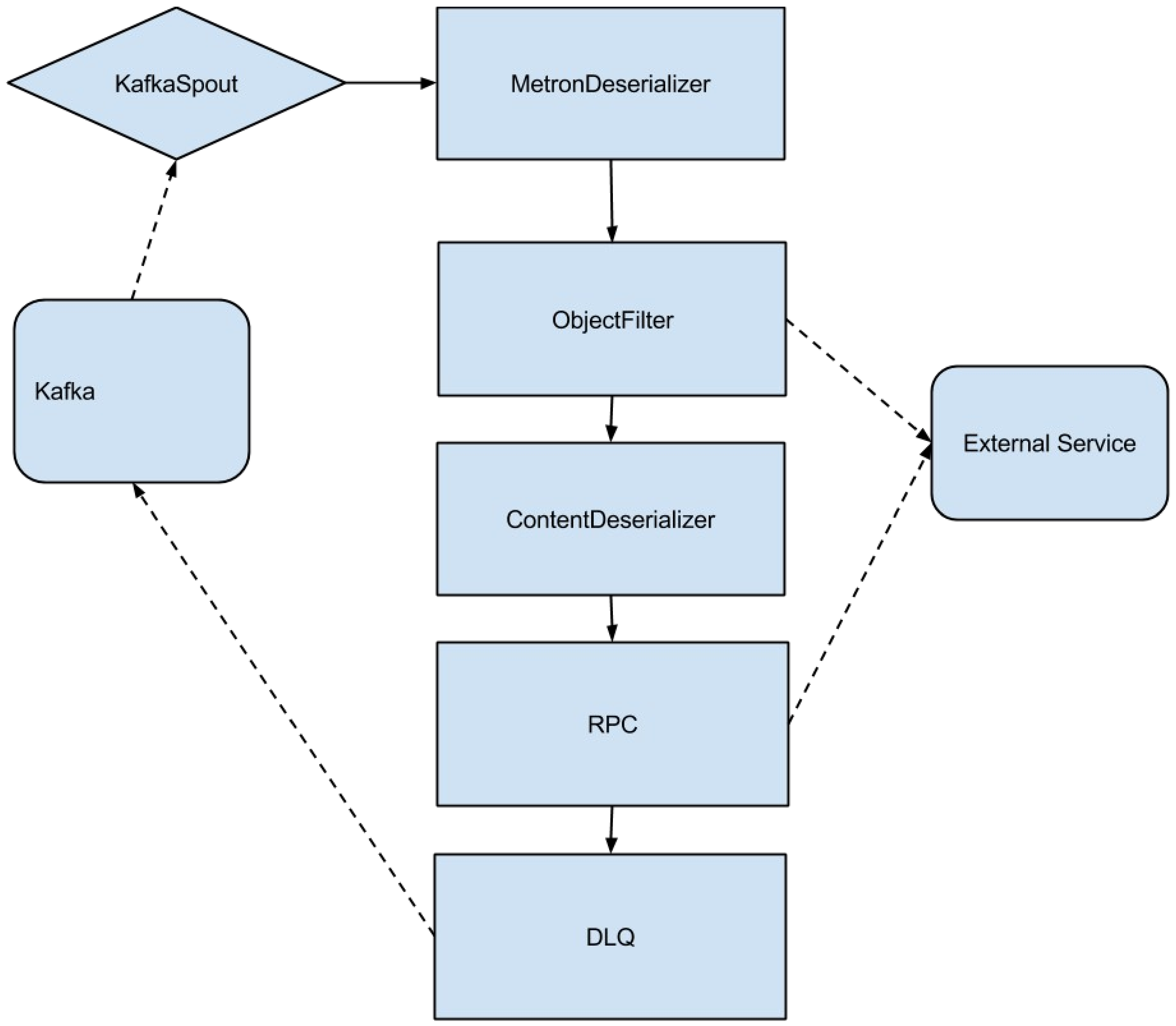
```
class HelloWorldBolt < RedStorm::DSL::Bolt
  on_receive :emit => false do |tuple|
    log.info(tuple[:word])
  end
end
```

```
class HelloWorldTopology
  spout HelloWorldSpout do
    output_fields :word
  end

  bolt HelloWorldBolt do
    source HelloWorldSpout, :shuffle
  end

  bolt AnotherBolt do
    source HelloWorldBolt, :shuffle
  end
end
```

Topology Design


KafkaSpout
MetronDeserializer
ObjectFilter
Kafka
External Service
ContentDeserializer
RPC
DLQ

```
class OurTopology < RedStorm::DSL::Topology
  def self.topology_name
    “#{self.name}_#{commit_hash}”
  end
```

```
class OurTopology < RedStorm::DSL::Topology
  spout_config = SpoutConfig.new(...)

  spout KafkaSpout, [spout_config] do
    output_fields :bytes
  end
```

```
class OurTopology < RedStorm::DSL::Topology
  bolt ContainerDeserializerBolt do
    source KafkaSpout, :shuffle
  end

  bolt ObjectFilterBolt do
    source ContainerDeserializerBolt
  end

  # ...
```

```ruby
class OurTopology < RedStorm::DSL::Topology
  submit_options do |env|
    # ...
  end

  configure self.topology_name do |env|
    # ...
  end
end
```

```
class OurBolt < RedStorm::DSL::Bolt
  output_fields :bytes, :dlq

  on_init do
    @connection = # ...
  end

  on_receive do |tuple|
   ...
  end
end
```

Lessons Learned / Pitfalls

Make sure your messages aren’t mangled

Lots of logging. Try isolating spouts from bolts.

`tuple[:foo]`

```
tuple[:foo]

tuple.value(:foo).to_s
```

```
tuple[:foo]

tuple.value(:foo).to_s

String.from_java_bytes(tuple.value(:foo))
```

The DSL doesn’t subclass directly

Use methods, not blocks

```ruby
class HelloWorldBolt < RedStorm::DSL::Bolt
  on_receive :emit => false do |tuple|
    log.info(tuple[:word])
  end
end
```

```
class GenericBolt < RedStorm::DSL::Bolt
  def on_receive(tuple)
    log.info(tuple[:word])
  end
end
```

```
# Topology-specific subclass
class HelloWorldBolt < GenericBolt
  on_receive :on_receive

  def log
   # topology-specific logging code
  end
end
```

Make sure you ack post-emit

You might emit multiple tuples

One topic is one topic

“Ok, you gave me a thing … what is it?”

Shared behavior

```
class Lookout::Bolt < RedStorm::DSL::Bolt
  # Wrap these calls in an exception handler

  def execute(*args)
    Raven.capture { super }
  end

  # Same with #prepare, #cleanup
end
```

```ruby
class Lookout::Bolt < RedStorm::DSL::Bolt
  def log
    # Custom logging
  end
end
```

Test with a cluster

Submit in inactive mode

Reduce downtime

Design a holistic system

Very few pieces operate independently

Questions?

Thanks