



For confidence, click here.

# Apache Storm 0.9 basic training

Michael G. Noll, Verisign

[mnoll@verisign.com](mailto:mnoll@verisign.com) / [@miguno](https://twitter.com/miguno)

July 2014

# Storm?

- **Part 1: Introducing Storm**
  - “Why should I stay awake for the full duration of this workshop?”
- **Part 2: Storm core concepts**
  - Topologies, tuples, spouts, bolts, groupings, parallelism
- **Part 3: Operating Storm**
  - Architecture, hardware specs, deploying, monitoring
- **Part 4: Developing Storm apps**
  - Bolts and topologies, Kafka integration, testing, serialization, example apps, P&S tuning
- **Part 5: Playing with Storm using Wirbelsturm**
- **Wrapping up**

# NOT covered in this workshop (too little time)

- Storm Trident

- High-level abstraction on top of Storm, which intermixes high throughput and stateful stream processing with low latency distributed querying.
  - Joins, aggregations, grouping, functions, filters.
  - Adds primitives for doing stateful, incremental processing on top of any database or persistence store.
- Has consistent, exactly-once semantics.
- Processes a stream as small batches of messages
  - (cf. Spark Streaming)

- Storm DRPC

- Parallelizes the computation of really intense functions on the fly.
- Input is a stream of function arguments, and output is a stream of the results for each of those function calls.

# Part 1: Introducing Storm

# Overview of Part 1: Introducing Storm

- Storm?
- Storm adoption and use cases in the wild
- Storm in a nutshell
- Motivation behind Storm

# Storm?



- “Distributed and fault-tolerant real-time computation”
- <http://storm.incubator.apache.org/>
- Originated at BackType/Twitter, open sourced in late 2011
- Implemented in Clojure, some Java
- 12 core committers, plus ~ 70 contributors

<https://github.com/apache/incubator-storm/#committers>  
<https://github.com/apache/incubator-storm/graphs/contributors>

# Storm adoption and use cases

- **Twitter:** personalization, search, revenue optimization, ...
  - [200 nodes, 30 topos, 50B msg/day, avg latency <50ms, Jun 2013](#)
- **Yahoo:** user events, content feeds, and application logs
  - [320 nodes \(YARN\), 130k msg/s, June 2013](#)
- **Spotify:** recommendation, ads, monitoring, ...
  - [v0.8.0, 22 nodes, 15+ topos, 200k msg/s, Mar 2014](#)
- Alibaba, Cisco, Flickr, PARC, WeatherChannel, ...
  - [Netflix is looking at Storm and Samza, too.](#)

<https://github.com/nathanmarz/storm/wiki/Powered-By>

# “A Storm in a Nutshell”





Engineers can solve tricky problems  
...as long as we have the right tools.







But without shooting ourselves in the foot.

# The Motivation of Storm

*“Show me your **code** and conceal your data structures, and I shall continue to be mystified. Show me your **data structures**, and I won't usually need your code; it'll be obvious.”*

-- Eric S. Raymond, The Cathedral and the Bazaar

# Data

# WordCount example

(1.1.1.1, "foo.com")  
(2.2.2.2, "bar.net")  
(3.3.3.3, "foo.com")  
(4.4.4.4, "foo.com")  
(5.5.5.5, "bar.net")

DNS queries

?

( ("foo.com", 3)  
("bar.net", 2) )

Top queried  
domains



# Functional Programming

( (1.1.1.1, "foo.com")  
(2.2.2.2, "bar.net")  
(3.3.3.3, "foo.com")  
(4.4.4.4, "foo.com")  
(5.5.5.5, "bar.net") )

DNS queries



("foo.com", "bar.net", "foo.com",  
"foo.com", "bar.net")

*f*



{"bar.net" -> 2, "foo.com" -> 3}

*g*



( ("foo.com", 3)  
("bar.net", 2) )

*h*

$h(g(f(\text{data})))$

→  $\lambda$ -calculus

$\lambda$  here



# Clojure

- Is a dialect of Lisp that targets the JVM (and JavaScript)
  - clojure-1.5.1.jar

# Wait a minute – LISP??

```
(me? (kidding (you (are))))
```

Yeah, those parentheses are annoying. *At first.*

Think: Like Python's significant whitespace.

# Clojure

- Is a dialect of Lisp that targets the JVM (and JavaScript)
  - clojure-1.5.1.jar
- "Dynamic, compiled programming language"
  - Predominantly functional programming
- Many interesting characteristics and value propositions for software development, notably for concurrent applications
- **Storm's core is implemented in Clojure**
  - And you will see why they match so well.

## Previous WordCount example in Clojure

*h*                      *g*                      *f*

(sort-by val > (frequencies (map second queries)))

Alternative, left-to-right syntax with `->>`:

```
(->> queries (map second) frequencies (sort-by val >))
```

```
$ cat input.txt | awk | sort # kinda
```



# Clojure REPL

```
user> queries
```

```
((("1.1.1.1" "foo.com") ("2.2.2.2" "bar.net")  
  ("3.3.3.3" "foo.com") ("4.4.4.4" "foo.com")  
  ("5.5.5.5" "bar.net")))
```

```
user> (map second queries)
```

```
("foo.com" "bar.net" "foo.com" "foo.com" "bar.net")
```

```
user> (frequencies (map second queries))
```

```
{"bar.net" 2, "foo.com" 3}
```

```
user> (sort-by val > (frequencies (map second queries)))
```

```
(["foo.com" 3] ["bar.net" 2])
```

# Scaling up

Clojure, Java, <your-pick> can turn the previous code into a multi-threaded app that utilizes all cores on your server.



But what if even a very big machine is not enough for your Internet-scale app?



And remember.



Enter:



# Part 2: Storm core concepts

# Overview of Part 2: Storm core concepts

- A first look
- Topology
- Data model
- Spouts and bolts
- Groupings
- Parallelism

## A first look

Storm is **distributed** FP-like processing of data streams.

Same idea, many machines.  
(but there's more of course)



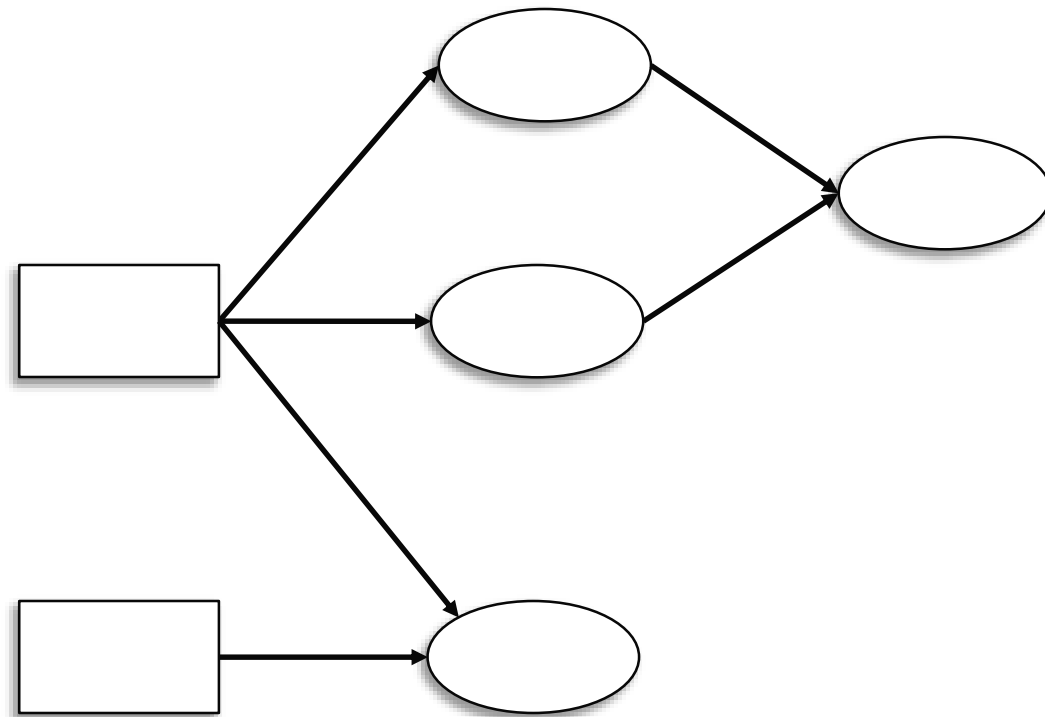
# Overview of Part 2: Storm core concepts

- A first look
- **Topology**
- Data model
- Spouts and bolts
- Groupings
- Parallelism

A topology in Storm wires  
**data** and *functions* via a **DAG**.

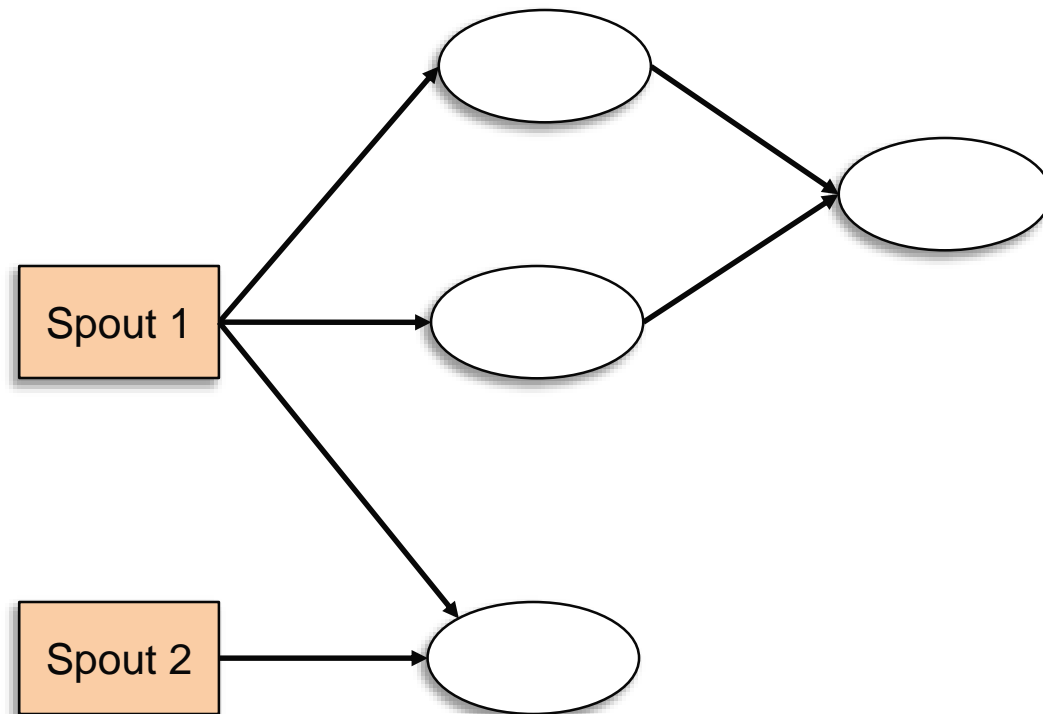
Executes on many machines  
like a MR job in Hadoop.

# Topology



# Topology

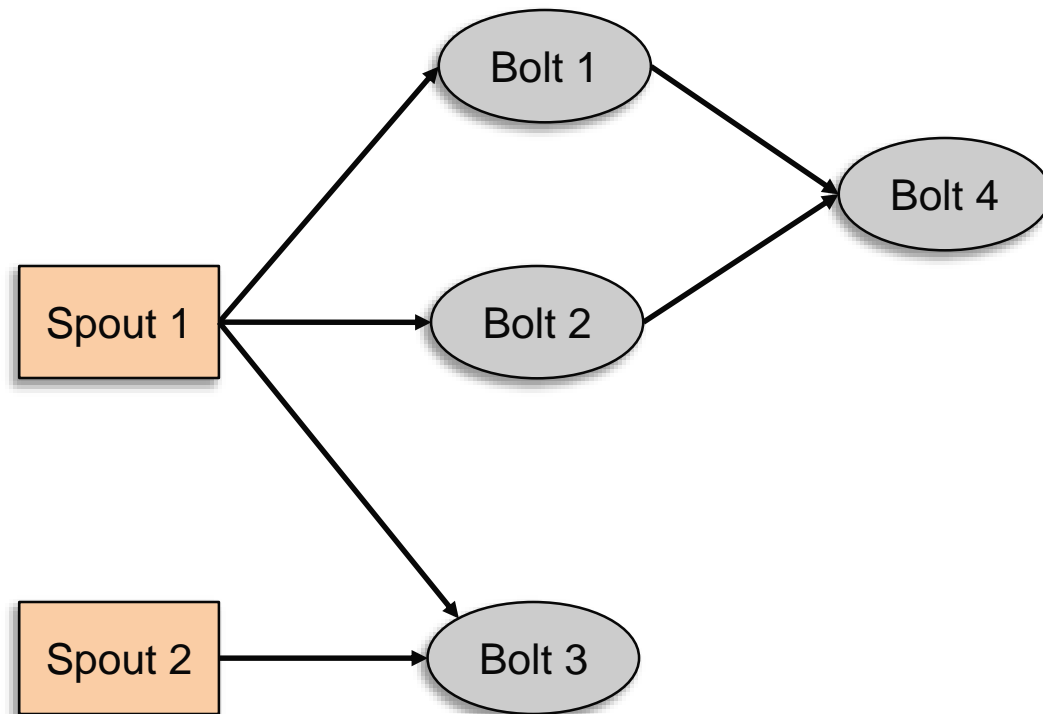
data



# Topology

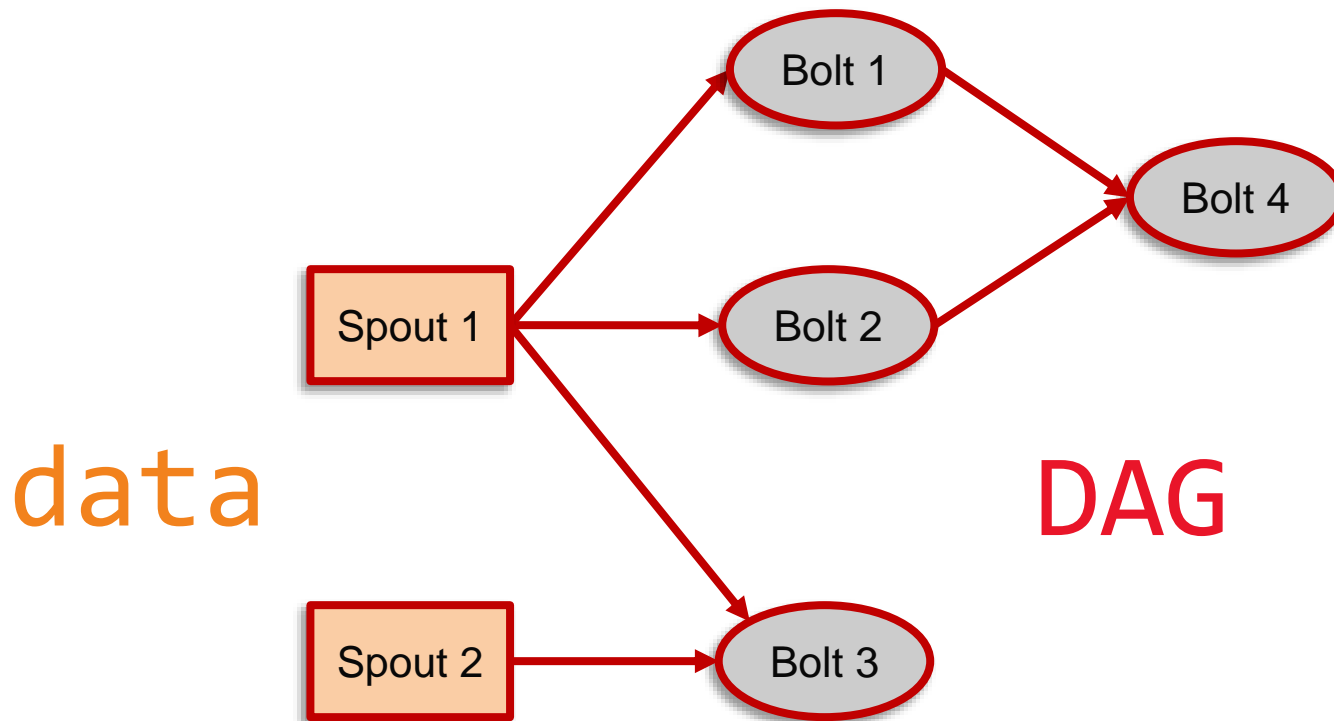
*functions*

data

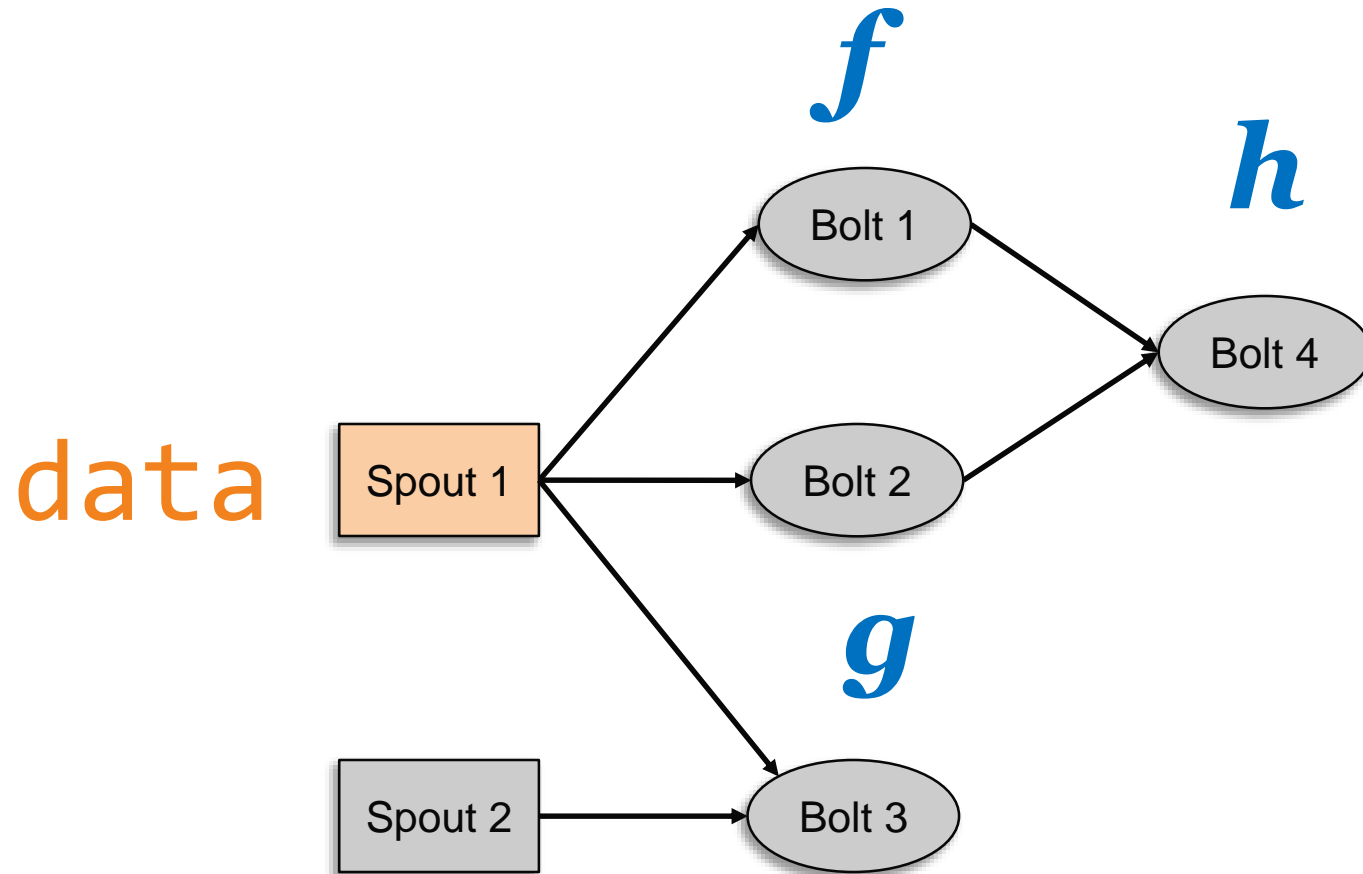


# Topology

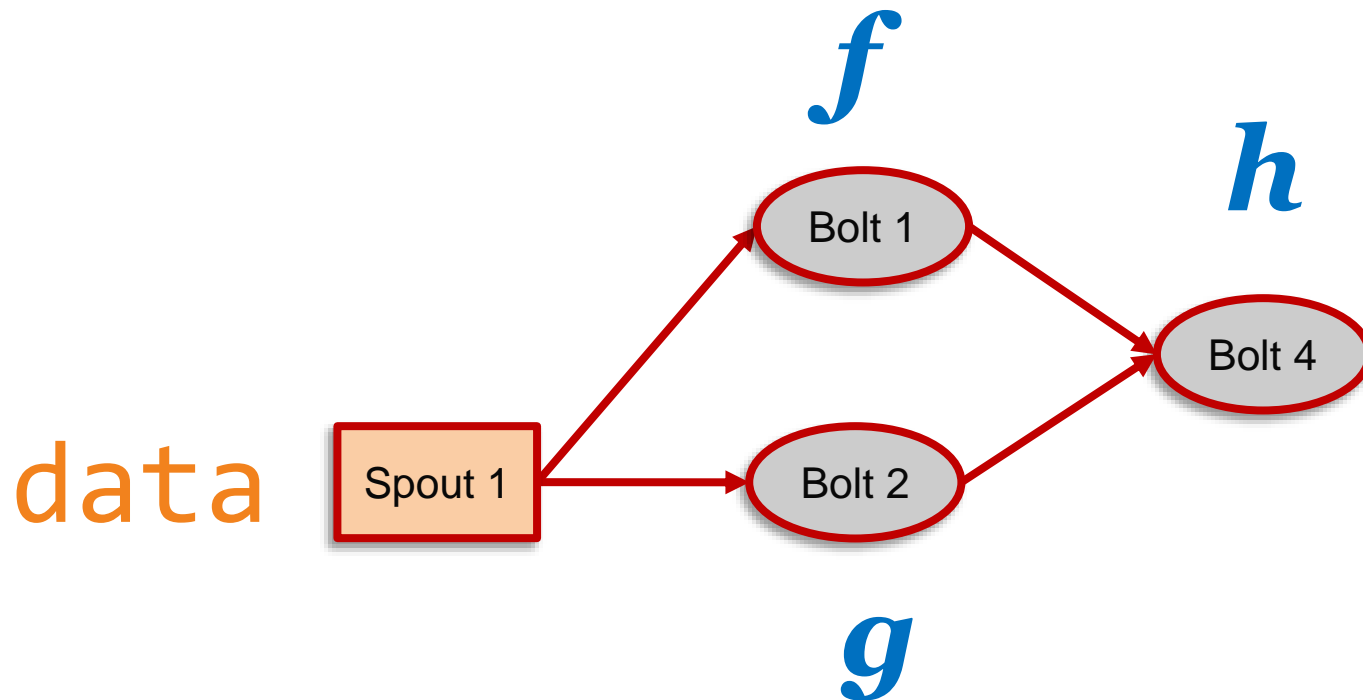
*functions*



# Relation of topologies to FP



# Relation of topologies to FP



DAG:  $h( f(\text{data}), g(\text{data}) )$

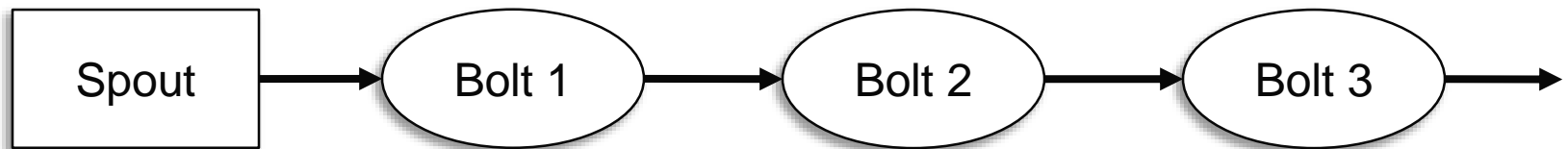


# Previous WordCount example in Storm (high-level)

Remember?

```
(->> queries (map second) frequencies (sort-by val >) )
```

*queries*      *f*      *g*      *h*



# Overview of Part 2: Storm core concepts

- A first look
- Topology
- **Data model**
- Spouts and bolts
- Groupings
- Parallelism

# Data model

Tuple = datum containing 1+ fields

(1.1.1.1, “foo.com”)

Values can be of any type such as Java primitive types, String, byte[]. Custom objects should provide their own Kryo serializer though.

Stream = unbounded sequence of tuples

...  
(1.1.1.1, “foo.com”)  
(2.2.2.2, “bar.net”)  
(3.3.3.3, “foo.com”)  
...

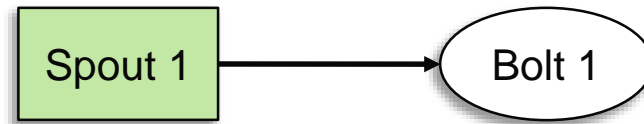
<http://storm.incubator.apache.org/documentation/Concepts.html>

# Overview of Part 2: Storm core concepts

- A first look
- Topology
- Data model
- **Spouts and bolts**
- Groupings
- Parallelism

# Spouts and bolts

Spout = source of data streams



Can be “unreliable” (fire-and-forget) or “reliable” (can replay failed tuples).

Example: Connect to the Twitter API and emit a stream of decoded URLs.

Bolt = consumes 1+ streams and potentially produces new streams



Can do anything from running functions, filter tuples, joins, talk to DB, etc.

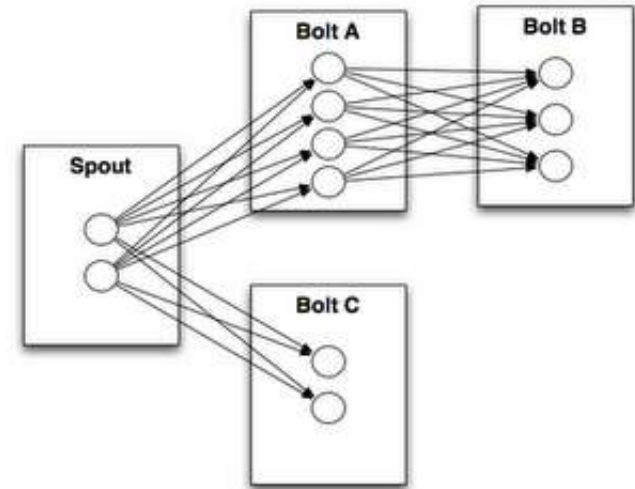
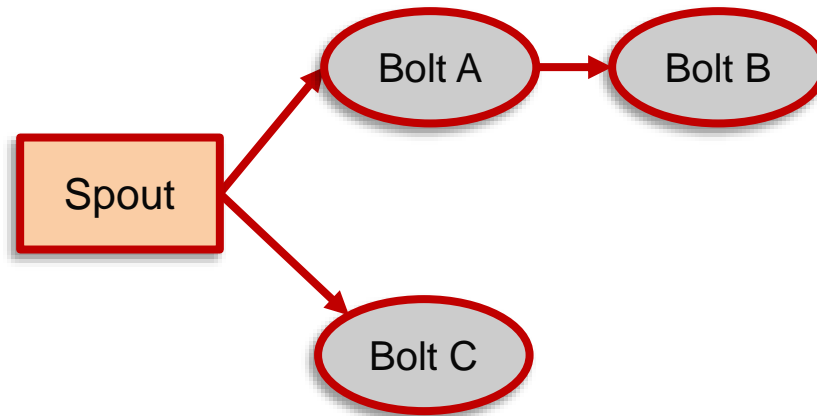
Complex stream transformations often require multiple steps and thus multiple bolts.

<http://storm.incubator.apache.org/documentation/Concepts.html>

# Overview of Part 2: Storm core concepts

- A first look
- Topology
- Data model
- Spouts and bolts
- **Groupings**
- Parallelism

# Stream groupings control the data flow in the DAG



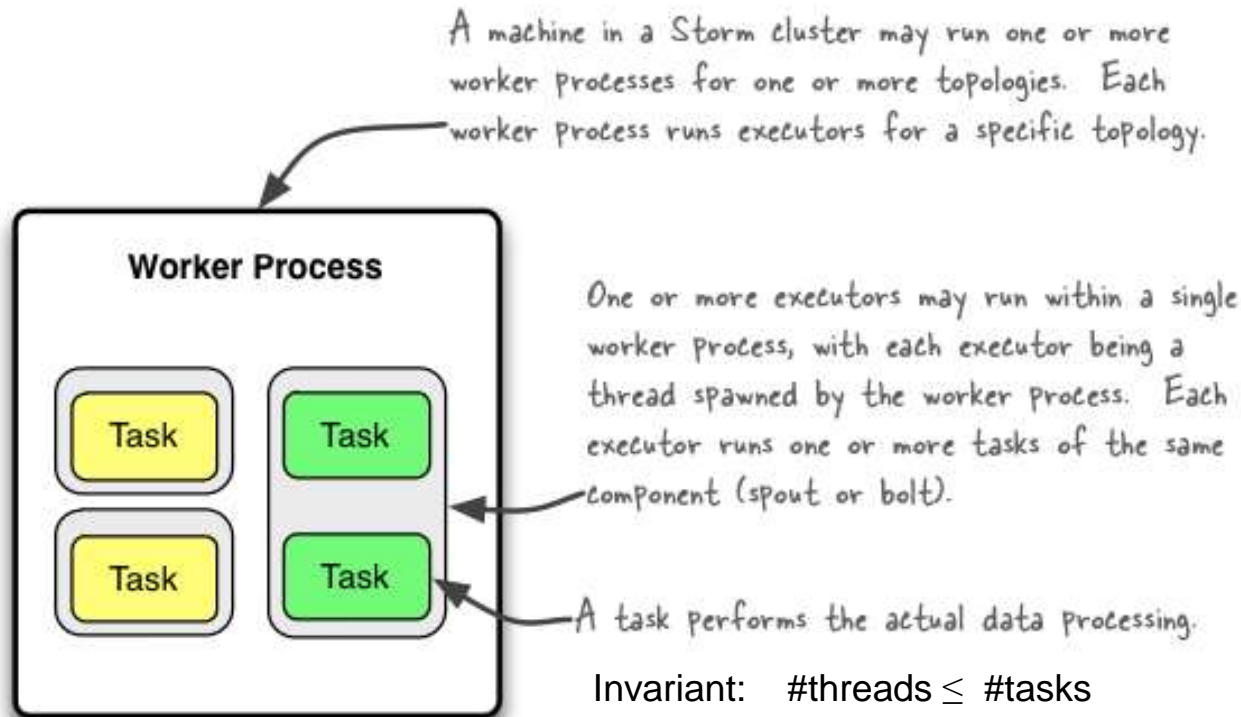
- **Shuffle grouping** = random; typically used to distribute load evenly to downstream bolts
- **Fields grouping** = GROUP BY field(s)
- **All grouping** = replicates stream across all the bolt's tasks; use with care
- **Global grouping** = stream goes to a single one of the bolt's tasks; don't overwhelm the target bolt!
- **Direct grouping** = producer of the tuple decides which task of the consumer will receive the tuple
- **LocalOrShuffle** = If the target bolt has one or more tasks in the same worker process, tuples will be shuffled to just those in-process tasks. Otherwise, same as normal shuffle.
- Custom groupings are possible, too.

# Overview of Part 2: Storm core concepts

- A first look
- Topology
- Data model
- Spouts and bolts
- Groupings
- **Parallelism – worker, executors, tasks**



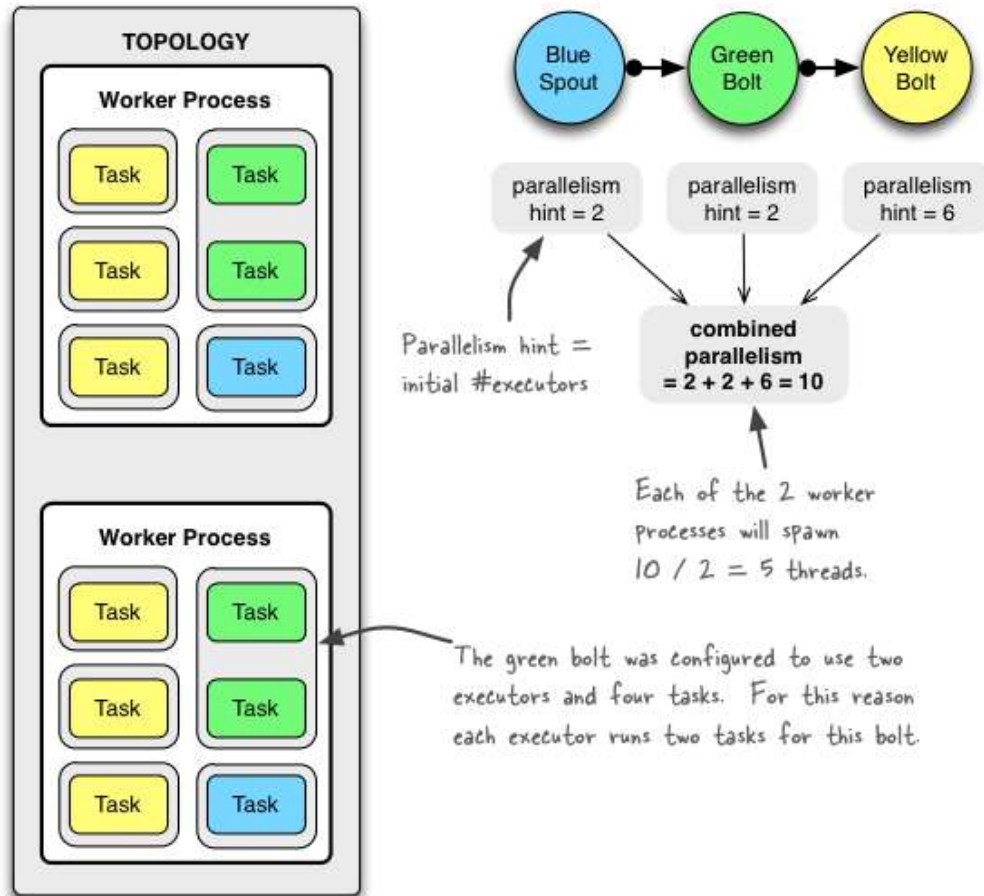
# Worker processes vs. Executors vs. Tasks



A worker process is either idle or being used by a single topology, and it is never shared across topologies. The same applies to its child executors and tasks.

<http://storm.incubator.apache.org/documentation/Understanding-the-parallelism-of-a-Storm-topology.html>

# Example of a running topology



# Code to configure this topology

```
Configuring the parallelism of a simple Storm topology

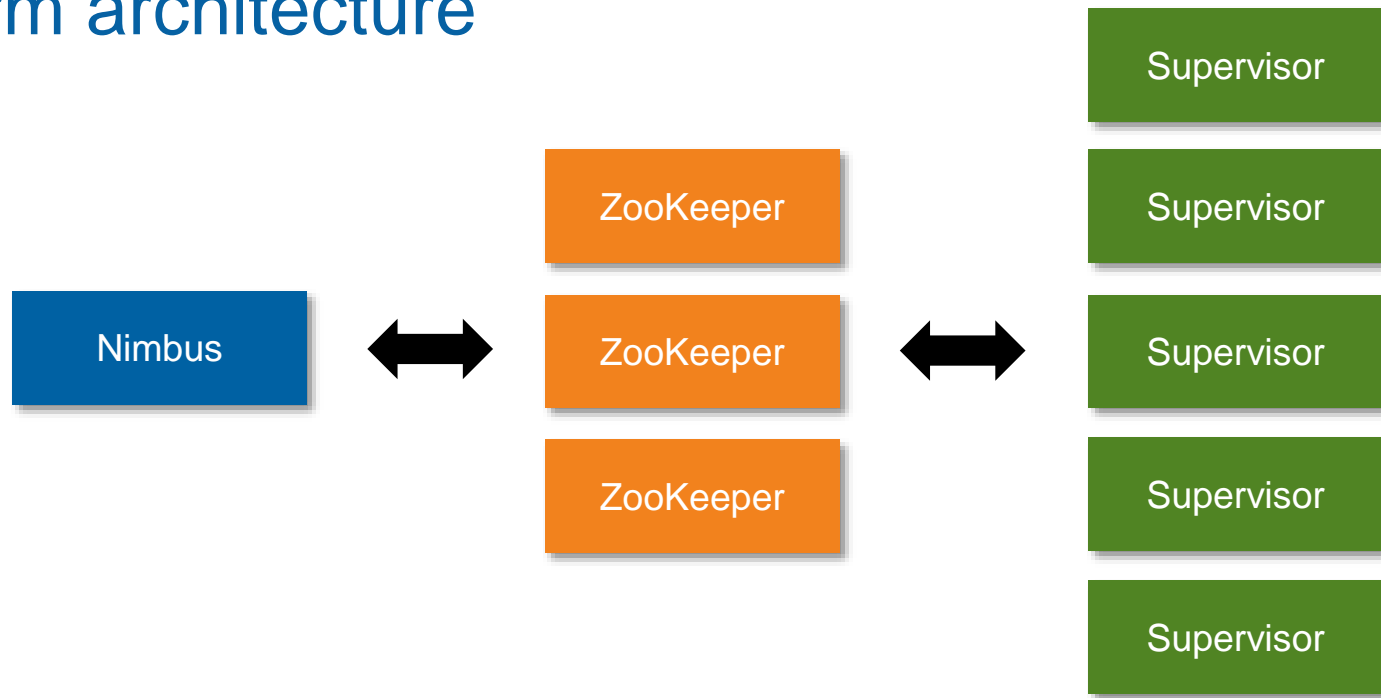
1  Config conf = new Config();
2  conf.setNumWorkers(2); // use two worker processes
3
4  topologyBuilder.setSpout("blue-spout", new BlueSpout(), 2); // parallelism hint
5
6  topologyBuilder.setBolt("green-bolt", new GreenBolt(), 2)
7      .setNumTasks(4)
8      .shuffleGrouping("blue-spout");
9
10 topologyBuilder.setBolt("yellow-bolt", new YellowBolt(), 6)
11     .shuffleGrouping("green-bolt");
12
13 StormSubmitter.submitTopology(
14     "mytopology",
15     conf,
16     topologyBuilder.createTopology()
17 );
```

# Part 3: Operating Storm

# Overview of Part 3: Operating Storm

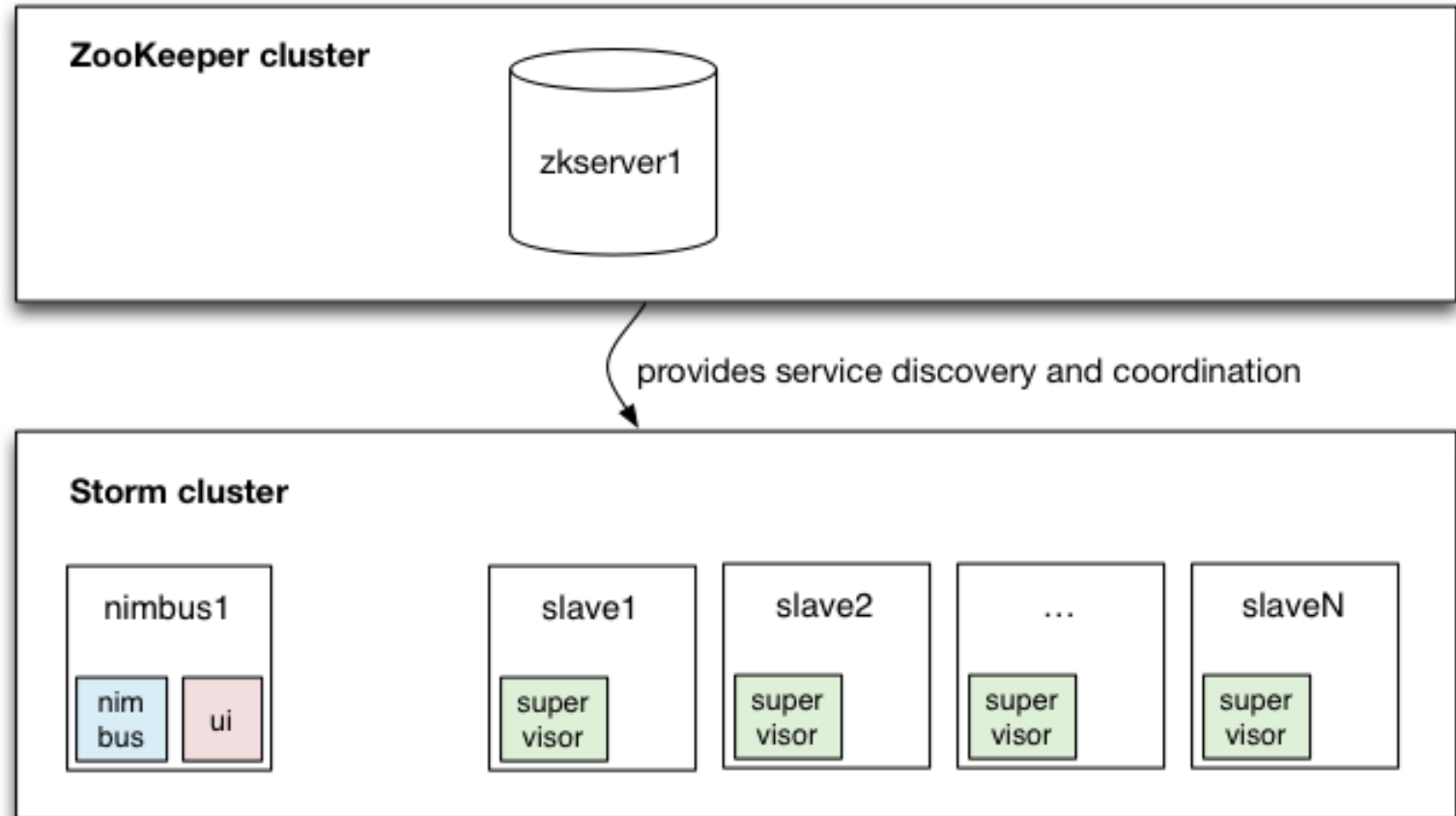
- Storm architecture
- Storm hardware specs
- Deploying Storm
- Monitoring Storm
  - Storm topologies
  - Storm itself
  - ZooKeeper
- Ops-related references

# Storm architecture



Hadoop v1	Storm	
JobTracker	<b>Nimbus (only 1)</b>	<ul style="list-style-type: none"> <li>▪ distributes code around cluster</li> <li>▪ assigns tasks to machines/supervisors</li> <li>▪ failure monitoring</li> <li>▪ is fail-fast and stateless (you can “kill -9” it)</li> </ul>
TaskTracker	<b>Supervisor (many)</b>	<ul style="list-style-type: none"> <li>▪ listens for work assigned to its machine</li> <li>▪ starts and stops worker processes as necessary based on Nimbus</li> <li>▪ is fail-fast and stateless (you can “kill -9” it)</li> <li>▪ shuts down worker processes with “kill -9”, too</li> </ul>
MR job	<b>Topology</b>	<ul style="list-style-type: none"> <li>▪ processes messages forever (or until you kill it)</li> <li>▪ a running topology consists of many worker processes spread across many machines</li> </ul>

# Storm architecture



# Storm architecture: ZooKeeper

- Storm requires **ZooKeeper**
  - 0.9.2+ uses ZK 3.4.5
  - Storm typically puts less load on ZK than Kafka (but ZK is still a bottleneck), but caution: often you have many more Storm nodes than Kafka nodes
- ZooKeeper
  - NOT used for message passing, which is done via Netty in 0.9
  - Used for coordination purposes, and to store state and statistics
    - Register + discover Supervisors, detect failed nodes, ...
      - Example: To add a new Supervisor node, just start it.
    - This allows Storm's components to be stateless. "kill -9" away!
      - Example: Supervisors/Nimbus can be restarted without affecting running topologies.
  - Used for heartbeats
    - Workers heartbeat the status of child executor threads to Nimbus via ZK.
    - Supervisor processes heartbeat their own status to Nimbus via ZK.
  - Stores recent task errors (deleted on topology shutdown)



# Storm architecture: fault tolerance

- What happens when **Nimbus** dies (master node)?
  - If Nimbus is run under process supervision as recommended (e.g. via [supervisord](#)), it will restart like nothing happened.
  - While Nimbus is down:
    - Existing topologies will continue to run, but you cannot submit new topologies.
    - Running worker processes will not be affected. Also, Supervisors will restart their (local) workers if needed. However, failed tasks will not be reassigned to other machines, as this is the responsibility of Nimbus.
- What happens when a **Supervisor** dies (slave node)?
  - If Supervisor run under process supervision as recommended (e.g. via supervisord), will restart like nothing happened.
  - Running worker processes will not be affected.
- What happens when a **worker process** dies?
  - It's parent Supervisor will restart it. If it continuously fails on startup and is unable to heartbeat to Nimbus, Nimbus will reassign the worker to another machine.

# Storm hardware specs

- ZooKeeper

- Preferably use dedicated machines because ZK is a bottleneck for Storm
  - 1 ZK instance per machine
  - Using VMs may work in some situations. Keep in mind other VMs or processes running on the shared host machine may impact ZK performance, particularly if they cause I/O load. ([source](#))
- I/O is a bottleneck for ZooKeeper
  - Put ZK storage on its own disk device
  - SSD's dramatically improve performance
  - Normally, ZK will sync to disk on every write, and that causes two seeks (1x for the data, 1x for the data log). This may add up significantly when all the workers are heartbeating to ZK. ([source](#))
  - Monitor I/O load on the ZK nodes
- Preferably run ZK ensembles with nodes  $\geq 3$  in production environments so that you can tolerate the failure of 1 ZK server (incl. e.g. maintenance)

# Storm hardware specs

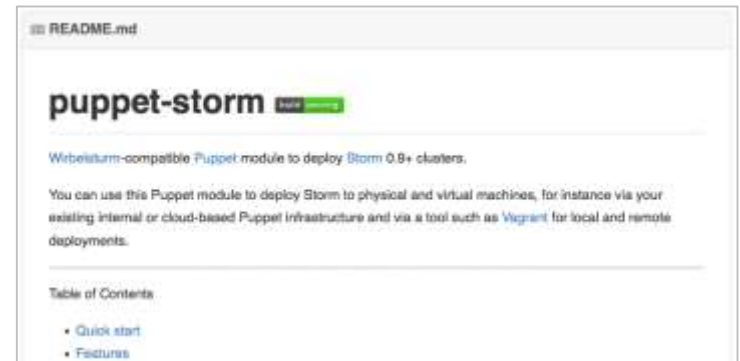
- Nimbus aka master node
  - Comparatively little load on Nimbus, so a medium-sized machine suffices
  - EC2 example: m1.xlarge @ \$0.27/hour
  - Check monitoring stats to see if the machine can keep up

# Storm hardware specs

- Storm Supervisor aka slave nodes
  - Exact specs depend on anticipated usage – e.g. CPU heavy, I/O heavy, ...
    - CPU heavy: e.g. machine learning
    - CPU light: e.g. rolling windows, pre-aggregation (here: get more RAM)
  - CPU cores
    - More is usually better – the more you have the more threads you can support (i.e. parallelism). And Storm potentially uses a *lot* of threads.
  - Memory
    - Highly specific to actual use case
    - Considerations: #workers (= JVMs) per node? Are you caching and/or holding in-memory state?
  - Network: 1GigE
    - Use bonded NICs or 10GigE if needed
  - EC2 examples: c1.xlarge @ \$0.36/hour, c3.2xlarges @ \$0.42/hour

# Deploying Storm

- Puppet module
  - <https://github.com/miguno/puppet-storm>
  - Hiera-compatible, rspec tests, Travis CI setup (e.g. to test against multiple versions of Puppet and Ruby, Puppet style checker/lint, etc.)
- RPM packaging script for RHEL 6
  - <https://github.com/miguno/wirbelsturm-rpm-storm>
  - Digitally signed by yum@michael-noll.com
  - RPM is built on a Wirbelsturm-managed build server
- See later slides on Wirbelsturm for 1-click off-the-shelf cluster setups.



# Deploying Storm

- Hierarchical example for a Storm slave node

```
---
classes:
  - storm::logviewer
  - storm::supervisor
  - supervisor

## Custom Storm settings
storm::zookeeper_servers:
  - 'zookeeper1'
storm::logviewer_childdopts: '-Xmx128m -Djava.net.preferIPv4Stack=true'
storm::nimbus_host: 'nimbus1'
storm::supervisor_childdopts: '-Xmx256m -Djava.net.preferIPv4Stack=true'
storm::worker_childdopts: '-Xmx1024m -Djava.net.preferIPv4Stack=true'
storm::supervisor_slots_ports:
  - 6700
  - 6701
  - 6702
  - 6703
```

# Operating Storm

- Typical operations tasks include:
  - Monitoring topologies for P&S (“Don’t let our pipes blow up!”)
    - Tackling P&S in Storm is a joint Ops-Dev effort.
  - Adding or removing slave nodes, i.e. nodes that run Supervisors
  - Apps management: new topologies, swapping topologies, ...
- See Ops-related references at the end of this part

# Storm security

- Original design was not created with security in mind.
- Security features are now being added, e.g. from Yahoo!'s fork.
- State of security in Storm 0.9.x:
  - No authentication, no authorization.
  - No encryption of data in transit, i.e. between workers.
  - No access restrictions on data stored in ZooKeeper.
  - Arbitrary user code can be run on nodes if Nimbus' Thrift port is not locked down.
  - This list goes on.
- Further details plus recommendations on hardening Storm:
  - <https://github.com/apache/incubator-storm/blob/master/SECURITY.md>



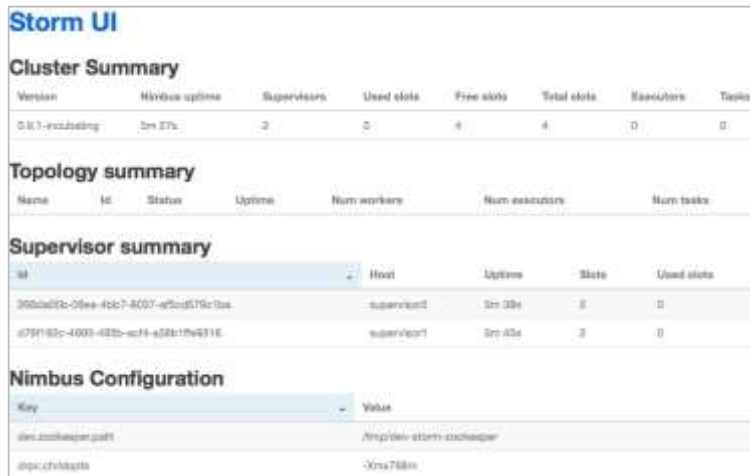
# Monitoring Storm

# Monitoring Storm

- Storm UI
- Use standard monitoring tools such as Graphite & friends
  - Graphite
    - <https://github.com/miguno/puppet-graphite>
    - Java API, also used by Kafka: <http://metrics.codahale.com/>
- Consider Storm's built-in metrics feature
- Collect logging files into a central place
  - Logstash/Kibana and friends
  - Helps with troubleshooting, debugging, etc. – notably if you can correlate logging data with numeric metrics

# Monitoring Storm

- Built-in Storm UI, listens on 8080/tcp by default



**Storm UI**

**Cluster Summary**

Version	Nimbus uptime	Supervisors	Used slots	Free slots	Total slots	Executors	Tasks
0.9.1-incubating	3m 27s	2	0	4	4	0	0

**Topology summary**

Name	ID	Status	Uptime	Num workers	Num executors	Num tasks
------	----	--------	--------	-------------	---------------	-----------

**Supervisor summary**

ID	Host	Uptime	State	Used slots
300d02c-06ee-4bc7-8007-e5f0d579c1ba	supervisor0	3m 33s	3	0
a79f183c-4003-482b-ec1f-a58b1f64311e	supervisor1	3m 43s	3	0

**Nimbus Configuration**

Key	Value
dev.zookeeper.path	/tmp/dev-storm-zookeeper
rpc.zookeeper	-Xms768m

- Storm REST API (new since in 0.9.2)
  - <https://github.com/apache/incubator-storm/blob/master/STORM-UI-REST-API.md>
- Third-party tools
  - <https://github.com/otoolep/stormkafkamon>

# Monitoring Storm topologies

- Wait – why does the Storm UI report seemingly incorrect numbers?
  - Storm *samples* incoming tuples when computing statistics in order to increase performance.
  - Sample rate is configured via `topology.stats.sample.rate`.
  - `0.05` is the default value
    - Here, Storm will pick a random event of the next 20 events in which to increase the metric count by 20. So if you have 20 tasks for that bolt, your stats could be off by +/- 380.
  - `1.00` forces Storm to count everything exactly
    - This gives you accurate numbers at the cost of a big performance hit. For testing purposes however this is acceptable and often quite helpful.

# Monitoring ZooKeeper

- Ensemble (= cluster) availability
  - LinkedIn run 5-node ensembles = tolerates 2 dead
  - Twitter run 13-node ensembles = tolerates 6 dead
- Latency of requests
  - Metric target is 0 ms when using SSD's in ZooKeeper machines.
    - Why? Because SSD's are so fast they typically bring down latency below ZK's metric granularity (which is per-ms).
- Outstanding requests
  - Metric target is 0.
  - Why? Because ZK processes all incoming requests serially. Non-zero values mean that requests are backing up.

# Ops-related references

- Storm documentation
  - <http://storm.incubator.apache.org/documentation/Home.html>
- Storm FAQ
  - <http://storm.incubator.apache.org/documentation/FAQ.html>
- Storm CLI
  - <http://storm.incubator.apache.org/documentation/Command-line-client.html>
- Storm fault-tolerance
  - <http://storm.incubator.apache.org/documentation/Fault-tolerance.html>
- Storm metrics
  - <http://storm.incubator.apache.org/documentation/Metrics.html>
  - <http://www.michael-noll.com/blog/2013/11/06/sending-metrics-from-storm-to-graphite/>
- Storm tutorials
  - <http://storm.incubator.apache.org/documentation/Tutorial.html>
  - <http://www.michael-noll.com/tutorials/running-multi-node-storm-cluster/>

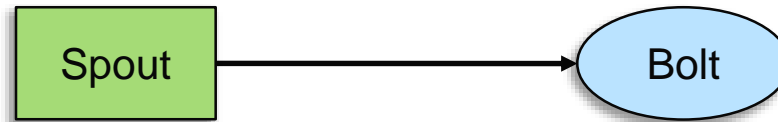
# Part 4: Developing Storm apps

# Overview of Part 4: Developing Storm apps

- Hello, Storm!
- Creating a bolt
- Creating a topology
- Running a topology
- Integrating Storm and Kafka
- Testing Storm topologies
- Serialization in Storm (Avro, Kryo)
- Example Storm apps
- P&S tuning



# A trivial “Hello, Storm” topology



“emit random  
number < 100”



“multiply  
by 2”

**(74)**

**(148)**

# Code

## Spout

```
1 ▾ public void nextTuple() {  
2     final Random rand = new Random(); // normally this should be an instance field  
3     int nextRandomNumber = rand.nextInt(100);  
4     collector.emit(new Values(nextRandomNumber)); // auto-boxing  
5 }
```

## Bolt

```
1  Override  
2 ▾ public void prepare(Map conf, TopologyContext context, OutputCollector collector) {  
3     this.collector = collector;  
4 }  
5  
6 @Override  
7 ▾ public void execute(Tuple tuple) {  
8     Integer inputNumber = tuple.getInteger(0);  
9     collector.emit(tuple, new Values(inputNumber * 2)); // auto-boxing  
10    collector.ack(tuple);  
11 }  
12  
13 @Override  
14 ▾ public void declareOutputFields(OutputFieldsDeclarer declarer) {  
15    declarer.declare(new Fields("doubled-number"));  
16 }
```

# Code

## Topology config – for running *on your local laptop*

```
1 Config conf = new Config();
2 conf.setNumWorkers(1);
3
4 topologyBuilder.setSpout("my-spout", new MySpout(), 2);
5
6 topologyBuilder.setBolt("my-bolt", new MyBolt(), 2)
7     .shuffleGrouping("my-spout");
8
9 StormSubmitter.submitTopology(
10     "mytopology",
11     conf,
12     topologyBuilder.createTopology()
13 );
```

# Code

## Topology config – for running *on a production Storm cluster*

```
1 Config conf = new Config();
2 conf.setNumWorkers(200);
3
4 topologyBuilder.setSpout("my-spout", new MySpout(), 100);
5
6 topologyBuilder.setBolt("my-bolt", new MyBolt(), 200)
7     .shuffleGrouping("my-spout");
8
9 StormSubmitter.submitTopology(
10     "mytopology",
11     conf,
12     topologyBuilder.createTopology()
13 );
```

# Creating a spout

- Won't cover implementing a spout in this workshop.
- This is because you typically use an existing spout (Kafka spout, Redis spout, etc). But you will definitely implement your own **bolts**.

# Creating a bolt

- Storm is polyglot – but in this workshop we focus on JVM languages.
- Two main options for JVM users:
  - Implement the [IRichBolt](#) or [IBasicBolt](#) interfaces
  - Extend the [BaseRichBolt](#) or [BaseBasicBolt](#) abstract classes
- BaseRichBolt
  - You must – and are able to – manually `ack()` an incoming tuple.
  - Can be used to delay acking a tuple, e.g. for algorithms that need to work across multiple incoming tuples.
- BaseBasicBolt
  - Auto-acks the incoming tuple at the end of its `execute()` method.
  - These bolts are typically simple functions or filters.

# Extending BaseRichBolt

- Let's re-use our previous example bolt.

```
1  Override
2  ▾ public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
3      this.collector = collector;
4  }
5
6  @Override
7  ▾ public void execute(Tuple tuple) {
8      Integer inputNumber = tuple.getInteger(0);
9      collector.emit(tuple, new Values(inputNumber * 2)); // auto-boxing
10     collector.ack(tuple);
11 }
12
13 @Override
14 ▾ public void declareOutputFields(OutputFieldsDeclarer declarer) {
15     declarer.declare(new Fields("doubled-number"));
16 }
```

# Extending BaseRichBolt

- `execute()` is the heart of the bolt.
- This is where you will focus most of your attention when implementing your bolt or when trying to understand somebody else's bolt.

```
1  Override
2  ▾ public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
3      ...   this.collector = collector;
4  }
5
6  @Override
7  ▾ public void execute(Tuple tuple) {
8      ...   Integer inputNumber = tuple.getInteger(0);
9      ...   collector.emit(tuple, new Values(inputNumber * 2)); // auto-boxing
10     ...   collector.ack(tuple);
11 }
12
13 @Override
14 ▾ public void declareOutputFields(OutputFieldsDeclarer declarer) {
15     ...   declarer.declare(new Fields("doubled-number"));
16 }
```



# Extending BaseRichBolt

- `prepare()` acts as a “second constructor” for the bolt’s class.
- Because of Storm’s distributed execution model and serialization, `prepare()` is often needed to fully initialize the bolt on the target JVM.

```
1  @Override
2  public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
3      this.collector = collector;
4  }
5
6  @Override
7  public void execute(Tuple tuple) {
8      Integer inputNumber = tuple.getInteger(0);
9      collector.emit(tuple, new Values(inputNumber * 2)); // auto-boxing
10     collector.ack(tuple);
11 }
12
13 @Override
14 public void declareOutputFields(OutputFieldsDeclarer declarer) {
15     declarer.declare(new Fields("doubled-number"));
16 }
```

# Extending BaseRichBolt

- **declareOutputFields()** tells downstream bolts about this bolt's output. What you declare must match what you actually emit().
  - You will use this information in downstream bolts to “extract” the data from the emitted tuples.
  - If your bolt only performs side effects (e.g. talk to a DB) but does not emit an actual tuple, override this method with an empty {} method.

```
1  Override
2  ▾ public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
3      this.collector = collector;
4  }
5
6  @Override
7  ▾ public void execute(Tuple tuple) {
8      Integer inputNumber = tuple.getInteger(0);
9      collector.emit(tuple, new Values(inputNumber * 2)); // auto-boxing
10     collector.ack(tuple);
11 }
12
13 @Override
14 ▾ public void declareOutputFields(OutputFieldsDeclarer declarer) {
15     declarer.declare(new Fields("doubled-number"));
16 }
```

# Common spout/bolt gotchas

- `NotSerializableException` at run-time of your topology
  - Typically you will run into this because your bolt has fields (instance or class members) that are not serializable. This recursively applies to each field.
  - The root cause is Storm's distributed execution model and serialization: Storm code will be shipped – first serialized and then deserialized – to a different machine/JVM, and then executed. (see docs for details)
  - How to fix?
    - Solution 1: Make the culprit class serializable, if possible.
    - Solution 2: Register a custom Kryo serializer for the class.
    - Solution 3a (Java): Make the field transient. If needed, initialize it in `prepare()`.
    - Solution 3b (Scala): Make the field `@transient lazy val` (Scala). If needed, turn it into a var and initialize it in `prepare()`.
      - For example, the `var/prepare()` approach may be needed if you use the factory pattern to create a specific type of a collaborator within a bolt. Factories come in handy to make the code testable. See [AvroKafkaSinkBolt](#) in `kafka-storm-starter` for such a case.

# Common spout/bolt gotchas

- Tick tuples are configured *per-component*, i.e. per bolt
  - Idiomatic approach to trigger periodic activities in your bolts: “Every 10s do XYZ.”
  - Don't configure them per-topology as this will throw a RuntimeException.

```
1  @Override
2  public Map<String, Object> getComponentConfiguration() {
3      Config config = new Config();
4      config.put(Config.TOPOLOGY_TICK_TUPLE_FREQ_SECS, 10);
5      return config;
6  }
```

- Tick tuples are not 100% guaranteed to arrive *in time*
  - They are sent to a bolt just like any other tuples, and will enter the same queues and buffers. Congestion, for example, may cause tick tuples to arrive too late.
  - Across different bolts, tick tuples are not guaranteed to arrive at the same time, even if the bolts are configured to use the same tick tuple frequency.
  - Currently, tick tuples for the same bolt will arrive at the same time at the bolt's various task instances. However, this property is not guaranteed for the future.

<http://www.michael-noll.com/blog/2013/01/18/implementing-real-time-trending-topics-in-storm/>

# Common spout/bolt gotchas

- When using tick tuples, forgetting to handle them "in a special way"
  - Trying to run your normal business logic on tick tuples – e.g. extracting a certain data field – will usually only work for normal tuples but fail for a tick tuple.

```
1  @Override
2  public void execute(Tuple tuple) {
3      if (isTickTuple(tuple)) {
4          // now you can trigger e.g. a periodic activity
5      }
6      else {
7          // do something with the normal tuple
8      }
9  }
10
11 private static boolean isTickTuple(Tuple tuple) {
12     return tuple.getSourceComponent().equals(Constants.SYSTEM_COMPONENT_ID)
13         && tuple.getSourceStreamId().equals(Constants.SYSTEM_TICK_STREAM_ID);
14 }
```

- When using tick tuples, forgetting to ack() them
  - Tick tuples must be acked like any other tuple.

<http://www.michael-noll.com/blog/2013/01/18/implementing-real-time-trending-topics-in-storm/>

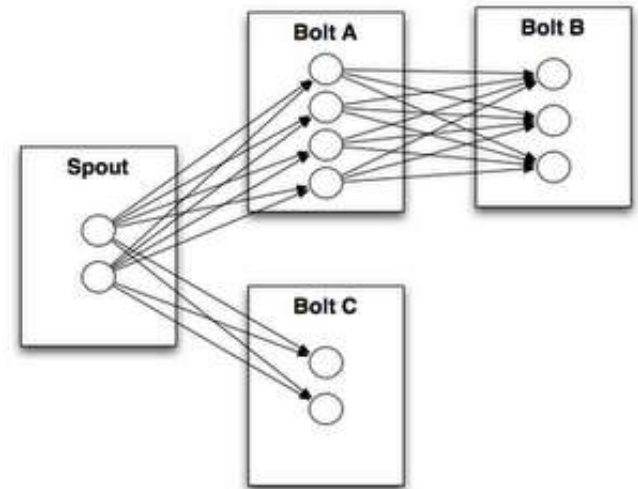
# Common spout/bolt gotchas

- `OutputCollector#emit()` can only be called from the "original" thread that runs a bolt
  - You can start additional threads in your bolt, but only the bolt's own thread may call `emit()` on the collector to write output tuples. If you try to emit tuples from any of the other threads, Storm will throw a `NullPointerException`.
    - If you need the additional-threads pattern, use e.g. a thread-safe queue to communicate between the threads and to collect [pun intended] the output tuples across threads.
  - This limitation is only relevant for output tuples, i.e. output that you want to send within the Storm framework to downstream consumer bolts.
  - If you want to write data to (say) Kafka instead – think of this as a side effect of your bolt – then you don't need the `emit()` anyways and can thus write the side-effect output in any way you want, and from any thread.

# Creating a topology

- When creating a topology you're essentially defining the DAG – that is, which spouts and bolts to use, and how they interconnect.
  - `TopologyBuilder#setSpout()` and `TopologyBuilder#setBolt()`
  - Groupings between spouts and bolts, e.g. `shuffleGrouping()`

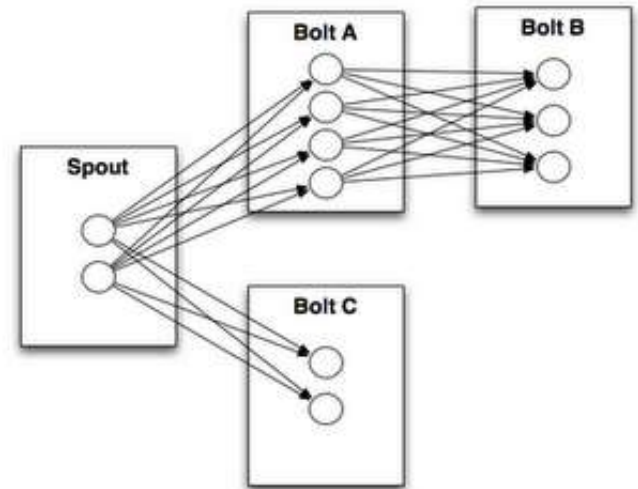
```
1 Config conf = new Config();  
2 conf.setNumWorkers(200);  
3  
4 TopologyBuilder builder = new TopologyBuilder();  
5 builder.setSpout("my-spout", new MySpout(), 100);  
6 builder.setBolt("my-bolt", new MyBolt(), 200)  
7     .shuffleGrouping("my-spout");  
8 StormTopology topology = builder.createTopology();
```



# Creating a topology

- You must specify the initial *parallelism* of the topology.
  - Crucial for P&S but no rule of thumb. We talk about tuning later.
  - You must understand concepts such as workers/executors/tasks.
- Only some aspects of parallelism can be changed later, i.e. at run-time.
  - You can change the #executors (threads).
  - You cannot change #tasks, which remains static during the topology's lifetime.

```
1 Config conf = new Config();
2 conf.setNumWorkers(200);
3
4 TopologyBuilder builder = new TopologyBuilder();
5 builder.setSpout("my-spout", new MySpout(), 100);
6 builder.setBolt("my-bolt", new MyBolt(), 200)
7     .shuffleGrouping("my-spout");
8 StormTopology topology = builder.createTopology();
```





# Creating a topology

- You submit a topology either to a “local” cluster or to a real cluster.
  - `LocalCluster#submitTopology`
  - `StormSubmitter#submitTopology()` and `#submitTopologyWithProgressBar()`
  - In your code you may want to use both approaches, e.g. to facilitate local testing.
- Notes
  - A `StormTopology` is a static, serializable Thrift data structure. It contains instructions that tell Storm how to deploy and run the topology in a cluster.
  - The `StormTopology` object will be *serialized*, including *all* the components in the topology's DAG. See later slides on serialization.
  - Only when the topology is *deployed* (and serialized in the process) and *initialized* (i.e. `prepare()` and other life cycle methods are called on components such as bolts) does it perform any actual message processing.

# Running a topology

- To run a topology you must first package your code into a “fat jar”.
  - You must includes all your code’s dependencies but:
  - Exclude the Storm dependency itself, as the Storm cluster will provide this.
    - Sbt: `"org.apache.storm" % "storm-core" % "0.9.2-incubating" % "provided"`
    - Maven: `<scope>provided</scope>`
    - Gradle with [gradle-fatjar-plugin](#): `compile '...', { ext { fatJarExclude = true } }`
  - Note: You may need to tweak your build script so that your local tests *do include* the Storm dependency. See e.g. [assembly.sbt](#) in kafka-storm-starter for an example.
- A topology is run via the [storm jar](#) command.

```
$ storm jar all-my-code.jar com.miguno.MyTopology arg1 arg2
```

- Will connects to Nimbus, upload your jar, and run the topology.
- Use any machine that can run "storm jar" and talk to Nimbus' Thrift port.
- You can pass additional JVM options via `$STORM_JAR_JVM_OPTS`.

# Alright, my topology runs – now what?

- The topology will run forever or until you kill it.
- Check the status of your topology
  - Storm UI (default: 8080/tcp)
  - [Storm CLI](#), e.g. `storm [list | kill | rebalance | deactivate | ...]`
  - Storm REST API
- FYI:
  - Storm will guarantee that no data is lost, even if machines go down and messages are dropped (as long as you don't disable this feature).
  - Storm will automatically restart failed tasks, and even re-assign tasks to different machines if e.g. a machine dies.
  - See Storm docs for further details.

# Integrating Storm and Kafka

# Reading from Kafka

- Use the official Kafka spout that ships in Storm 0.9.2
  - <https://github.com/apache/incubator-storm/tree/master/external/storm-kafka>
  - Compatible with Kafka 0.8, available on Maven Central
  - Based on wurstmeister's spout, now part of Storm  
<https://github.com/wurstmeister/storm-kafka-0.8-plus>

```
"org.apache.storm" % "storm-kafka" % "0.9.2-incubating"
```

- Alternatives to official Kafka spout
  - NFI: <https://github.com/HolmesNL/kafka-spout>
  - A detailed comparison is beyond the scope of this workshop, but:
    - Official Kafka spout uses Kafka's Simple Consumer API, NFI uses High-level API.
    - Official spout can read from multiple topics, NFI can't.
    - Official spout's replay-failed-tuples functionality is better than NFI's.

# Reading from Kafka

- Spout configuration via [KafkaConfig](#)
- In the following example:
  - Connect to the target Kafka cluster via the ZK ensemble at zookeeper1:2181.
  - We want to read from the Kafka topic “my-kafka-input-topic”, which has 10 partitions.
  - By default, the spout stores its own state incl. Kafka offsets in the *Storm* cluster's ZK.
    - Can be changed by setting the field [SpoutConfig.zkServers](#). See source, no docs yet.

```
1 val inputTopic = "my-kafka-input-topic"
2 val inputTopicNumPartitions = 10
3 val builder = new TopologyBuilder
4 val kafkaSpoutId = "kafka-spout"
5 val kafkaSpoutConfig = {
6     val zkHosts = new ZkHosts("zookeeper1:2181")
7     val zkRoot = "/kafka-storm-starter-spout"
8     val zkId = "kafka-spout"
9     new SpoutConfig(zkHosts, inputTopic, zkRoot, zkId)
10 }
11 val kafkaSpout = new KafkaSpout(kafkaSpoutConfig)
12 val numSpoutExecutors = inputTopicNumPartitions
13 builder.setSpout(kafkaSpoutId, kafkaSpout, numSpoutExecutors)
```

- Full example at [KafkaStormSpec](#) in kafka-storm-starter

# Writing to Kafka

- Use a normal Kafka producer in your bolt, no special magic needed
- Base setup:
  - Serialize the desired output data in the way you need, e.g. via Avro.
  - Write to Kafka, typically in your bolt's `emit()` method.
  - If you are not emitting any Storm tuples, i.e. if you write to Kafka only, make sure you override `declareOutputFields()` with an empty `{}` method

```
1 override def execute(tuple: Tuple, collector: BasicOutputCollector) {  
2   tuple.getValueByField(inputField) match {  
3     case pojo: T =>  
4       val bytes = Injection[T, Array[Byte]](pojo)  
5       log.debug("Encoded pojo " + pojo + " to Avro binary format")  
6       producer.send(bytes)  
7     case _ => log.error("Could not decode binary data")  
8   }  
9 }
```

- Full example at [AvroKafkaSinkBolt](#) in kafka-storm-starter

# Testing Storm topologies



# Testing Storm topologies

- Won't have the time to cover testing in this workshop.
- Some hints:
  - Unit-test your individual classes like usual, e.g. bolts
  - When integration testing, use in-memory instances of Storm and ZK
  - Try Storm's built-in testing API (cf. kafka-storm-starter below)
  - Test-drive topologies in virtual Storm clusters via Wirbelsturm
- Starting points:
  - storm-core test suite
    - <https://github.com/apache/incubator-storm/tree/master/storm-core/test/>
  - storm-kafka test suite
    - <https://github.com/apache/incubator-storm/tree/master/external/storm-kafka/src/test>
  - kafka-storm-starter tests related to Storm
    - <https://github.com/miguno/kafka-storm-starter/>

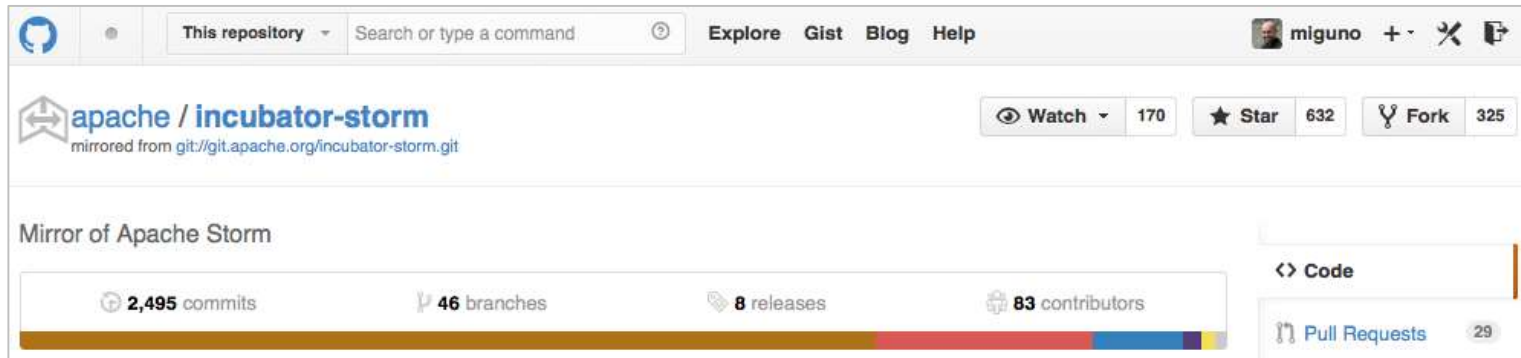
# Serialization in Storm

# Serialization in Storm

- Required because Storm processes data across JVMs and machines
- When/where/how serialization happens is often critical for P&S tuning
- Storm uses [Kryo](#) for serialization, falls back on Java serialization
  - By default, Storm can serialize primitive types, strings, byte arrays, ArrayList, HashMap, HashSet, and the Clojure collection types.
  - Anything else needs a custom Kryo serializer, which must be “registered” with Storm.
  - Storm falls back on Java serialization if needed. But this serialization is slow.
    - Tip: Disable `topology.fall.back.on.java.serialization` to spot missing serializers.
- Examples in kafka-storm-starter, all of which make use of Twitter Bijection/Chill
  - [AvroScheme\[T\]](#) – enable automatic Avro-decoding in Kafka spout
  - [AvroDecoderBolt\[T\]](#) – decode Avro data in a bolt
  - [AvroKafkaSinkBolt\[T\]](#) – encode Avro data in a bolt
  - [TweetAvroKryoDecorator](#) – a custom Kryo serializer
  - [KafkaStormSpec](#) – shows how to register a custom Kryo serializer
- More details at [Storm serialization](#)

# Example Storm apps

# storm-starter



- storm-starter is part of core Storm project since 0.9.2
- <https://github.com/apache/incubator-storm/tree/master/examples/storm-starter>
- Since 0.9.2 also published to Maven Central = you can re-use its spouts/bolts

```
$ git clone https://github.com/apache/incubator-storm.git
$ cd incubator-storm/
$ mvn clean install -DskipTests=true # build Storm locally
$ cd examples/storm-starter          # go to storm-starter
```

(Must have Maven 3.x and JDK installed.)

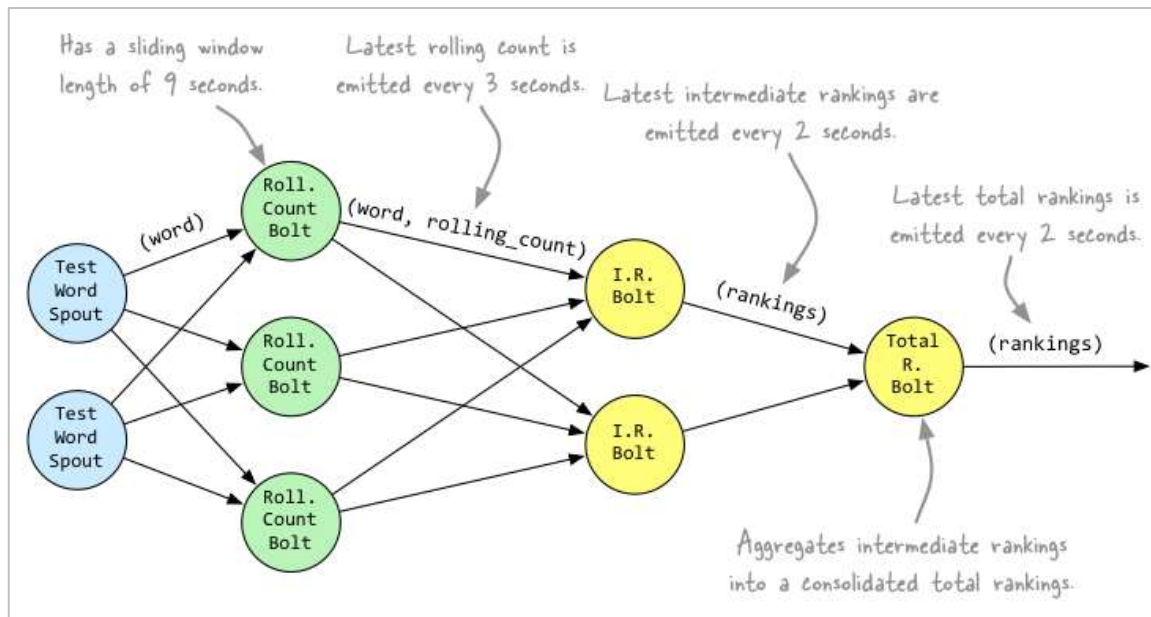
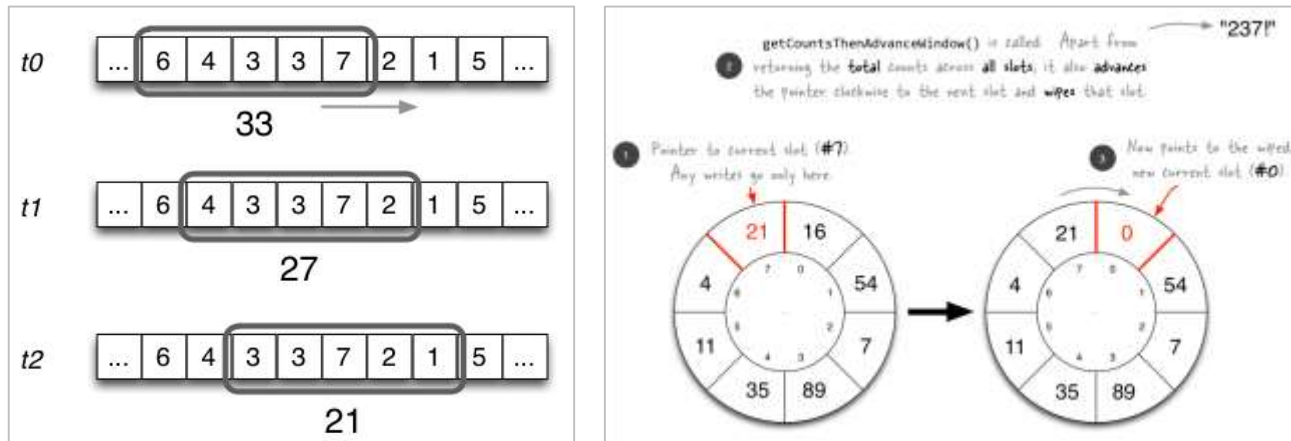
# storm-starter: RollingTopWords

```
$ mvn compile exec:java -Dstorm.topology=storm.starter.RollingTopWords
```

- Will run a topology that implements trending topics.
- <http://www.michael-noll.com/blog/2013/01/18/implementing-real-time-trending-topics-in-storm/>

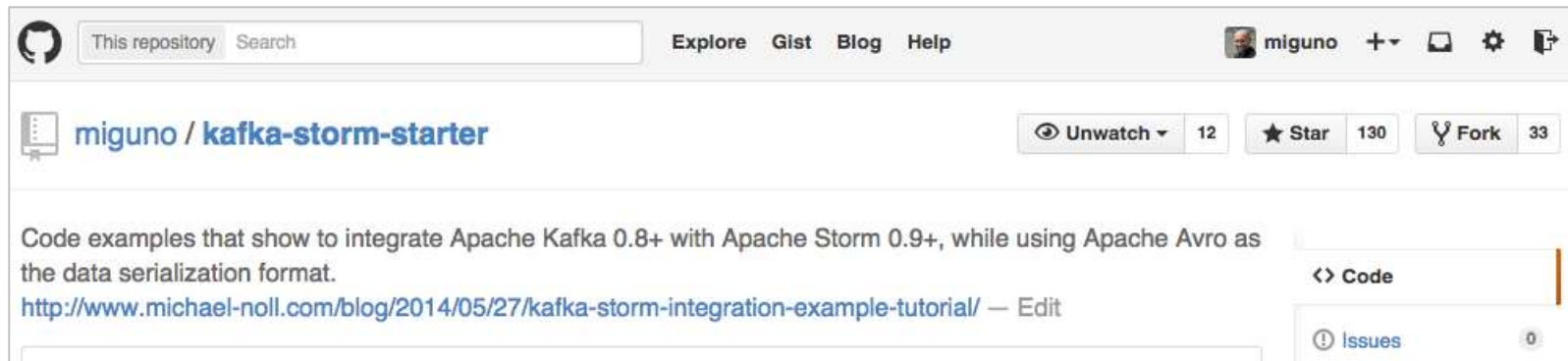
```
76793 [Thread-42-wordGenerator] INFO backtype.storm.daemon.task - Emitting: wordGenerator default [nathan]
76793 [Thread-38-wordGenerator] INFO backtype.storm.daemon.task - Emitting: wordGenerator default [nathan]
76793 [Thread-44-wordGenerator] INFO backtype.storm.daemon.task - Emitting: wordGenerator default [nathan]
76793 [Thread-40-wordGenerator] INFO backtype.storm.daemon.task - Emitting: wordGenerator default [golda]
76793 [Thread-36-wordGenerator] INFO backtype.storm.daemon.task - Emitting: wordGenerator default [jackson]
76793 [Thread-24-counter] INFO backtype.storm.daemon.executor - Processing received message source: wordGenerator:14, stream: default, id: {}, [nathan]
76793 [Thread-18-counter] INFO backtype.storm.daemon.executor - Processing received message source: wordGenerator:13, stream: default, id: {}, [golda]
76793 [Thread-24-counter] INFO backtype.storm.daemon.executor - Processing received message source: wordGenerator:12, stream: default, id: {}, [nathan]
76794 [Thread-22-counter] INFO backtype.storm.daemon.executor - Processing received message source: wordGenerator:11, stream: default, id: {}, [jackson]
76794 [Thread-24-counter] INFO backtype.storm.daemon.executor - Processing received message source: wordGenerator:15, stream: default, id: {}, [nathan]
76893 [Thread-42-wordGenerator] INFO backtype.storm.daemon.task - Emitting: wordGenerator default [jackson]
76894 [Thread-36-wordGenerator] INFO backtype.storm.daemon.task - Emitting: wordGenerator default [mike]
76893 [Thread-38-wordGenerator] INFO backtype.storm.daemon.task - Emitting: wordGenerator default [nathan]
76894 [Thread-44-wordGenerator] INFO backtype.storm.daemon.task - Emitting: wordGenerator default [jackson]
76894 [Thread-40-wordGenerator] INFO backtype.storm.daemon.task - Emitting: wordGenerator default [jackson]
76894 [Thread-22-counter] INFO backtype.storm.daemon.executor - Processing received message source: wordGenerator:14, stream: default, id: {}, [jackson]
76894 [Thread-20-counter] INFO backtype.storm.daemon.executor - Processing received message source: wordGenerator:11, stream: default, id: {}, [mike]
76894 [Thread-24-counter] INFO backtype.storm.daemon.executor - Processing received message source: wordGenerator:12, stream: default, id: {}, [nathan]
76894 [Thread-22-counter] INFO backtype.storm.daemon.executor - Processing received message source: wordGenerator:15, stream: default, id: {}, [jackson]
76894 [Thread-22-counter] INFO backtype.storm.daemon.executor - Processing received message source: wordGenerator:13, stream: default, id: {}, [jackson]
76985 [Thread-26-finalRanker] INFO backtype.storm.daemon.executor - Processing received message source: __system:-1, stream: __tick, id: {}, [2]
76985 [Thread-26-finalRanker] INFO backtype.storm.daemon.task - Emitting: finalRanker default [[[mike|108|[[[9]]]], [golda|91|[[[9]]]], [jackson|90|[[[9]]]]], [nathan|73|[[[9]]]]]]
76986 [Thread-28-intermediateRanker] INFO backtype.storm.daemon.executor - Processing received message source: __system:-1, stream: __tick, id: {}, [2]
76987 [Thread-28-intermediateRanker] INFO backtype.storm.daemon.task - Emitting: intermediateRanker default [[[golda|98|[[[9]]]]]]
76987 [Thread-26-finalRanker] INFO backtype.storm.daemon.executor - Processing received message source: intermediateRanker:7, stream: default, id: {}, [[
76995 [Thread-36-wordGenerator] INFO backtype.storm.daemon.task - Emitting: wordGenerator default [nathan]
76995 [Thread-44-wordGenerator] INFO backtype.storm.daemon.task - Emitting: wordGenerator default [nathan]
76995 [Thread-40-wordGenerator] INFO backtype.storm.daemon.task - Emitting: wordGenerator default [nathan]
76995 [Thread-42-wordGenerator] INFO backtype.storm.daemon.task - Emitting: wordGenerator default [nathan]
76995 [Thread-38-wordGenerator] INFO backtype.storm.daemon.task - Emitting: wordGenerator default [nathan]
76995 [Thread-24-counter] INFO backtype.storm.daemon.executor - Processing received message source: wordGenerator:15, stream: default, id: {}, [nathan]
76995 [Thread-24-counter] INFO backtype.storm.daemon.executor - Processing received message source: wordGenerator:13, stream: default, id: {}, [nathan]
76995 [Thread-24-counter] INFO backtype.storm.daemon.executor - Processing received message source: wordGenerator:14, stream: default, id: {}, [nathan]
```

# Behind the scenes of RollingTopWords



<http://www.michael-noll.com/blog/2013/01/18/implementing-real-time-trending-topics-in-storm/>

# kafka-storm-starter



- Written by yours truly
- <https://github.com/miguno/kafka-storm-starter>

```
$ git clone https://github.com/miguno/kafka-storm-starter
$ cd kafka-storm-starter

# Now ready for mayhem!
```

(Must have JDK 7 installed.)



# kafka-storm-starter: run the test suite

```
$ ./sbt test
```

- Will run unit tests plus end-to-end tests of Kafka, Storm, and Kafka-Storm integration.

```
[info] And I Avro-encode the tweets and use the Kafka producer app to sent them to Kafka
[info] Then the Kafka consumer app should receive the original tweets from the Storm topology
[info] Feature: AvroScheme[T] for Kafka spout
[info] Scenario: User creates a Storm topology that uses AvroScheme in Kafka spout
[info] Given a ZooKeeper instance
[info] And a Kafka broker instance
[info] And a Storm topology that uses AvroScheme and that reads tweets from topic testing-input and writes them as-is to topic tes
[info] And some tweets
[info] And a synchronous Kafka producer app that writes to the topic testing-input
[info] And a single-threaded Kafka consumer app that reads from topic testing-output and Avro-decodes the incoming data
[info] And a Storm topology configuration that registers an Avro Kryo decorator for Tweet
[info] When I run the Storm topology
[info] And I Avro-encode the tweets and use the Kafka producer app to sent them to Kafka
[info] Then the Kafka consumer app should receive the original tweets from the Storm topology
[info] Run completed in 28 seconds, 597 milliseconds.
[info] Total number of tests run: 25
[info] Suites: completed 8, aborted 0
[info] Tests: succeeded 25, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
27901 [Thread-147-kafka-spout-EventThread] INFO com.netflix.curator.framework.state.ConnectionStateManager - State change: SUSPENDED
27901 [Thread-106-kafka-spout-EventThread] INFO com.netflix.curator.framework.state.ConnectionStateManager - State change: SUSPENDED
27901 [Thread-147-kafka-spout-EventThread] INFO com.netflix.curator.framework.state.ConnectionStateManager - State change: SUSPENDED
27901 [Thread-106-kafka-spout-EventThread] INFO com.netflix.curator.framework.state.ConnectionStateManager - State change: SUSPENDED
27901 [ConnectionStateManager-0] WARN com.netflix.curator.framework.state.ConnectionStateManager - There are no ConnectionStateLister
27903 [ConnectionStateManager-0] WARN com.netflix.curator.framework.state.ConnectionStateManager - There are no ConnectionStateLister
27905 [ConnectionStateManager-0] WARN com.netflix.curator.framework.state.ConnectionStateManager - There are no ConnectionStateLister
27905 [ConnectionStateManager-0] WARN com.netflix.curator.framework.state.ConnectionStateManager - There are no ConnectionStateLister
[success] Total time: 40 s, completed Jun 10, 2014 2:44:03 PM
➤:kafka-storm-starter (develop) $
```

# kafka-storm-starter: run the KafkaStormDemo app

```
$ ./sbt run
```

- Starts in-memory instances of ZooKeeper, Kafka, and Storm. Then runs a Storm topology that reads from Kafka.

```
7031 [Thread-19] INFO backtype.storm.daemon.worker - Worker 3f7f1a51-5c9e-43a5-b431-e39a7272215e for storm l
7033 [Thread-29-kafka-spout] INFO storm.kafka.DynamicBrokersReader - Read partition info from zookeeper: Glo
7050 [Thread-29-kafka-spout] INFO backtype.storm.daemon.executor - Opened spout kafka-spout:(1)
7051 [Thread-29-kafka-spout] INFO backtype.storm.daemon.executor - Activating spout kafka-spout:(1)
7051 [Thread-29-kafka-spout] INFO storm.kafka.ZkCoordinator - Refreshing partition manager connections
7065 [Thread-29-kafka-spout] INFO storm.kafka.DynamicBrokersReader - Read partition info from zookeeper: Glo
7066 [Thread-29-kafka-spout] INFO storm.kafka.ZkCoordinator - Deleted partition managers: []
7066 [Thread-29-kafka-spout] INFO storm.kafka.ZkCoordinator - New partition managers: [Partition{host=127.0
7083 [Thread-29-kafka-spout] INFO storm.kafka.PartitionManager - Read partition information from: /kafka-sp
7100 [Thread-29-kafka-spout] INFO storm.kafka.PartitionManager - No partition information found, using conf:
7105 [Thread-29-kafka-spout] INFO storm.kafka.PartitionManager - Starting Kafka 127.0.0.1:0 from offset 18
7106 [Thread-29-kafka-spout] INFO storm.kafka.ZkCoordinator - Finished refreshing
7126 [Thread-29-kafka-spout] INFO storm.kafka.PartitionManager - Committing offset for Partition{host=127.0
7126 [Thread-29-kafka-spout] INFO storm.kafka.PartitionManager - Committed offset 18 for Partition{host=127
9128 [Thread-29-kafka-spout] INFO storm.kafka.PartitionManager - Committing offset for Partition{host=127.0
9129 [Thread-29-kafka-spout] INFO storm.kafka.PartitionManager - Committed offset 18 for Partition{host=127
```

# Storm related code in kafka-storm-starter

- AvroDecoderBolt[T]
  - <https://github.com/miguno/kafka-storm-starter/blob/develop/src/main/scala/com/miguno/kafkastorm/storm/AvroDecoderBolt.scala>
- AvroScheme[T]
  - <https://github.com/miguno/kafka-storm-starter/blob/develop/src/main/scala/com/miguno/kafkastorm/storm/AvroScheme.scala>
- AvroKafkaSinkBolt[T]
  - <https://github.com/miguno/kafka-storm-starter/blob/develop/src/main/scala/com/miguno/kafkastorm/storm/AvroKafkaSinkBolt.scala>
- StormSpec: test-drives Storm topologies
  - <https://github.com/miguno/kafka-storm-starter/blob/develop/src/test/scala/com/miguno/kafkastorm/integration/StormSpec.scala>

# Storm performance tuning

# Storm performance tuning

- Unfortunately, no silver bullet and no free lunch. Witchcraft?
- And what is “the best” performance in the first place?
  - Some users require a low latency, and are willing to let most of the cluster sit idle as long they can process a new event quickly once it happens.
  - Other users are willing to sacrifice latency for minimizing the hardware footprint to save \$\$\$\$. And so on.
- P&S tuning depends very much on the actual use cases
  - Hardware specs, data volume/velocity/..., etc.
  - Which means in practice:
    - What works with sampled data may not work with production-scale data.
    - What works for topology A may not work for topology B.
    - What works for team A may not work for team B.
  - Tip: Be careful when adopting other people’s recommendations if you don’t fully understand what’s being tuned, why, and in which context.

# General considerations

- **Test + measure: use Storm UI, Graphite & friends**
- Understand your topology's DAG on a macro level
  - Where and how data flows, its volume, joins/splits, etc.
    - Trivial example: Shoveling 1Gbps into a “singleton” bolt = WHOOPS
- Understand ... on a micro level
  - How your data flows between machines, workers, executors, tasks.
  - Where and when serialization happens.
  - Which queues and buffers your data will hit.
  - We talk about this in detail in the next slides!
- Best performance optimization is often to stop doing something.
  - Example: If you can cut out (de-)serialization and sending tuples to another process, even over the loopback device, then that is potentially a big win.

<http://www.slideshare.net/ptgoetz/scaling-storm-hadoop-summit-2014>

<http://www.slideshare.net/JamesSirota/cisco-opensoc>

# How to approach P&S tuning

- Optimize locally before trying to optimize globally
  - Tune individual spouts/bolts before tuning entire topology.
    - Write simple data generator spouts and no-op bolts to facilitate this.
  - Even small things count at scale
    - A simple string operation can slowdown throughput when processing 1M tuples/s
- Turn knobs slowly, one at a time
  - A common advice when fiddling with a complex system.
- Add your own knobs
  - It helps to make as many things configurable as possible.
- Error handling is critical
  - Poorly handled errors can lead to topology failure, data loss or data duplication.
  - Particularly important when interfacing Storm with other systems such as Kafka.

<http://www.slideshare.net/ptgoetz/scaling-storm-hadoop-summit-2014>

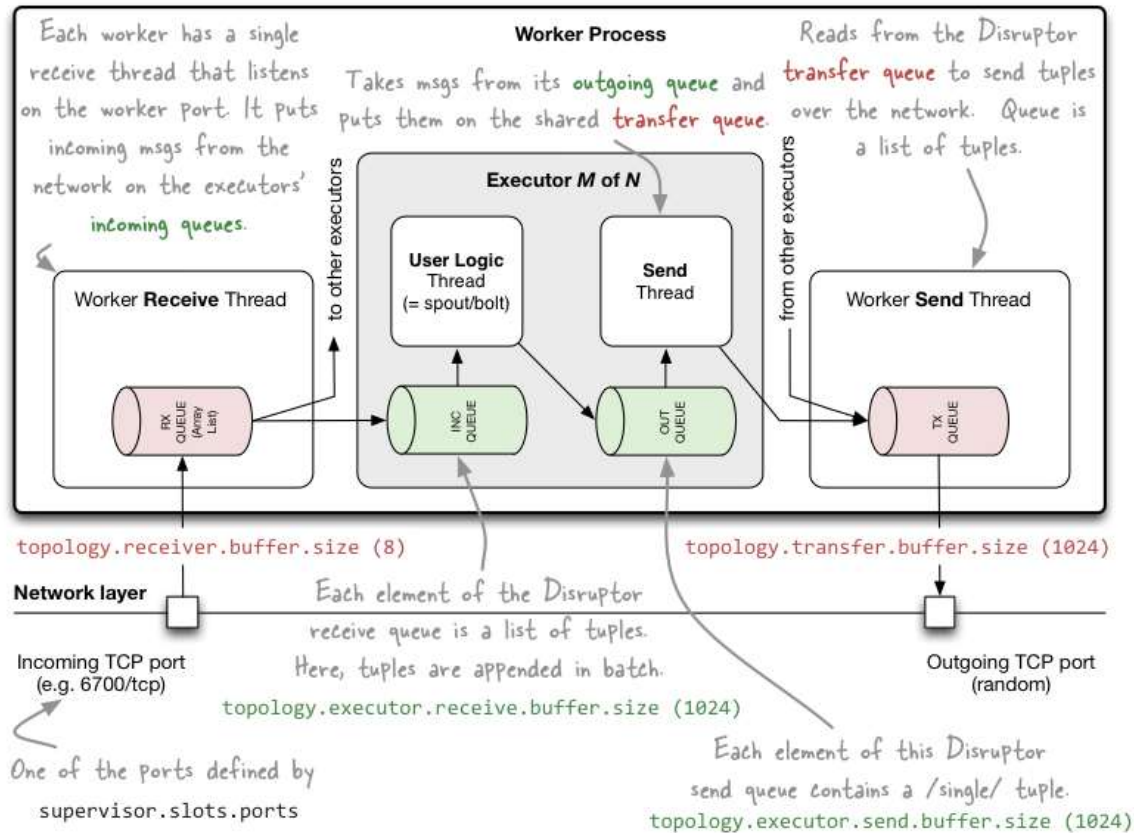
<http://www.slideshare.net/JamesSirota/cisco-opensoc>

# Some rules of thumb, for guidance

- CPU-bound topology?
  - Try to spread and parallelize the load across cores (think: workers).
    - Local cores: may incur serialization/deserialization costs, see later slides.
    - Remote cores: will incur serialization/deserialization costs, plus network I/O and additional Storm coordination work.
- Network I/O bound topology?
  - Collocate your cores, e.g. try to perform more logical operations per bolt.
  - Breaks single responsibility principle (SRP) in favor of performance.
- But what if topology is CPU-bound *and* I/O-bound *and* ...?
  - It becomes very tricky when parts of your topology are CPU-bound, other parts are I/O bound, and other parts are constrained by memory (which has it's own limitations).
  - Grab a lot of coffee, and good luck!



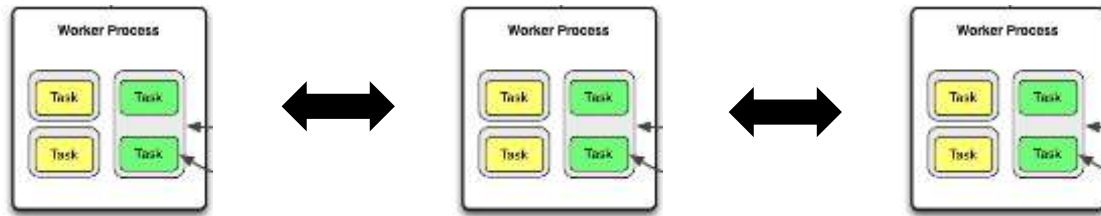
# Internal message buffers of Storm (as of 0.9.1)



**Update August 2014:** This setup may have changed due to recent P&S work in [STORM-297](https://storm.apache.org/releases/2.0.0/2014-08-08.html).

<http://www.michael-noll.com/blog/2013/06/21/understanding-storm-internal-message-buffers/>

# Communication within a Storm cluster



- **Intra-worker** communication: [LMAX Disruptor](#) <<< awesome library!
  - Between local threads within the same worker process (JVM), e.g. between local tasks/executors of same topology
  - Flow is: emit() -> executor A's send buffer -> executor B's receive buffer.
  - Does not hit the parent worker's transfer buffer. Does not incur serialization because it's in the same JVM.
- **Inter-worker** communication: Netty in Storm 0.9+, ZeroMQ in 0.8
  - Different JVMs/workers on same machine
    - emit() -> exec send buffer -> worker A's transfer queue -> **local socket** -> worker B's recv queue -> exec recv buffer
  - Different machines.
    - Same as above, but uses a **network socket** and thus also hits the NIC. Incurs additional latency because of network.
  - Inter-worker communication incurs serialization overhead (passes JVM boundaries), cf. [Storm serialization with Kryo](#)
- **Inter-topology** communication:
  - Nothing built into Storm – up to you! Common choices are a messaging system such as Kafka or Redis, an RDBMS or NOSQL database, etc.
  - Inter-topology communication incurs serialization overhead, details depend on your setup

# Tuning internal message buffers

- Start with the following settings if you think the defaults aren't adequate

Config	Default	Tuning guess	Notes
<code>topology.receiver.buffer.size</code>	8	8	
<code>topology.transfer.buffer.size</code>	1,024	32	<i>Batches of messages</i>
<code>topology.executor.receive.buffer.size</code>	1,024	16,384	<i>Batches of messages</i>
<code>topology.executor.send.buffer.size</code>	1,024	16,384	<i>Individual messages</i>

- Helpful references
  - [Storm default configuration \(defaults.yaml\)](#)
  - [Tuning and Productionization of Storm](#), by Nathan Marz
  - [Notes on Storm+Trident Tuning](#), by Philip Kromer
  - [Understanding the Internal Message Buffers of Storm](#), by /me

# JVM garbage collection and RAM

- Garbage collection woes
  - If you are GC'ing too much and failing a lot of tuples (which may be in part due to GCs) it is possible that you are out of memory.
  - Try to increase the JVM heap size (-Xmx) that is allocated for each worker.
  - Try the G1 garbage collector in JDK7u4 and later.
- But: A larger JVM heap size is not always better.
  - When the JVM will eventually garbage-collect, the GC pause may take much longer for larger heaps.
  - Example: A GC pause will also temporarily stop those threads in Storm that perform the heartbeating. So GC pauses can potentially make Storm think that workers have died, which will trigger “recovery” actions etc. This can cause cascading effects.

# Rate-limiting topologies

- `topology.max.spout.pending`
  - Max number of tuples that can be pending on a single spout task at once. “Pending” means the tuple has either failed or has not been acked yet.
  - Typically, increasing max pending tuples will increase the throughput of your topology. But in some cases decreasing the value may be required to increase throughput.
  - Caveats:
    - This setting has no effect for unreliable spouts, which don't tag their tuples with a message id.
    - For Trident, `maxSpoutPending` refers to the number of pipelined *batches* of tuples.
      - Recommended to not setting this parameter very high for Trident topologies (start testing with ~ 10).
  - Primarily used a) to throttle your spouts and b) to make sure your spouts don't emit more than your topology can handle.
    - If the complete latency of your topology is increasing then your tuples are getting backed up (bottlenecked) somewhere downstream in the topology.
    - If some tasks run into “OOM: GC overhead limit exceeded” exception, then typically your upstream spouts/bolts are outpacing your downstream bolts.
  - Apart from throttling your spouts with this setting you can of course also try to increase the topology's parallelism (maybe you actually need to combine the two).

# Acking strategies

- `topology.acker.executors`
  - Determines the number of executor threads (or tasks?) that will track tuple trees and detect when a tuple has been fully processed.
  - Disabling acking trades reliability for performance.
  - If you want to **enable** acking and thus [guaranteed message processing](#)
    - Rule of thumb: 1 acker/worker (which is also the default in Storm 0.9)
  - If you want to **disable** acking and thus guaranteed message processing
    - Set value to 0. Here, Storm will immediately ack tuples as soon as they come off the spout, effectively disabling acking and thus reliability.
    - Note that there are two additional ways to fine-tune acking behavior, and notably to disable acking:
      1. Turn off acking for an individual spout by omitting a message id in the `SpoutOutputCollector.emit()` method.
      2. If you don't care if a particular subset of tuples is failed to be processed downstream in the topology, you can emit them as unanchored tuples. Since they're not anchored to any spout tuples, they won't cause any spout tuples to fail if they aren't acked.

# Miscellaneous

- A worker process is never shared across topologies.
  - If you have Storm configured to run only a single worker on a machine, then you can't run multiple topologies on that machine.
  - Spare worker capacity can't be used by other topos. All the worker's child executors and tasks will only ever be used to run code for a single topology.
- All executors/tasks on a worker run in the same JVM.
  - In some cases – e.g. a `localOrShuffleGrouping()` – this improves performance.
  - In other cases this can cause issues.
    - If a task crashes the JVM/worker or causes the JVM to run out of memory, then all other tasks/executors of the worker die, too.
    - Some applications may malfunction if they co-exist as multiple instances in the same JVM, e.g. when relying on static variables.

# Miscellaneous

- Consider the use of Trident to increase throughput
  - Trident inherently operates on batches of tuples.
  - Drawback is typically a higher latency.
  - Trident is not covered in this workshop. 😊
- Experiment with batching messages/tuples manually
  - Keep in mind that here a failed tuple actually corresponds to multiple data records.
  - For instance, if a batch “tuple” fails and gets replayed, all the batched data records will be replayed, which may lead to data duplication.
  - If you don’t like the idea of manual batching, try Trident!



# When using Storm with Kafka

- Storm's parallelism is controlled by Kafka's "parallelism"
  - Set Kafka spout's parallelism to #partitions of source topic.
- Other key parameters that determine performance
  - [KafkaConfig.fetchSizeBytes](#) (default: 1 MB)
  - [KafkaConfig.bufferSizeBytes](#) (default: 1 MB)

# TL;DR: Start with this, then measure/improve/repeat

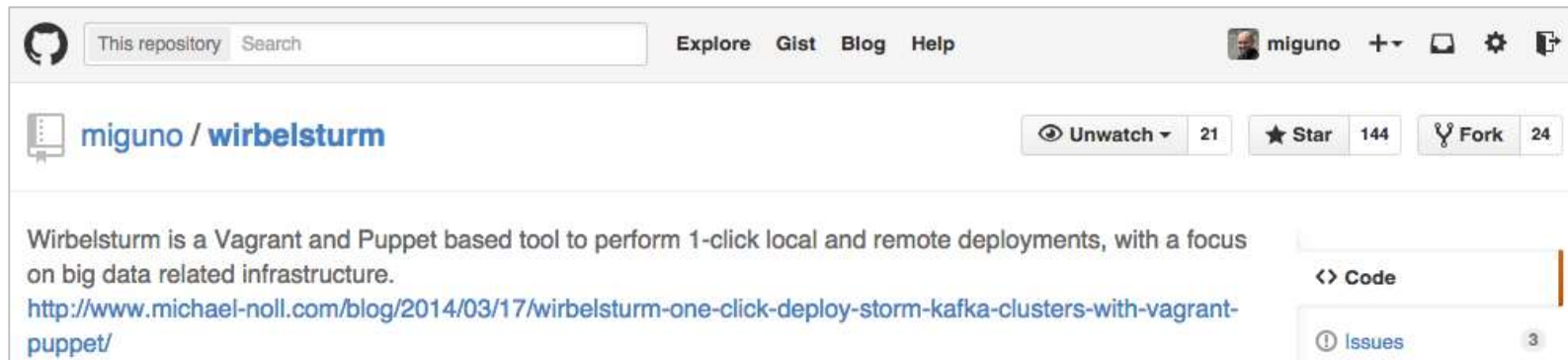
- 1 worker / machine / topology
  - Minimize unnecessary network transfer
- 1 acker / worker
  - This is also the default in Storm 0.9
- CPU-bound use cases:
  - 1 executor thread / CPU core, to optimize thread and CPU usage
- I/O-bound use cases:
  - 10-100 executor threads / CPU core

<http://www.slideshare.net/ptgoetz/scaling-storm-hadoop-summit-2014>

# Part 5: Playing with Storm using Wirbelsturm

1-click Storm deployments

# Deploying Storm via Wirbelsturm



- Written by yours truly
- <https://github.com/miguno/wirbelsturm>

```
$ git clone https://github.com/miguno/wirbelsturm.git
$ cd wirbelsturm
$ ./bootstrap
$ vagrant up zookeeper1 nimbus1 supervisor1 supervisor2
```

(Must have Vagrant 1.6.1+ and VirtualBox 4.3+ installed.)

# Deploying Storm via Wirbelsturm

- By default, the Storm UI runs on nimbus1 at:
  - <http://localhost:28080/>
- You can build and run a topology:
  - Beyond the scope of this workshop.
  - Use e.g. an Ansible playbook to submit topologies to make this task simple, easy, and fun.

# What can I do with Wirbelsturm?

- Get a first impression of Storm
- Test-drive your topologies
- Test failure handling
  - Stop/kill Nimbus, check what happens to Supervisors.
  - Stop/kill ZooKeeper instances, check what happens to topology.
- Use as sandbox environment to test/validate deployments
  - “What will actually happen when I deactivate this topology?”
  - “Will my Hiera changes actually work?”
- Reproduce production issues, share results with Dev
  - Also helpful when reporting back to Storm project and mailing lists.
- Any further cool ideas? 😊

# Wrapping up

# Where to go from here

- A few Storm books are already available.
- Storm documentation
  - <http://storm.incubator.apache.org/documentation/Home.html>
- storm-kafka
  - <https://github.com/apache/incubator-storm/tree/master/external/storm-kafka>
- Mailing lists
  - <http://storm.incubator.apache.org/community.html>
- Code examples
  - <https://github.com/apache/incubator-storm/tree/master/examples/storm-starter>
  - <https://github.com/miguno/kafka-storm-starter/>
- Related work aka tools that are similar to Storm – try them, too!
  - [Spark Streaming](#)
    - See comparison [Apache Storm vs. Apache Spark Streaming](#), by P. Taylor Goetz (Storm committer)



powered by



**VERISIGN®**