第七章　流计算系统Storm

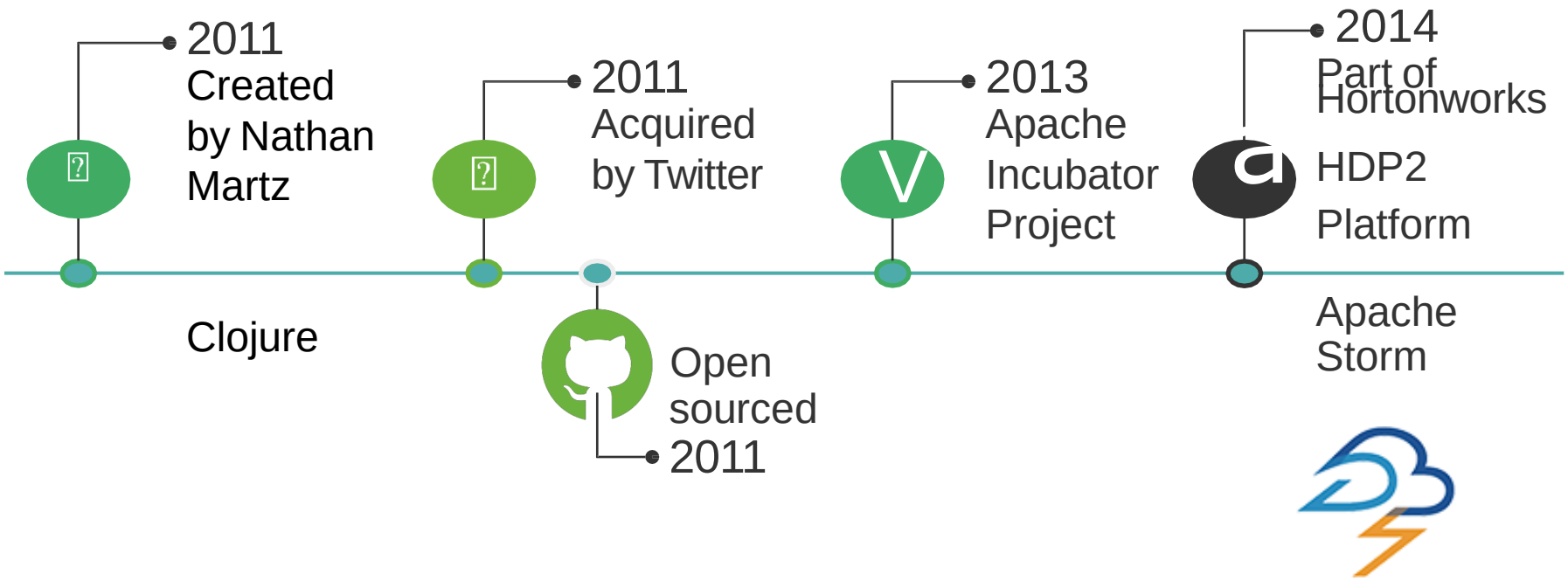
徐 辰
cxu@dase.ecnu.edu.cn

華東師範大學

流计算 vs. 批处理

- 批处理：处理的输入数据是静态的，即输入数据在计算开始前就已经确定好了
- 流计算：处理的输入数据是动态的，即输入数据在计算开始后才逐步到达
  - 在网络监控、传感监测等其它领域，计算结果往往需要根据实时采集到的数据实时反馈
  - 例子：为准确测量实际温度值，可以利用传感器采集温度并实时计算某一时段内的平均值
    - “求平均值”这一计算是预先确定好的
    - 数据是不断到达的

Storm历史


2011
Created
by Nathan
Martz
2011
Acquired
by Twitter
2013
Apache
Incubator
Project
2014
Part of
Hortonworks
HDP2
Platform
Clojure
Open
sourced
2011
Apache
Storm

大纲

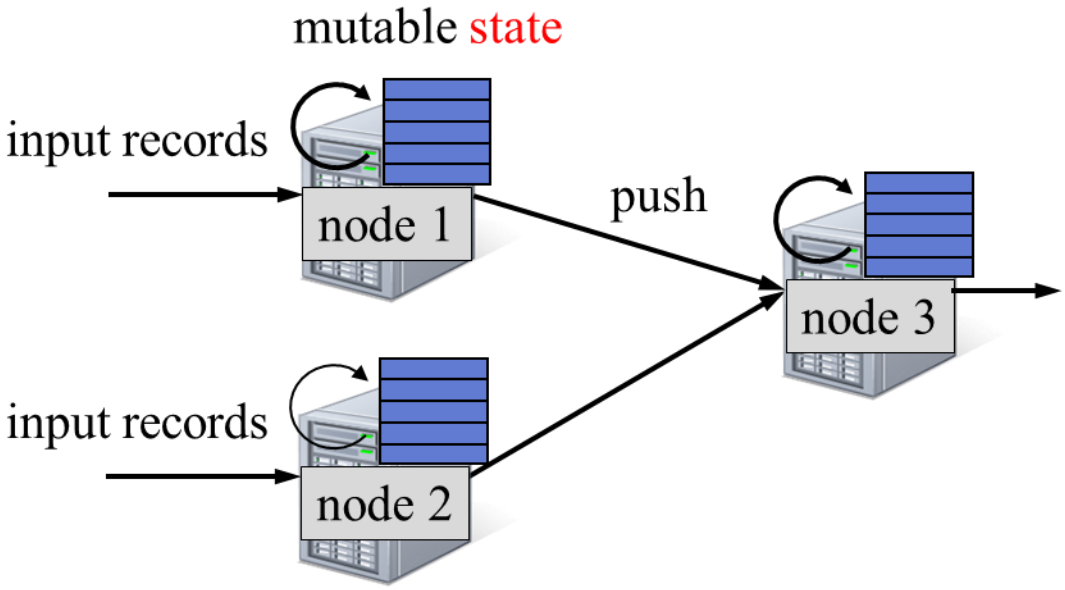
- 设计思想
  - 连续处理
  - 数据模型
  - 计算模型
- 体系架构
- 工作原理
- 容错机制
- 编程示例

短时运行 vs. 长期驻留

- 批处理系统中负责执行计算的任务是短时运行的
- 流计算系统中负责执行计算任务的线程或进程长期驻留在系统中

连续处理(Continuous processing)

- 输入的流数据记录不断地进入系统
- 计算任务长期驻留并且更新自身的状态
  - 状态是一种特殊的数据，用于保存从流计算开始到目前为止得到的计算结果

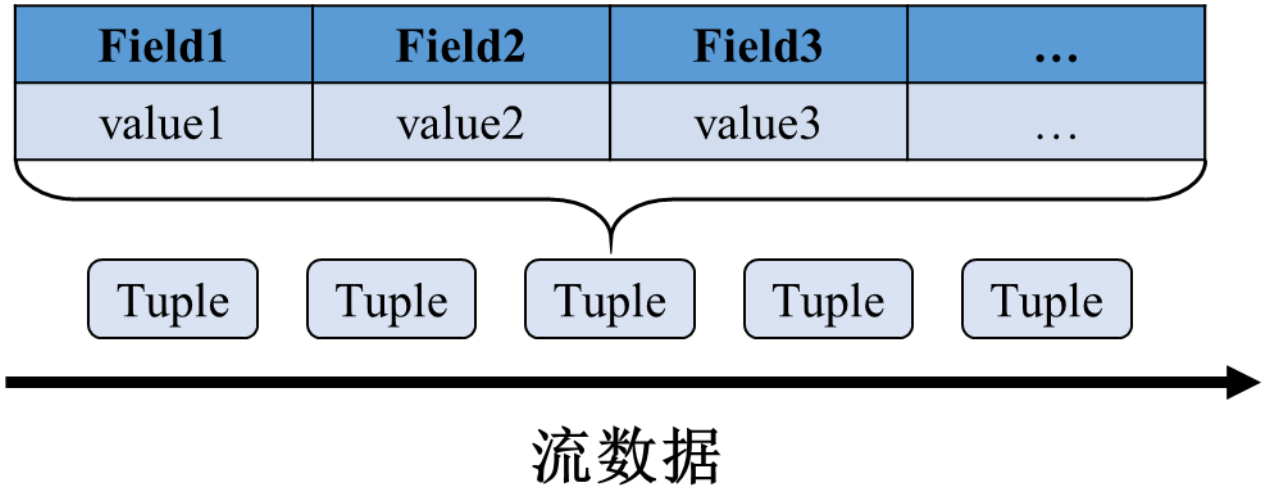


大纲

- 设计思想
  - 连续处理
  - 数据模型
  - 计算模型
- 体系架构
- 工作原理
- 容错机制
- 编程示例

数据模型：Tuple

- 流数据是一个无界的、连续的元组序列
- 一个元组就是系统处理的一条记录，每一条记录（元组）包含若干个字段



数据模型的比较

Key-value Pair

Tuple

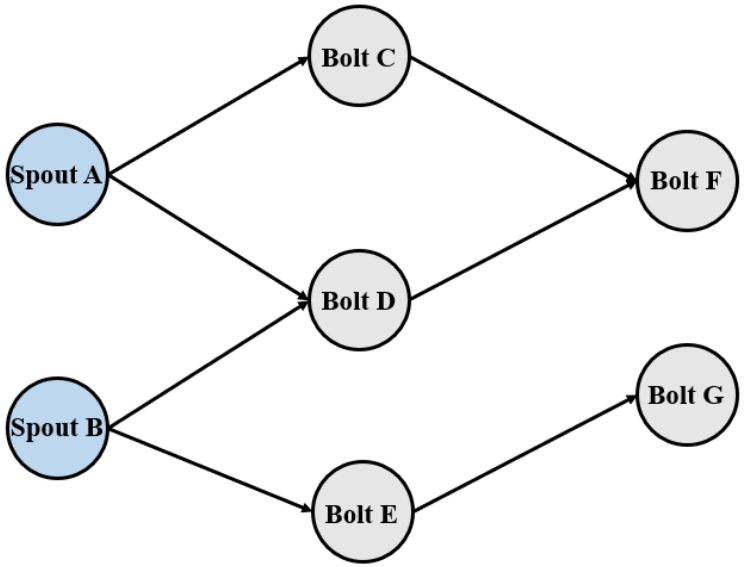
RDD

大纲

- 设计思想
  - 连续处理
  - 数据模型
  - 计算模型
- 体系架构
- 工作原理
- 容错机制
- 编程示例

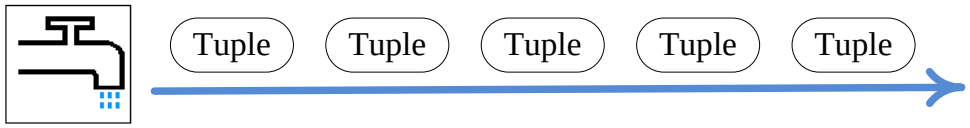
逻辑计算模型：Topology

- 由Spouts和Bolts组成的DAG
  - 顶点：Spout或Bolt（数据处理逻辑）
  - 边：Bolt订阅的流数据（数据流动的方向）

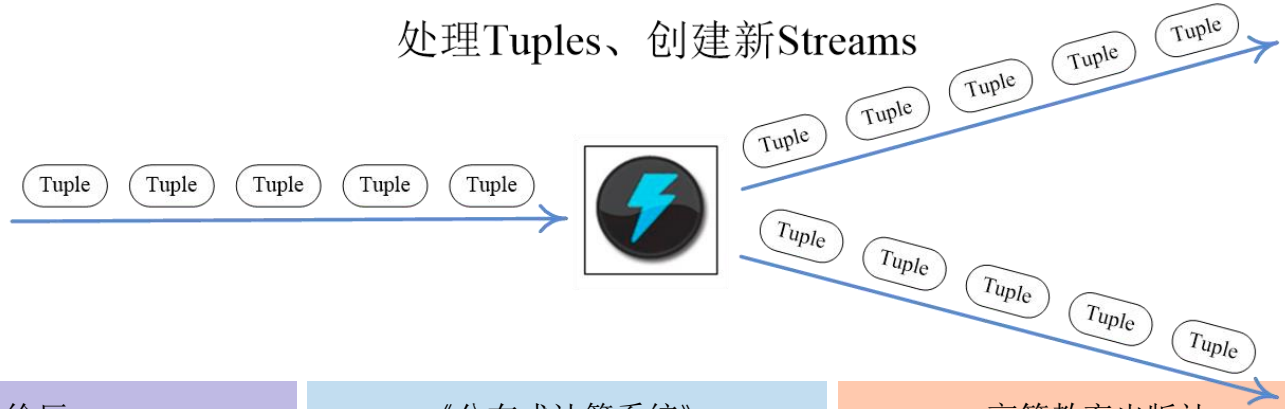


逻辑计算描述：Topology

- Spout：Stream的源头，从外部数据源（Kafka、数据库等）读取数据，然后封装成Tuple，发送给Bolt

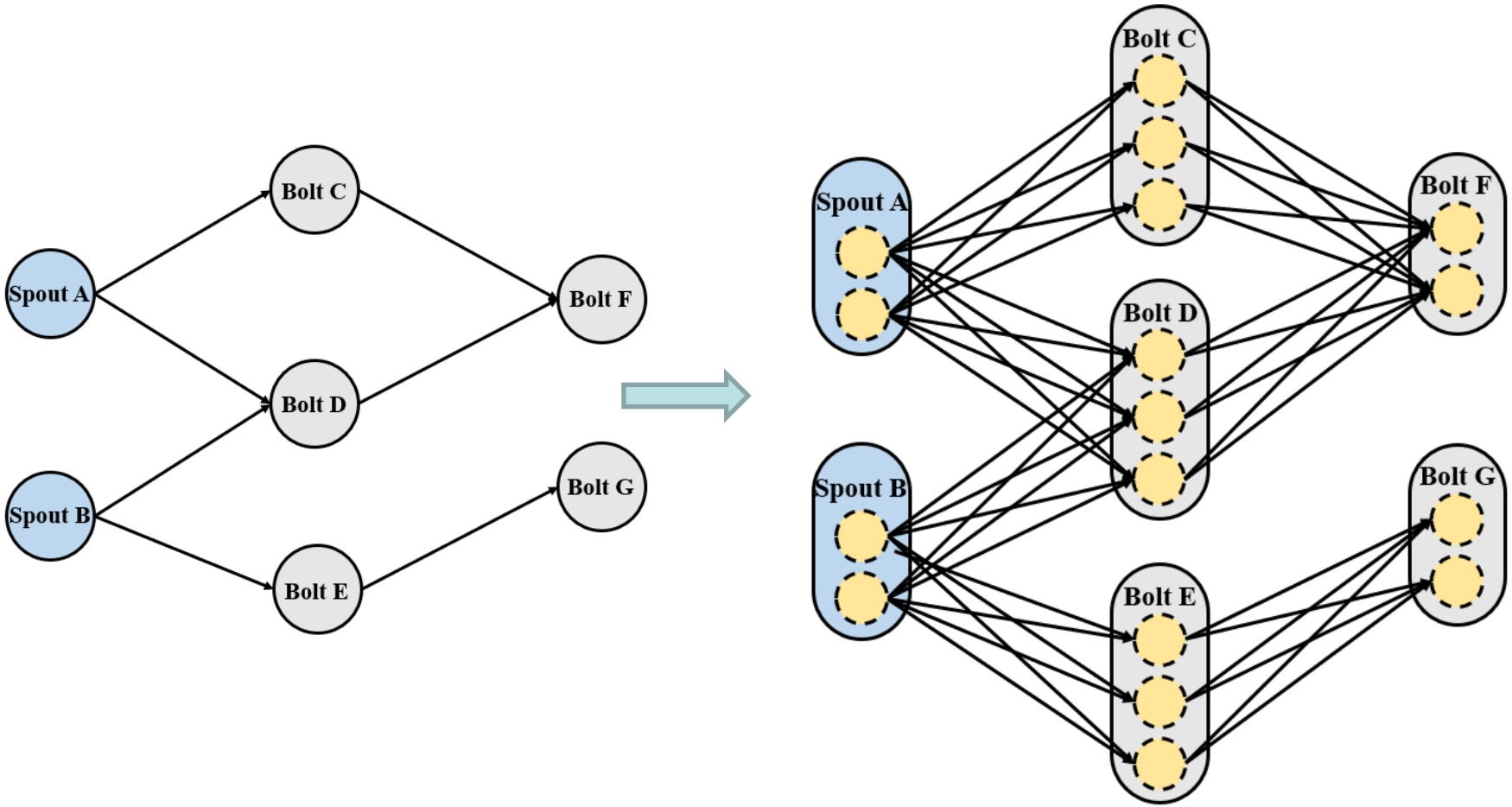


- Bolt：描述Streams的转换过程，将处理后的Tuple作为新的Streams发送给其他Bolt



物理计算模型

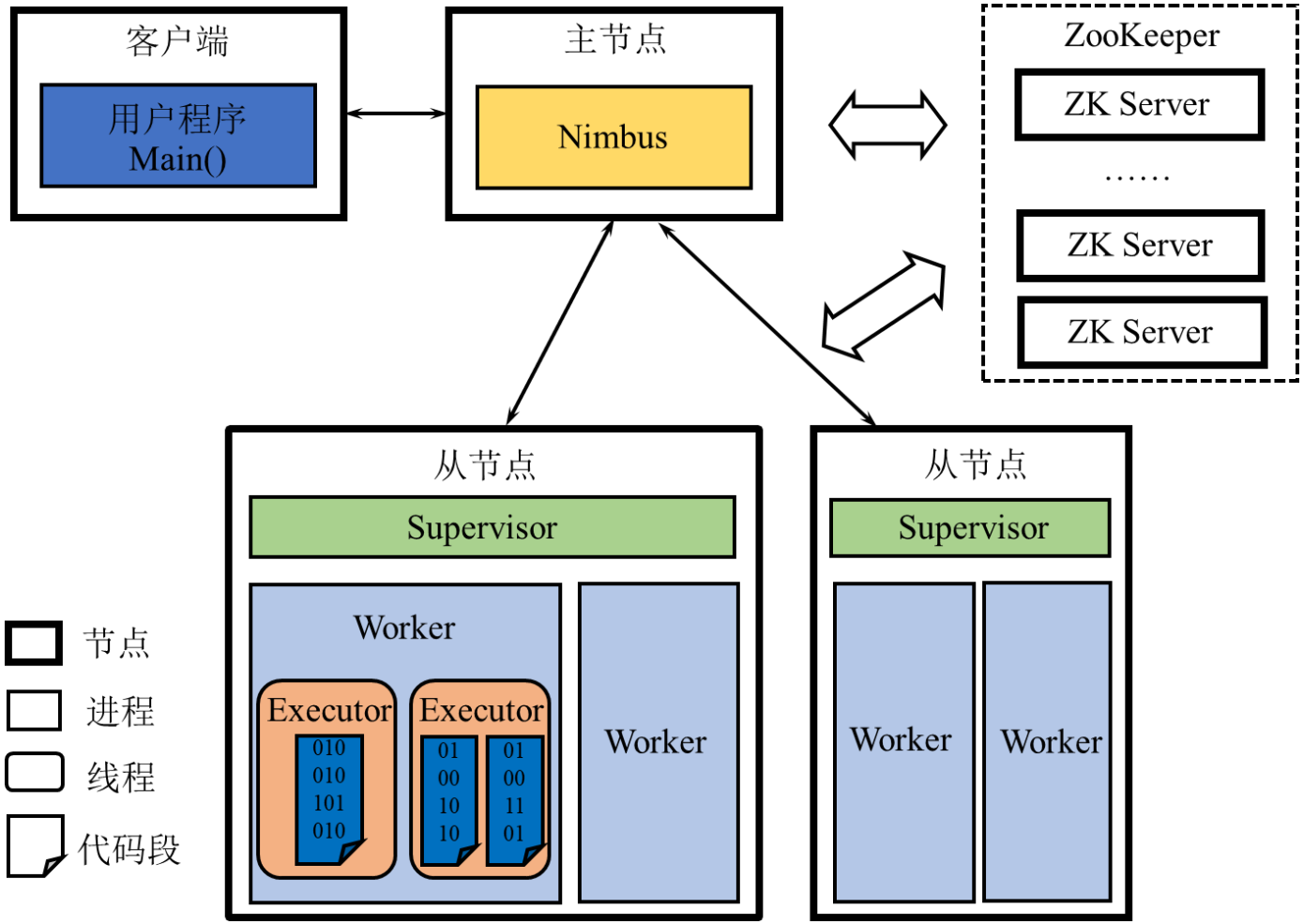
- Spout/Bolt物理上由若干个task来实现



大纲

- 设计思想
- 体系架构
  - 架构图
  - 应用程序执行流程
- 工作原理
- 容错机制
- 编程示例

Storm架构

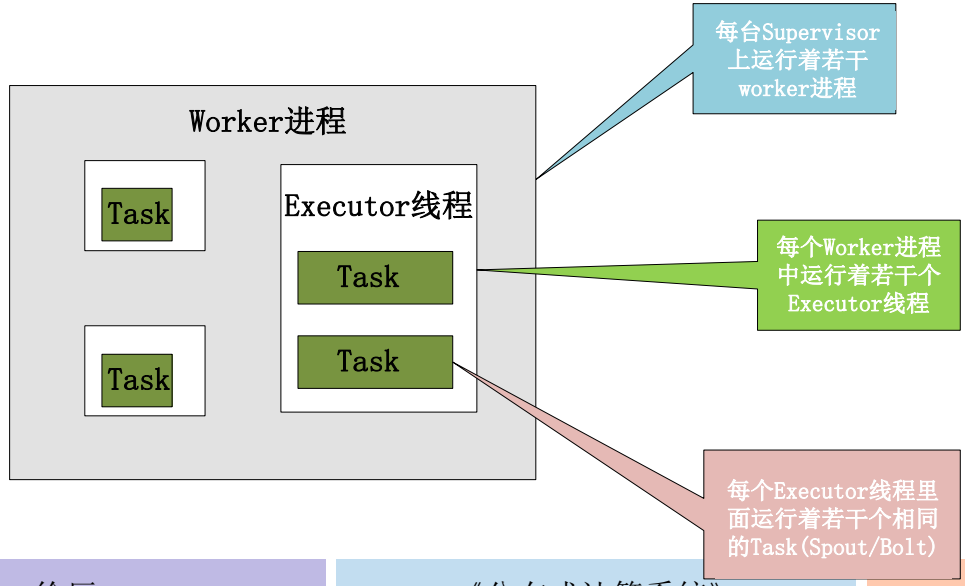

客户端
用户程序
Main()
主节点
Nimbus
ZooKeeper
ZK Server
.....
ZK Server
ZK Server
从节点
Supervisor
Worker
Executor
010
010
101
010
Executor
01
00
10
10
01
00
11
01
Worker
从节点
Supervisor
Worker
Worker
节点
进程
线程
代码段

Storm角色

- Nimbus：主节点运行的后台程序，负责分发代码、分配任务和监测故障
- Supervisor：从节点运行的后台程序
  - 负责监听所在机器的工作，根据Nimbus分配的任务来决定启动或停止Worker进程
  - 一个从节点上同时运行若干个Worker进程
- Zookeeper：负责Nimbus和Supervisor之间的所有协调工作
  - 若Nimbus进程或Supervisor进程意外终止，重启时也能读取、恢复之前的状态并继续工作

Storm角色

- Worker：Worker进程内部运行一个或多个Executor线程，从而实际执行任务
  - executor：产生于worker进程内部的线程，会执行同一个组件的一个或者多个task
  - Task: 执行数据处理的代码实例（spout/bolt）



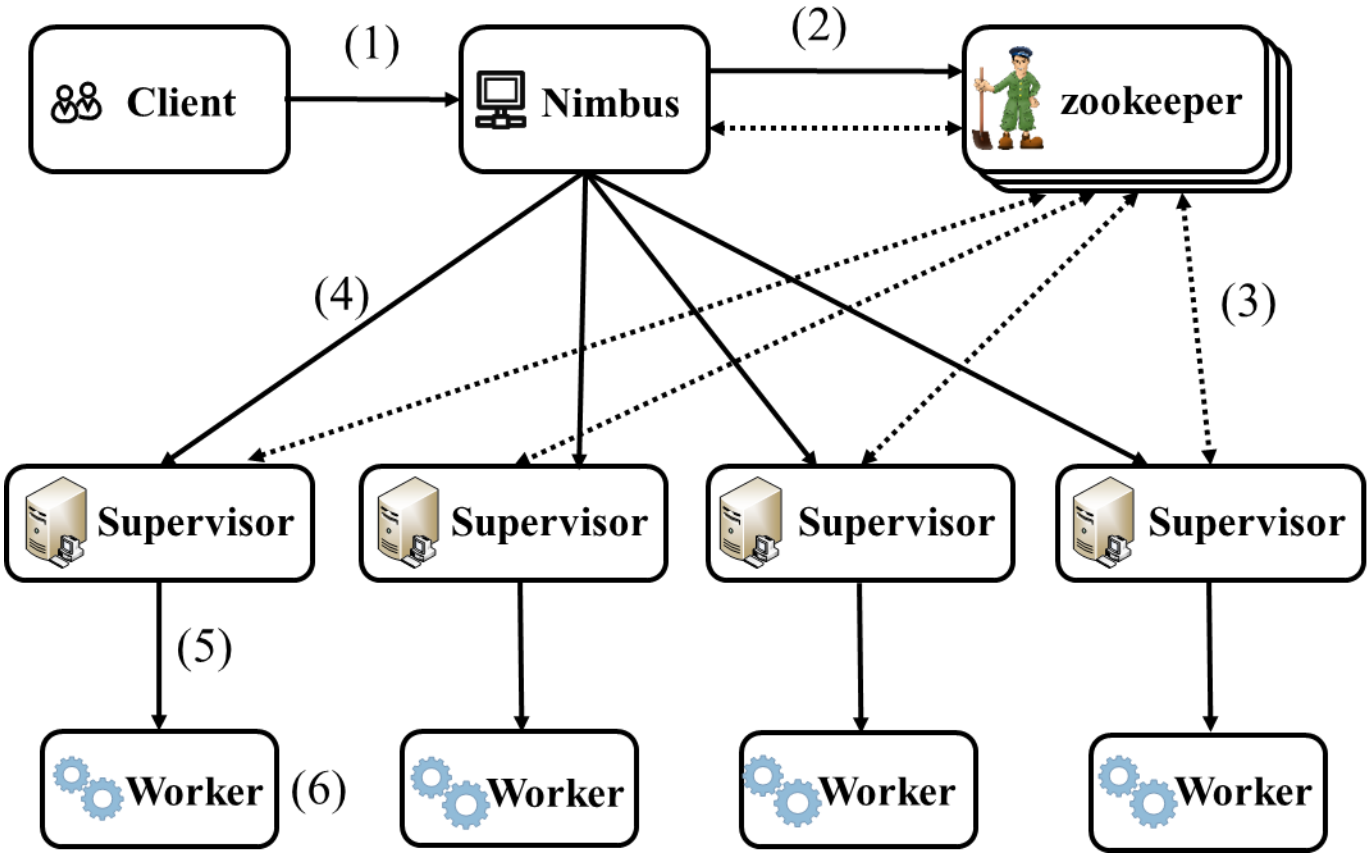
Storm与MapReduce/Spark比较

	MapReduce	Storm	Spark
系统进程	JobTracker	Nimbus	Master
	TaskTracker	Supervisor	Worker
	Child	Worker	CoarseGrainedExecutorBackend
工作线程	/	Executor	Task
任务代码	Task	Task	
基础接口	Map/Reduce	Spout/Bolt	RDD API

大纲

- 设计思想
- 体系架构
  - 架构图
  - 应用程序执行流程
- 工作原理
- 容错机制
- 编程示例

执行流程


Client
(1)
Nimbus
(2)
zookeeper
(4)
(3)
Supervisor
Supervisor
Supervisor
Supervisor
(5)
Worker
(6)
Worker
Worker
Worker

执行流程

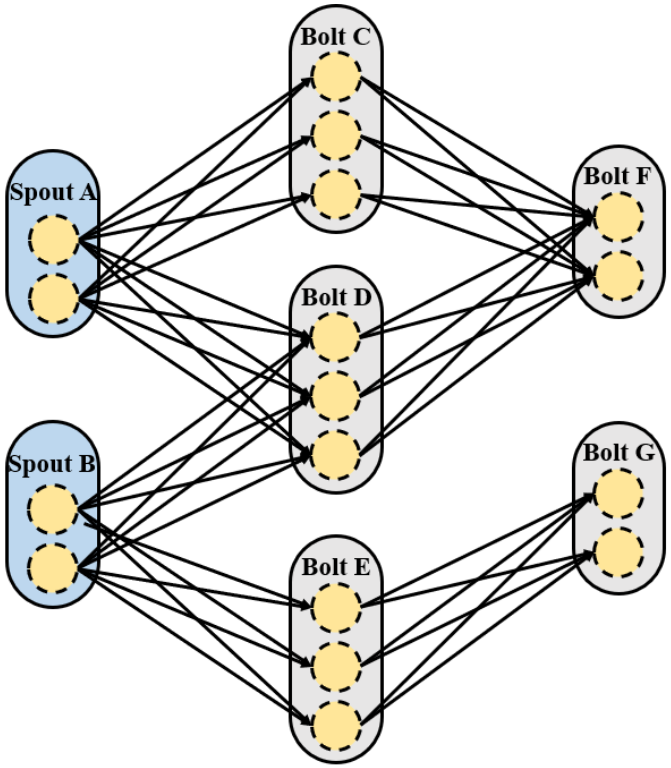
1. 用户编写的Topology程序，经过序列化、打包并提交给主节点Nimbus
2. Nimbus创建一个组件与物理节点的对应关系文件，将该文件原子地写入Zookeeper中某Znode
3. 所有Supervisor监听Znode来得到通知从而获取所在节点所需执行的组件任务
4. Supervisor从Nimbus处拉取可执行的代码
5. Supervisor启动若干Worker进程执行具体的任务
6. Worker进程根据ZooKeeper中获取的文件信息，启动若干个Executor线程，该线程负责执行组件（Spout或Bolt）所描述的任务（Task）

大纲

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例

Task之间的Tuple传输

- Spout和Bolt之间，或者不同的Bolt之间的Tuple传输表现为属于上游组件的task和属于下游组件的task之间的Tuple传输



Tuple传输中的问题

- 对于一组Tuple来说，上游组件的task发送哪些Tuple给下游组件的task?
  - 如何对这组Tuple进行划分?
- 对于一条Tuple来说，上游组件的task如何向下游组件的task传递Tuple?
  - 立即传输还是等待多条Tuple成批次传输?

大纲

- 设计思想
- 体系架构
- 工作原理
  - 流数据分组策略
  - 元组传递方式
- 容错机制
- 编程示例

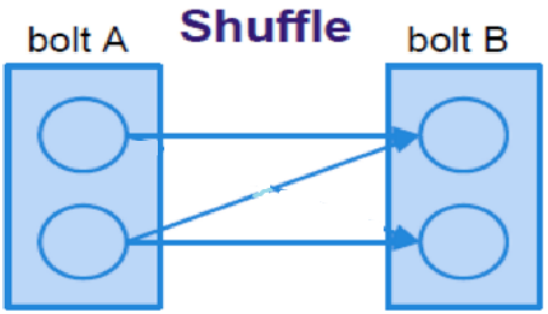
流数据分组策略

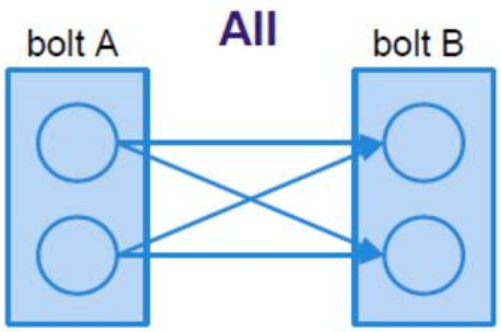
- 定义了两个有订阅关系的组件间（如Spout和Bolt之间，或者不同的Bolt之间）进行元组传输的方式
  - 对于上游的组件来说，流数据的分组策略定义了属于该组件的多个任务发送哪些元组给下游组件的任务
  - 对于下游的组件来说，流数据的分组策略定义了下游组件的多个任务之间对于元组的划分策略

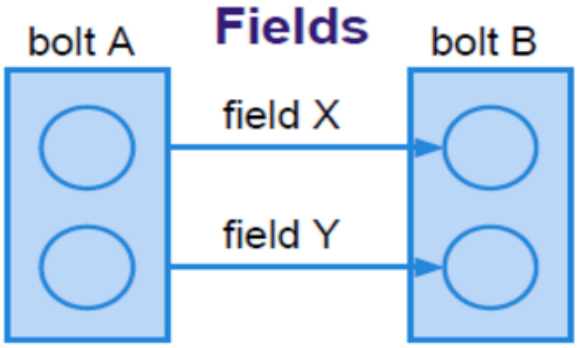
Stream Groupings

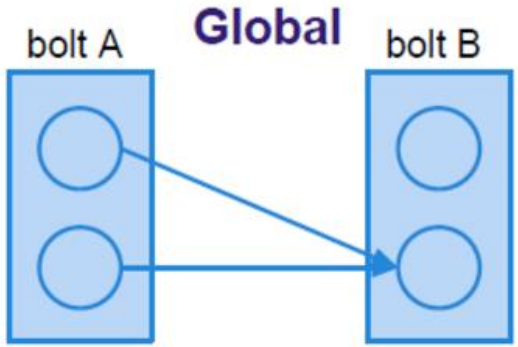
- ShuffleGrouping：随机分组，随机分发Stream中的Tuple，保证每个Bolt的Task接收Tuple数量大致一致
- FieldsGrouping：按照字段分组，保证相同字段的Tuple分配到同一个Task中
- AllGrouping：广播发送，每一个Task都会收到所有的Tuple
- GlobalGrouping：全局分组，所有的Tuple都发送到同一个Task中
- DirectGrouping：直接分组，直接指定由某个Task来执行Tuple的处理

Stream Groupings









大纲

- 设计思想
- 体系架构
- 工作原理
  - 流数据分组策略
  - 元组传递方式
- 容错机制
- 编程示例

消息传递（Message Passing）机制

- 一次一元组或一次一记录
  - 一旦上游组件的任务处理完一条元组，就立即发送给下游组件的任务，且一次发送一条元组
  - 对于连续两条元组，上游组件的任务处理了第一条元组就可立即发送给下游组件的任务，而不必等待下一条元组的处理结果
  - 这种立即发送的消息传递机制有利于减少处理的延迟，从而满足实时性需求

讨论

- 流计算
  - 一次一元组
  - 数据传输立即进行，无阻塞
- 批处理
  - 数据传输是成块进行的
  - MapReduce和Spark的shuffle阶段存在阻塞

大纲

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例

故障类型

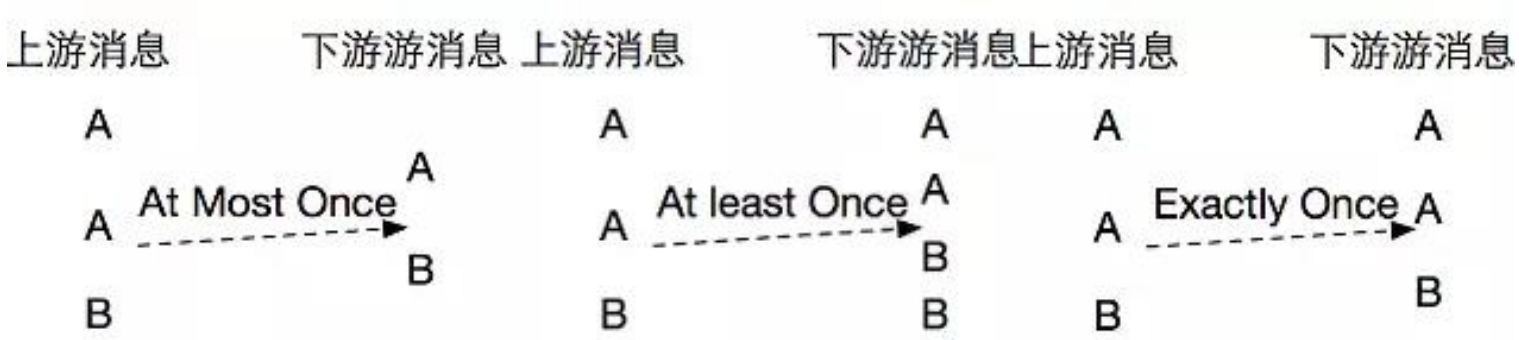
- ZooKeeper故障：极端情况
- Nimbus故障：怎么办?
- Supervisor故障
  - 重启新的Supervisor
  - 原来监控的所有Worker重新调度、启动
- Worker故障
  - 重新启动

大纲

- 设计思想
- 体系架构
- 工作原理
- 容错机制
  - 容错语义
  - 元组树
  - ACK机制
  - 消息重放
- 编程示例

容错语义

- 流计算系统容错语义
  - At Most Once：消息可能会丢失
  - At Least Once：消息不会丢失，可能会重复
  - Exactly Once：消息不丢失，不重复



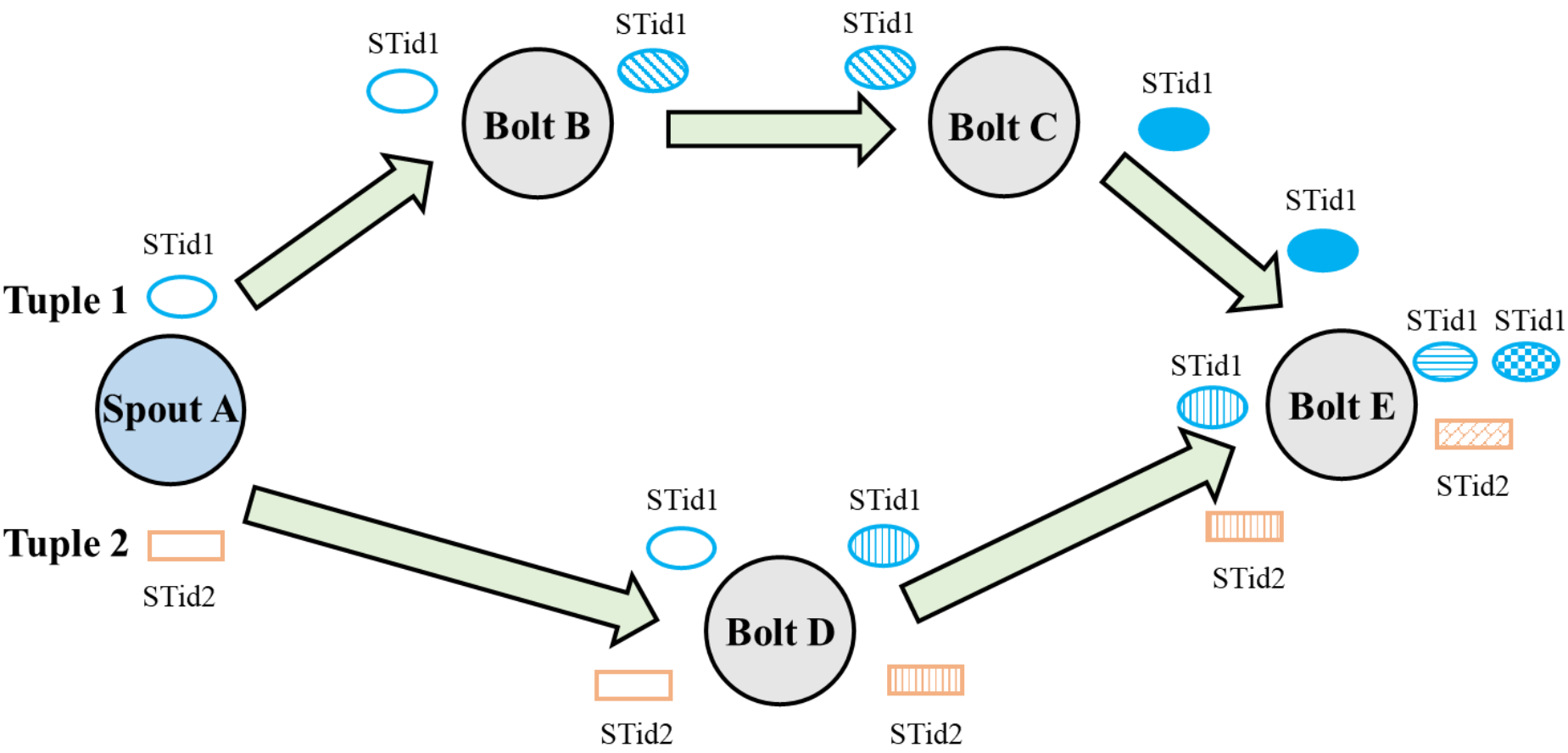
大纲

- 设计思想
- 体系架构
- 工作原理
- 容错机制
  - 容错语义
  - 元组树
  - ACK机制
  - 消息重放
- 编程示例

元组树

- Topology中的Spout会源源不断地发射出Tuple，每一条Tuple都会由后续的Bolt处理并演化为新的Tuple
- 元组树：Spout发出的Tuple及其衍生出来的Tuple抽象为一棵树
  - Spout中每一条元组都对应一棵元组树
- Spout-Tuple-id(STid)：Spout中发射Tuple时用户可以为其指定标识
  - 该STid伴随着元组在处理过程中的演化

元组树


STid1
STid1
STid1
STid1
Bolt B
Bolt C
STid1
STid1
Tuple 1
STid1
STid1
Spout A
Bolt E
STid1
STid2
Tuple 2
STid2
STid1
STid1
Bolt D
STid2
STid2
STid2

大纲

- 设计思想
- 体系架构
- 工作原理
- 容错机制
  - 容错语义
  - 元组树
  - ACK机制
  - 消息重放
- 编程示例

Mid vs. STid

- 元组树中的Tuple传输物理上表现为组件的Task之间的消息传输，该消息有一个64位标识，称为Mid
  - Mid是系统定义的消息传递标识，每条消息的Mid都不同
  - STid是用户定义的标识，同一元组树的STid都相同

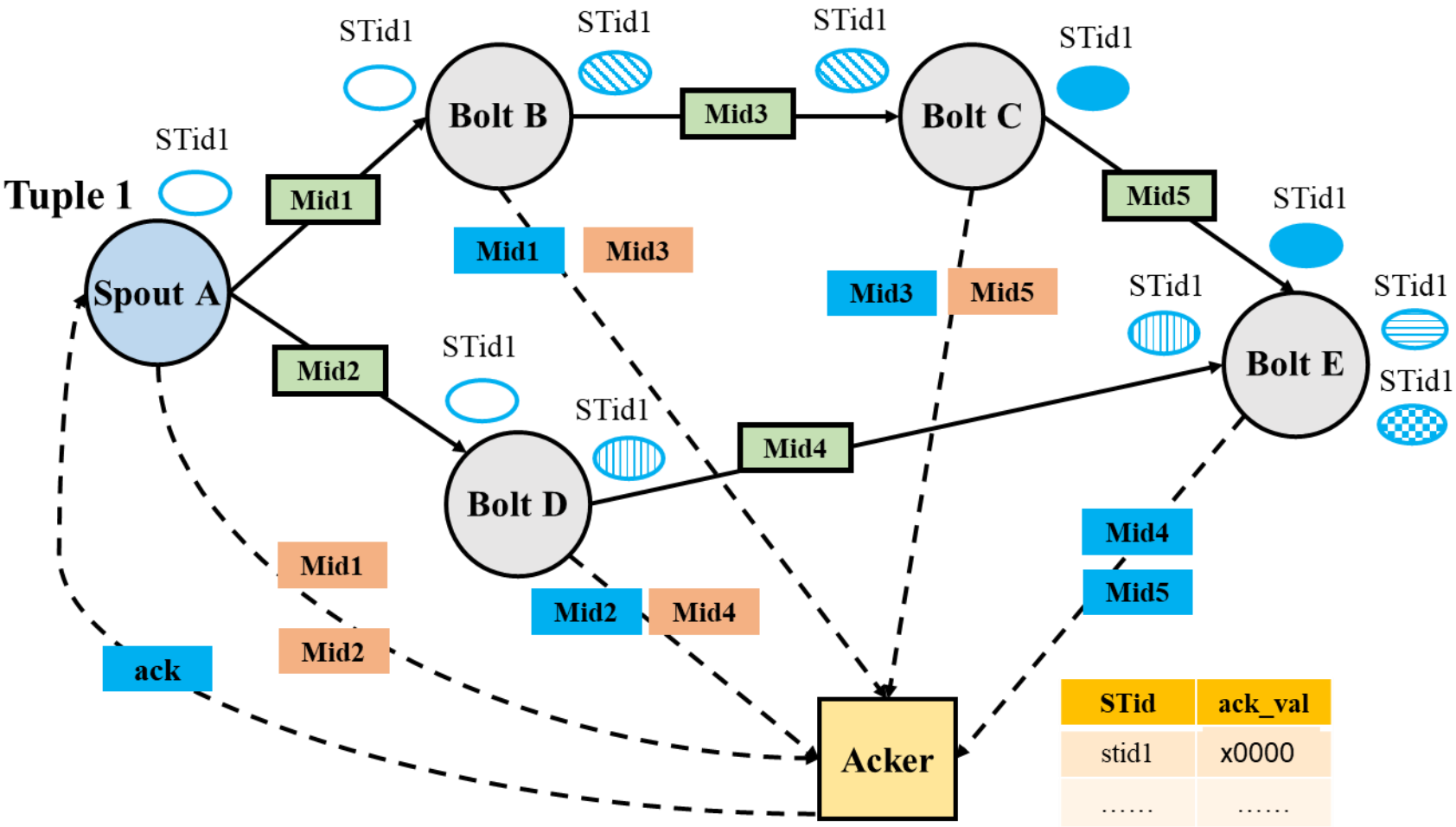
Ack机制

- Storm里面有一类特殊的Task作为Acker，负责跟踪Spout发出的元组及其元组树

- Ack机制
  - 上游组件的Task发射消息的同时，会向Acker报告Mid及STid
  - 当下游组件的Task接收到消息时向Acker报告Mid及STid

Ack机制

- 同一个元组树（STid1）



Acker数据结构

- 元组树中的元组非常多，并且往往并行地向Acker报告Mid和STid
- Acker端需要维护类似于<STid, list<Mid>>的映射表
  - 维护list对于Acker来说内存开销太大

Acker数据结构

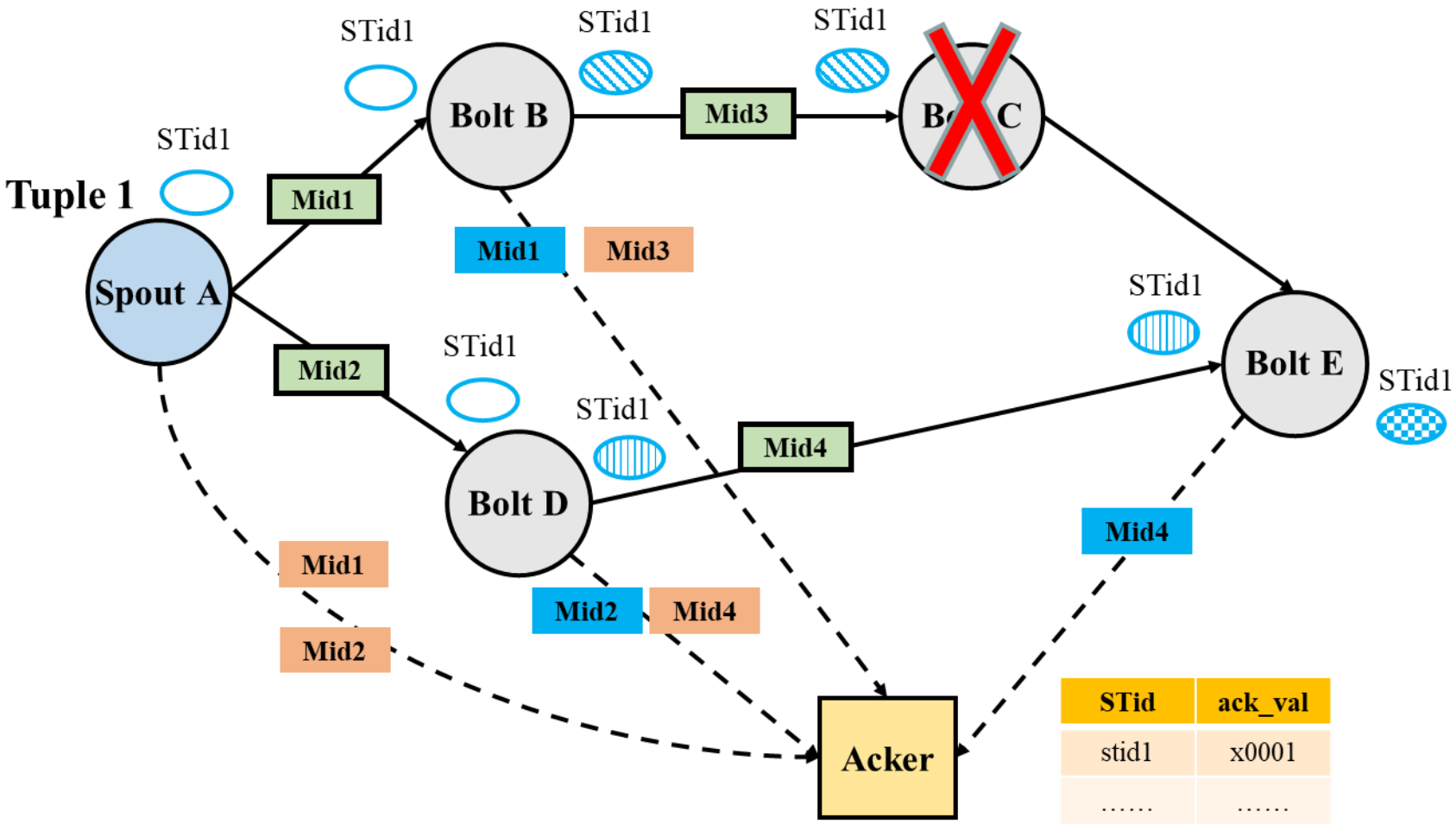
- <STid, ack_val>映射表
  - 收到Spout发来消息时将相应STid的ack_val初始化为0
  - 无论Acker接收到上游组件还是下游组件报告的消息，均将其中的Mid与映射表中相应STid的ack_val进行异或（XOR）操作
  - 如果Acker在设定的时间范围内收到处于拓扑最末端的Bolt报告并且ack_val为0，那么Acker会告诉相应的Spout：STid对应的元组树已经成功地处理完

STid	ack_val
stid1	x0000
……	……

大纲

- 设计思想
- 体系架构
- 工作原理
- 容错机制
  - 容错语义
  - 元组树
  - ACK机制
  - 消息重放
- 编程示例

故障发生

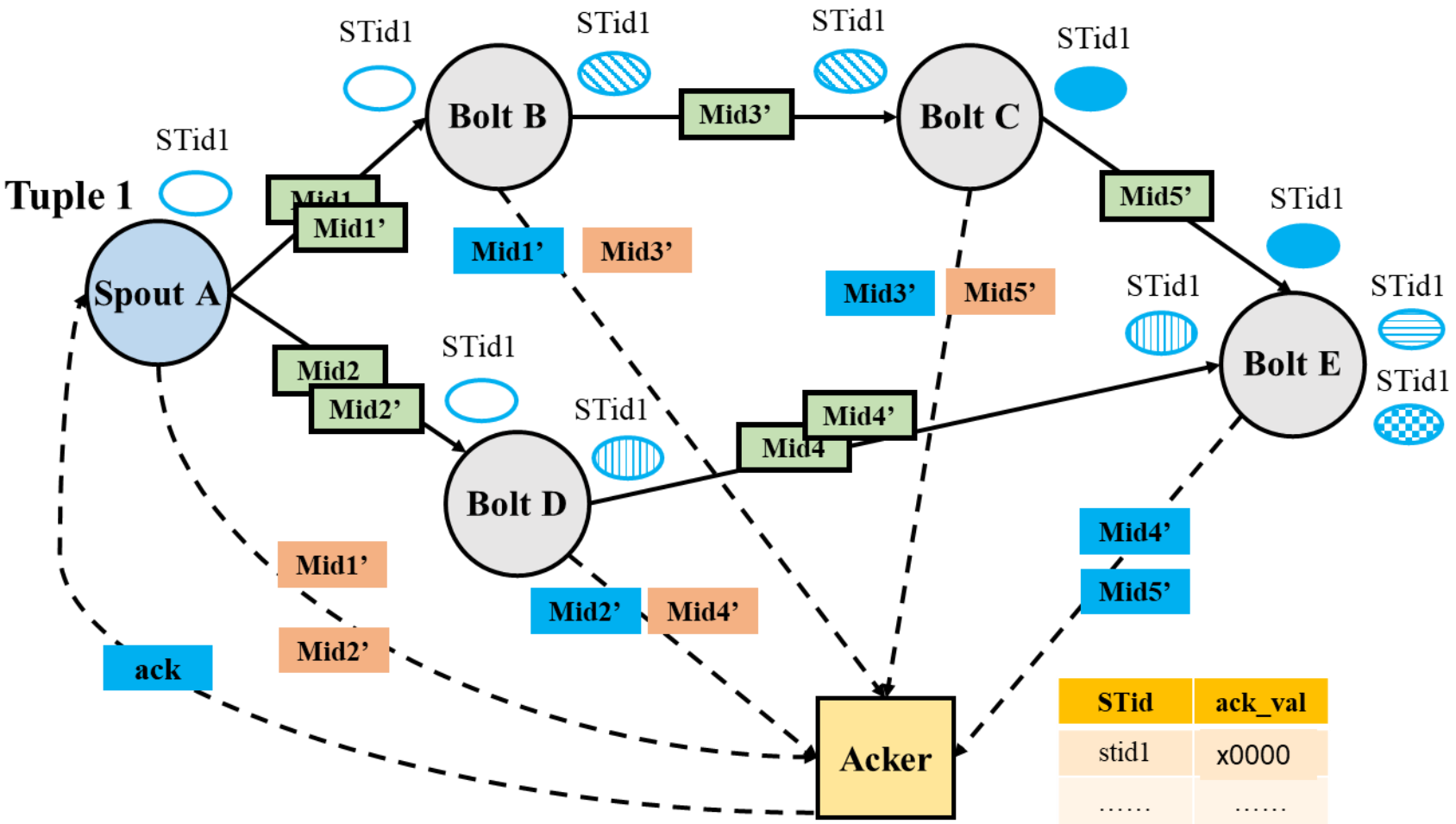

STid1
STid1
STid1
Bolt B
Mid3
C
STid1
Tuple 1
Mid1
Spout A
Mid1
Mid3
STid1
Bolt E
STid1
Mid2
STid1
STid1
Mid4
Bolt D
Mid4
Mid1
Mid2
Mid4
Mid2
Acker
STid
ack_val
stid1
x0001
……
……

故障处理

- Storm认为消息的传输发生了故障
  - Acker在设定的时间范围内，STid对应的ack_val不为0
- Spout重新发送以STid为标识的元组
  - 但这种消息重放机制可能会导致消息的重复计算，达到的是至少一次的容错语义级别

消息重放

- 重放可能导致元组的重复计算



大纲

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例

自定义CustomSpout类的框架


```
import org.apache.storm.*;
......

public class CustomSpout extends BaseRichSpout {
  private SpoutOutputCollector collector;
  ......
  /* 步骤1：初始化Spout */
  @Override
  public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
    this.collector = collector;
    ......
  }
  /* 步骤2：读取并发送元组 */
  @Override
  public void nextTuple() {
    // 读取元组
    ......
    // 发出一条元组，依次列出各字段的值，并且不启用ACK机制
    collector.emit(new Values(...));
    // 如需启用ACK机制,将第19行替换为第21行
    // collector.emit(new Values(...), STid);
  }

  /* 步骤3：声明输出元组的字段名称 */
  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    // 依次声明各字段的名称
    declarer.declare(new Fields(...));
  }

  /* 步骤4（可选）：描述元组ACK成功或失败后的处理逻辑 */
  @Override
  public void ack(Object STid) {
    // 当STid所对应的元组树中所有元组都得到成功处理时调用该方法
    // 清除STid对应的元组
    .....
  }

  @Override
  public void fail(Object STid) {
    // 当STid所对应的元组树中存在元组未得到成功处理时调用该方法
    // 重新发送STid所对应的元组
    ......
  }
}
```

自定义CustomBolt类的框架


```
1  import org.apache.storm.*;
2  .....
3  public class CustomBolt extends BaseBasicBolt {
4    /* 步骤1: 描述元组的处理逻辑 */
5    @Override
6    public void execute(Tuple tuple, BasicOutputCollector collector) {
7      ......
8      collector.emit(new Values(...));
9    }
10   /* 步骤2: 声明输出元组的字段名称 */
11   @Override
12   public void declareOutputFields(OutputFieldsDeclarer declarer) {
13     declarer.declare(new Fields(...));
14   }
15 }
```

构建拓扑的主类框架


```
import org.apache.storm.*
......
public class Topology {
  public static void main(String[] args) {
    /* 步骤1：构建拓扑 */
    TopologyBuilder builder = new TopologyBuilder();
    // 将拓扑中的Spout命名为SpoutName，并设置Spout的任务实例数为spout_numTask，运行这些任务
        的Executor线程数为spout_parallelism_hint
    builder.setSpout("SpoutName", new CustomSpout(),
        spout_parallelism_hint).setNumTasks(spout_numTask);
    // 定义拓扑中SpoutName和BoltName两个组件之间流数据分组策略为shuffleGrouping
    builder.setBolt("BoltName", new CustomBolt(),
        bolt_parallelism_hint).setNumTasks(bolt_numTask).shuffleGrouping("SpoutName");
    ......

    /* 步骤2：设置配置信息 */
    Config conf = new Config();
    conf.setDebug(false); // 设置是否开启调试模式
    conf.setNumWorkers(numWorker); // 设置worker的数量为numWorker
    conf.setNumAckers(numAcker); // 设置Acker的数量为numAcker

    /* 步骤3：指定程序运行的方式 */
    if(args[0].equals("cluster")) { // 在集群运行程序，拓扑的名称为TopologyName
      StormSubmitter.submitTopology("TopologyName", conf, builder.createTopology());
    } else if(args[0].equals("local")){ // 在本地IDE调试程序，拓扑的名称为TopologyName
      LocalCluster cluster = new LocalCluster();
      cluster.submitTopology("TopologyName", conf, builder.createTopology());
    }
    ......
    }
}
```

大纲

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例
  - 词频统计
  - 支持容错的词频统计
  - 简化的窗口操作
  - 异常检测

词频统计

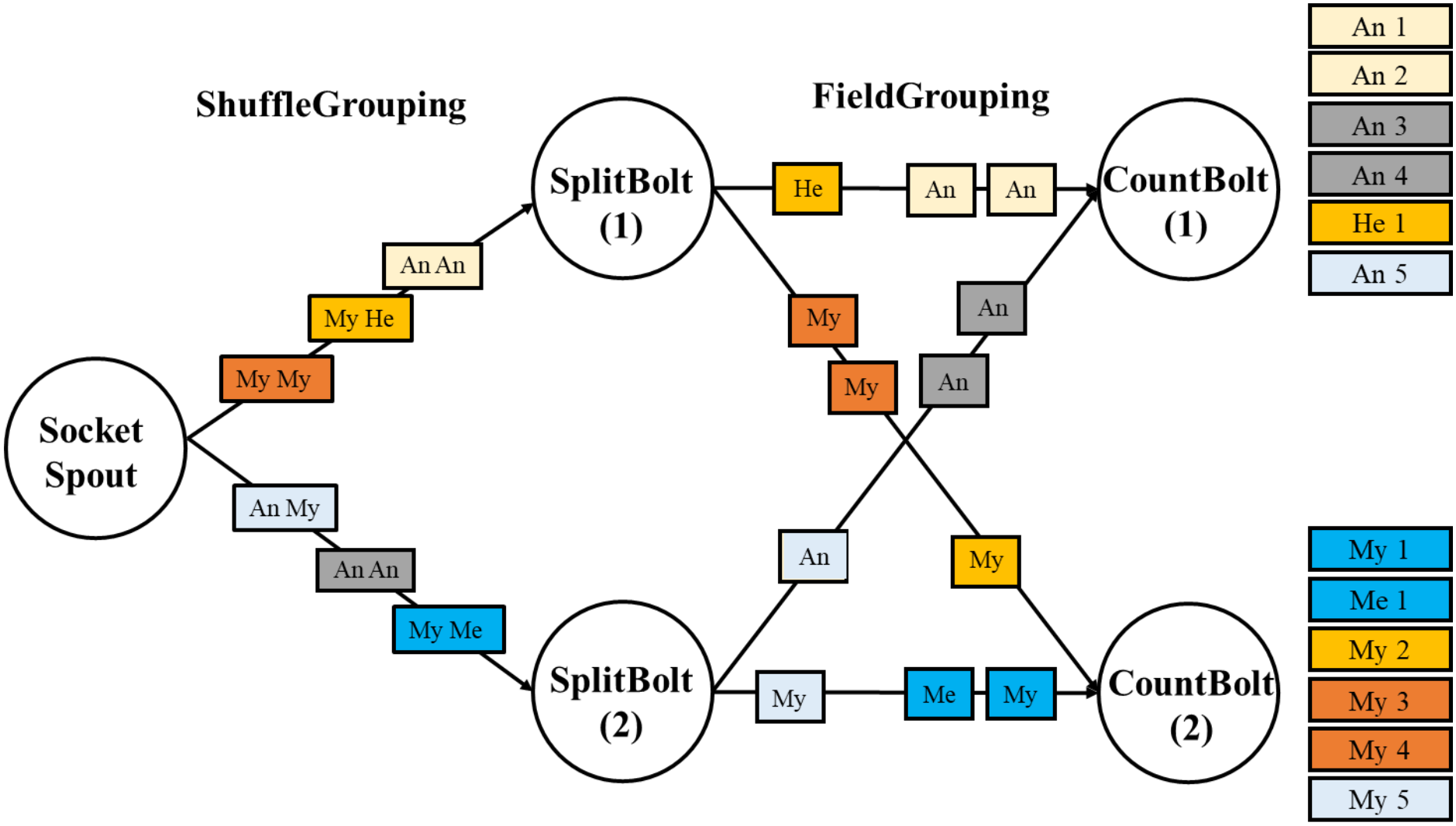
输入和输出示例

输入	实例1输出	实例2输出
An An	An 1 An 2	
My Me		My 1 Me 1
An An	An 3 An 4	
My He	He 1	My 2
My My		My 3 My 4
An My	An 5	My 5
…	…	…

解决方案

- 从Spout中发送Stream（每个英文句子为一个Tuple）
- 用于分割单词的Bolt将接收的句子分解为独立的单词，将单词作为Tuple的字段名发送出去
- 用于计数的Bolt接收表示单词的Tuple，并对其进行统计，输出每个单词以及单词出现过的次数

运行过程


ShuffleGrouping
FieldGrouping
Socket Spout
SplitBolt (1)
SplitBolt (2)
CountBolt (1)
CountBolt (2)
An An
My He
My My
An My
An An
My Me
He
An
An
My
My
An
An
My
An
My
My
Me
My
An 1
An 2
An 3
An 4
He 1
An 5
My 1
Me 1
My 2
My 3
My 4
My 5

编写WordCountTopology


```
1  import org.apache.storm.*;
2
3  public class WordCountTopology {
4
5    public static void main(String[] args) throws Exception {
6      ......
7      /* 步骤1：构建拓扑 */
8      TopologyBuilder builder = new TopologyBuilder();
9      // 设置Spout的名称为"SPOUT"，executor数量为1，任务数量为1
10     builder.setSpout("SPOUT", new SocketSpout(args[1], args[2]), 1).setNumTasks(1);
11     // 设置Bolt的名称为"SPLIT"，executor数量为2，任务数量为2，与"SPOUT"之间的流分组策略为随
           机分组
12     builder.setBolt("SPLIT", new SplitBolt(),
           2).setNumTasks(2).shuffleGrouping("SPOUT");
13     // 设置Bolt取名"COUNT"，executor数量为2，任务数量为2，订阅策略为fieldsGrouping
14     builder.setBolt("COUNT", new CountBolt(), 2).fieldsGrouping("SPLIT", new
           Fields("word"));
15
16     /* 步骤2：设置配置信息 */
17     Config conf = new Config();
18     conf.setDebug(false); // 关闭调试模式
19     conf.setNumWorkers(2); // 设置Worker数量为2
20     conf.setNumAckers(0); // 设置Acker数量为0
21
22     /* 步骤3：指定程序运行的方式 */
23     if (args[0].equals("cluster")) { // 在集群运行程序，拓扑的名称为WORDCOUNT
24       StormSubmitter.submitTopology("WORDCOUNT", conf, builder.createTopology());
25     } else if (args[0].equals("local")) { // 在本地IDE调试程序，拓扑的名称为WORDCOUNT
26       LocalCluster cluster = new LocalCluster();
27       cluster.submitTopology("WORDCOUNT", conf, builder.createTopology());
28     }
29     ......
30   }
31 }
```

构建拓扑

设置配置信息

指定程序运行方式

编写SocketSpout


```
public class SocketSpout extends BaseRichSpout {
  SpoutOutputCollector collector;
  String ip;
  int port;
  BufferedReader br;
  Socket socket;
  ......

  public SocketSpoutWithAck(String ip, String port) {
    this.ip = ip;
    this.port = Integer.valueOf(port);
  }

  /* 步骤1：初始化Spout */
  @Override
  public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
    this.collector = collector;
    ......
    socket = new Socket(ip, port);
    br = new BufferedReader(new InputStreamReader(socket.getInputStream()));
    ......
  }

  /* 步骤2：读取并发送元组 */
  @Override
  public void nextTuple() {
    ......
    String tuple;
    if ((tuple = br.readLine()) != null) { // 读取元组
      collector.emit(new Values(tuple)); // 发送元组
    }
    ......
  }

  /* 步骤3：声明输出元组的字段名称 */
  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    // 该输出元组仅有一个字段sentence
    declarer.declare(new Fields("sentence"));
  }
}
```

声明输出元组的字段名称

编写SplitBolt


```java
import org.apache.storm.*;
import java.util.StringTokenizer;
......

public class SplitBolt extends BaseBasicBolt {
  /* 步骤1：描述元组的处理逻辑 */
  @Override
  public void execute(Tuple tuple, BasicOutputCollector collector) {
    String sentence = tuple.getStringByField("sentence");
    StringTokenizer iter = new StringTokenizer(sentence);
    while (iter.hasMoreElements()) {
      collector.emit(new Values(iter.nextToken()));
    }
  }

  /* 步骤2：声明输出元组的字段名称 */
  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    // 该元组仅有一个字段
    declarer.declare(new Fields("word"));
  }
}
```

元组的处理逻辑

声明输出元组的字段名称

编写CountBolt


```java
import org.apache.storm.*;
import java.util.*;

public class CountBolt extends BaseBasicBolt {
  // 保存单词的频数
  Map<String, Integer> counts = new HashMap<String, Integer>();

  /* 步骤1：描述元组的处理逻辑 */
  @Override
  public void execute(Tuple tuple, BasicOutputCollector collector) {
    // 从接收到的元组中按字段提取单词
    String word = tuple.getStringByField("word");
    // 获取该单词对应的频数
    Integer count = counts.get(word);
    ......
    // 计数增加，并将单词和对应的频数加入map中
    count++;
    counts.put(word, count);
    // 输出结果，也可采用写入文件等其它方式
    System.out.println(word + " " + count);
  }

  /* 步骤2：声明输出元组的字段名称 */
  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    // 为空
```

词频统计

大纲

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例
  - 词频统计
  - 支持容错的词频统计
  - 简化的窗口操作
  - 异常检测

支持容错的词频统计

- 实现思路
  - Storm通过ACK机制提供至少一次的容错语义
  - 开启ACK机制需要对Spout和拓扑的实现进行修改，而Bolt的实现无需修改

- 程序不同之处
  - 增加用于缓存元组的Map对象
  - 为每一条元组绑定STid
  - 增加元组ACK成功或失败后的处理逻辑

编写SocketSpoutWithAck


```
public class SocketSpoutWithAck extends BaseRichSpout {
  SpoutOutputCollector collector;
  String ip;
  int port;
  BufferedReader br;
  Socket socket;
  ......
  // 该Map的键为STid，值为源元组的值
  private Map<String, String> waitAck;

  public SocketSpoutWithAck(String ip, String port) {
    this.ip = ip;
    this.port = Integer.valueOf(port);
  }

  /* 步骤1：  初始化Spout */
  @Override
  public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
    this.collector = collector;
    waitAck = new HashMap<>();
    ......
    socket = new Socket(ip, port);
    br = new BufferedReader(new InputStreamReader(socket.getInputStream()));
    ......
  }

  /* 步骤2：接收网络元组，将元组绑定一个STid后发送到下游Bolt */
  @Override
  public void nextTuple() {
    ......
    String tuple;
    String STid = UUID.randomUUID().toString();
    if ((tuple = br.readLine()) != null) {
      waitAck.put(STid, tuple);
      collector.emit(new Values(tuple), STid);
    }
    ......
  }
```

缓存元组，并为每一个元组绑定STid

编写SocketSpoutWithAck


```
46    /* 步骤3：声明输出元组的字段名称 */
47    @Override
48    public void declareOutputFields(OutputFieldsDeclarer declarer) {
49      declarer.declare(new Fields("sentence"));
50    }
51
52    /* 步骤4：描述元组ACK成功或失败后的处理逻辑 */
53    @Override
54    public void ack(Object STid) {
55      // 当STid所对应的元组树中所有元组都得到成功处理时调用该方法
56      // 根据STid删除waitAck中对应的元组
57      waitAck.remove(STid);
58    }
59
60    @Override
61    public void fail(Object STid) {
62      // 当STid所对应的元组树中所有元组未得到成功处理时调用该方法
63      // 重新发送waitAck中STid对应的元组
64      collector.emit(new Values(waitAck.get(STid)), STid);
65    }
66  }
```

成功，根据STid删除waitAck中对应的元组

失败，重新发送waitAck中Stid对应的元组

编写WordCountWithAckTopology


```
import org.apache.storm.*;

public class WordCountWithAckTopology {

  public static void main(String[] args) throws Exception {
    ......
    /* 步骤1：构建拓扑 */
    TopologyBuilder builder = new TopologyBuilder();
    builder.setSpout("SPOUT", new SocketSpoutWithAck(args[1], args[2]), 1);
    builder.setBolt("SPLIT", new SplitBolt(),
        2).setNumTasks(2).shuffleGrouping("SPOUT");
    builder.setBolt("COUNT", new CountBolt(), 2).fieldsGrouping("SPLIT", new
        Fields("word"));

    /* 步骤2：设置配置信息 */
    Config conf = new Config();
    conf.setDebug(false);
    conf.setNumWorkers(2);
    conf.setNumAckers(2); // 设置Acker的数量为2

    /* 步骤3：指定运行程序的方式 */
    if (args[0].equals("cluster")) { // 在集群运行程序，拓扑的名称为WORDCOUNTwithack
      StormSubmitter.submitTopology("WORDCOUNTwithack", conf, builder.createTopology());
    } else if (args[0].equals("local")) {
      // 在本地IDE调试程序，拓扑的名称为WORDCOUNTwithack
      LocalCluster cluster = new LocalCluster();
      cluster.submitTopology("WORDCOUNTwithack", conf, builder.createTopology());
    }
    ......
  }
}
```

设置Acker数量大于0

大纲

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例
  - 词频统计
  - 支持容错的词频统计
  - 简化的窗口操作
  - 异常检测

简化的窗口操作

- 基于滑动窗口的词频统计
  - 在收到M条记录时才对最近收到的N条记录执行操作并输出结果，这种操作被称为窗口操作
  - 其中M和N分别表示窗口间隔和窗口大小
- 输入和输出示例

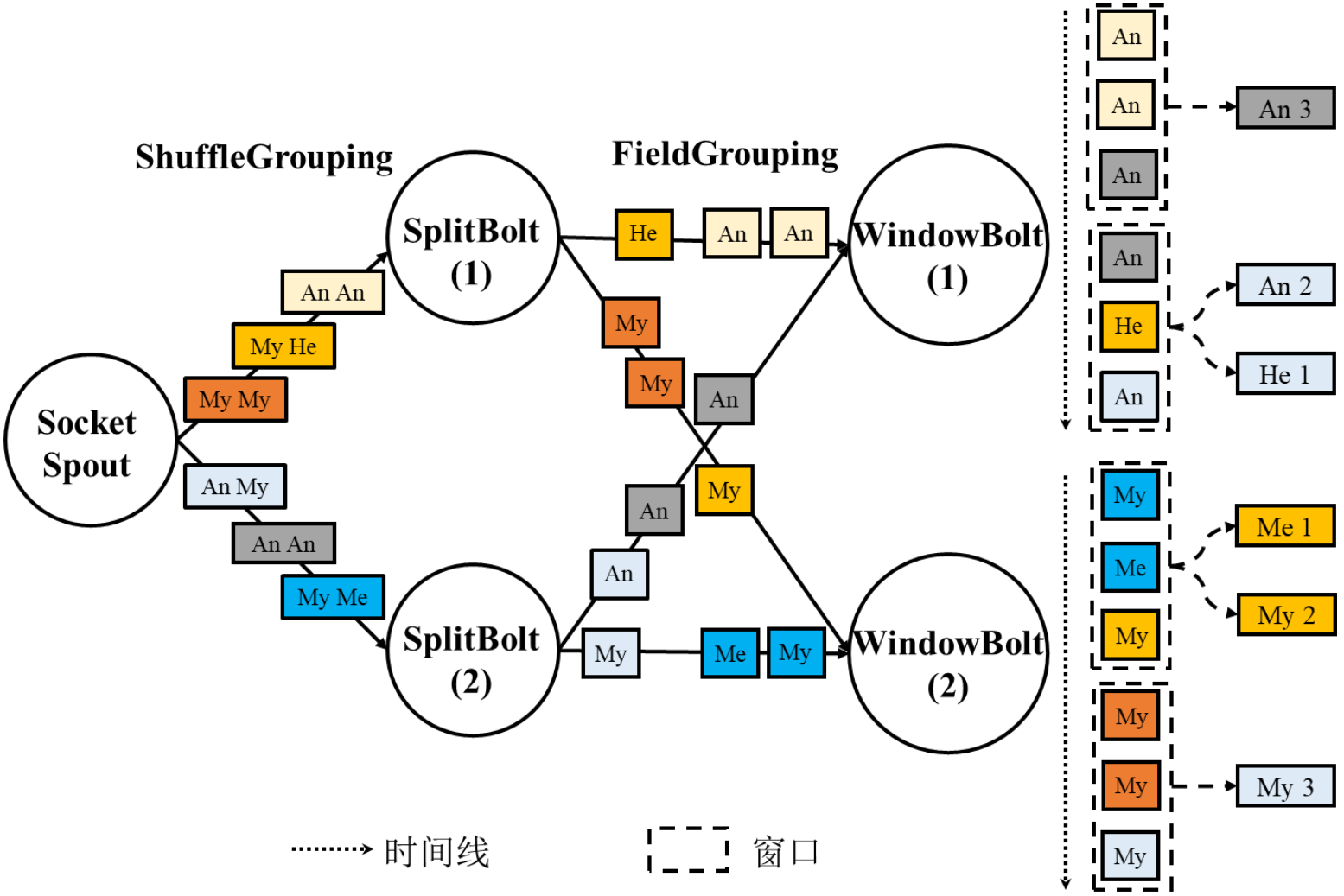
输入	实例1输出	实例2输出
An An		
My Me		
An An	An 3	
My He		Me 1 My 2
My My		
An My	An 2 He 1	My 3
…	…	…

解决方案

- 窗口操作仅影响计数过程，并不影响分词过程

- 在计数过程中，负责计数的Bolt组件需要维护窗口的内容，即缓存最近到达的M个单词元组
  - 每接收到M个单词元组，负责计数的Bolt组件就触发一次针对窗口内容的词频统计

运行过程


ShuffleGrouping
FieldGrouping
Socket Spout
SplitBolt (1)
SplitBolt (2)
WindowBolt (1)
WindowBolt (2)
An An
My He
My My
An My
An An
My Me
He
An
An
My
My
An
An
My
My
Me
My
An
An
An
My
Me
My
An 3
An 2
He 1
Me 1
My 2
My 3
时间线
窗口

WindowBolt


```java
public class WindowBolt extends BaseBasicBolt {

  // 窗口内的元组
  private final List<String> window = new ArrayList<>();
  // 窗口的大小和间隔
  private static final int LENGTH_AND_INTERVAL = 3;

  /* 步骤1：描述元组的处理逻辑*/
  @Override
  public void execute(Tuple tuple, BasicOutputCollector collector) {
    // 缓存接收到的单词元组
    String word = tuple.getStringByField("word");
    window.add(word);
    // 接收的单词元组数量等于窗口间隔时，触发计数操作
    if (window.size() == LENGTH_AND_INTERVAL) {
      // 计数
      Map<String, Integer> wordCounts = new HashMap<>();
      for (String wordInWindow : window) {
        if (wordCounts.containsKey(wordInWindow)) {
          wordCounts.put(wordInWindow, wordCounts.get(wordInWindow) + 1);
        } else {
          wordCounts.put(wordInWindow, 1);
        }
      }

      // 输出计数结果
      for (Entry<String, Integer> entry : wordCounts.entrySet()) {
        System.out.println(entry.getKey() + " " + entry.getValue());
      }

      // 清除窗口内容
      window.clear();
    }
  }

  /* 步骤2：声明输出元组的字段名称*/
  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    // 为空
  }
}
```

缓存元组

接收的单词元组数量等于窗口间隔时，触发计数操作

清空窗口

大纲

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例
  - 词频统计
  - 支持容错的词频统计
  - 简化的窗口操作
  - 异常检测

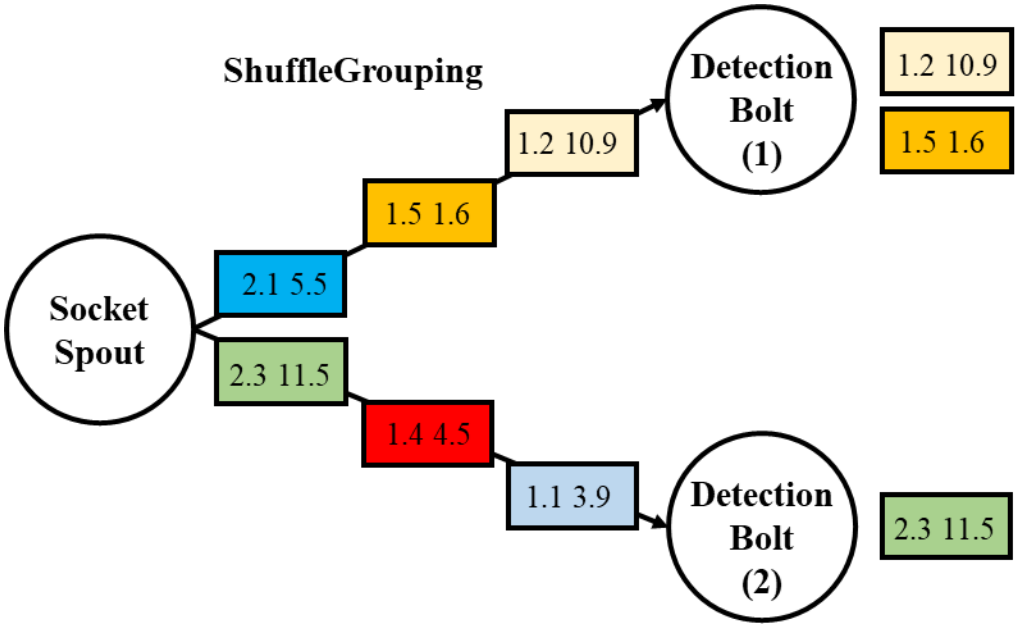
异常检测

- 线性模型y = wx +b,对输入向量（x`,y`），若满足 $|wx' + b - y'| > \epsilon$，认为（x`,y`）是异常值并予以输出
- 输入和输出示例

输入	输出
1.2 10.9 1.1 3.9 1.5 1.6 1.4 4.5 2.1 5.5 2.3 11.5 …	1.2 10.9 1.5 1.6 2.3 11.5 …

解决方案

- Spout负责接收表示向量的数据
- Bolt负责检测向量并输出异常值



编写SocketSpout


```
7  public class SocketSpout extends BaseRichSpout {
8
9    SpoutOutputCollector collector;
10   String ip;
11   int port;
12   BufferedReader br;
13   Socket socket;
14   ......
15
16   SocketSpout(String ip, String port) {
17     this.ip = ip;
18     this.port = Integer.valueOf(port);
19   }
20
21   /* 步骤1:  初始化Spout */
22   @Override
23   public void open(Map map, TopologyContext topologyContext, SpoutOutputCollector
         collector) {
24     this.collector = collector;
25     ......
26    socket = new Socket(ip, port);
27    br = new BufferedReader(new InputStreamReader(socket.getInputStream()));
28    ......
29  }
30
31  /* 步骤2: 读取并发送元组 */
32  @Override
33  public void nextTuple() {
34    ......
35    String tuple;
36    if ((tuple = br.readLine()) != null) {
37      String[] vec = tuple.split(" ");
38      double x = Double.parseDouble(vec[0]);
39      double y = Double.parseDouble(vec[1]);
40      collector.emit(new Values(x, y)); // 输出元组包含两个字段
41    }
42    ......
43  }
44
45  /* 步骤3: 声明输出元组的字段名称 */
46  @Override
47  public void declareOutputFields(OutputFieldsDeclarer declarer) {
48    // 该元组有两个字段，分别为x和y
49    declarer.declare(new Fields("x", "y"));
50  }
51 }
```

读取并发送元组

编写DetectionBolt


```
4  public class DetectionBolt extends BaseBasicBolt {
5
6    double w, b, delta; // 线性模型参数
7
8    public DetectionBolt(double w , double b, double delta) {
9      this.w = w;
10     this.b = b;
11     this.delta = delta;
12   }
13
14   /* 步骤1：定义元组处理逻辑 */
15   @Override
16   public void execute(Tuple tuple, BasicOutputCollector basicOutputCollector) {
17     // 获取数据
18     double x = tuple.getDoubleByField("x");
19     double y = tuple.getDoubleByField("y");
20     // 判断数据是否异常
21     if (Math.abs(w * x + b - y) > delta) {
22       System.out.println(x + " " + y);
23     }
24   }
25
26   /* 步骤2：声明输出元组的字段名称 */
27   @Override
28   public void declareOutputFields(OutputFieldsDeclarer outputFieldsDeclarer) {
29     // 为空
30   }
```

检测数据是否异常

编写OutlierTopology


```
public class OutlierTopology {
  public static void main(String[] args) throws Exception {
    ......
    /* 步骤1：构建拓扑 */
    TopologyBuilder builder = new TopologyBuilder();
    builder.setSpout("SPOUT", new SocketSpout(args[1], args[2]), 1);
    builder.setBolt("DETECTION", new DetectionBolt(1.5, 2.5, 0.5),
        2).setNumTasks(2).shuffleGrouping("SPOUT");

    /* 步骤2：设置配置信息 */
    Config conf = new Config();
    conf.setDebug(false);
    conf.setNumWorkers(2);
    conf.setNumAckers(0);

    /* 步骤3：指定运行程序的方式 */
    if (args[0].equals("cluster")) { // 在集群运行程序，拓扑名称为OUTLIERTOPOLOGY
      StormSubmitter.submitTopology("OUTLIERTOPOLOGY", conf, builder.createTopology());
    } else if (args[0].equals("local")) {
      // 在本地IDE调试程序，拓扑的名称为OUTLIERTOPOLOGY
      LocalCluster cluster = new LocalCluster();
      cluster.submitTopology("OUTLIERTOPOLOGY", conf, builder.createTopology());
    }
    ......
  }
}
```

课后阅读

- 论文
  - Toshniwal, A., Donham, J., Bhagat, N., Mittal, S., Ryaboy, D., Taneja, S., … Fu, M. (2014). Storm@twitter. In SIGMOD Conference (pp. 147–156).

本章小结

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例