

DS256:Jan17 (3:1)

Tutorial: Apache Storm

Anshu Shukla

16 Feb, 2017

©Yogesh Simmhan & Partha Talukdar, 2016

This work is licensed under a [Creative Commons Attribution 4.0 International License](https://creativecommons.org/licenses/by/4.0/)

Copyright for external content used with attribution is retained by their original authors



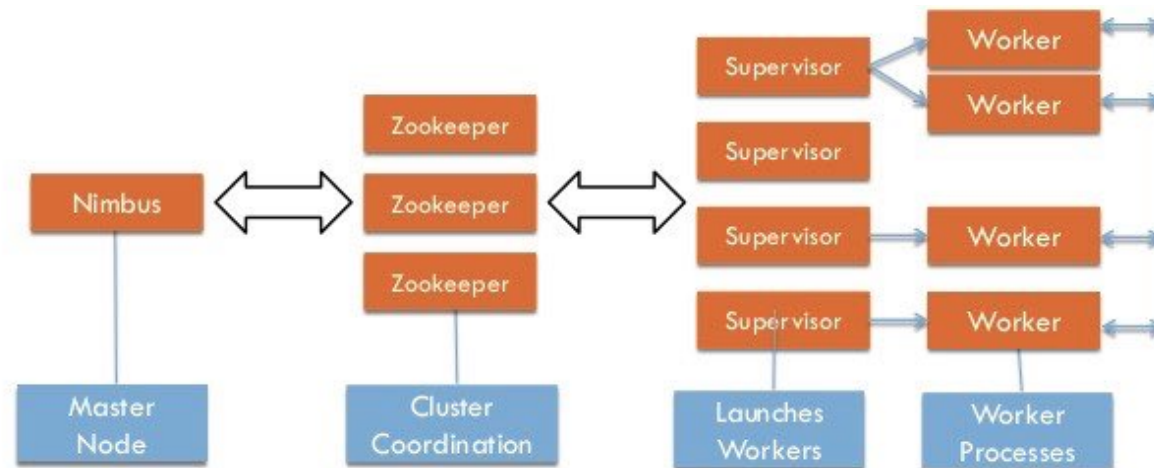


Apache Storm

- Open source distributed realtime computation system
- Can process million tuples processed per second per node.
- Scalable, fault-tolerant, guarantees your data will be processed
- Does for realtime processing what Hadoop did for batch processing.
- Key difference is that a MapReduce job eventually finishes, whereas a topology processes messages forever (or until you kill it).

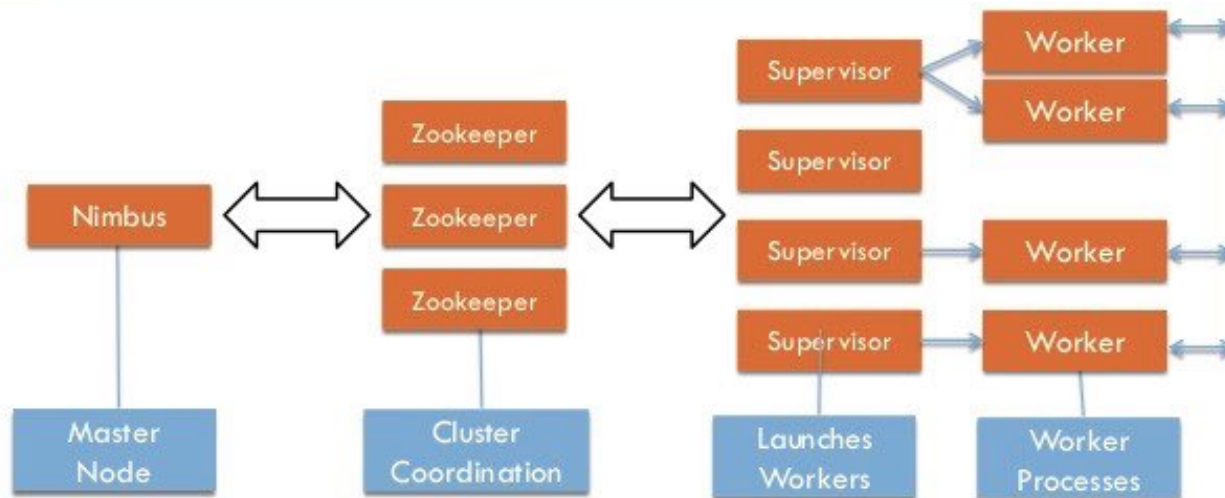
Storm Architecture:

- Two kinds of nodes on a Storm cluster: Master node and the worker nodes
 - Master node**
 - › runs a daemon called "Nimbus"
 - › distributing code around the cluster, assigning tasks to machines, and monitoring for failures.
 - Worker node**
 - › runs a daemon called the "Supervisor"
 - › listens for work assigned by nimbus to its machine and starts and stops worker processes
 - › Worker process executes a subset of a topology, a running topology consists of many worker processes spread across many machines.



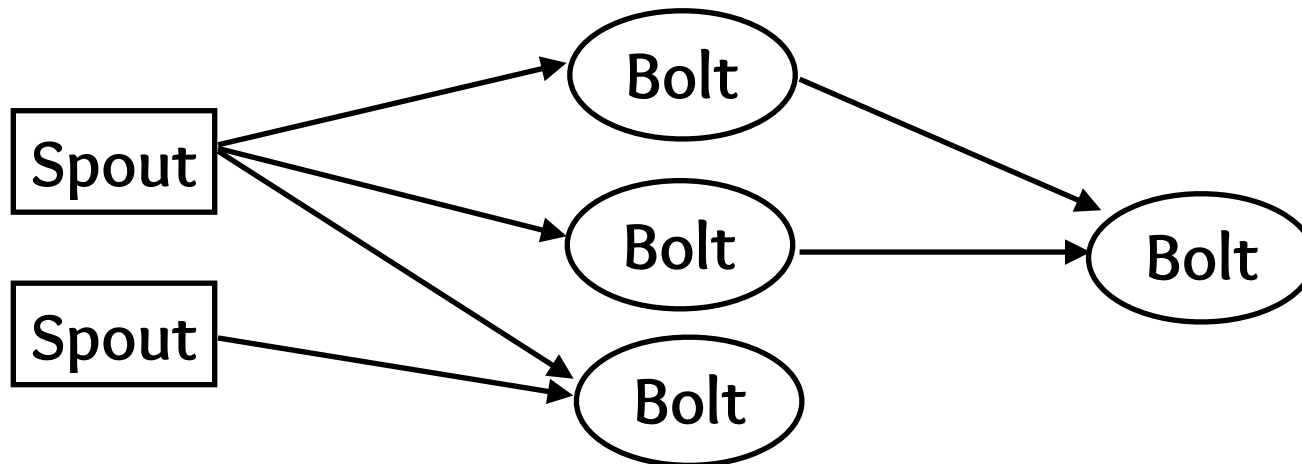
Storm Architecture:

- Zookeeper
 - Coordination between Nimbus and the Supervisors is done through a Zookeeper cluster
 - Nimbus daemon and Supervisor daemons are fail-fast and stateless, state is kept in Zookeeper
 - »can kill Nimbus or the Supervisors and they'll start back up like nothing happened.



Key abstractions

- Tuples: an ordered list of elements.
- Streams: an unbounded sequence of tuples.
- Spouts: sources of streams in a computation (e.g. a Twitter API)
- Bolts:
 - process input streams and produce output streams.
 - run functions (filter, aggregate, or join data or talk to databases).
- Topologies: Computation DAG, each node contains processing logic, and links between nodes indicate how data streams



Topology Example

```
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");
```

- Contains a spout and two bolts, Spout emits words, and each bolt appends the string "!!!" to its input
- Nodes are arranged in a line
 - e.g. spout emits the tuples ["bob"] and ["john"], then the second bolt will emit the words ["bob!!!!!!"] and ["john!!!!!!"]
- Last parameter, **parallelism**: how many threads should run for that component across the cluster
- **"shuffle grouping"** means that tuples should be randomly distributed to downstream tasks.

Spout and Bolt

- Processing logic implements the *IRichSpout* & *IRichBolt* interface for spouts & bolts.
- *open/prepare method* provides the bolt with an `OutputCollector` that is used for emitting tuples from this bolt, **executed once**.
- **Execute** method receives a tuple from one of the bolt's inputs, **executes for every tuple**.
- Cleanup method is called when a Bolt is being shutdown, **executed once**.

```
public static class ExclamationBolt implements IRichBolt {
    OutputCollector _collector;

    @Override
    public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
        _collector = collector;
    }

    public void nextTuple() {
        Utils.sleep(100);
        final String[] words = new String[] {"nathan", "mike", "jackson", "golda", "bertels"};
        final Random rand = new Random();
        final String word = words[rand.nextInt(words.length)];
        _collector.emit(new Values(word));
    }

    public void cleanup() {
    }

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

    @Override
    public Map<String, Object> getComponentConfiguration() {
        return null;
    }
}
```

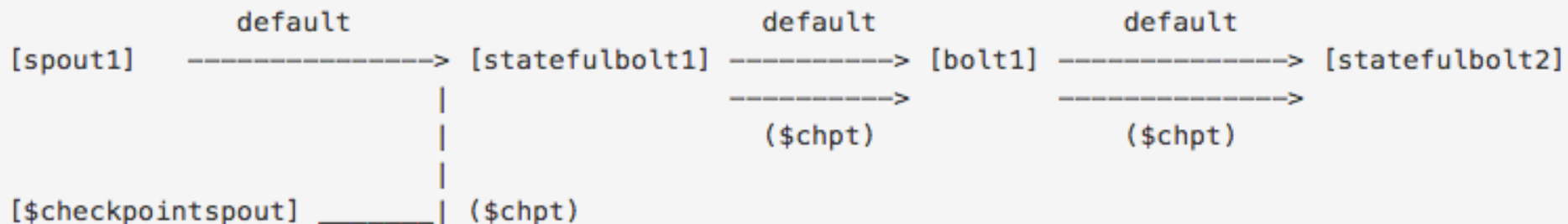
Stateful bolts (from v1.0.1)

- Abstractions for bolts to save and retrieve the state of its operations.
- By extending the **BaseStatefulBolt** and implement **initState(T state)** method.
- **initState method** is invoked by the framework during the bolt initialization (after `prepare()`) with the previously saved state of the bolt.

```
public class WordCountBolt extends BaseStatefulBolt<KeyValueState<String, Long>> {  
    private KeyValueState<String, Long> wordCounts;  
    private OutputCollector collector;  
    ...  
    @Override  
    public void prepare(Map stormConf, TopologyContext context, OutputCollector collector) {  
        this.collector = collector;  
    }  
    @Override  
    public void initState(KeyValueState<String, Long> state) {  
        wordCounts = state;  
    }  
    @Override  
    public void execute(Tuple tuple) {  
        String word = tuple.getString(0);  
        Integer count = wordCounts.get(word, 0);  
        count++;  
        wordCounts.put(word, count);  
        collector.emit(tuple, new Values(word, count));  
        collector.ack(tuple);  
    }  
    ...  
}
```


Stateful bolts (from v1.0.1)

- The framework periodically checkpoints the state of the bolt (default every second).
- Checkpoint is triggered by an internal checkpoint spout.
- If there is at-least one **IStatefulBolt** in the topology, the checkpoint spout is automatically added by the topology builder.
- Checkpoint tuples flow through a separate internal stream namely **\$checkpoint**
- Non stateful bolts just forwards the checkpoint tuples so that the checkpoint tuples can flow through the topology DAG.





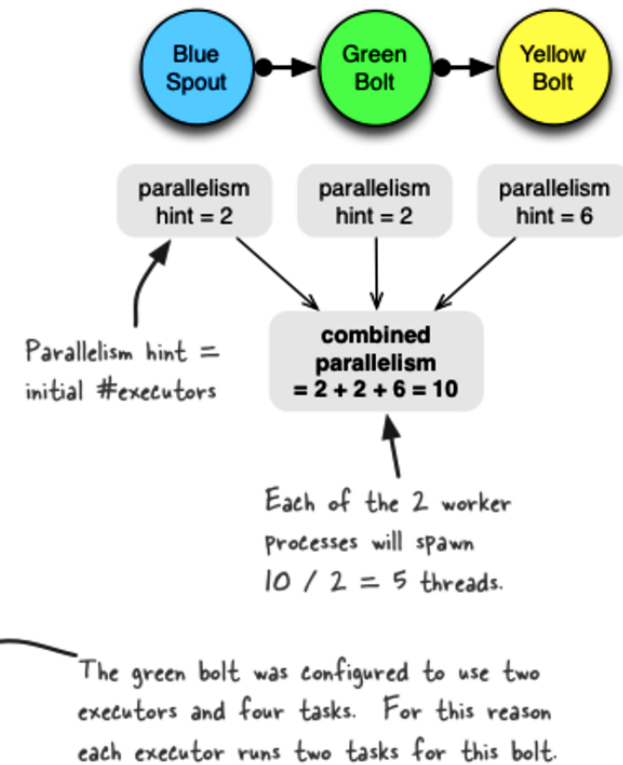
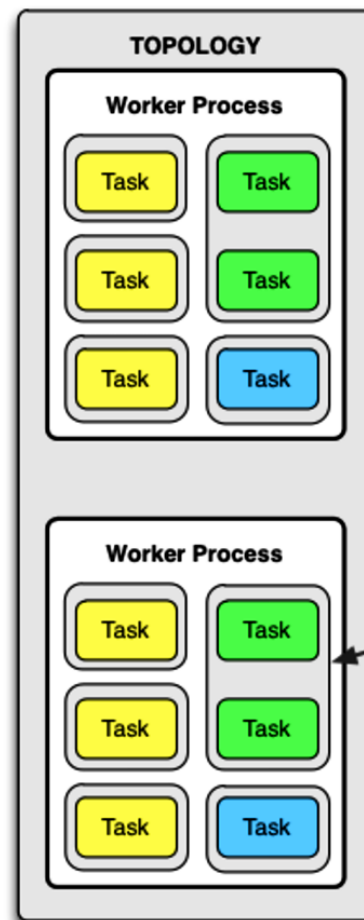
Example of a running topology

- Topology consists of **three components**: one BlueSpout and two bolts, GreenBolt and YellowBolt
- #worker processes=2
- for green Bolt:
 - #executors = 2
 - #tasks = 4

Configuring the parallelism of a simple Storm topology

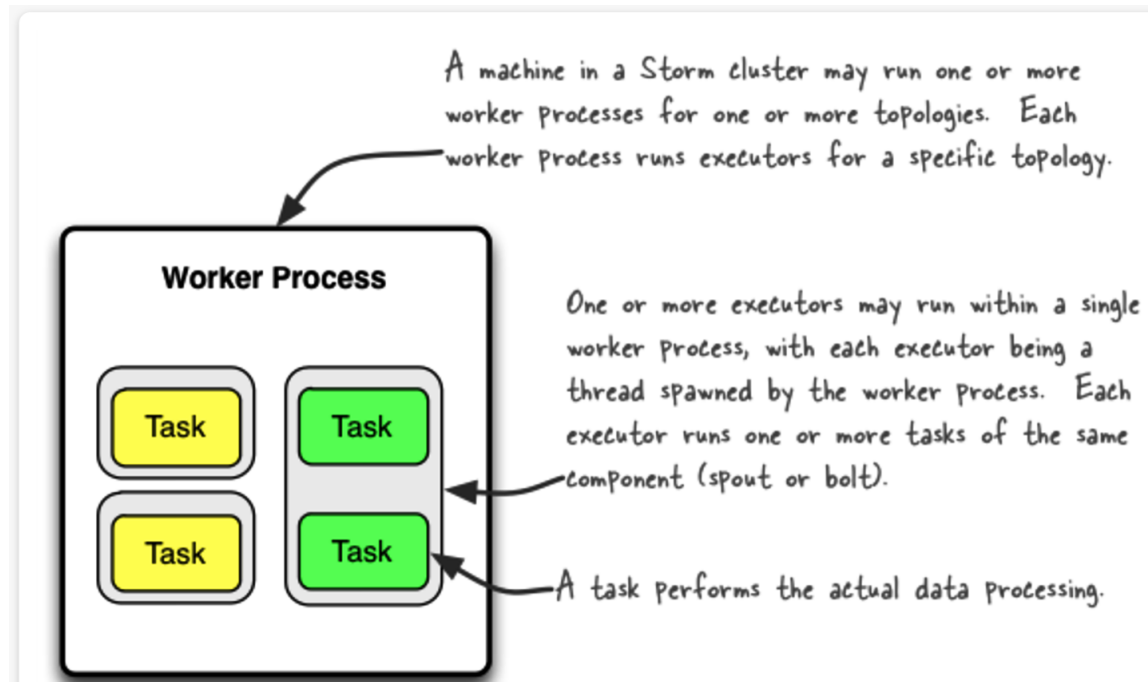
```

1 Config conf = new Config();
2 conf.setNumWorkers(2); // use two worker processes
3
4 topologyBuilder.setSpout("blue-spout", new BlueSpout(), 2);
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 );
  
```



Running topology: worker processes, executors and tasks

- Worker processes executes a subset of a topology, and runs in its own JVM.
- An executor is a thread that is spawned by a worker process and runs within the worker's JVM (*parallelism hint*).
- A task performs the actual data processing and is run within its parent executor's thread of execution.
- # threads can change at run time, *but not # tasks*
- #threads \leq #tasks



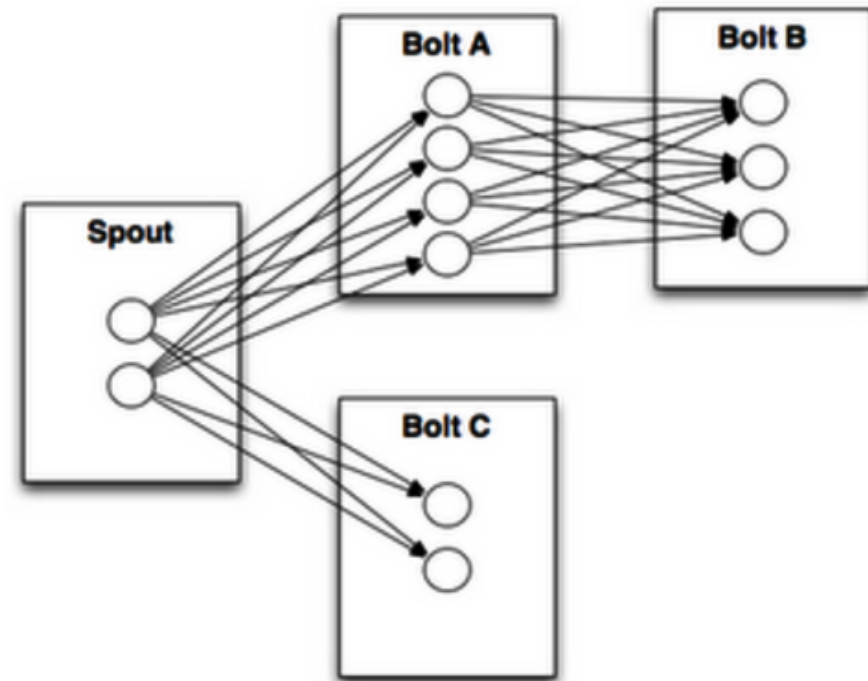


Updating the parallelism of a running topology

- Rebalancing: Increase or decrease the number of worker processes and/or executors without being required to restart the cluster or the topology, but not tasks.
- e.g. To reconfigure the topology "mytopology" to use 5 worker processes, # the spout "blue-spout" to use 3 executors.
 - **storm rebalance mytopology -n 5 -e blue-spout=3**
- Demo:

Stream groupings

- Stream grouping defines how that stream should be partitioned among the bolt's tasks.
 - **Shuffle grouping:** random distribution, each bolt is guaranteed to get an equal number of tuples
 - **Fields grouping:** stream is partitioned by the fields specified in the grouping
 - **Global grouping:** entire stream goes to a single one of the bolt's tasks.
 - **All grouping:** The stream is replicated across all the bolt's tasks.
 - etc ..





Guaranteeing Message Processing

- Storm can guarantee *at least once processing*.
- Tuple coming off the spout triggers many tuples being created based on it forming *Tuple tree*.
- **"fully processed" tuple**: tuple tree has been exhausted and every message in the tree has been processed (*within a specified timeout*).
 - Spout while emitting provides a "message id" that will be used to identify the tuple later.
 - Storm takes care of **tracking the tree of messages** that is created.
 - **if fully processed**, Storm will call the **ack method** on the originating Spout task with its message id.
 - **if tuple times-out** Storm will call the **fail method** on the Spout.

Guaranteeing Message Processing...

- Things user have to do to achieve at-least once semantics.
 - **Anchoring**: creating a new link in the tree of tuples.
 - **Acking**: finished processing an individual tuple.
 - **Failing**: to immediately fail tuple at the root of the tuple tree, to replay faster than waiting for the tuple to time-out.

```
public class SplitSentence extends BaseRichBolt {
    OutputCollector _collector;

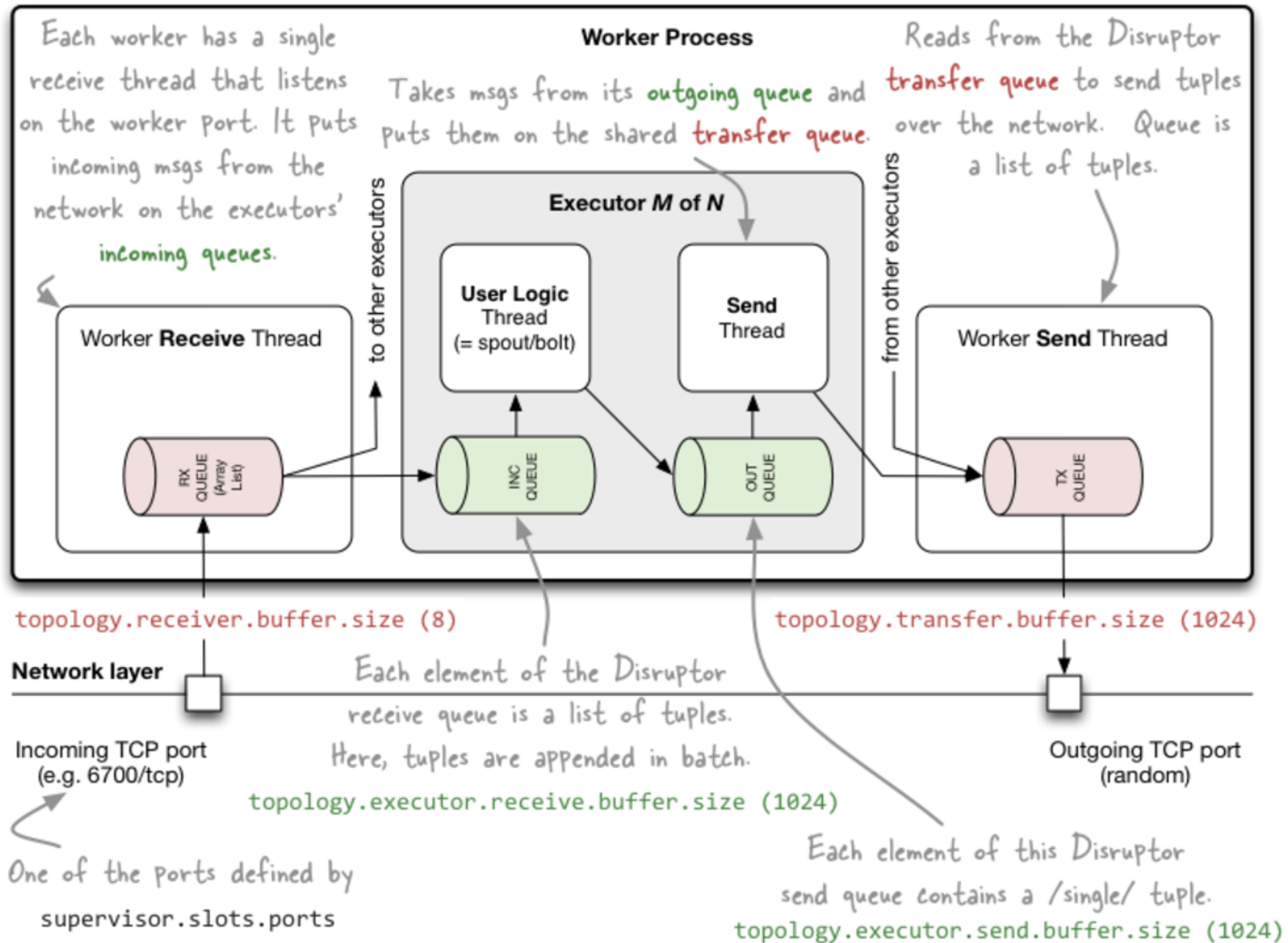
    public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
        _collector = collector;
    }

    public void execute(Tuple tuple) {
        String sentence = tuple.getString(0);
        for(String word: sentence.split(" ")) {
            _collector.emit(tuple, new Values(word));
        }
        _collector.ack(tuple);
    }

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

The diagram highlights two key operations in the code: **Anchoring**, which occurs at the line `_collector.emit(tuple, new Values(word));`, and **Acking**, which occurs at the line `_collector.ack(tuple);`.

Internal messaging within Storm worker processes





Resource Scheduling for DSPS

- Scheduling for the DSPS has two parts:
 - *Resource allocation* -
 - Determining the appropriate **degrees of parallelism** per task (i.e., threads of execution)
 - Amount of **computing resources** per task (e.g., Virtual Machines (VMs)) for the given dataflow
 - *Resource mapping* -
 - Deciding the specific **assignment of the threads to the VMs** ensuring that the expected performance behavior and resource utilization is met.



Resource Allocation

- For a given DAG and input rate, allocation determines the number of resource slots(ρ) for DAG & number of threads(q), resources required for each task.
- Resource allocation algorithms:
 - *Linear Storm Allocation (LSA)*
 - *Model Based Allocation (MBA)* [3]
- Requires input rate to each task for finding the resource needs and data parallelism for that task.
- # of slots:

$$\rho = \max \left(\left[\sum_{t_i \in \mathbb{T}} (c_i) \right], \left[\sum_{t_i \in \mathbb{T}} (m_i) \right] \right)$$

Linear Storm Allocation

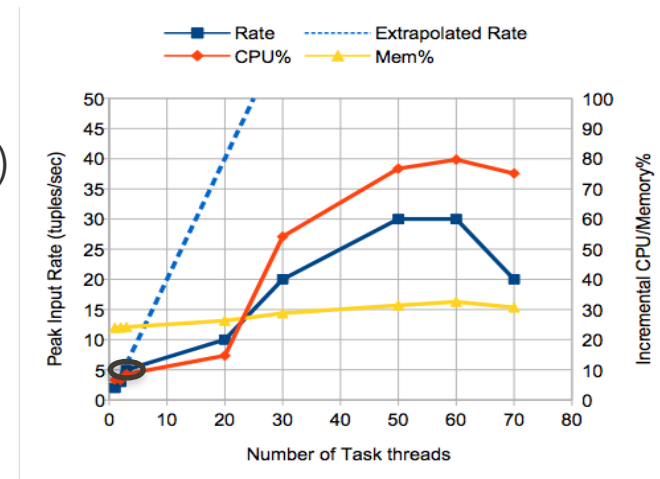
Pseudocode:

```

AllocateLSA(G,Ω){
  for each task in DAG
    while(input rate for task ti > Peak rate with 1 thread)
      {
        add 1 thread
        decrease required rate by Peak rate with 1 thread
        increase resources allocated with that required for 1 thread
      }
    if(remaining input rate for task ti>0)
      {
        increase number of threads by 1
        set input rate to zero
        add resources by scaling remaining rate with peak rate
      }
  return <#threads,CPU%,Memory%> for each task

```

- e.g. For 105 tuples/sec rate.
 - Threads=(52 thread * 2 tuples/sec)+(1 thread*1 tuple/sec)
 - CPU% =52*6.73+3.3=353%
 - Memory%=52*23.92+11.16=1255.8%
 - Required #Slots=ceil (353%,1255.8%)=13



Default Mapping

- Not “**resource aware**”, so does not use the output of the performance model
- Threads are picked in any arbitrary order for mapping
- For each thread, next slot is picked in **round-robin** fashion
- Unbalanced load distribution across the slots

Pseudocode:

```
MapDSM(R, S){
```

```
  M ← new map()
```

```
  get the list of slots
```

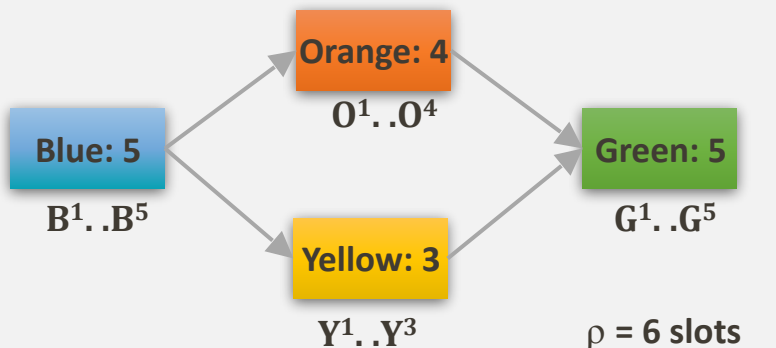
```
  for each thread
```

```
    pick slots in round robin manner
```

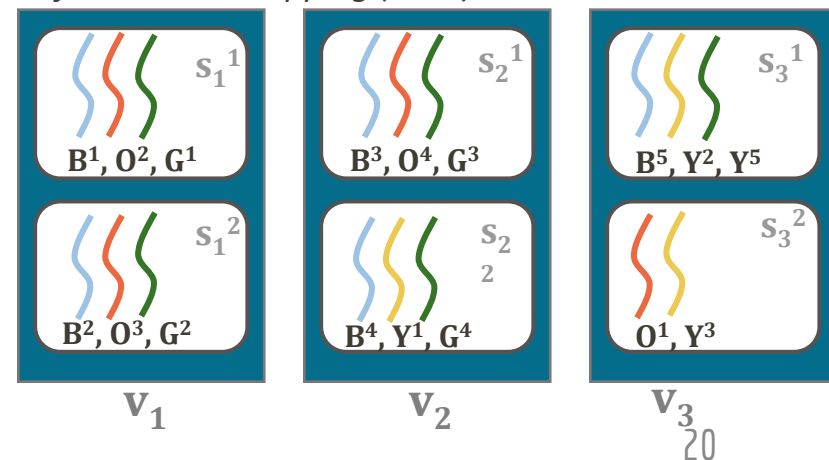
```
    store mapping of thread to slot in M
```

```
  return M
```

DAG with Thread Allocation for Tasks using Model



Default Storm Mapping (DSM)

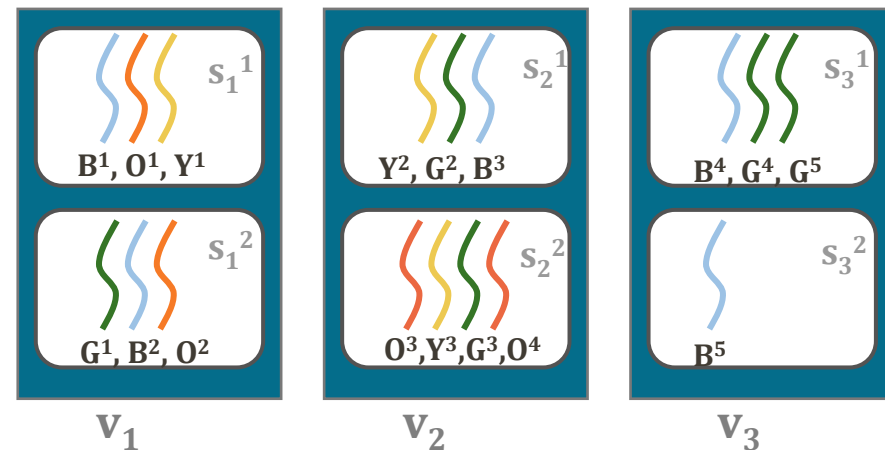
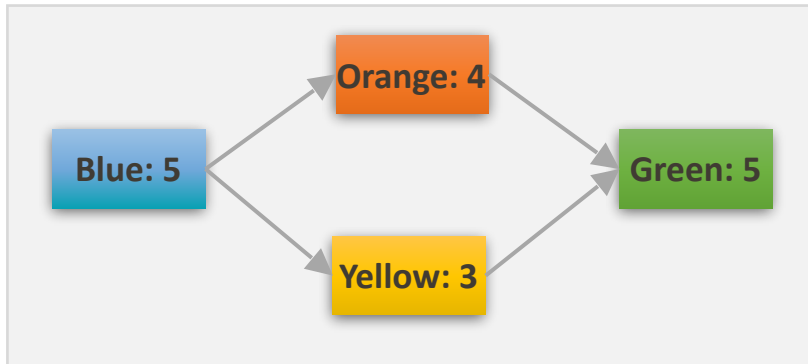


Resource Aware Mapping^[4]

- Use only **resource usage for single thread** from the performance model
- “**Network aware**”, places the threads on slots such that communication latency between adjacent tasks is reduced
- Threads are picked in order of BFS traversal of the DAG for locality.
- Slots are chosen by **Distance function** (minimum value) based on the available and required resources, and a network latency measure

$$d = w_M \times (M_j - \bar{m}_i)^2 + w_C \times (C_j - \bar{c}_i)^2 + w_N \times \text{NWDIST}(\widehat{v}, v_j)$$

DAG with Thread Allocation for Tasks using Model





References

- Apache Storm concepts

<http://storm.apache.org/releases/current/Concepts.html>

- Understanding the Parallelism of a Storm Topology

<http://www.michael-noll.com/blog/2012/10/16/understanding-the-parallelism-of-a-storm-topology/>

- Model-driven Scheduling for Distributed Stream Processing Systems, Shukla et. al. {under review}

- R-storm: Resource-aware scheduling in storm, Peng et.al.,in: Middleware 2016