



# Big Data Analytics: Approaches and Tools

Московский городской педагогический университет

## Лекция 7. Поточковая обработка

➤ Существующие платформы

➤ Storm



# Существующие платформы

# Существующие системы

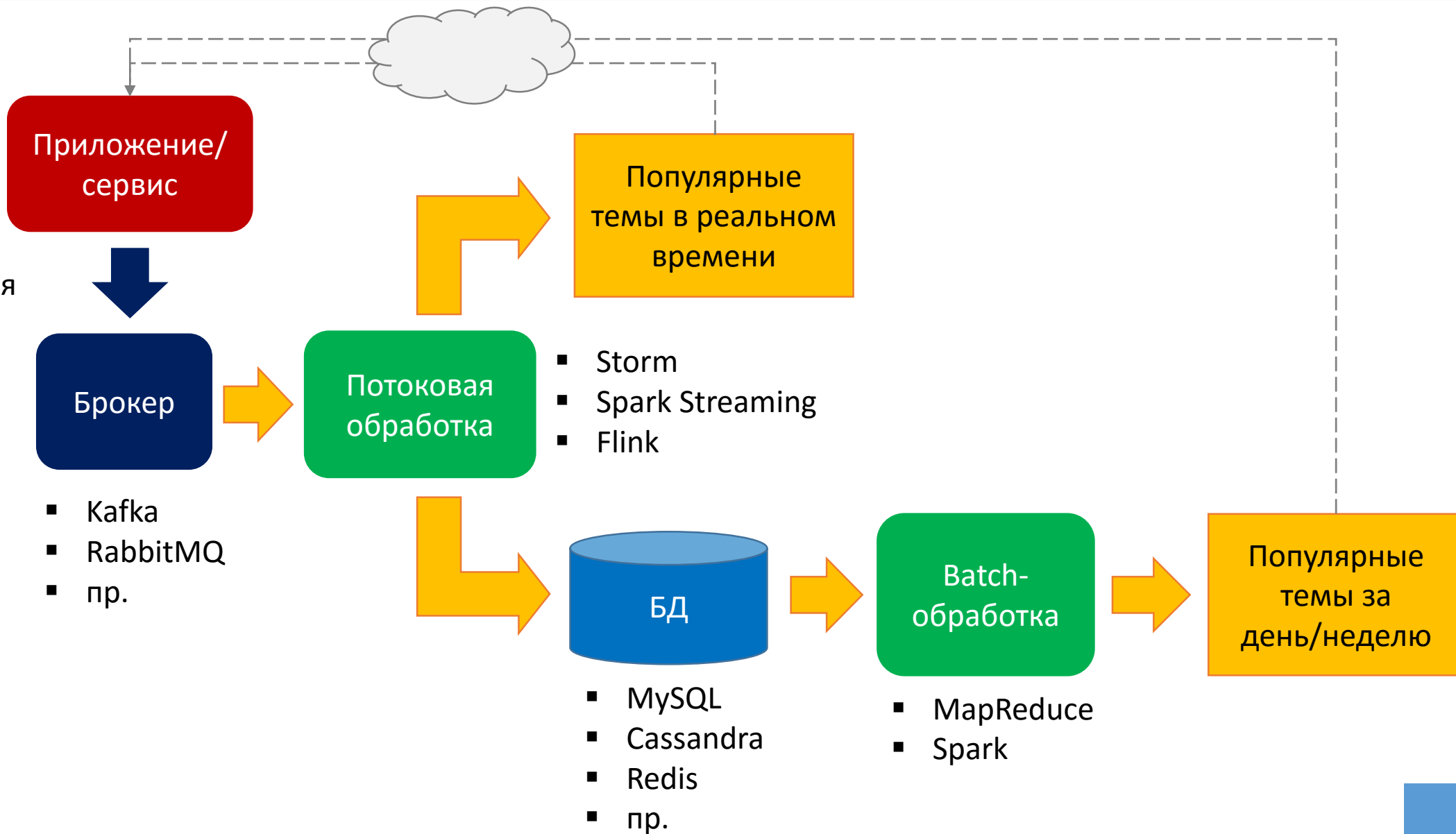


Streaming Model
API
Guarantees
Fault Tolerance
State Management
Latency
Throughput
Maturity

Native	Micro-batching	Micro-batching	Native	Native
Compositional		Declarative	Compositional	Declarative
At-least-once	Exactly-once	Exactly-once	At-least-once	Exactly-once
Record ACKs		RDD based Checkpointing	Log-based	Checkpointing
Not build-in	Dedicated Operators	Dedicated DStream	Stateful Operators	Stateful Operators
Very Low	Medium	Medium	Low	Low
Low	Medium	High	High	High
High		High	Medium	Low

# Пример архитектуры

- сообщения
- ТВИТЫ
- посты
- GPS
- действия пользователя
- пр








# Apache Storm

# Особенности Apache Storm

- Apache Storm – распределенная вычислительная система для обработки данных, поступающих в реальном времени
- Обеспечивает надежную обработки потоков данных
- Можно использовать различные языки программирования – Java, Ruby, Python, Javascript, Perl
- Масштабируется горизонтально
- Используется для ETL, в задачах по анализу данных
- Более 1000000 обрабатываемых записей в секунду на одном узле
- Множество решений для интеграции с существующими системами обработки и хранения данных

<https://storm.apache.org/>

# Особенности Apache Storm

		 TRIDENT			
Streaming Model	Native	Micro-batching	Micro-batching	Native	Native
API	Compositional		Declarative	Compositional	Declarative
Guarantees	At-least-once	Exactly-once	Exactly-once	At-least-once	Exactly-once
Fault Tolerance	Record ACKs		RDD based Checkpointing	Log-based	Checkpointing
State Management	Not build-in	Dedicated Operators	Dedicated DStream	Stateful Operators	Stateful Operators
Latency	Very Low	Medium	Medium	Low	Low
Throughput	Low	Medium	High	High	High
Maturity	High		High	Medium	Low



# Словарь Storm

➤ Topology

➤ Stream

➤ Spout

➤ Bolt

➤ Nimbus

➤ Supervisor

➤ Worker process

➤ Executor

➤ Task

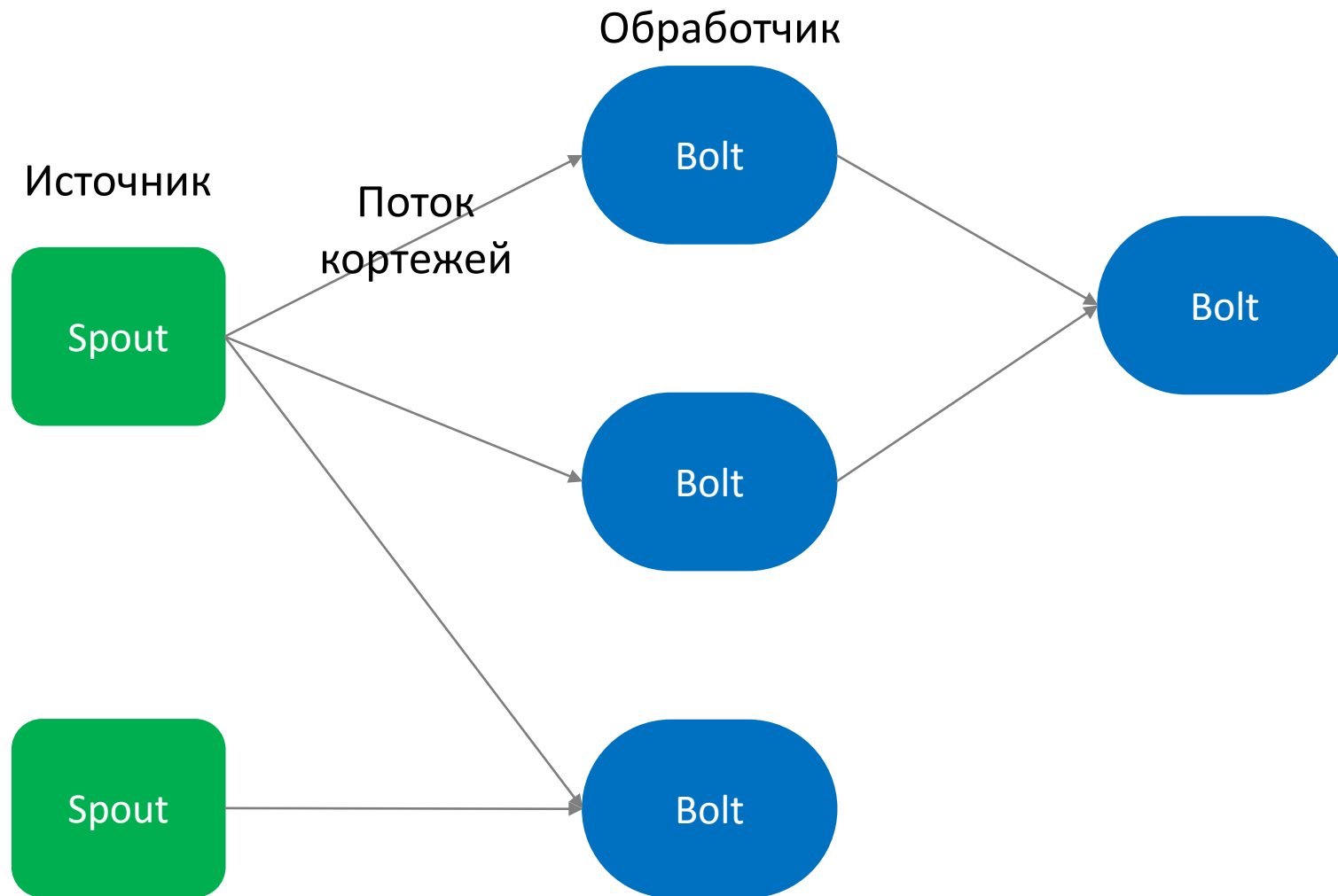
➤ Zookeeper

# Топология

# Топология. Основные компоненты

- Topology – DAG выполнения
- Stream – поток кортежей
- Spout – источник кортежей
- Bolt – обработка
- Task – экземпляр **spout** или **bolt**

# Топология. DAG выполнения



**Spout** - источник потоков в топологии Storm

- Как правило, spout читает данные из брокеров, таких как Kestrel, RabbitMQ, Kafka
- Может генерировать собственный поток
- Запрашивать данные из других внешних источников, пример, Twitter API

**Методы:**

## **nextTuple**

Опрашивает источник данных на предмет новых событий (сообщений). Если доступно новое событие, оно передается обработчику (bolt) в соответствии с топологией

## **ack**

Вызывается, когда кортеж успешно обработан всей топологией

## **fail**

- где-то в топологии произошла ошибка при обработке кортежа
- истекло отведенное на выполнение время (timeout)

**Bolt** – обрабатывает входные кортежи

**Методы:**

Примеры:

- Фильтрация
- Парсинг
- Трансформации
- Агрегация
- Объединение
- Взаимодействие с БД

**prepare**

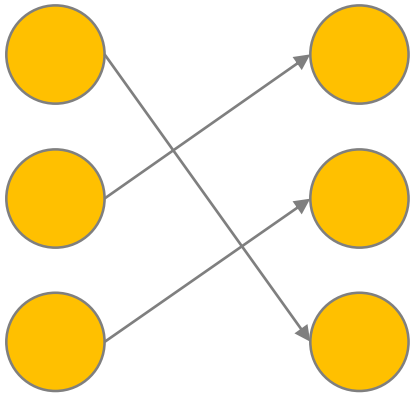
- Вызывается при инициализации экземпляра bolt

**execute**

- Выполнение обработки входных данных. Выполняется для каждого кортежа

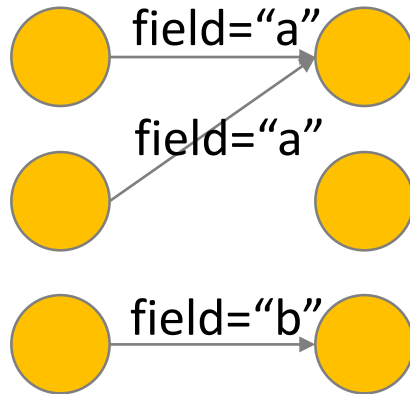
# Группировка потоков

## Shuffle/random



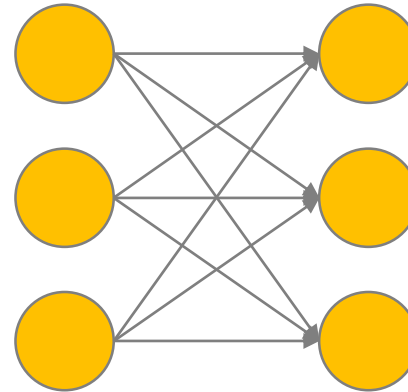
Сбалансированно  
случайным  
образом

## Fields



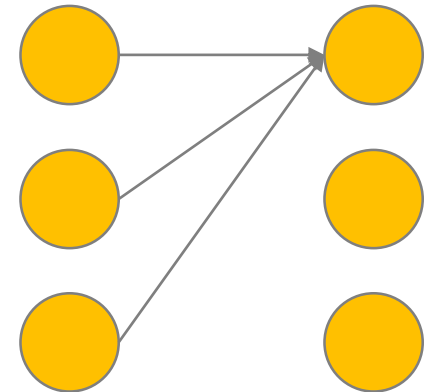
Все записи с  
одинаковым field идут в  
один экземпляр

## All grouping



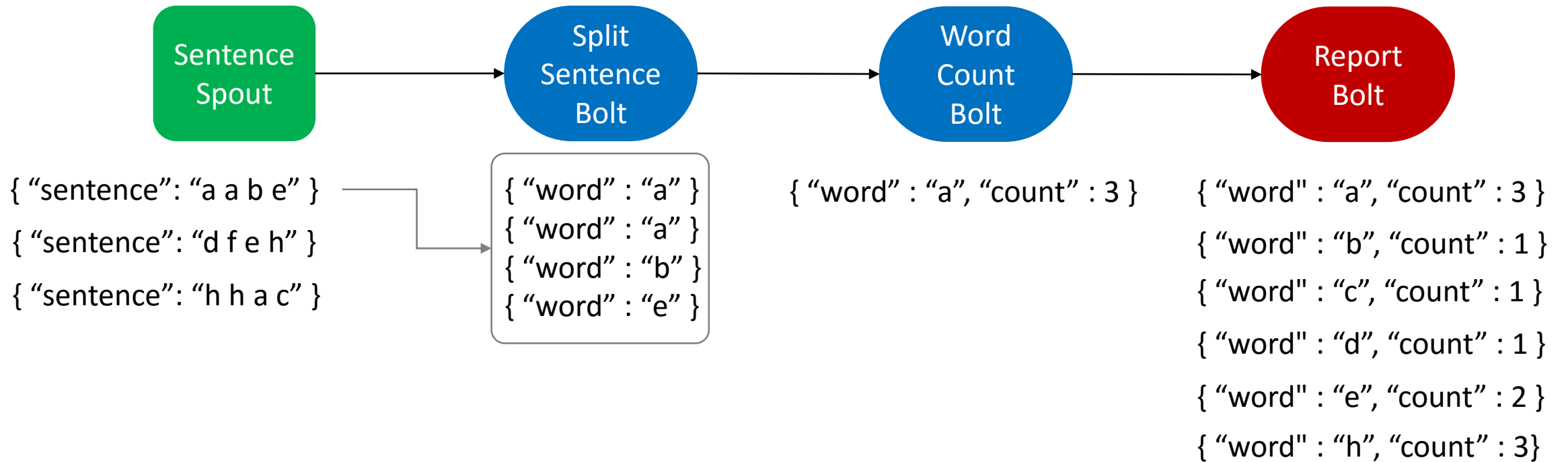
Каждая запись  
поступает на все  
экземпляры

## Global



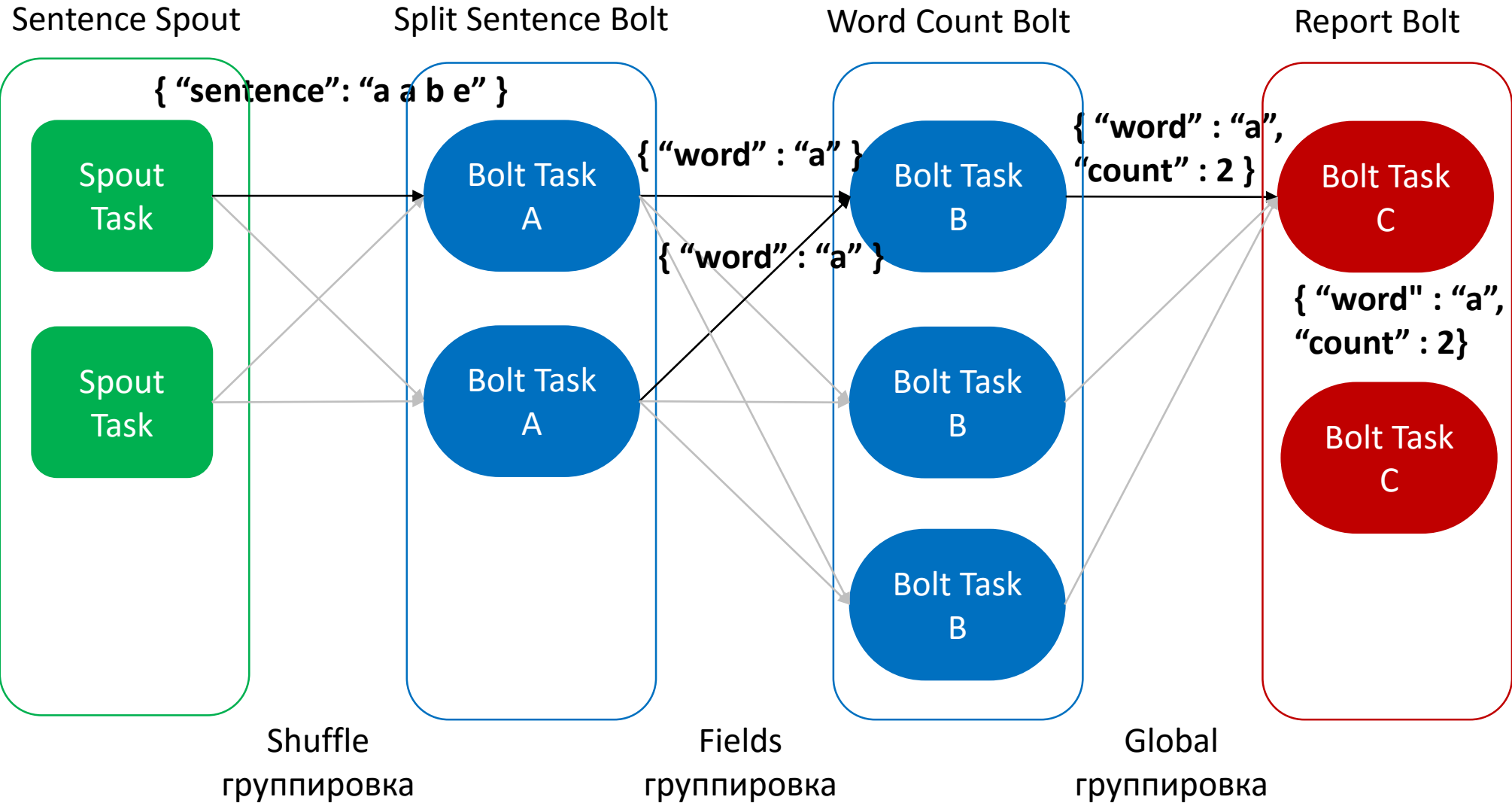
Все записи  
собираются на  
одном экземпляре

# Пример Word Count. DAG





# Пример Word Count. Задачи



# Пример. Spout

```
public class SentenceSpout extends BaseRichSpout {
    private SpoutOutputCollector collector;
    private String[] sentences = {
        "my dog has fleas",
        "i like cold beverages",
        "the dog ate my homework",
        "don't have a cow man",
        "i don't think i like fleas"
    };
    private int index = 0;
    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() {
        this.collector.emit(new Values(sentences[index]));
        index++;
        if (index >= sentences.length) {
            index = 0;
        }
        Utils.waitForMillis(1);
    }
}
```

# Пример. Bolt A

```
public class SplitSentenceBolt extends BaseRichBolt{
    private OutputCollector collector;
    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));
        }
    }
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
}
```

# Пример. Bolt B

```
public class WordCountBolt extends BaseRichBolt {
    private OutputCollector collector;
    private HashMap<String, Long> counts = null;
    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");
        Long count = this.counts.get(word);
        if(count == null){
            count = 0L;
        }
        count++;
        this.counts.put(word, count);
        this.collector.emit(new Values(word, count));
    }
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word", "count"));
    }
}
```

# Пример. Bolt C

```
public class ReportBolt extends BaseRichBolt {  
    private HashMap<String, Long> counts = null;  
    public void prepare(Map config, TopologyContext context,  
        OutputCollector collector) {  
        this.counts = new HashMap<String, Long>();  
    }  
    public void execute(Tuple tuple) {  
        String word = tuple.getStringByField("word");  
        Long count = tuple.getLongByField("count");  
        this.counts.put(word, count);  
    }  
    public void declareOutputFields(OutputFieldsDeclarer declarer) {  
        // this bolt does not emit anything  
    }  
    public void cleanup() {  
        System.out.println("--- FINAL COUNTS ---");  
        List<String> keys = new ArrayList<String>();  
        keys.addAll(this.counts.keySet());  
        Collections.sort(keys);  
        for (String key : keys) {  
            System.out.println(key + " : " + this.counts.get(key));  
        }  
        System.out.println("-----");  
    }  
}
```

# Пример. Driver

```
public class WordCountTopology {  
    public static void main(String[] args) throws Exception {  
  
        SentenceSpout spout = new SentenceSpout();  
        SplitSentenceBolt splitBolt = new SplitSentenceBolt();  
        WordCountBolt countBolt = new WordCountBolt();  
        ReportBolt reportBolt = new ReportBolt();  
  
        TopologyBuilder builder = new TopologyBuilder();  
        builder.setSpout("sentence-spout", spout);  
        builder.setBolt("split-bolt", splitBolt).shuffleGrouping("sentence-spout");  
        builder.setBolt("count-bolt", countBolt).fieldsGrouping("split-bolt", new Fields("word"));  
        builder.setBolt("report-bolt", reportBolt).globalGrouping("count-bolt");  
  
        Config config = new Config();  
        LocalCluster cluster = new LocalCluster(); // StormSubmitter for cluster  
        cluster.submitTopology("word-count-topology", config, builder.createTopology());  
  
        waitForSeconds(10);  
        cluster.killTopology("word-count-topology");  
        cluster.shutdown();  
    }  
}
```

# Гарантия выполнения

# Гарантия выполнения. Spout

- Источник (**spout**) выдает кортежи (**tuple**) обработчикам (**bolt**)
- Каждое сообщение (**tuple**) в режиме надежного выполнения имеет идентификатор
- Выходом обработчика (**bolt**) является дочерний кортеж по отношению к исходному от источника (**spout**)
- Если все обработчики (**bolts**), принимающие участие в обработке исходного кортежа, подтверждают успешность выполнения, то в **spout** вызывается метод **ack** и считается, что входное сообщение успешно обработано.
- Если на каком-то **bolt** произошел сбой или истекло время для обработки (timeout), то вызывается метод **fail**
- В методе **fail** можно задать повторную передачу сообщения



Гарантированная обработка происходит в два этапа:

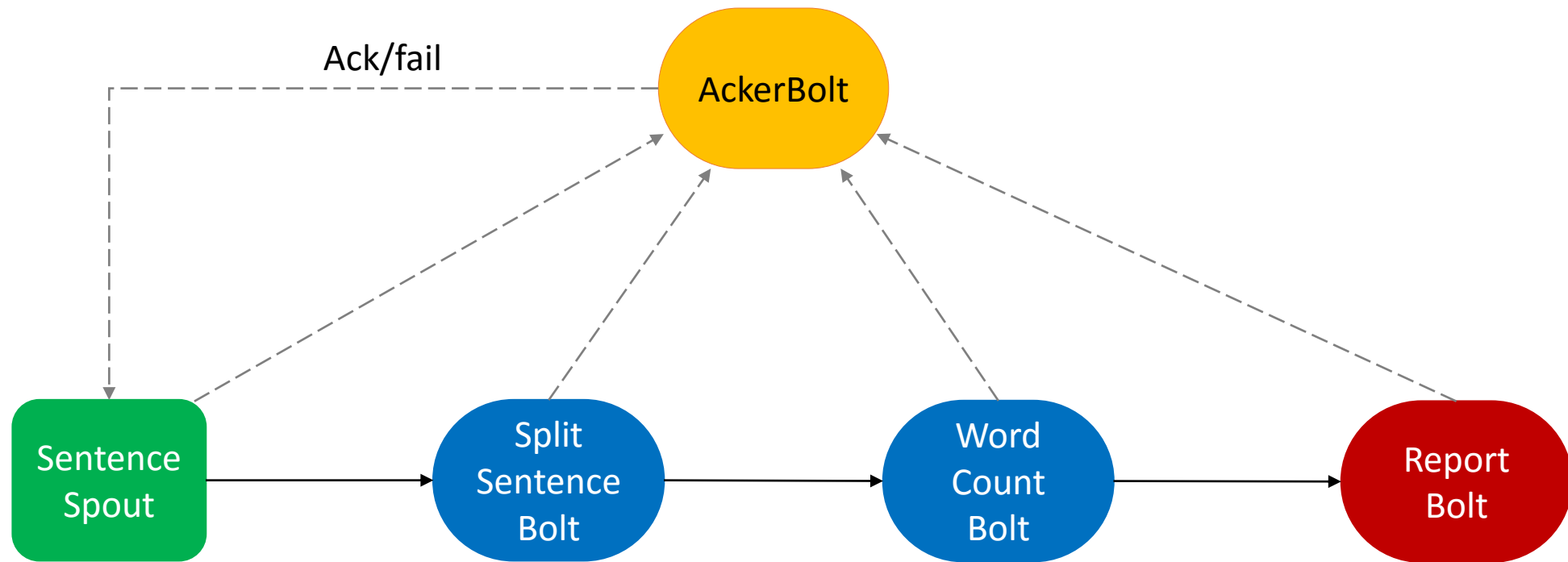
- Фиксация (anchoring) входящего кортежа (**tuple**)

```
collector.emit(tuple, new Values(word));
```

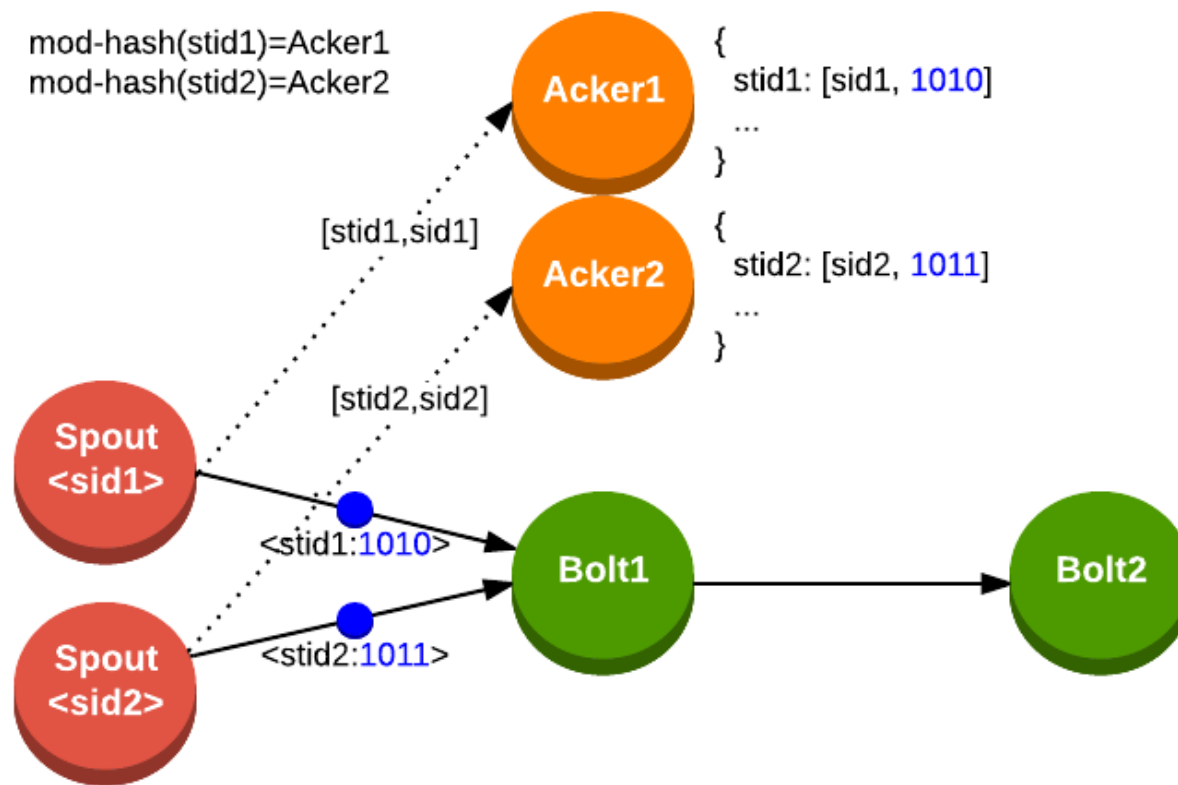
- Подтверждение успешности обработки кортежа или отмечает его как failed

```
this.collector.ack(tuple), this.collector.fail(tuple)
```

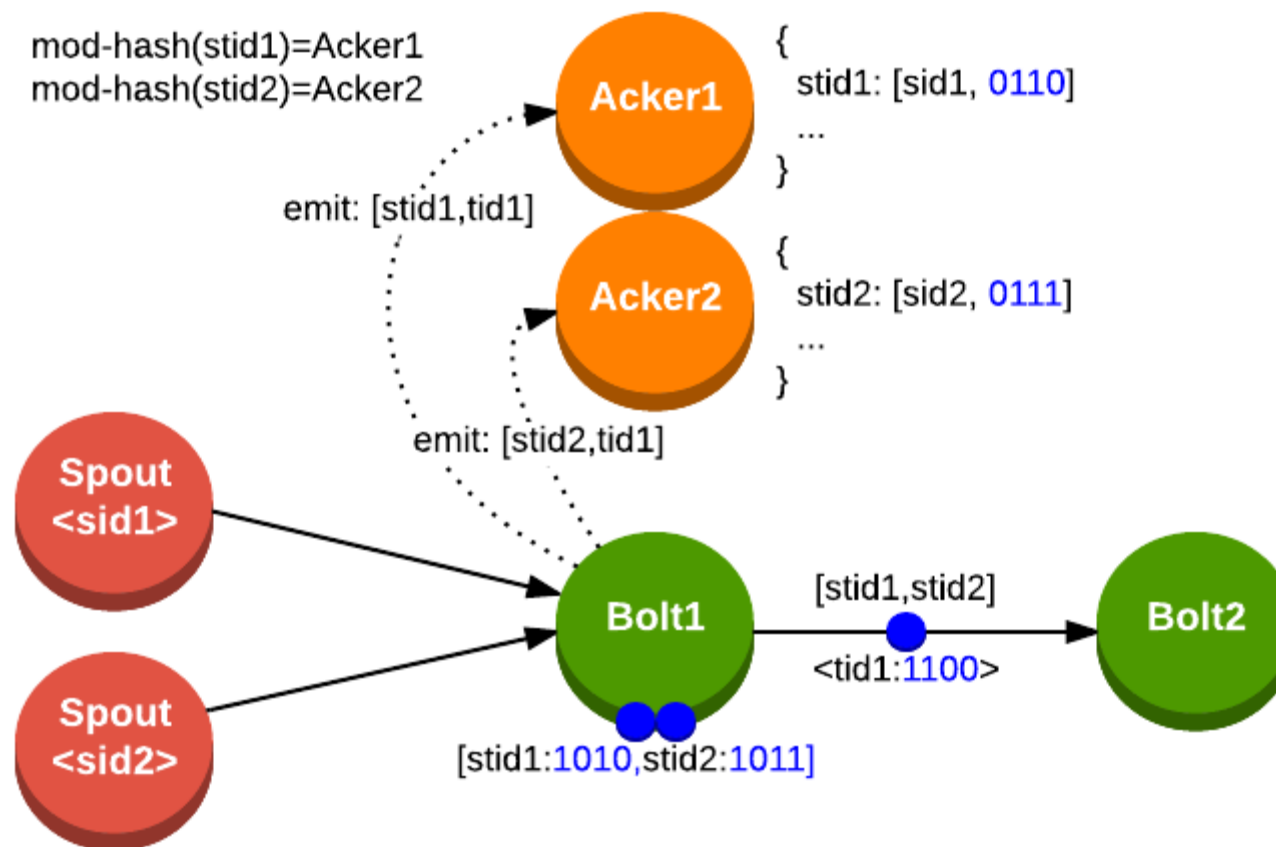
# Гарантия выполнения. Топология



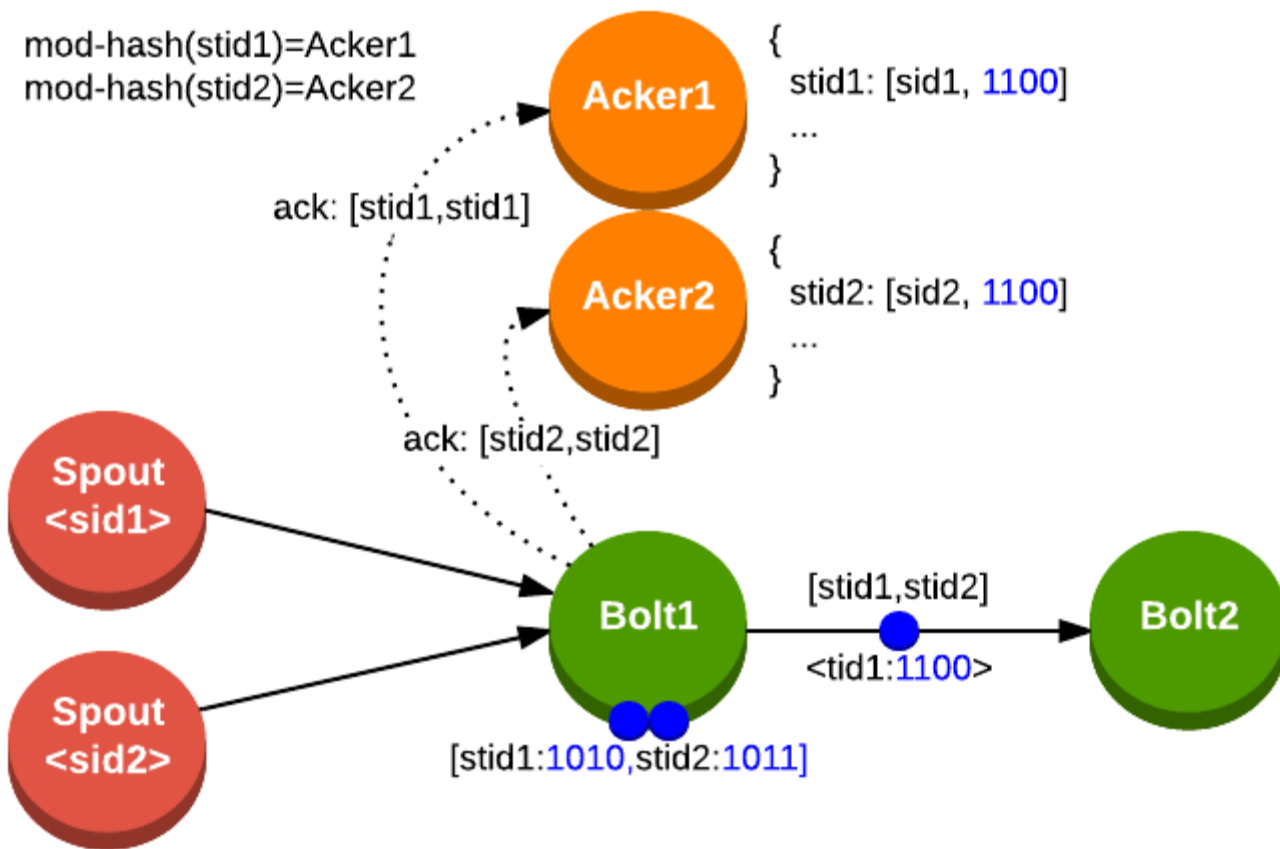
# Гарантия выполнения. Алгоритм. Шаг 1



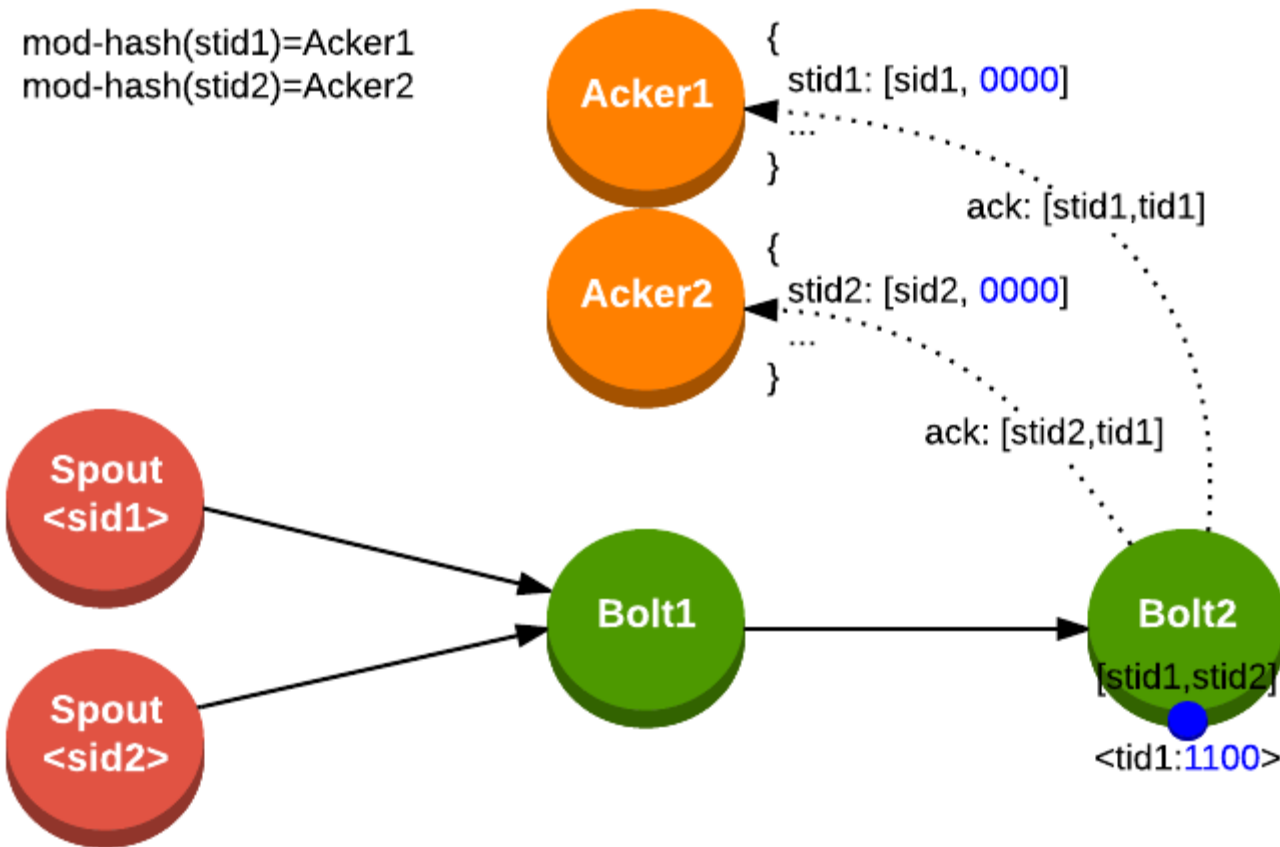
## Гарантия выполнения. Алгоритм. Шаг 2



# Гарантия выполнения. Алгоритм. Шаг 3

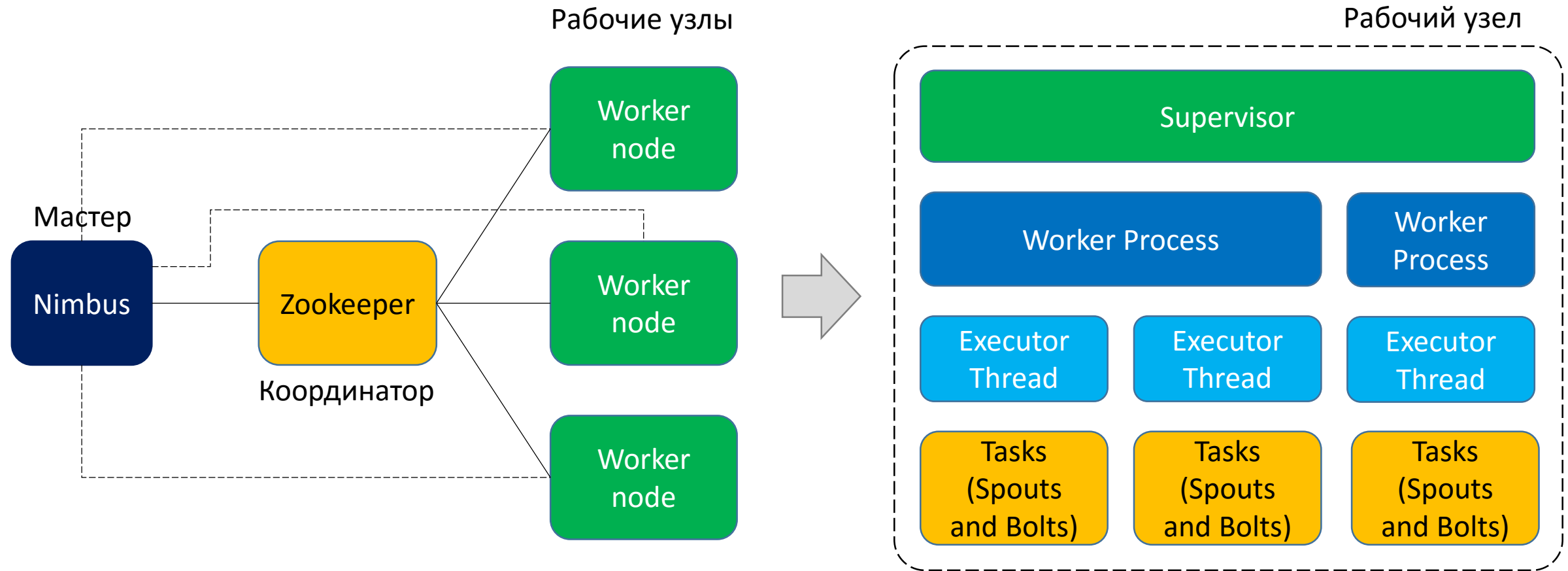


# Гарантия выполнения. Алгоритм. Шаг 4



# Архитектура

# Архитектура





## **Nimbus** (демон) – Apache Thrift сервис

- Распределение кода программы
- Планирование выполнения и назначение задач
- Мониторинг отказов

## **Supervisor** (демон)

- Получает задание от Nimbus
- Запускает worker процессы
- Мониторит состояние worker процессов
- Перезапускает worker'ы при необходимости

## **Worker** (процесс)

- Выполняет часть задач топологии
- Запускает Executor
- Перенаправляет потоки кортежей на соответствующие очереди executor'ов

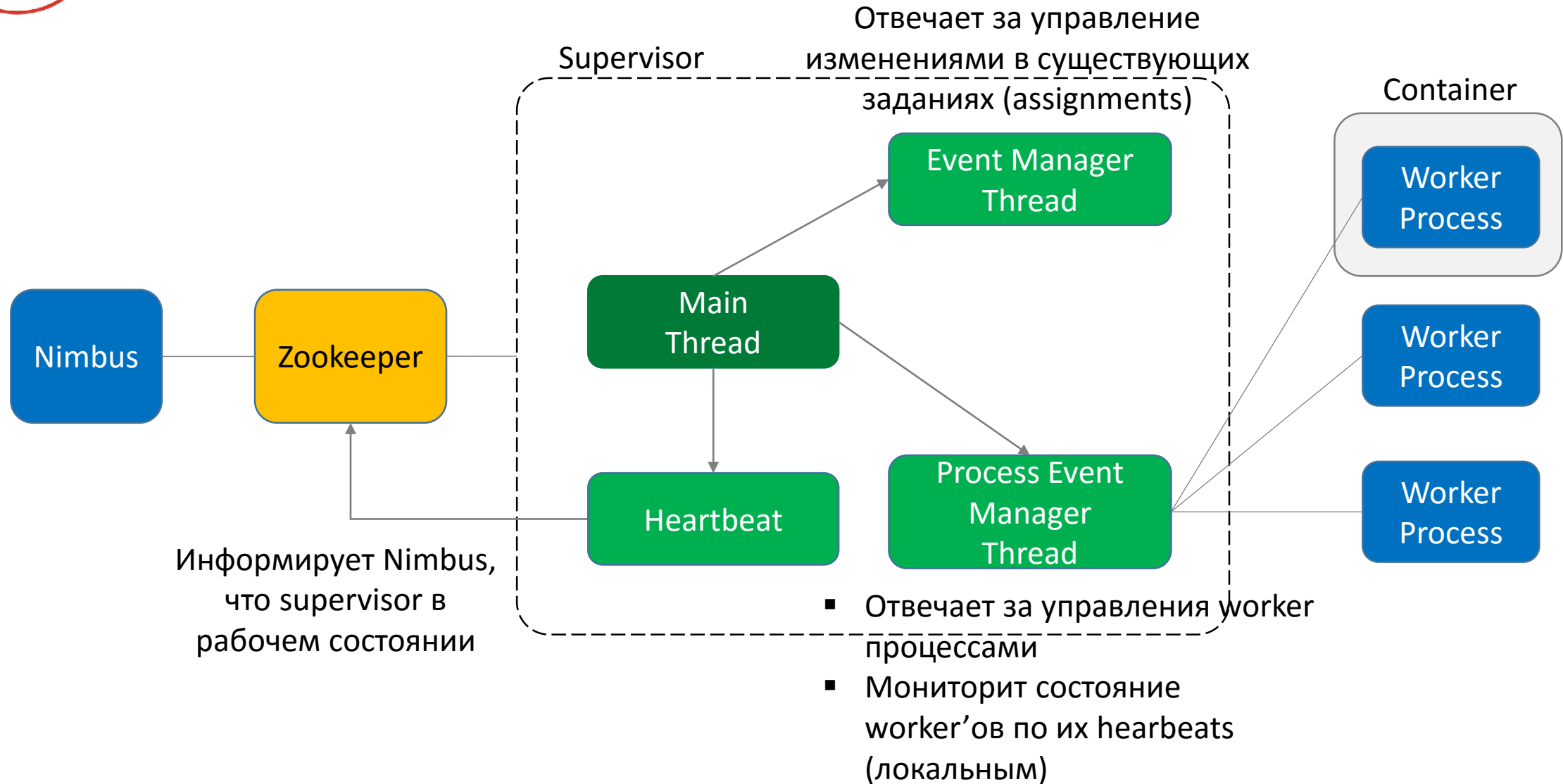
## **Executor** (поток)

- Запускает задачи (task)

## **Task**

- Выполняет непосредственную обработку входного потока

# Архитектура. Supervisor. Компоненты



# Параллелизм в Storm



Количество worker процессов

```
Config.setNumWorkers(int workers)
```



Количество executors (потоков)

```
TopologyBuilder.setBolt(String id, IBasicBolt bolt, Number parallelism_hint)
```

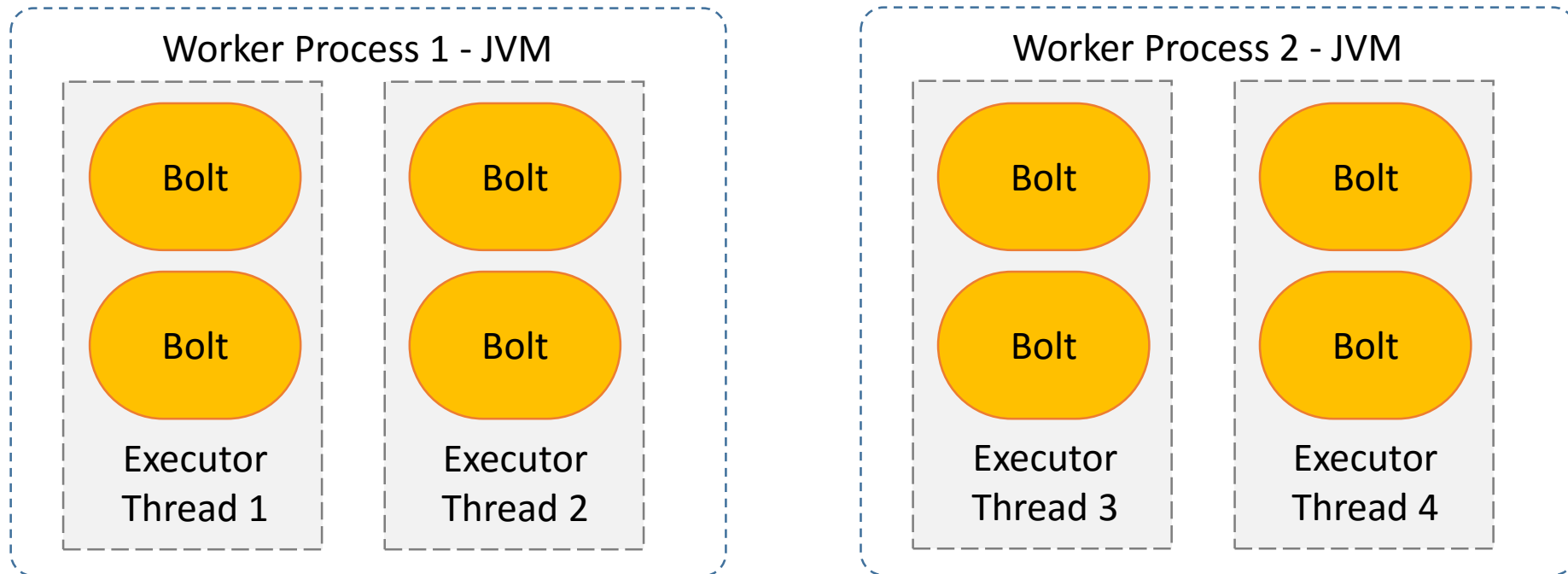


Количество tasks

```
BaseConfigurationDeclarer.setNumTasks(Number val)
```

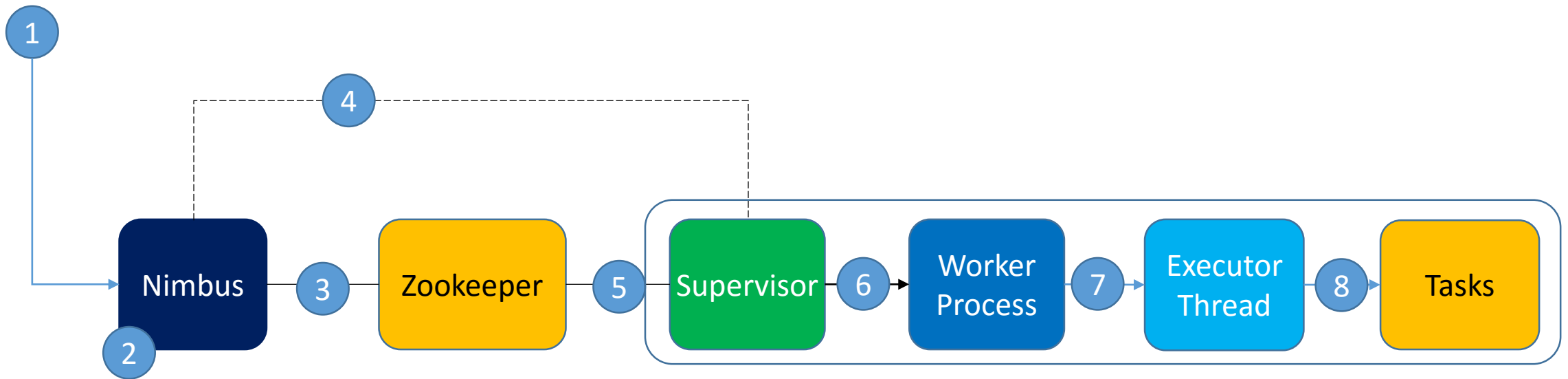
# Распараллеливание выполнения в Storm

Количество рабочих процессов (**worker**'ов)  
↓  
`conf.setNumWorkers(2)`  
Количество исполнителей (**executor**'ов)  
↓  
`builder.setBolt("Bolt", new Bolt(), 4)`  
Количество задач (**task**'ов)  
↙  
`.setNumTasks(8)`



# Запуск приложения

# Запуск приложения



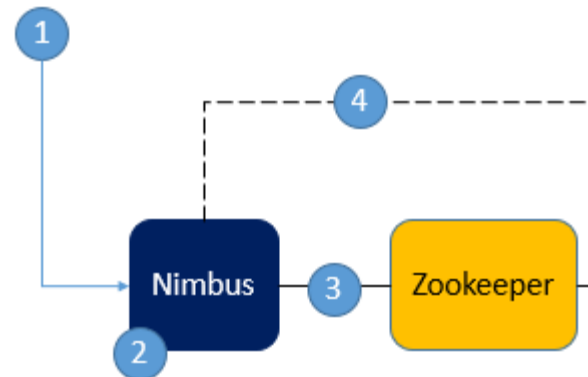
# Запуск приложения

## StormSubmitter

- Отправляет Thift объект топологии на Nimbus (хранится в ZK) (**шаг 1**)
- Загружает jar файлы с кодом на Nimbus (хранится локально)
- Загружает конфигурации в формате JSON

## Nimbus

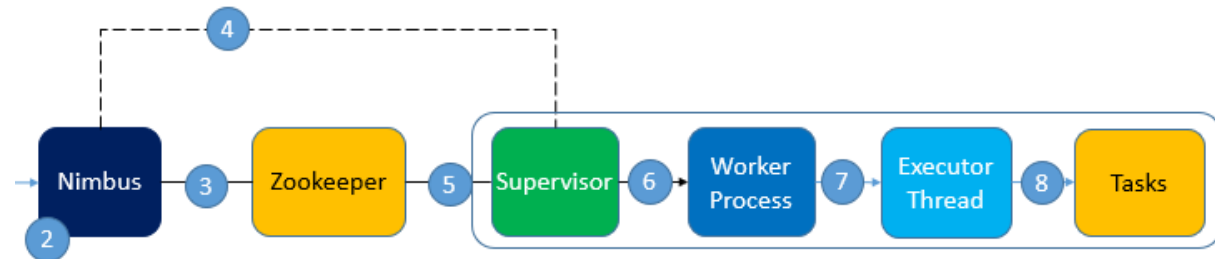
- Сохраняет полученные файлы (**шаг 2**)
- Запускает планировщик заданий – назначает какие задачи будут выполняться и на каких узлах (**шаг 3**):
  - master-code-dir (для загрузки jar и конфигураций из Nimbus)
  - task->node+port (worker)
  - node->host
  - task->start-time-secs
- Записывает данные в ZK и запускается выполнение топологии



# Запуск приложения

## Supervisor (демон)

- Загружает топологию из ZK (**шаг 4**)
- Загружает jar и конфигурации с Nimbus
- Записывает задание в файловую систему (**шаг 5**)
- Синхронизирует задания
- Запускает worker'ы на основе полученных заданий (**шаг 6**)



## Worker (процесс)

- Создает executor'ы (**шаг 7**)
- Устанавливает соединение с другими worker'ами

## Executor (поток)

- Создает task'и (**шаг 8**)

## Task

- Выполняет задачу (spout, bolt)

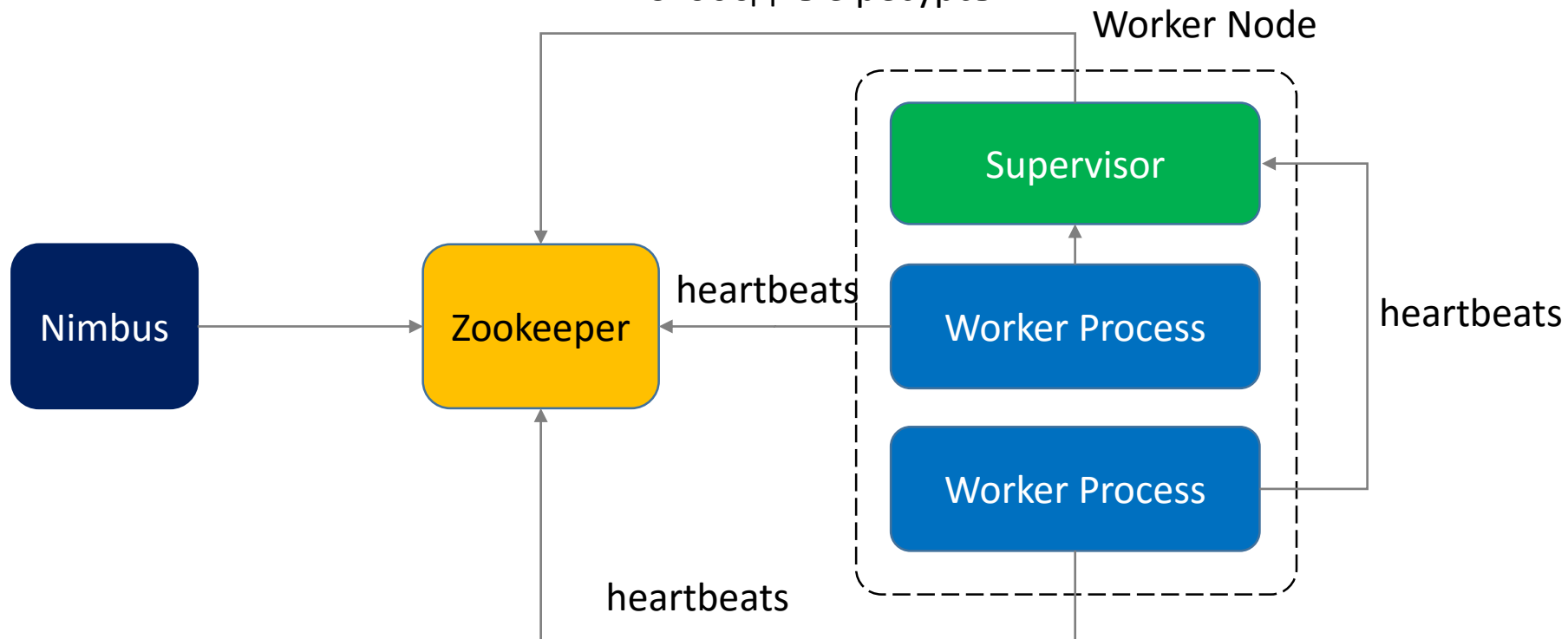


# Отказоустойчивость

# Отказоустойчивость. Heartbeat

## Heartbeats

- Запущенные задачи
- Свободные ресурсы



Текущие состояния хранятся в Zookeeper'е

## Отказ **Nimbus**'а

- Worker'ы продолжают работать
- Supervisor повторно запускает worker'ы, они отказывают
- Пользователь не может запустить новую топологию
- Если отказывает узел, то задачи не могут быть перезапущены на другом узле

## Отказ **Supervisor**'а

- Обработка продолжается
- Задания не синхронизируются с Nimbus
- Nimbus перераспределит работу другому Supervisor'у на другом узле

## Отказ **узла** с Supervisor'ом

- Nimbus перераспределит работу другому Supervisor'у на другом узле

## Отказ **Worker**'а

- Supervisor перезапускает его
- Если не может запустить или Nimbus не получает heartbeat'ы, Nimbus переназначит задачи другой машине

# Коммуникация между компонентами Storm



## Nimbus - Supervisor

Jar-файлы/конфигурация – Thrift



## Nimbus – Supervisor, Nimbus - Worker

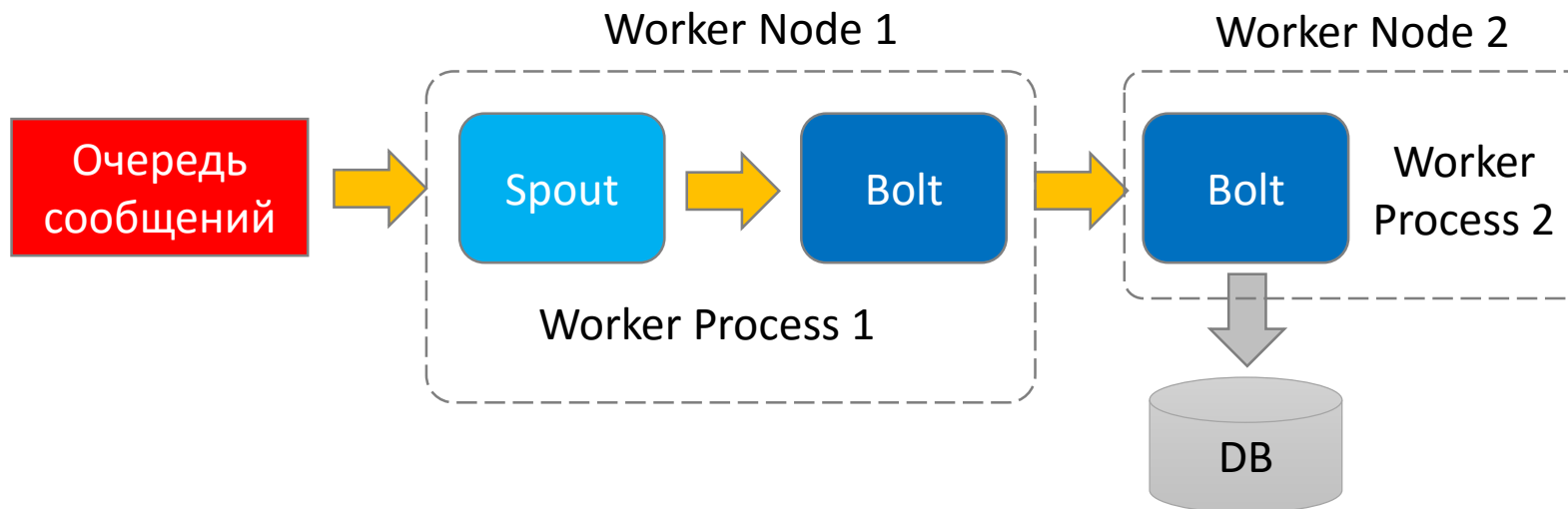
Состояния, heartbeat, задачи и назначенные задачи – Zookeeper



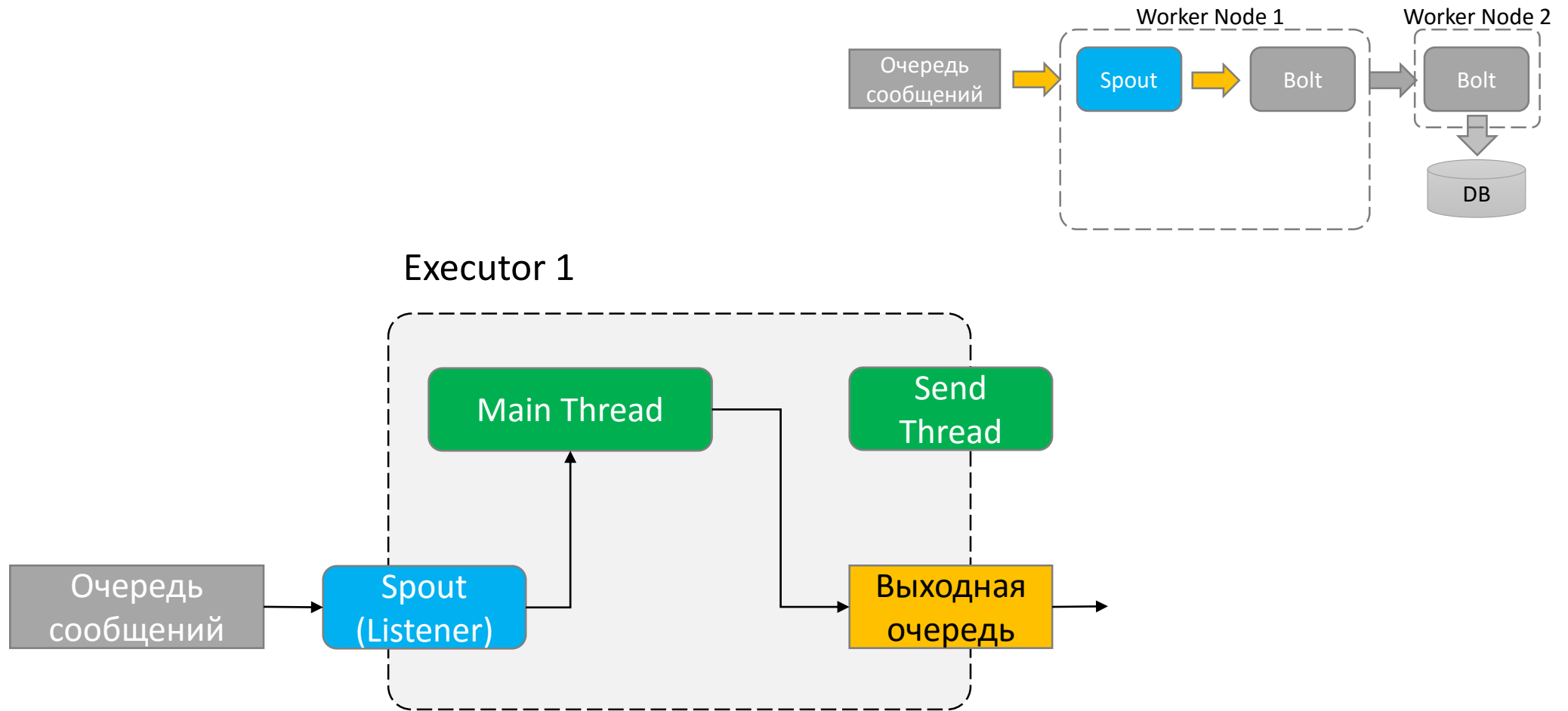
## Executor - Executor

Поток кортежей – LMAX Disruptor, Netty

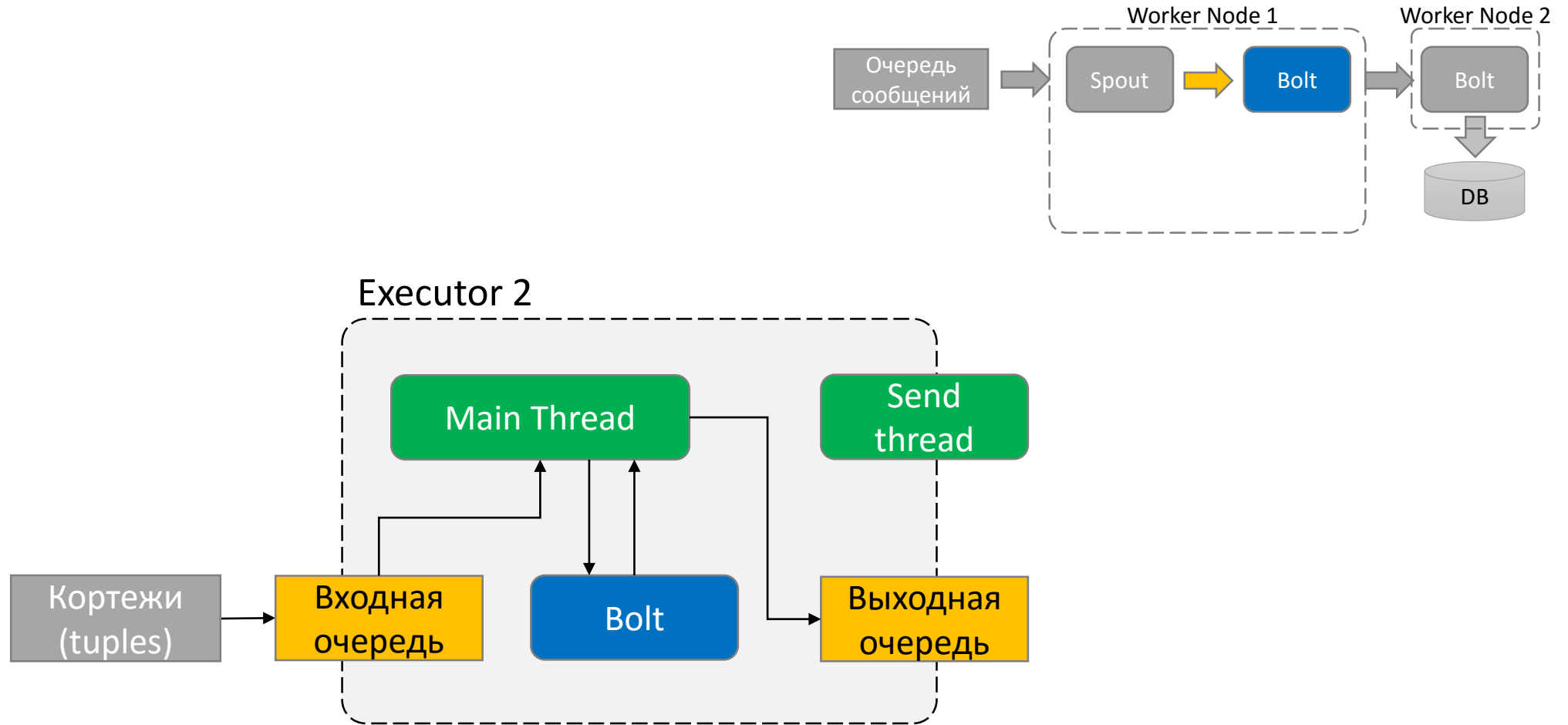
# Продвижение данных



# Продвижение данных. Executor. Task “Spout”

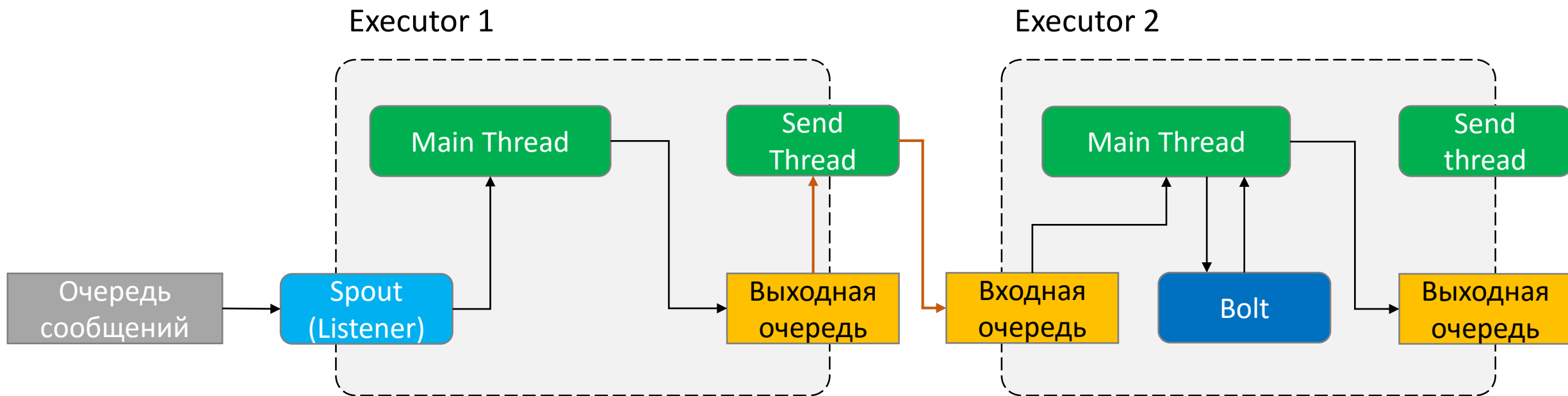
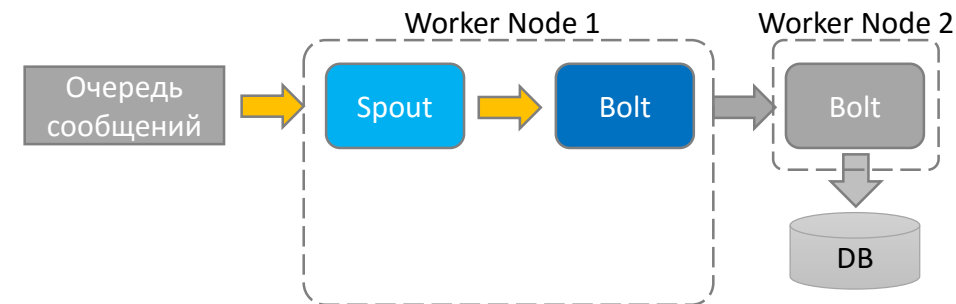


# Продвижение данных. Executor. Task “Bolt”

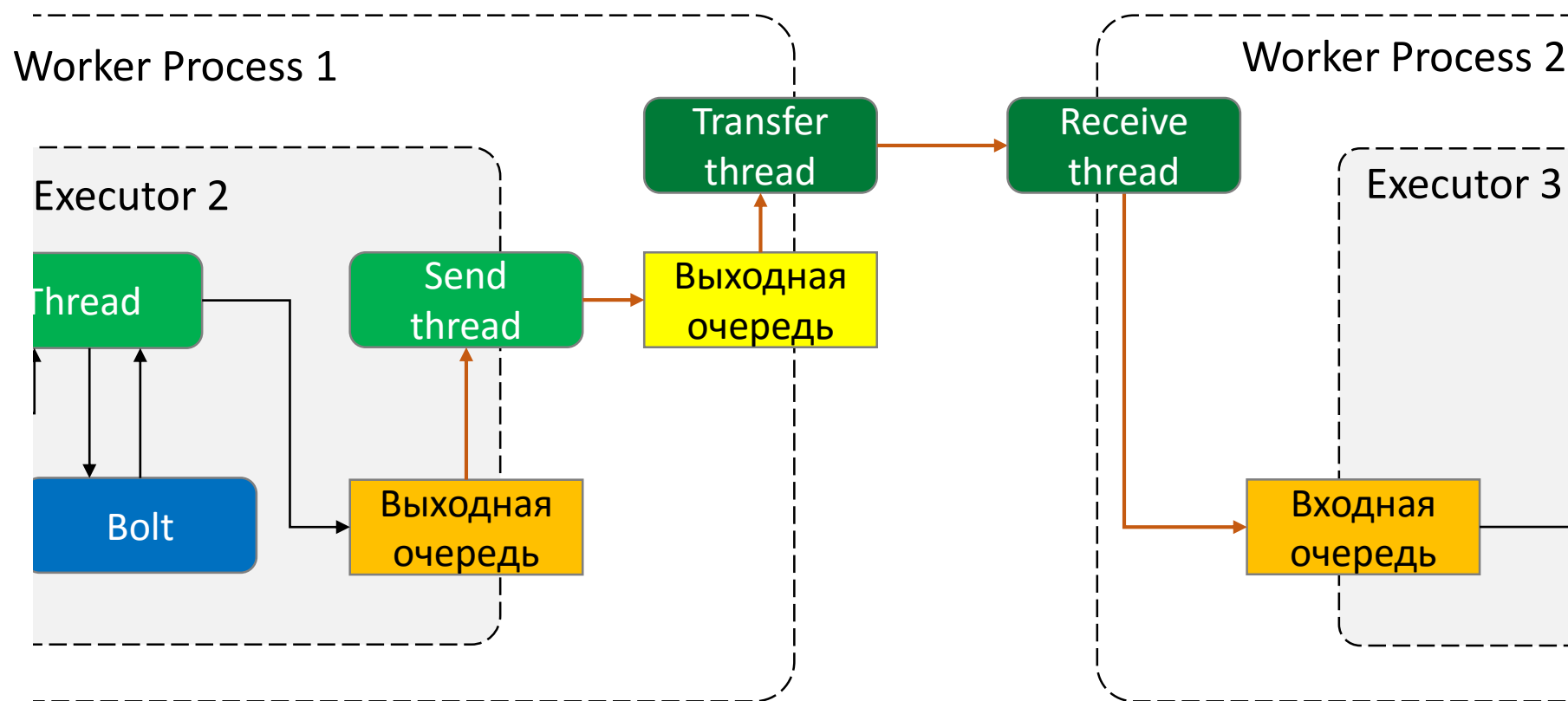
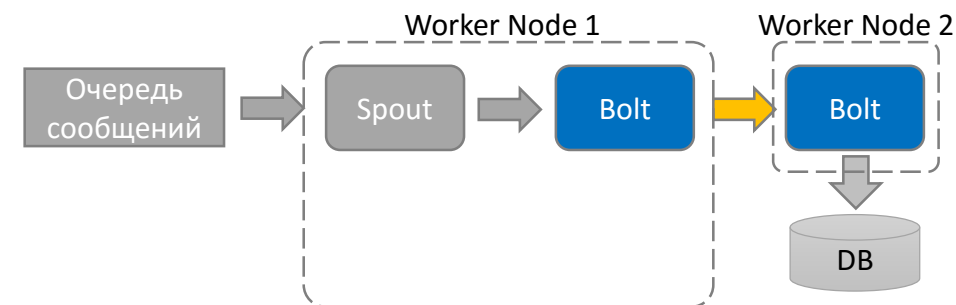




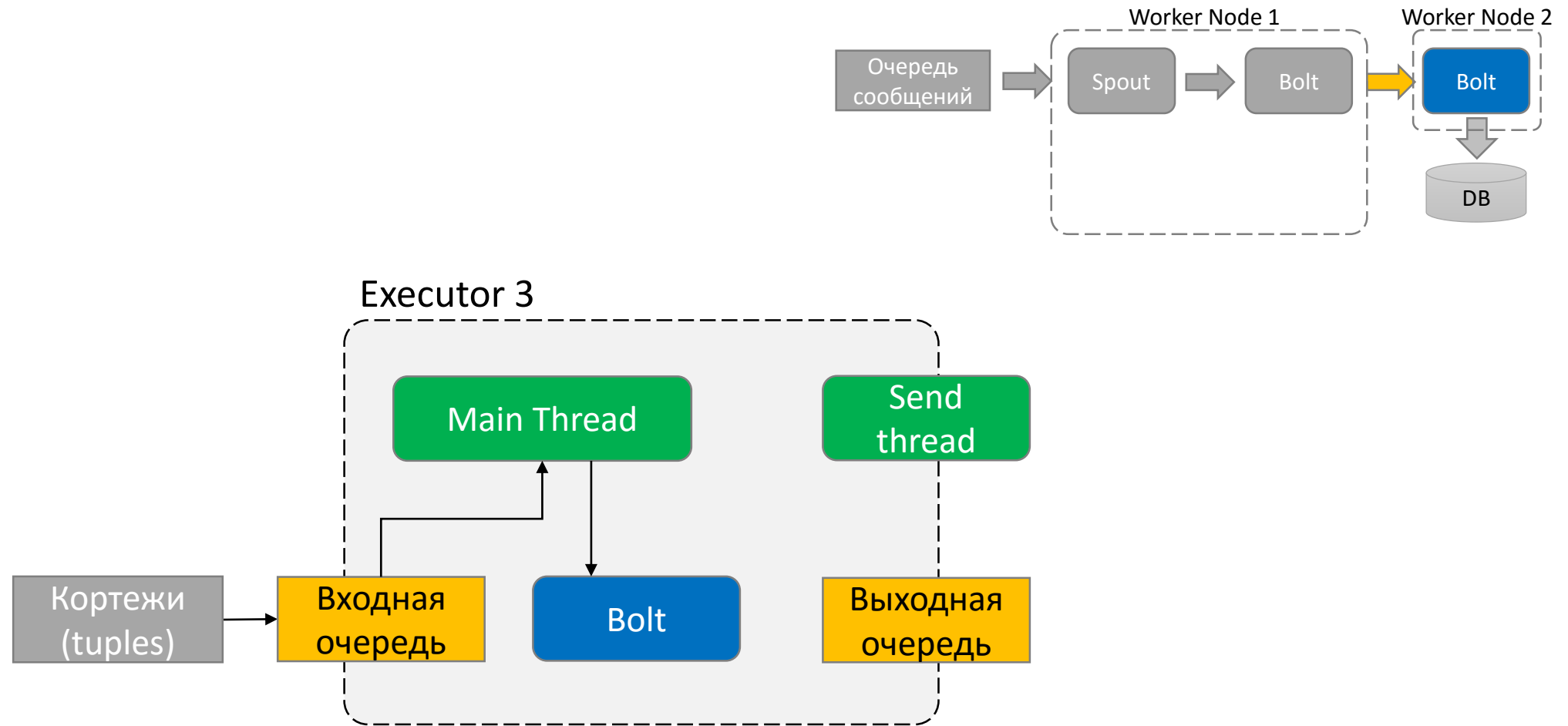
# Продвижение данных между Executors. Один worker процесс



# Продвижение данных между Executors. Два worker процесса

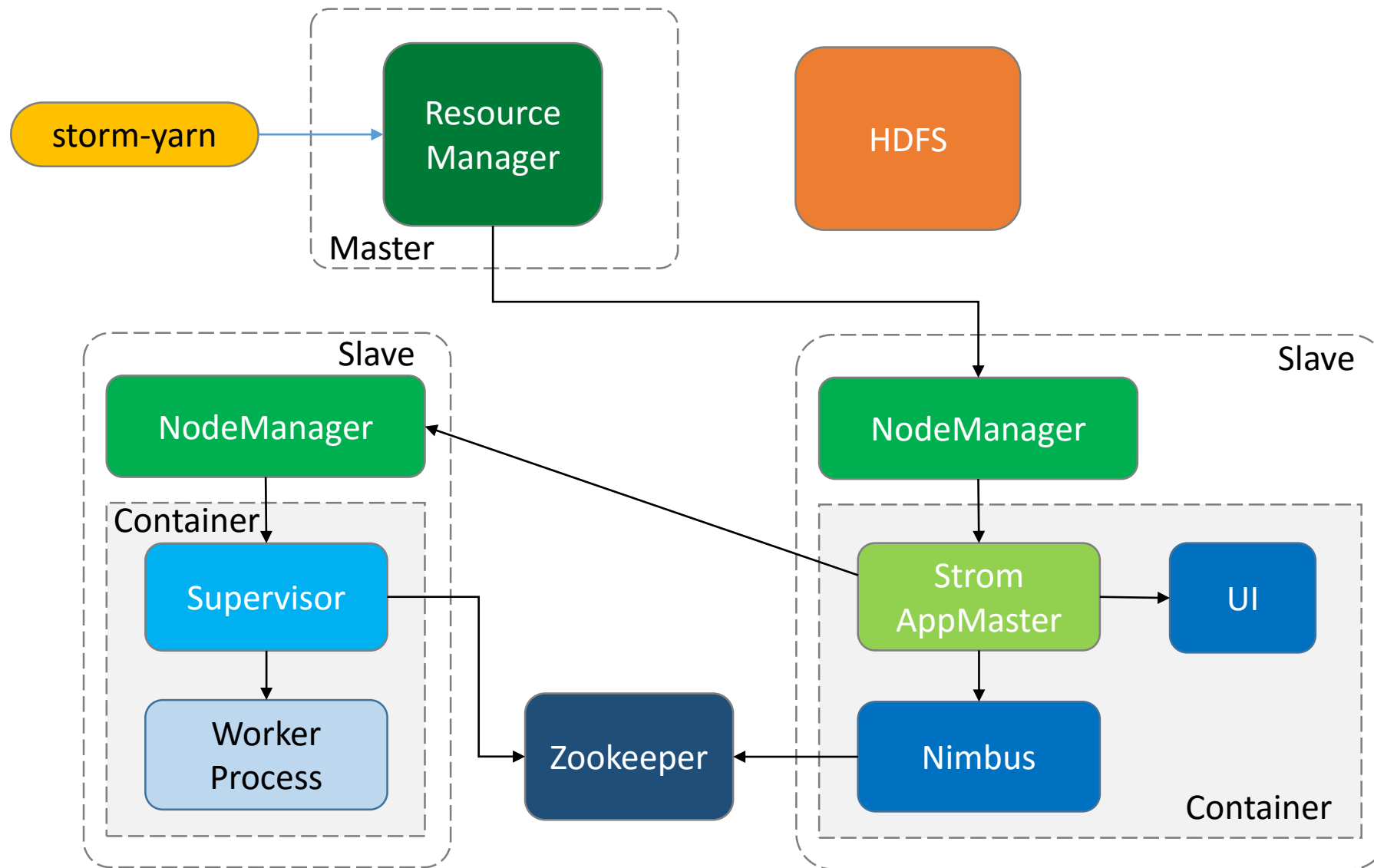


# Продвижение данных. Executor. Терминальный “Bolt”



# Запуск Storm на YARN

# Storm и YARN



# Надстройки

- Trident
- Storm SQL
- Flux Data Driven Topology Builder
- HDFS, Hbase, Cassandra, Redis, MongoDB и пр.
- YARN, Mesos

# ИСТОЧНИКИ

Storm Applied by Sean T. Allen, Matthew Jankowski, Peter Pathirana (book)

Building Python Real-Time Applications with Storm by Kartik Bhatnagar Barry Hart (book)

[Apache Storm](#) (official website)

[Concepts](#) (doc)

[Understanding the Parallelism of a Storm Topology](#) (doc)

[Fault Tolerant Message Processing in Storm](#) (blog)

[Storm: Real-Time Data Processing](#) (blog)

[Deploying Storm on Hadoop for Advertising Analysis](#) (blog)