Scalable Realtime Architectures in Python

Jim Baker

jim.baker@{python.org, rackspace.com} @jimbaker

Overview

- Key ideas in implementing scalable realtime architectures: **partitioning** and **fault tolerance**
- Talk focuses on implications for architectures on Storm, but ideas applicable to other tools
- Such architectures have increasing importance, especially need to be more responsive
- Python code will be shown!

About me

- Core developer of Jython
- Co-author of *Definitive Guide to Jython* from Apress
- Software developer at Rackspace
- Formerly, founding team member, Ubuntu Juju
- Lecturer in CS at Univ of Colorado at Boulder
- Leader, Boulder/Denver Storm Users meetup

Example realtime architectures

Some examples of what might you want to build, **at scale**:

- Realtime aggregation - map/reduce, but current, event-at-a-time, not batched - not even in hourly batches
- Realtime dashboards - pulling together relevant info on a service/customer/etc, push to distributed file system (especially in cloud)
- Realtime decision making - to control or optimize some system through a feedback loop

Common realtime characteristics

- Consume stream(s) of data as events
- Event-oriented - as an event occurs possibly compute, act, and/or emit to output
- Best effort low latency - milliseconds to seconds. Not hours.
- Often called complex event processing or stream processing
- You might have written your own!

(Hard realtime means **provable bounded latency**; we are not doing that here!)

What you used to be able to do

Realtime analysis could be as simple as writing a Unix pipeline

```
$ a *.log | b | c | d > out.txt
```

with nice properties:

- Composition of reusable filters by using the pipe operator
- Reliability - if pipeline fails, rerun against data files

Problems with writing your own

- Some elided details in previous command line example (tailing, rotation, record format)
- Support for common ops: joins, windows, aggregation
- But much more important: **how to scale**

Example: implementing global alarms

Issues with homegrown code or using frameworks like Esper:

- Problem: need to ensure all relevant events about customer is in one place
- Also let customers define policies about what is an alarm
- Then run some computation using policy and events $\Rightarrow$ produce alarms
- Sharding is only a partial solution - what if you have already sharded on some other key?

Observations

- Small problems are comparatively easy
- $\Rightarrow$ So make large problems smaller with *divide & conquer*
- $\Rightarrow$ Need horizontal scaling
- $\Rightarrow$ But failure becomes increasingly likely at scale
- $\Rightarrow$ And distributed coordination is tough!

Cannot easily sweep away problems

- Not sufficient to simply add a box labeled "ZooKeeper" on your architecture diagram
- Even if ZK is awesome!
- Need to work with managing failure - ZK supports distributed recovery by tracking configuration, possibly ephemeral and linked to hearbeats
- Don't even ask me about supporting distributed locks ;)

Enter Storm

- Lingo: spouts (event sources), bolts (process events), topology (connect together spouts & bolts)
- Strong support for partitioning and fault tolerance
- Written in Clojure, exposes a Java API to any JVM language
- Uses ZooKeeper internally
- Part of Apache incubator program

Other open source scalable streaming systems

Also Apache incubator projects:

- UC Berkeley's Spark Streaming - mini-batch approach, written in Scala
- LinkedIn's Samza - message processor companion to Kafka written in Scala
- Yahoo's S4 - written in Java, seems to be crowded out

Partitioning

- Storm lets you partition streams, so you can break down the size of your problem
- If node running your code fails, Storm will restart it

Topology and its invariants

- Spouts source events, providing for reliable tracking - notified when the event is fully acked, or it has failed
- Bolts consume events, possible emitting to downstream bolts
- Topology is a **directed acyclic graph** for event routing
- Topology also describes how many nodes of a spout/bolt
- Storm ensures an **invariant** - the number of nodes for spouts and bolts is held constant
- Failure is easy - no rebalancing, just need to restart node

Computation locality

- Storm routes events consistently to specific node for a bolt
- Use field grouping to route (map!) all events of a given key to a node for some bolt
- **Q**: What possibilities do you have if **all current data** about some key - a customer - is in one place?
- Of course, you might have many such customers on a given node

Invariant implication

Event consistently will be **on this node and only this node**

Other routing possibilities

- Random shuffling for load balancing - good for ingest
- Global grouping - ensure all events go to one node

Retries

- Storm tracks success and failure of events being processed efficiently through a batching scheme and other cleverness
- Your code can then choose to retry as necessary, supporting **at-least-once** event processing
- Your code must then tolerate retry - be **idempotent** with appropriate merge function

Storm also supports exactly-once event processing semantics.

Handling retries

Retries can be readily handled - this might be your merge function

```
seen = set()
for record in stream:
  k = uniquifier(record)
  if k not in seen:
    seen.add(k)
    process(record)
```

Important caveat

- Any such real usage must not attempt to store all observations (first, *download the Internet!* ;),
- Use some sort of window mechanism to manage
- No built-in query language for this - Storm just provides routing and underlying retry support
- But can address with tools like Summingbird DSL or various libraries - or make your own

Managing retries

- Spouts are responsible for managing retries
- Consensus: use Storm's ZooKeeper cluster to track (in some namespace)
- Updating persistent ZK znodes is readily retryable - no out-of-band recovery required
- Handshaking - as events are acked, periodically record in ZK
- Just don't record any tracking in ZK per event!
- Or if available use upstream source's built-in tracking
- More advice: use Apache Curator for your client

Kafka handshaking

- Use low-level consumer API for given topic(s)
- Track message offsets as read
- Write to ZK in batches as acks are returned, rereading messages as necessary with failures
- Possible problems? of course - nature of eventually consistent systems - will converge, just a question of when
- For more insight, look at ACID 2.0 model and eventual consistency - associative, commutative, idempotent, distributed
- Latency is best effort in Storm, so write good code, run on (reasonably) reliable clusters

Example spouts

- Message queues - AMPQ (example Rabbit), Kafka, JMS
- Feeds like Twitter
- Pull in from Cassandra
- What would you like to use? Straightforward to implement
- Can support either push or pull - Storm will ask for events periodically, but can emit at any time to the corresponding collector
- Needs to manage state with respect to upstream sources
- Use topology sizing invariant

Example bolts

- Realtime dashboards - consolidate events from various sources, write to Cloud Files, send notification via pusher.com like service
- Realtime decision making - pull together all of possibly contradictory info about an auto scale group - scheduled scale up! policy scale down! replace lost servers! - and converge on action
- Realtime aggregation - apply map/reduce style aggregation, but with windows

Running Python on Storm with Clamp

- Python is a natural fit for writing Storm code
- Clamp provides support for wrapping Python classes so can be used directly by Java
- Or other JVM languages, like Storm's actual implementation in Clojure

Clamp a Python class

Example: module clamped containing the BarClamp class:

```
from java.io import Serializable
from java.util.concurrent import Callable
from clamp import clamp_base

BarBase = clamp_base("bar")

class BarClamp(BarBase, Callable, Serializable):
  def call(self):
    return 42
```

setup.py

Describe what modules you need to clamp:

```
import ez_setup
ez_setup.use_setuptools()
from setuptools import setup, find_packages
from clamp.commands import clamp_command

setup(
  name = "clamped",
  version = "0.1",
  packages = find_packages(),
  install_requires = ["clamp>=0.4"],
  clamp = { "modules": ["clamped"] },
  cmdclass = { "install": clamp_command }
)
```

Build with setuptools support

Construct uber jar:

```
$ jython setup.py install singlejar
```

Use from Java

Now directly import Python classes into Java code!

```
import bar.clamped.BarClamp;

public class UseClamped {

  public static void main(String[] args) {
    BarClamp barclamp = new BarClamp();
    try {
      System.out.println("BarClamp: " + barclamp.call
    } catch (Exception ex) {
      System.err.println("Exception: " + ex);
    }
  }
}
```

Python spout

Basic outline of your code - can readily write in Python:

```
class MonitoringSpout(BaseRichSpout, MyBase):
    def open(self, conf, context, collector):
        # connect to Kafka, AtomHopper, etc

    def nextTuple(self):
        # read, parse, and emit event from upstream
        self._collector.emit(event, offset)

    def fail(self, offset):
        # resend event some number of times,
        # else send to error stream

    def ack(self, offet):
        # batch successful offsets, write to ZK
```

Python bolt

Set up topology such that all events are field grouped by customer:

```
class GlobalAlarmBolt(BaseRichBolt, MyBase):
    def prepare(self, conf, context, collector):
        # setup for any subsequent computations

    def execute(self, t):
        # read input stream
        # get customer from event tuple
        # compute in a window for given customer
        # ack and optionally emit more events

    def declareOutputFields(self, declarer):
        declarer.declare(Fields(...))
```

Conclusions

- Storm can let you horizontally scale out your realtime architecture
- Must carefully consider the implications of **partitioning** and **fault tolerance**
- Choose your favorite language implemented on the JVM - Clojure, Groovy, Java, JRuby, **Jython**, Scala, . . .
- Your development strategies should work with Storm - TDD, CI, and more
- Have fun!

Questions

Any questions?

Now or for later:

`jim.baker@{python.org, rackspace.com}`
`@jimbaker`

Talk available at github.com/jimbaker/talks