

DSP Frameworks

Corso di Sistemi e Architetture per Big Data

A.A. 2024/25

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

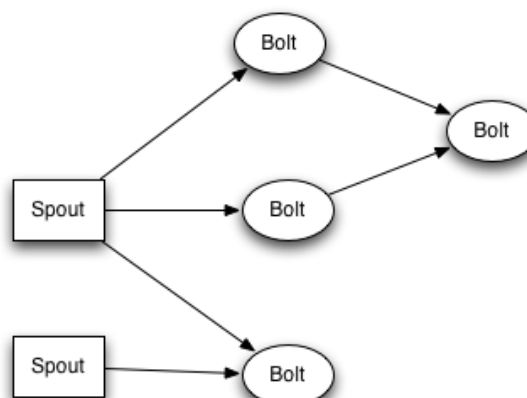
DSP frameworks we consider

- Apache Storm
- Apache Flink (plus hands-on lesson)
- Apache Spark Streaming (plus hands-on lesson)
- Kafka Streaming (hands-on lesson)
- Cloud-based frameworks
 - Google Cloud Dataflow
 - Amazon Kinesis

- Open-source, real-time, scalable streaming system
<https://storm.apache.org/>
- A distributed system, which provides an abstraction layer to execute DSP applications
- Many use cases, including real-time analytics, online ML, continuous computation, distributed RPC, ETL
- Fast: 1M tuples/sec. processed per node
- Easy to integrate with different sources (e.g., messaging systems)
- Initially developed by Twitter
- Current version: 2.8

Storm: topology

- Main Storm's concept: **topology**
 - Where the application logic is packaged into
 - Long-running
 - DAG of **spouts** (sources of streams) and **bolts** (operators that do processing as well as data sinks)
 - Top-level abstraction submitted to Storm for execution



Storm: streams and tuples

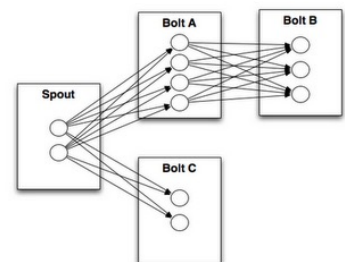
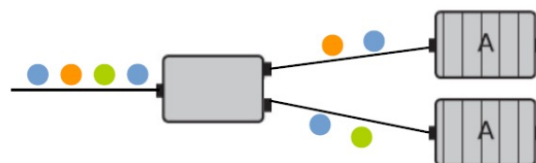
- Storm uses **streams and tuples** as its data model
 - Stream: core abstraction in Storm
 - Unbounded sequence of tuples
 - Storm provides functions for transforming a stream into a new stream in a distributed and reliable way
 - Streams are defined with a schema that names the fields in the stream's tuples
 - Tuple: named list of values
 - A field in a tuple can be an object of any type
 - Storm supports all primitive types, strings, and byte arrays as tuple field values
 - To use a custom data type, you need to define the corresponding serializer

Storm: stream grouping

- Stream grouping defines how to send tuples between two adjacent nodes in the topology
 - Remember of **data parallelism**: spouts and bolts execute in parallel (multiple threads of execution)

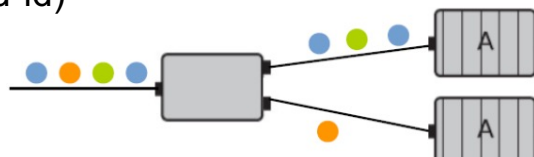
- **Shuffle grouping**

- Tuples are randomly partitioned



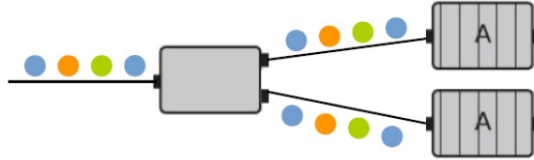
- **Field grouping**

- Stream is partitioned by the fields specified in the grouping (e.g., used-id)



Storm: stream grouping

- **All grouping** (i.e., broadcast)
 - Stream is replicated across all the bolt's replicas (use with care)



- **Global grouping**
 - Stream goes to a single one of the bolt's replicas (specifically, to the replica with the lowest id)
- **Direct grouping**
 - The producer of the tuple decides which replica of the consumer will receive this tuple

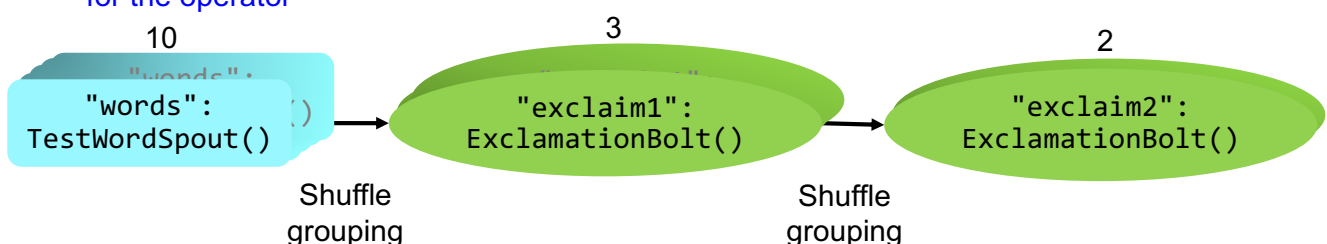
Storm: a simple topology

- **First example: exclamation**
 - Spout emits words, each bolt appends "!!!" to its input
- <https://github.com/apache/storm/blob/master/examples/storm-starter/src/jvm/org/apache/storm/starter/ExclamationTopology.java>

setSpout and setBolt methods take as input:

- user-specified id
- object containing the processing logic of the operator
- amount of parallelism for the operator

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("words", new TestWordSpout(), 10);
builder.setBolt("exclaim1", new ExclamationBolt(), 3)
    .shuffleGrouping("words");
builder.setBolt("exclaim2", new ExclamationBolt(), 2)
    .shuffleGrouping("exclaim1");
```



Storm: another topology

- Example: WordCount

```
TopologyBuilder builder = new TopologyBuilder();

builder.setSpout("sentences", new RandomSentenceSpout(), 5);
builder.setBolt("split", new SplitSentence(), 8)
    .shuffleGrouping("sentences");
builder.setBolt("count", new WordCount(), 12)
    .fieldsGrouping("split", new Fields("word"));
```


<https://github.com/apache/storm/blob/master/examples/storm-starter/src/jvm/org/apache/storm/starter/WordCountTopology.java>

- Bolts can be defined in any language
 - Bolts written in another language are executed as subprocesses, and Storm communicates with them using JSON messages over stdin/stdout
 - Communication protocol for Python available in an adapter library <https://streamparse.readthedocs.io>

Storm: windowing

- Storm supports both sliding and tumbling windows
<https://storm.apache.org/releases/current/Windowing.html>
- Windows can be based on time duration or event count
 - Count-based windows
 - Based on tuples count (no relation to clock time)
 - Time-based windows
 - Based on time duration
- Bolt that requires windowing support must implement **IWindowedBolt** interface

```
public interface IWindowedBolt extends IComponent {
    void prepare(Map stormConf, TopologyContext context, OutputCollector collector);
    /**
     * Process tuples falling within the window and optionally emit
     * new tuples based on the tuples in the input window.
     */
    void execute(TupleWindow inputWindow);
    void cleanup();
}
```

 **execute** is invoked every time the window activates

Storm: windowing

- Different window configurations, including
 - Sliding windows

```
withWindow(Count windowLength, Count slidingInterval)
Tuple count based sliding window that slides after slidingInterval number of tuples.

withWindow(Duration windowLength, Duration slidingInterval)
Time duration based sliding window that slides after slidingInterval time duration.
```

- Tumbling windows

```
withTumblingWindow(BaseWindowedBolt.Count count)
Count based tumbling window that tumbles after the specified count of tuples.

withTumblingWindow(BaseWindowedBolt.Duration duration)
Time duration based tumbling window that tumbles after the specified time duration.
```

<https://github.com/apache/storm/blob/master/examples/storm-starter/src/jvm/org/apache/storm/starter/SlidingWindowTopology.java>

- By default, tuples in the window are stored in memory until they are processed and expired
 - ✗ Windows need to fit entirely in memory
- Storm also supports stateful windowing

Storm: time and out-of-order tuples

- By default, Storm supports **processing time**
 - Time when an event is actually *processed* by DSP system
 - In Storm, the timestamp tracked in the window is the time when the tuple is processed by the bolt
- Source-generated timestamps are also supported
 - It requires that the spout generates a timestamp and includes it with each tuple it emits
- Out-of-order tuples are handled through
 - **Lateness bound** (aka *time lag*): specify max time limit for tuples with out-of-order timestamps; by default, late tuples are not processed
 - **Watermark**: defined as the minimum of the latest tuple timestamps (minus the lag) across all the input streams
 - User can change the interval at which watermarks are emitted

Storm: Stream API

- Alternative interface: provides a typed API for expressing streaming computations and supports **functional style** operations
 - Similarly to Spark and Flink, still experimental
<https://storm.apache.org/releases/current/Stream-API.html>
- Stream APIs include Stream and PairStream (key-value pair streams)
 - Support a wide range of operations, including
 - Basic transformations, which produce an output stream transforming the input one (e.g., filter, map, flatMap)
 - Windowing
 - Aggregations (e.g., reduce, aggregate, reduceByKey, countByKey)
 - Joins
 - Output, which produce a result (e.g., print, forEach)
 - State management, to save and update state and also to query it

Storm: Stream API example

- The usual WordCount using Stream API

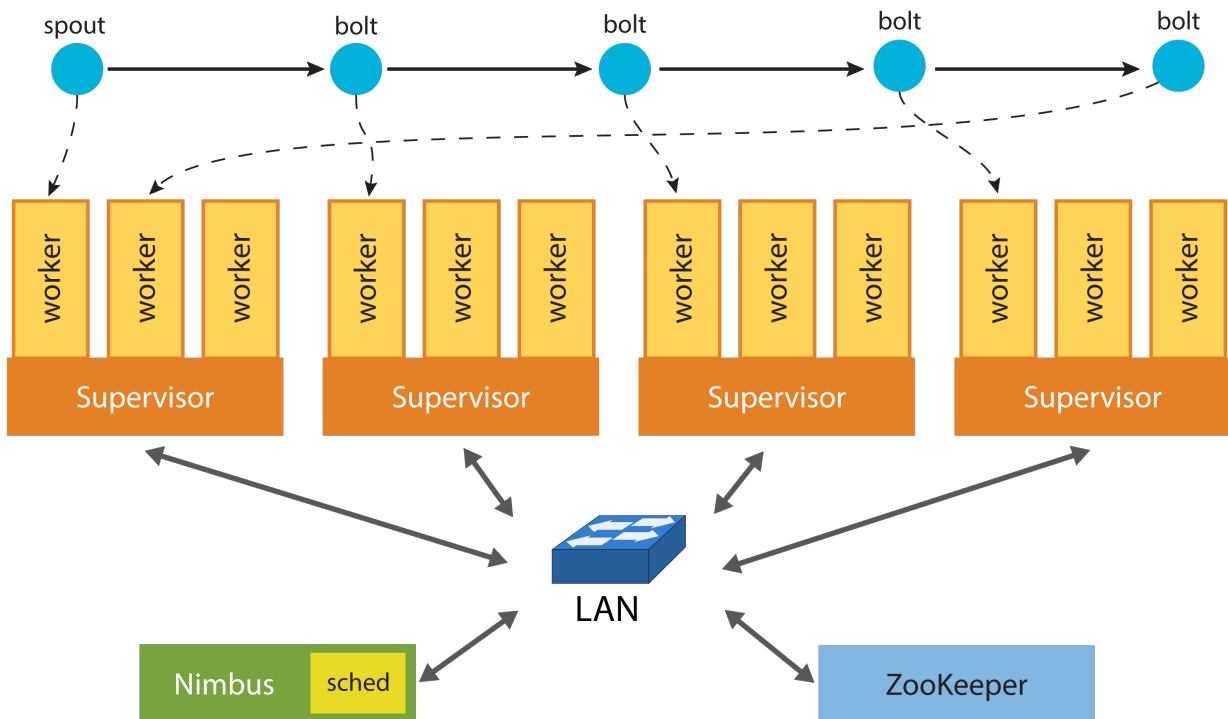
```
StreamBuilder builder = new StreamBuilder();

builder
    // A stream of random sentences with two partitions
    .newStream(new RandomSentenceSpout(), new ValueMapper<String>(), 2)
    // a two seconds tumbling window
    .window(TumblingWindows.of(Duration.seconds(2)))
    // split the sentences to words
    .flatMap(s -> Arrays.asList(s.split(" ")))
    // create a stream of (word, 1) pairs
    .mapToPair(w -> Pair.of(w, 1))
    // compute the word counts in the last two second window
    .countByKey()
    // print the results to stdout
    .print();
```

<https://github.com/apache/storm/blob/master/examples/storm-starter/src/jvm/org/apache/storm/starter/streams/WindowedWordCount.java>

Storm: architecture

- Master-worker architecture



V. Cardellini - SABD 2024/25

14

Storm components: master and Zookeeper

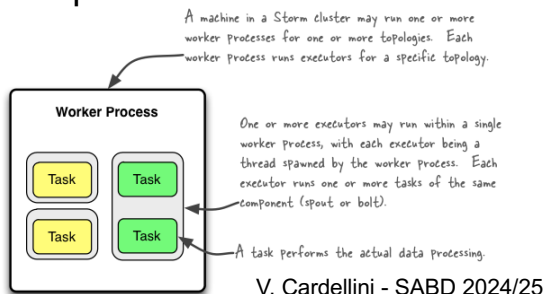
- Nimbus
 - Master node
 - Users submit topologies to it
 - Responsible for distributing and coordinating the topology execution
- Zookeeper
 - Nimbus uses a combination of local disk(s) and Zookeeper to store info about application topology

V. Cardellini - SABD 2024/25

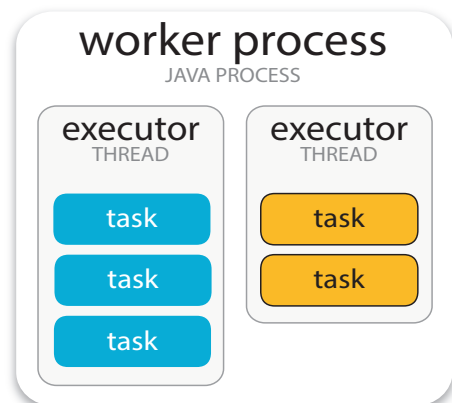
15

Storm components: worker node

- **Worker node**: computing resource, contains one or more worker processes
- **Worker process**: Java process running one or more executors, belongs to a specific topology
- **Executor**: thread spawned by a worker process, the smallest schedulable entity
 - May run one or more tasks for the same operator
- **Task**: operator instance
 - Does the actual work for the operator



V. Cardellini - SABD 2024/25



16

Storm components: supervisor

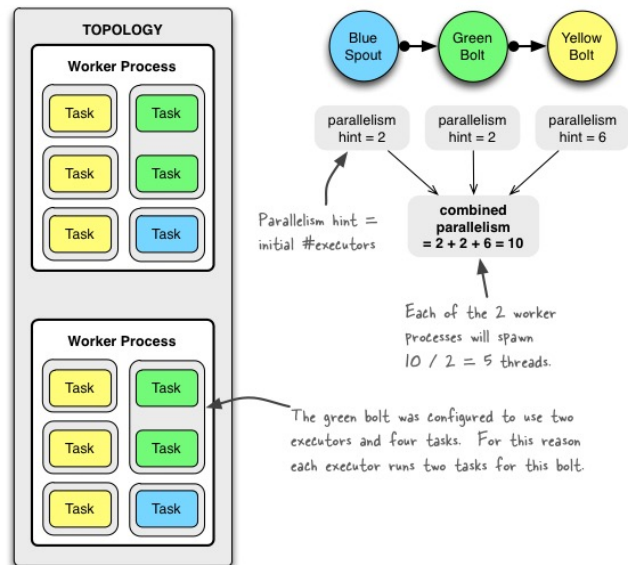
- Each worker node runs a **supervisor**
- Each supervisor:
 - Receives assignments from Nimbus (through ZooKeeper) and spawns workers based on the assignment
 - Sends to Nimbus (through ZooKeeper) a periodic heartbeat
 - Advertises the topologies that the worker node is currently running, and any vacancies that are available to run more topologies

Storm: running a topology

- User can configure topology parallelism
 - Number of worker processes: `setNumWorkers` method
 - Number of executors (threads): `parallelism_hint` parameter in `setSpout` or `setBolt`
 - Number of tasks : `setNumTasks` method

- Parallelism of running topology can be *manually* changed using `rebalance` command

<https://storm.apache.org/releases/current/Understanding-the-parallelism-of-a-Storm-topology.html>



V. Cardellini - SABD 2024/25

18

Storm: reliable message processing

- What happens if a bolt fails to process a tuple?
- Storm provides a mechanism by which the originating spout can replay the failed tuple
 - Storm needs to maintain the link between every spout tuple and its tree of tuples so to detect when the tree of tuples has been successfully processed: *anchoring*
 - And needs to *ack* (or fail) the spout tuple appropriately
 - If ack is not received within a specified timeout time period, the tuple processing is considered as failed and the tuple is replayed
- Storm provides *at-least-once* semantics
 - Best effort semantics if acking is disabled

<https://storm.apache.org/releases/current/Guaranteeing-message-processing.html>

V. Cardellini - SABD 2024/25

19

Storm: application monitoring

- Storm has a built-in monitoring and metrics system
 - Built-in and user-defined metrics
- Built-in metrics include:
 - **Capacity**
 - # of messages executed * average execute latency / time window
 - **Latency**
 - For spouts: **completeLatency** (total latency for processing the message)
 - Ignore value if acking is disabled
 - For bolts: **executeLatency** (avg time the bolt spends to run the execute method) and **processLatency** (avg time from starting execute to ack)
 - **JVM memory usage and garbage collection**
- Metrics can be queried via Storm's UI REST API or reported to a registered consumer (e.g., Graphite)

```
"bolts": [  
  {  
    "executors": 12,  
    "emitted": 184580,  
    "transferred": 0,  
    "acked": 184640,  
    "executeLatency": "0.048",  
    "tasks": 12,  
    "executed": 184620,  
    "processLatency": "0.043",  
    "boltId": "count",  
    "lastError": "",  
    "errorLapsedSecs": null,  
    "capacity": "0.003",  
    "failed": 0  
  },  
  ...  
]
```

<https://storm.apache.org/releases/current/STORM-UI-REST-API.html>

V. Cardellini - SABD 2024/25

20

Layers on top of Storm

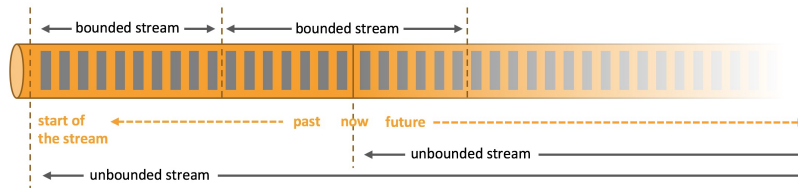
- Trident
 - High-level abstraction on top of Storm to provide **exactly-once processing**

<https://storm.apache.org/releases/current/Trident-API-Overview.html>
- SQL
 - To run SQL queries over streaming data

<https://storm.apache.org/releases/current/storm-sql.html>

Unbounded vs. bounded streams

- Data can be processed as bounded or unbounded streams



Unbounded streams

- Have a start but **no defined end**
- Provide data as it is generated
- Must be **continuously processed**
- Not possible to wait for all input data to arrive
- Processing them often requires that events are ingested in a **specific order**

Bounded streams

- Have a **defined start and end**
- Can be processed by **ingesting all data before performing any computations**
- Ordered ingestion is **not required** because bounded data can always be sorted

Batch processing vs. stream processing

- Batched/stateless: scheduled in batches
 - Short-lived tasks (Hadoop, Spark)
 - Distributed streaming over batches (Spark Streaming)
- Dataflow/stateful: continuously processed, typically scheduled once (Storm, Flink)
 - Long-lived task execution
 - State is kept inside tasks