INTRODUCTION TO APACHE STORM

Tiziano De Matteis

Ph.D. Course in Perspective in Parallel Computing

OUTLINE

- Introduction
- Apache Storm:
  - Basics
  - Guaranteed Processing
  - Internals
  - Usage examples
- Discussion
- Conclusions

INTRODUCTION

Apache Storm is a real-time fault-tolerant and distributed Stream Processing Engine (SPE)

SPEs tackle an application space in which programs have in input continuous streams of information that has to be processed as it arrives :

- ▷ use cases: *financial applications*, *network monitoring*, *social network analysis*, etc…
- ▷ different from traditional *batch systems* (store and process).

This is in line to what we have studied in the Ph.D. course (data arrives from external source and computations are long running and often stateful).

SPEs EVOLUTION

1. Early **2000s**, centralized systems (*DSMS*, *CEP*);
2. Till **2008**: evolution to distributed systems;
3. From **2009**: “general-purpose” systems (*SPS* o *SPE*) from academia or big companies for which Hadoop was not sufficient

<u>Some names</u>: *S4* (Yahoo), *Samza* (Linkedin), *Millwheel*(Google), *Storm* (Twitter), *Spark Streaming*, *InfoSphereStream* (IBM), Kinesis (Amazon) …

APACHE STORM

History: Backtype → Twitter (2011) → Apache Incubator (sept. 2013) → Apache (sept. 2014)

Written in Clojure, it is *language agnostic* (Java is the natural choice and we will use it in the examples).

Core concepts:

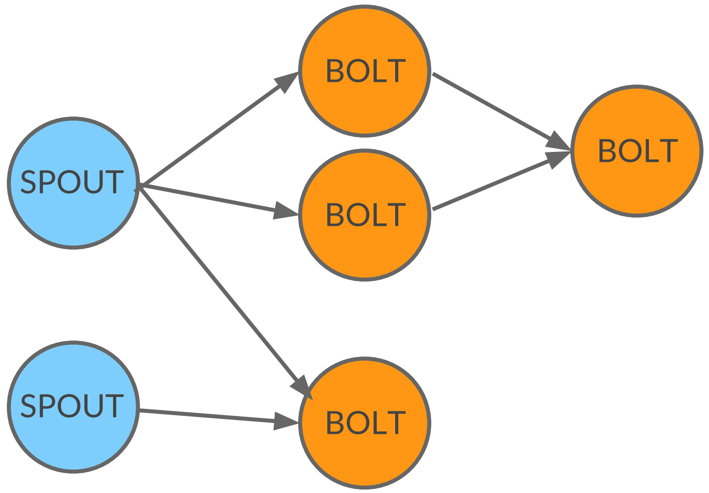
- ***tuple***: a named list of values;
- ***stream***: a (possibly) unbounded sequence of tuples processed by the application.

Basics

BASICS

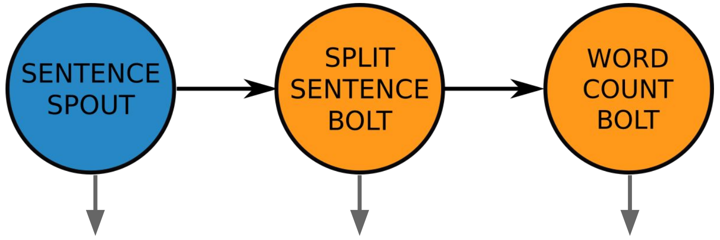
A Streaming Application is defined in Storm by means of a ***topology*** that describes its logic as a graph of operators and streams. We can have two types of operators:

- ***spouts***: are the sources of streams in a topology. Generally will read tuples from external sources (e.g. Twitter API) or from disk and emit them in the topology;
- ***bolts***: processes input streams and produce output streams. They encapsulate the application logic.



TOPOLOGY EXAMPLE

Word Count: count the different words in a stream of sentences



Implemented by 3 *classes* and composed to obtain the desired topology:

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("sentencs-spout", new SentenceSpout());
builder.setBolt("split-bolt", new SplitSentenceBolt())
     .shuffleGrouping("sentences-spout");
builder.setBolt("count-bolt", new WordCountBolt())
     .fieldsGrouping("split-bolt", new Fields("word"));
```

SentenceSpout: emits a stream of tuples that represent sentences:

```
{sentence: "my dog has fleas"}
```

SplitSentenceBolt: emits a tuple for each word in the sentences it receives:

```
{word: "my"}; {word:"dog"}; …
```

WordCountBolt: updates the count and at save it to file.

TOPOLOGY EXAMPLE

Each node extends some abstract classes and must implements some basic methods for defining the format of the tuple emitted, the logic and so on….

```
public class SentenceSpout extends BaseRichSpout {
     private SpoutOutputCollector collector;

     public void declareOutputFields(OutputFieldsDeclarer declarer) {
          declarer.declare(new Fields("sentence"));
     }

     public void open(Map config, TopologyContext context,SpoutOutputCollector collector) {
          this.collector = collector;
     }

     public void nextTuple() {
          //prepare the next sentence S to emit
          this.collector.emit(new Values(S));
          //...
     }
}
```

TOPOLOGY EXAMPLE

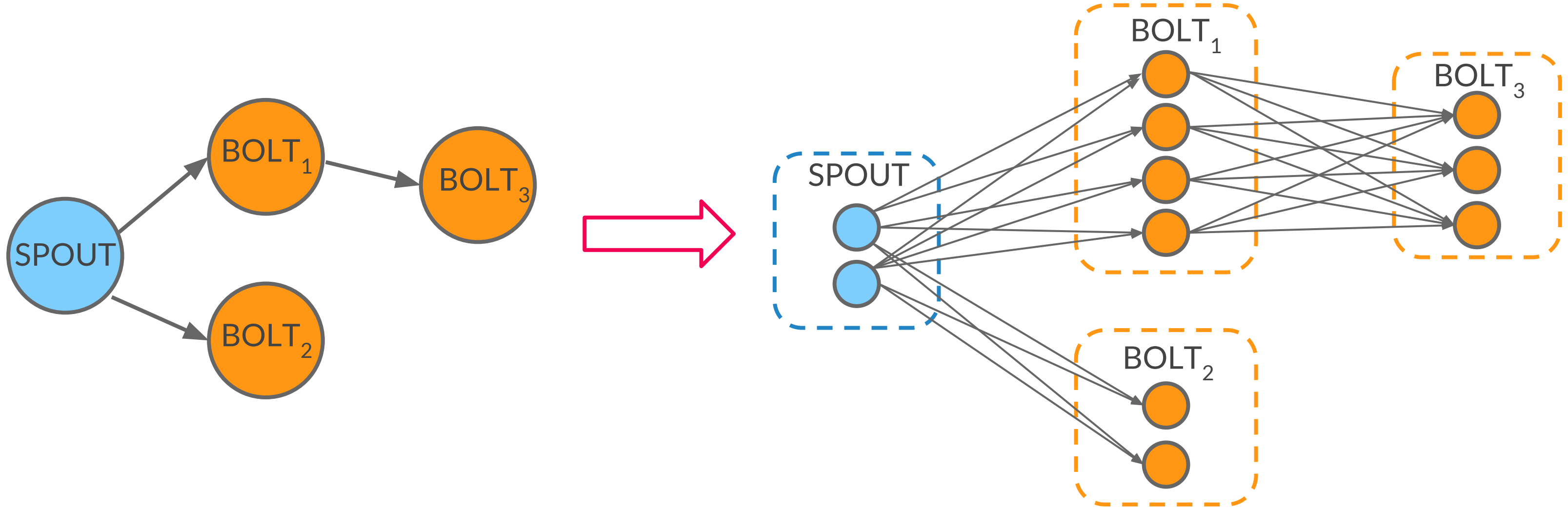
```
public class SplitSentenceBolt extends BaseRichBolt{
     private OutputCollector collector;

     public void declareOutputFields(OutputFieldsDeclarer declarer) {
          declarer.declare(new Fields("word"));
     }

     public void prepare(Map config, TopologyContext context,OutputCollector collector) {
          this.collector = collector;
     }

     public void execute(Tuple tuple) {
          String sentence = tuple.getStringByField("sentence");
          String[] words = sentence.split(" ");
          for(String word : words){
               this.collector.emit(new Values(word));
          }
     }
}
```

TOPOLOGY EXAMPLE

```
public class WordCountBolt extends BaseRichBolt{
     private OutputCollector collector;
     private HashMap<String,Long> counts=null;

     public void declareOutputFields(OutputFieldsDeclarer declarer) {
          //this bolt does not emit anything
     }

     public void prepare(Map config, TopologyContext context,OutputCollector collector) {
          this.collector = collector;
          this.counts = new HashMap<String,Long>();
     }

     public void execute(Tuple tuple) {
          String word = tuple.getStringByField("word");
          //...increments count..
     }
}
```

APPLICATION DEPLOYING

When executed, the topology is deployed as a set of processing entities over a set of computational resources (typically a cluster). Parallelism is achieved in Storm by running multiple replicas of the same spout/bolt:



Groupings specify how tuples are routed to the various replicas

GROUPINGS

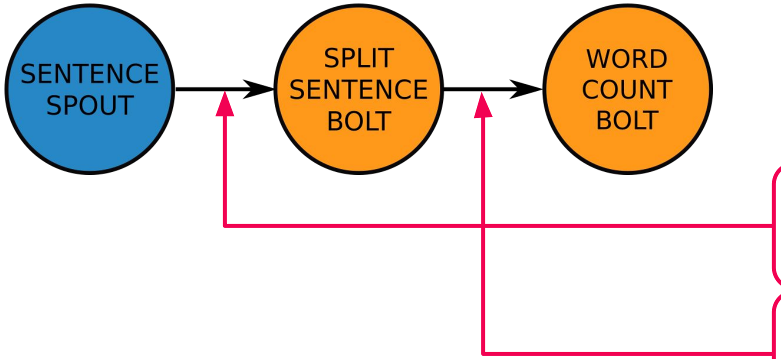
There are 7 built-in possibilities, the most interesting are:

- *shuffle grouping*: tuples are randomly distributed;
- *field grouping*: the stream is partitioned according to a tuple attribute. Tuples with the same attribute will be scheduled to the same replica;
- *all grouping*: tuples are replicated to all replicas;
- *direct grouping*: the producer decides the destination replica
- *global grouping*: all the tuples go to the same replica (low. ID).

Users have also the possibility of implementing their own grouping through the `CustomStreamGrouping` interface

EXAMPLE

Back to the Words Count example: grouping are specified while we build the topology



```
TopologyBuilder builder = new TopologyBuilder();


builder.setSpout("sentences-spout", new SentenceSpout());
builder.setBolt("split-bolt", new SplitSentenceBolt())
        .shuffleGrouping("sentences-spout");
builder.setBolt("count-bolt", new WordCountBolt())
        .fieldsGrouping("split-bolt", new Fields("word"));
```

TOPOLOGY EXAMPLE

Finally, let’s see how the main will look like:

```
public class WordCountTopology {
     public static void main(String[] args) throws Exception {

          //…Topology construction…
          Config config = new Config();

          LocalCluster cluster = new LocalCluster();

          cluster.submitTopology(TOPOLOGY_NAME, config, builder.createTopology());

          waitForSeconds(10);
          cluster.killTopology(TOPOLOGY_NAME);
          cluster.shutdown();
     }
}
```

Guaranteed Processing

GUARANTEED PROCESSING

Storm provides an API to guarantee that a tuple emitted by a spout is fully processed by the topology (*at-least-once* semantic).

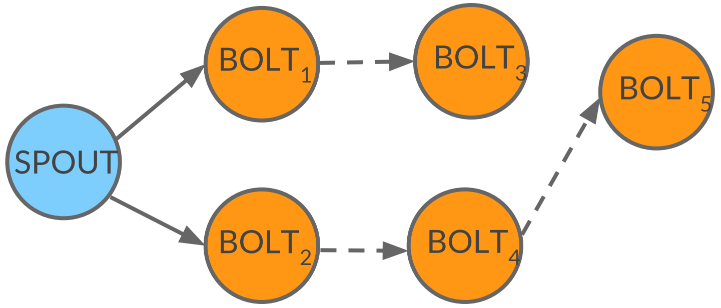
A tuple coming off a spout can trigger thousands of tuples to be created based on it. Consider the WordCount example:

- the spout generates sentences (*tuples*);
- the bolt generates a set of words for each sentence (*child tuples*).

A tuple is fully processed *iff it and all its child tuples have been correctly processed* by the topology.

GUARANTEED PROCESSING

Another way to look at it is to consider the tuple tree:



- the solid lines represent the tuple emitted by the spout;
- the dashed ones the child tuples generated by the bolts.

With guaranteed processing, each bolt in the tree can either *acknowledge* or *fail* a tuple:

- If all bolts in the tree acknowledge the tuple and child tuples the message processing is *complete*;
- if any bolts explicitly fail a tuple, or we exceed a time-out period, the processing is failed.

GUARANTEED PROCESSING

From the Spout side, we have to keep track of the tuple emitted and be prepared to handle fails:

```
public void nextTuple() {
     //prepare the next sentence S to emit
     this.collector.emit(new Values(S), msgID);
     //...
}
```

Assign a unique ID to any emitted tuple

```
public void ack(Object msgID) {
     //handle success
     //...
}

public void fail(Object msgID) {
     //handle failure
     //...
}
```

Implement the `ack` and `fail` methods for handling successes and failures

GUARANTEED PROCESSING

On the Bolts side, we have to *anchor* any emitted tuple to the originating one and acknowledging or failing tuples

```
public void execute(Tuple tuple) {
     //… processing...
     this.collector.emit(tuple, new Values(word));


     //acknowledgment
     this.collector.ack(tuple);

     //or, if something goes wrong, fail
     this.collector.fail(tuple);

}
```

← Anchoring (through overloaded `emit` method)

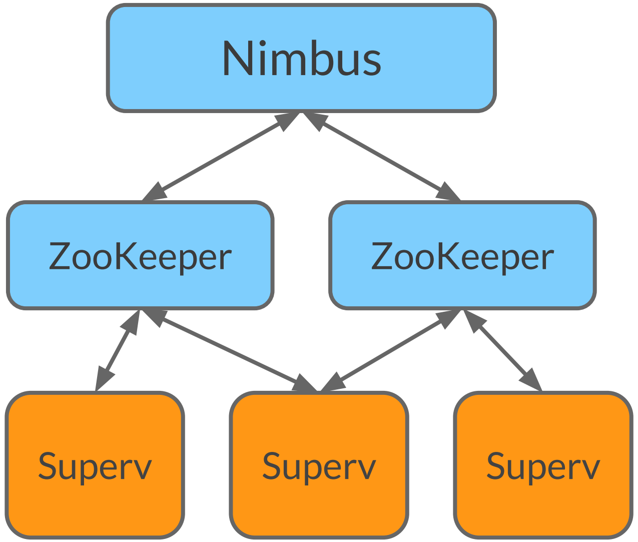
← Ack or fail the tuple

Internal Architecture

STORM ARCHITECTURE

Two kinds of nodes in a Storm cluster:

- **Master node:** runs the *Nimbus*, a central job master to which topologies are submitted . It is in charge of scheduling, job orchestration, communication and fault tolerance;
- **Worker nodes**: nodes of the cluster in which applications are executed. Each of them run a Supervisor, that communicates with the Nimbus about topologies and available resources.

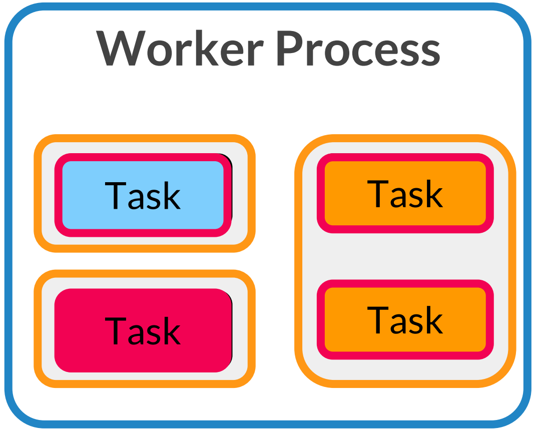


The coordination between this two entities is done through *Zookeper* that is used also for their fault tolerance

STORM ARCHITECTURE

Three entities are involved in running a topology:

▷ **Worker**: 1+ per cluster node, each one is related to one topology;



▷ Executor: thread spawned by the Worker. It runs one or more tasks for the same component (bolt or spout);

▷ Task: a component replica.

Therefore Workers provide inter-topology parallelism, Executors intra-topology and Tasks intra-component.

By default there is a 1:1 association between Executor and Tasks

```
builder.setBolt("split-bolt", new SplitSentenceBolt(),2).setNumTasks(4)
        .shuffleGrouping("sentences-spout");
```

Parallelism Hint

Usage Examples

ON TOP OF STORM

Storm is having a discrete success and various libraries/frameworks have been developed on top of it. Just to name a few:

▷ *Storm Trident*: a library that provides micro-batching and high level contructs (e.g. groupBy, aggregates, join);

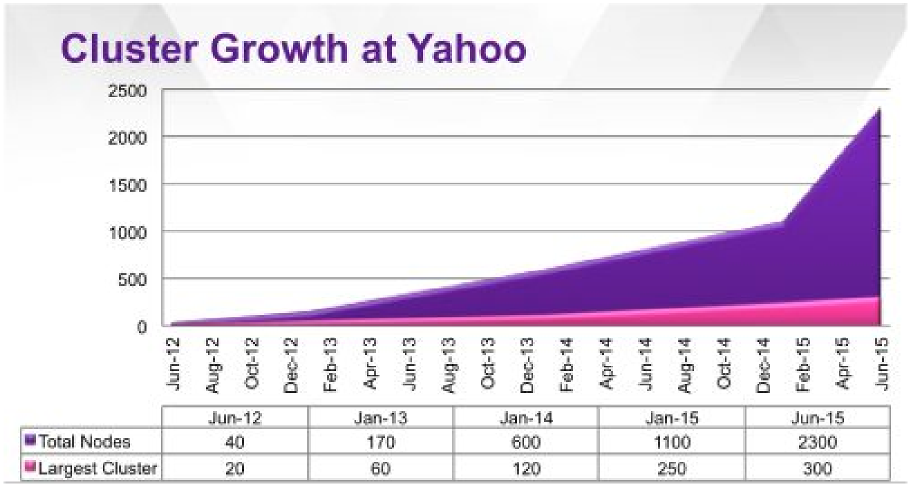
▷ *Yahoo/Apache Samoa* [3]: a distributed streaming machine learning (ML) framework that can run on top of Storm;

▷ *Twitter Summingbird* [4]: streaming Map Reduce.

USAGE EXAMPLES: STORM @YAHOO

Various applications: identifying a breaking news story and promoting it, showing trending search terms as they happen, helping to identify and block SPAM, or letting advertisers see the impact of their campaigns as quickly as possible.

The number of nodes over which they deployed Storm applications constantly increases (doubled in the last six month) [5]



	Jun-12	Jan-13	Jan-14	Jan-15	Jun-15
■ Total Nodes	40	170	600	1100	2300
■ Largest Cluster	20	60	120	250	300

USAGE EXAMPLES: STORM @TWITTER

Twitter contributed a lot in the development of Storm

It runs on hundreds of servers, with several hundreds of topologies deployed [6]:

- these are used by various groups at Twitter like revenue, search, content discovery,...
- perform simple things (like filtering and aggregating the content of various streams) or complex things (like running machine learning algorithms on stream data);

BUT THIS WAS 2014...

USAGE EXAMPLES: STORM @TWITTER

In June 2015: Storm has been decommissioned and Heron (their own SPE, [7]) is now the de-facto streaming system at Twitter, for a variety of reasons:

- Each worker runs a mix of tasks, making it difficult to reason about the behaviour and performance of a particular task;
- Each tuple has to pass through four threads in the worker process. This design leads to significant overhead;
- Nimbus is functionally overloaded and becomes an operational bottleneck;
- Storm workers belonging to different topologies but running on the same machine can interfere with each other;
- ...and others, mainly related to its implementation

Results: 7-10X improvements in throughput, and 5-10X reductions in tuple latencies, 3x reduction in resource consumption

Same philosophy adopted by other companies, e.g. *librato.com*

Discussion

COMPARISONS

We want to make a comparison between Storm wrt what we have seen in the Ph.D. course. We can do it from two points of view:

- methodological one: in the course we have seen how algo. skeletons/pattern could help programmer in developing their parallel application in a structured way. How Storm relate to this?
- technological one: as a framework

COMPARISON: METHODOLOGICAL

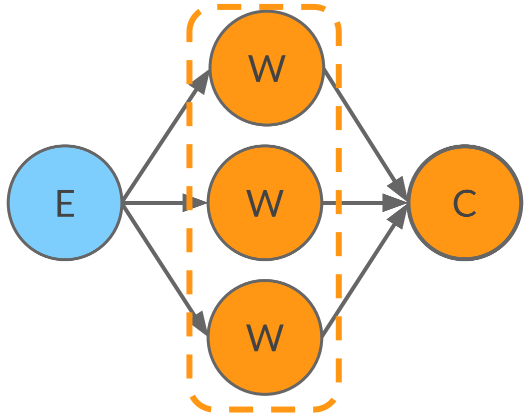
Storm provides a way of defining loosely structured parallel programs:

- the programmer specifies the topologies (i.e. the computational graphs);
- but can exploit limited possibilities for what concern the internal parallelization of the operator.

An idea could be of implement algorithmic skeletons as an abstraction on top of Storm: they will be “translated” in a set of spout/bolts that mimic their behaviour.

EXAMPLE: FARM ABSTRACTION

Example: Farm skeleton (Emitter, Worker, Collector) on top of Storm



- ▷ the Emitter is implemented as a Spout that receives/generates the stream of data and send it to Workers
- ▷ Workers have the same logic, therefore we can implement them as a Bolt whose number of Executor/Task is the equivalent of the desired parallelism degree. Shuffle grouping is used to distribute data
- ▷ Collector is implemented as another bolt (parallelism hint=1 if sequential)

This can be done also for other skeletons such as Pipeline, Map,...

COMPARISON: AS A FRAMEWORK

During the course we have seen various parallel framework (e.g. Fastflow). How they relate to Storm?

- we can express the same type of computations. In Storm it is required a major effort to the programmers since it does not exploit any structured way of composing parallel programs;

- on the other side, it should be noticed that Storm is a “production framework” in contrast to the others that are mainly research/academic products. It allows an easy the deploy of multiple application on a set of distributed resources, it takes into account fault tolerance mechanisms … but still there is a lot of work to do (e.g. autonomic management)

CONCLUSIONS

We have take a rapid tour on the features of Storm.

It is a complete framework for the development, deployment and maintenance of distributed stream applications.

But

- it is not a structured parallel frameworks;
- its internal architecture is a little bit confusing;
- actually it is not fault tolerant but provides mechanisms to implement this feature

BIBLIOGRAPHY

[1] Apache Storm Documentation: https://storm.apache.org/documentation/Home.html

[2] Goetz, P. Taylor, and Brian O'Neill. Storm blueprints: Patterns for distributed real-time computation. Packt Publishing Ltd, 2014.

[3] Morales, Gianmarco De Francisci, and Albert Bifet. "SAMOA: Scalable Advanced Massive Online Analysis." Journal of Machine Learning Research 16 (2015): 149-153.

[4] Oscar Boykin, Sam Ritchie, Ian O'Connell, and Jimmy Lin. 2014. Summingbird: a framework for integrating batch and online MapReduce computations. Proc. VLDB Endow. 7, 13 (August 2014), 1441-1451.

[5] Apache Storm Roadmap at Yahoo: http://yahoohadoop.tumblr.com/post/122544585051/apache-storm-roadmap-at-yahoo

BIBLIOGRAPHY

[6] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. 2014. Storm@twitter. In Proceedings of the 2014 ACM SIGMOD international conference on Management of data (SIGMOD '14). ACM, New York, NY, USA, 147-156.

[7] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15).

Thank you!

Questions?