Developing Storm Applications

A trivial running example topology

This spout emits a stream of random integer numbers between 0 and 99. It emits one number every 100ms.

This bolt multiplies each element of the input stream by 2 and prints the result on the standard output.

(,, 74, 10, 5, ..) → (,, 148, 20, 10, ..)
Developing a Storm application

- You must implement
  - One class for each spout of your topology
    - However, in real applications, you typically use an existing spout (Kafka spout, Redis spout, etc)
  - One class for each bolt of your topology
  - One class, with the main method, to define and submit the topology

Implementing Spouts
Implementing Spouts

- For each spout you must specify
  - The format of the emitted tuples
    - The names of the fields
  - How tuples are generated

Implementing Spouts

- Spouts implement the BaseRichSpout abstract class
  - BaseRichSpout implements the following interfaces
    - Serializable, ISpout, IComponent, IRichSpout
  - The methods to be implemented are
    - public void open(Map conf, TopologyContext context, SpoutOutputCollector collector)
    - public void declareOutputFields(OutputFieldsDeclarer declarer)
    - public void nextTuple()
Implementing Spouts

- public void open(Map conf, TopologyContext context, SpoutOutputCollector collector)
  - It is called when a task for this component is initialized within a worker on the cluster
  - It provides the spout with the environment in which it executes
  - Parameters
    - conf
      - The Storm configuration for this spout
    - context
      - It can be used to get information about this task's place within the topology, including the task id and component id of this task
    - collector:
      - The collector is used to emit tuples from this spout
      - Tuples can be emitted at any time, including the open and close methods
      - The collector is thread-safe and should be saved as an instance variable of this spout object

Implementing Spouts

- public void declareOutputFields(OutputFieldsDeclarer declarer)
  - Declares the output schema for all the streams of this spout
    - An spout can emit more than one stream
  - Parameter
    - declarer
      - It is used to declare output stream ids, output fields, and whether or not each output stream is a direct stream
Implementing Spouts

- public void nextTuple()
  - It is used to emit the next tuple(s) of the stream(s) generated by this spout by calling the emit method on the output collector
  - When this method is called, Storm is requesting that the Spout emits tuples to the output collector
  - This method should be non-blocking
    - So if the Spout has no tuples to emit, this method should return

Running example Spout

```java
package ...
import ...

@ SuppressWarnings("serial")
public class EmitRandomIntSpout extends BaseRichSpout {
    private SpoutOutputCollector collector;
    private Random rand;

    @Override
    public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
        this.collector = collector;
        this.rand = new Random();
    }

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

@SuppressWarnings("serial")
public class EmitRandomIntSpout extends BaseRichSpout {
    private SpoutOutputCollector collector;
    private Random rand;

    @Override
    public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
        this.collector = collector;
        this.rand = new Random();
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("IntValue"));
    }
}
@Override
public void nextTuple() {
    Utils.sleep(100);
    collector.emit(new Values(rand.nextInt(100)));
}
}
Implementing Bolts

- For each bolt you must specify
  - How the input tuples are processed
  - The format of the emitted tuples
    - The final bolt of a path of the topology does not emit a new stream of tuples
    - For the final bolts the tuple format is not specified
Implementing Bolts

- Bolts implement the BaseRichBolt abstract class
  - BaseRichBolt implements the following interfaces
    - Serializable, IBolt, IComponent, IRichBolt
- The methods to be implemented are
  - public void prepare(Map conf, TopologyContext context, OutputCollector collector)
  - public void declareOutputFields(OutputFieldsDeclarer declarer)
  - public void execute(Tuple tuple)

Implementing Bolts

- public void prepare(Map conf, TopologyContext context, OutputCollector collector)
  - It is called when a task for this component is initialized within a worker on the cluster
  - It provides the bolt with the environment in which it executes
  - Parameters
    - conf
      - The Storm configuration for this spout
    - context
      - It can be used to get information about this task’s place within the topology, including the task id and component id of this task
    - collector:
      - The collector is used to emittuples from this bolt
      - Tuples can be emitted at any time, including the prepare and cleanup methods
      - The collector is thread-safe and should be saved as an instance variable of this bolt object
Implementing Bolts

- public void declareOutputFields(OutputFieldsDeclarer declarer)
  - Declares the output schema for all the streams of this bolt
    - A bolt can emit zero or many streams
  - Parameter
    - declarer
      - It is used to declare output stream ids, output fields, and whether or not each output stream is a direct stream

Implementing Bolts

- public void execute(Tuple tuple)
  - It is used to process a single tuple of input
  - The Tuple object contains metadata on it about which component/stream/task it came from
  - The values of the Tuple can be accessed using the getValue* methods
  - The Bolt does not have to process the Tuple immediately
    - It is perfectly fine to hang onto a tuple and process it later (for instance, to do an aggregation or a join)
package ...
import ...

@SuppressWarnings("serial")
public class MultiplyBy2Bolt extends BaseRichBolt {
    private OutputCollector collector;
    
    @Override
    public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
        this.collector = collector;
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
    }
}
Running example Bolt

```java
package ...
import ...

@SuppressWarnings("serial")
public class MultiplyBy2Bolt extends BaseRichBolt {
    private OutputCollector collector;

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

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
    }
}
```

The schema of the output tuples is not defined because this bolt does not emit a new stream

```java
@Override
public void execute(Tuple tuple) {
    // Print the computed value on the standard output
    System.out.println(tuple.getIntegerByField("intValue") * 2);
}
```
Running example Bolt

```java
@override
public void execute(Tuple tuple) {
    // Print the computed value on the standard output
    System.out.println(tuple.getIntegerByField("intValue") * 2);
}
```
Implementing Topologies

- For each topology you must specify
  - Which spouts and bolts are part of the topology
  - How spouts and bolts are connected
  - Which stream grouping is used for each stream
    - It depends on the pair (emitter spout/bolt, consumer bolt) and the performed stream transformation/processing
  - You must specify the initial parallelism of the topology
    - Pay attention: The maximum number of tasks cannot be changed at runtime

Implementing Topologies

- Topologies are created and configured by means of the TopologyBuilder class
- The main methods to be used are
  - public SpoutDeclarer setSpout(String id, IRichSpout spout)
  - public BoltDeclarer setBolt(String id, IRichBolt bolt)
  - public StormTopology createTopology()
Implementing Topologies

- public SpoutDeclarer setSpout(String id, IRichSpout spout, Number parallelism_hint)
  - It is used to add a spout to the topology
  - Parameters
    - id
      - The id of this component
      - Usually it is the “name” of the spout
      - This id is referenced by other components that want to consume this spout’s outputs
    - spout
      - An instance of the class implementing this spout
    - parallelism_hint
      - Number of executors that should be assigned to execute this spout

Implementing Topologies

- public BoltDeclarer setBolt(String id, IRichBolt bolt, Number parallelism_hint)
  - It is used to add a bolt to the topology
  - Parameters
    - id
      - The id of this component
      - Usually it is the “name” of the bolt
      - This id is referenced by other components that want to consume this bolt’s outputs
    - bolt
      - An instance of the class implementing this bolt
    - parallelism_hint
      - Number of executors that should be assigned to execute this bolt
Implementing Topologies

- Use the object returned by setBolt to declare the inputs of the bolt
  - Specify the input streams and the stream grouping technique
- Use one of the following methods of the BoltDeclarer class
  - shuffleGrouping(..), localOrShuffleGrouping(..), fieldsGrouping(..), partialKeyGrouping(..), allGrouping(..), globalGrouping(..), noneGrouping(..), directGrouping(..), customGrouping(..)

Implementing Topologies

- public StormTopology createTopology()
  - It is used to create an instance of the defined topology
Implementing Topologies

- public static void submitTopology(String name, Map stormConf, StormTopology topology) of StormSubmitter is used to submit the topology
  - Submits a topology to run on the cluster
    - A topology runs forever or until explicitly killed.
  - Parameters
    - name
      - name of the topology
    - stormConf
      - the topology-specific configuration
    - Topology
      - An instance of the topology to execute

Running example Topology

```
package ...
import ...

public class MultiplyBy2Topology {
    public static void main(String[] args) throws Exception {
        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("streamIntegers", new EmitRandomIntSpout(), 1);
        builder.setBolt("multiply", new MultiplyBy2Bolt(), 2)
                     .shuffleGrouping("streamIntegers");

        Config conf = new Config();
        conf.setDebug(false);
        conf.setNumWorkers(3);
    }
```
package ...
import ...

public class MultiplyBy2Topology {

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

        TopologyBuilder builder = new TopologyBuilder();

        builder.setSpout("streamIntegers", new EmitRandomIntSpout(), 1);
        builder.setBolt("multiply", new MultiplyBy2Bolt(), 2)
            .shuffleGrouping("streamIntegers");

        Config conf = new Config();
        conf.setDebug(false);
        conf.setNumWorkers(3);
    }
}
Running example Topology

package ...
import ...

public class MultiplyBy2Topology {
    public static void main(String[] args) throws Exception {
        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("streamIntegers", new EmitRandomIntSpout(), 1);
        builder.setBolt("multiply", new MultiplyBy2Bolt(), 2)
            .shuffleGrouping("streamIntegers");

        Config conf = new Config();
        conf.setDebug(false);
        conf.setNumWorkers(3);
    }
}

Running example Topology

package ...
import ...

public class MultiplyBy2Topology {
    public static void main(String[] args) throws Exception {
        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("streamIntegers", new EmitRandomIntSpout(), 1);
        builder.setBolt("multiply", new MultiplyBy2Bolt(), 2)
            .shuffleGrouping("streamIntegers");

        Config conf = new Config();
        conf.setDebug(false);
        conf.setNumWorkers(3);
    }
}
Running example Topology

```java
package ...
import ...

public class MultiplyBy2Topology {
    public static void main(String[] args) throws Exception {
        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("streamIntegers", new EmitRandomIntSpout(), 1);
        builder.setBolt("multiply", new MultiplyBy2Bolt(), 2)
            .shuffleGrouping("streamIntegers");

        Config conf = new Config();
        conf.setDebug(false);
        conf.setNumWorkers(3); Specifying the number of workers used to deploy the topology

        if (args != null && args.length > 0) {
            String topologyName = args[0];

            StormSubmitter.submitTopology(topologyName, conf,
                builder.createTopology());
        } else {
            System.out.println("storm jar example-1.0.0.jar
                                 storm_example.multiplyby2 <topology name>");
        }
    }
}
```
Running example Topology

```java
if (args != null && args.length > 0) {
    String topologyName = args[0];
    StormSubmitter.submitTopology(topologyName, conf,
    builder.createTopology());
} else {
    System.out.println("storm jar example-4.0.0.jar
    storm_example.multiply two topology
    topologyName ");
    }
```
Reliable vs unreliable spouts

- Spouts can be reliable or unreliable
- A reliable spout is capable of replaying a tuple if it failed to be processed by Storm
- An unreliable spout forgets about the tuple as soon as it is emitted
  - It does not reemit the tuple if it processing fails
- Unreliable spouts are faster
  - Use them if you need high-performance and you can “lose” some tuples

Reliable vs unreliable spouts

- Each reliable spout maintains a queue with the emitted tuples
- The ack() and fail() methods of BaseRichSpout are used to update the content of the queue
  - ack is used to remove from the queue a tuple that has been fully processed
  - fail is usually used to resend a tuple that has not been properly processed
Ack and Fail methods

- **BaseRichSpout** has also the following methods
  - `void ack(Object msgId)`
    - This method of the spout is invoked when the tuple emitted by this spout with the msgId identifier has been fully processed
  - `void fail(Object msgId)`
    - This method of the spout is invoked when the tuple emitted by this spout with the msgId identifier has failed to be fully processed

---

Reliable implementation of the running example topology: Spout

```java
package ...
import ...*/
/**
 * Emita random integer every 100 ms.
 *
 * @SuppressWarnings("serial")
 * public class EmitRandomIntSpoutReliable extends BaseRichSpout {
 *     private SpoutOutputCollector collector;
 *     private Random rand;
 *     private Integer msgId;
 *     HashMap<Integer, Integer> sentTuples;
 *     
 *     @Override
 *     public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
 *         this.collector = collector;
 *         this.msgId = 0;
 *         this.rand = new Random();
 *         this.sentTuples = new HashMap<Integer, Integer>();
 *     }
 *     
 *     @Override
 *     public void declareOutputFields(OutputFieldsDeclarer declare) {
 *         declare.declare(new Fields("intValue"));
 *     }
 * }
```
Reliable implementation of the running example topology: Spout

```java
@Override
public void nextTuple() {
    Util.sleep(100);
    Integer val = rand.nextInt(100);
    msgId++;
    collector.emit(new Values(val), msgId);
    // Store the sent tuple until the ack is received
    sentTuples.put(msgId, val);
}

@Override
public void ack(Object id) {
    sentTuples.remove(msgId);
}

@Override
public void fail(Object id) {
    // Send again the number associated with this msgId
    Integer val = sentTuples.get(id);
    collector.emit(new Values(val), msgId);
    // Store the sent tuple until the ack is received
    sentTuples.put(msgId, val);
}
```

Remove the tuple when the ack is received

Store the sent tuples
### Reliable implementation of the running example topology: Spout

```java
@Override
public void nextTuple() {
    Util.sleep(100);
    int val = rand.nextInt(100);
    msgId++;
    collector.emit(new Values(val, msgId));
    // Store the sent tuple until the ack is received
    sentTuples.put(msgId, val);
}
```
Send again the tuple if a fail is received.

```java
@Override
public void ack(Object id) {
    sentTuples.remove(msgId);
}
```

```java
@Override
public void fail(Object id) {
    // Send again the number associated with this msgId
    int val = sentTuples.get(id);
    collector.emit(new Values(val, msgId));
}
```

---

### Reliable implementation of the running example topology: Bolt

```java
package ...
import ...

@SuppressWarning("serial")
public class MultiplyBy2BoltReliable extends BaseRichBolt {
    private OutputCollector collector;

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

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
    }

    @Override
    public void execute(Tuple tuple) {
        // Print the computed value on the standard output
        System.out.println(tuple.getFieldAs(IntegerByField.class, "intValue") * 2);
        collector.ack(tuple);
    }
}
```
Ack the processing of the tuple.
package ...
import ...

public class MultiplyBy2TopologyReliable {
    public static void main(String[] args) throws Exception {
        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("streamIntegers", new EmitRandomIntSpoutReliable(), 1);
        builder.setBolt("multiply", new MultiplyBy2BoltReliable(), 2, shuffleGrouping("streamIntegers"));
        Config conf = new Config();
        conf.setDebug(false);
        conf.setNumWorkers(3);
        if (args != null && args.length > 0) {
            String topologyName = args[0];
            StormSubmitter.submitTopology(topologyName, conf, builder.createTopology());
        } else {
            System.out.println("stormjar example-s.o.o.jar storm_example.multiplyby2Reliable <topologyname>");
        }
    }
}
Bolts emitting tuples: Example

- Run a topology with one spout and two bolts
- The spout emits random integer numbers
- The first bolt reads the stream emitted by the spout and multiplies each number by 2
  - It emits the output as a new stream
- The second bolt reads the stream emitted by the first bolt and sums 1 to each number
  - It prints the output on the standard output

This spout emits random integer numbers between 0 and 99.

\((\ldots, 1, 10, 5, \ldots) \rightarrow (\ldots, 2, 20, 10, \ldots) \rightarrow (\ldots, 3, 21, 11, \ldots)\)
Bolts emitting tuples: Example - Topology

```java
package ....
import .......

public class BoltEmitStreamTopology{

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

        TopologyBuilder builder = new TopologyBuilder();

        builder.setSpout("spout", new EmitRandomIntSpout(), 1);
        builder.setBolt("multiplyBy2", new MultiplyBy2Bolt(), 2).shuffleGrouping("spout");
        builder.setBolt("sum1", new Sum1Bolt(), 2).shuffleGrouping("multiplyBy2");

        Config conf = new Config();
        conf.setDebug(false);
        conf.setNumWorkers(3);

        sum1 processes the data emitted by multiplyBy2

    }

    if (args != null && args.length > 0) {
        String topologyName = args[0];

        StormSubmitter.submitTopology(topologyName, conf, builder.createTopology());
    } else {
        System.out.println("Storm jar target/example-1.0.0.jar storm_example.bolt_emitting_stream.BoltEmitStreamTopology <topologyName>");
    }
}
```
package ...
import......

@SuppressWarnings("serial")
public class EmitRandomIntSpout extends BaseRichSpout{
    private SpoutOutputCollector collector;
    private Random rand;
    private Integer msgId;
    HashMap<Integer, Integer> sentTuples;

    @Override
    public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
        this.collector = collector;
        this.msgId = 0;
        this.rand = new Random();
        this.sentTuples = new HashMap<Integer, Integer>();
    }
}

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

@Override
public void nextTuple() {
    Utils.sleep(100);
    Integer val = rand.nextInt(100);
    msgId++;
    collector.emit(new Values(val), msgId);
    // Store the sent tuple until the ack is received
    sentTuples.put(msgId, val);
}
Bolts emitting tuples: Example - 
Spout

```java
@Override
public void ack(Object id) {
    sentTuples.remove(msgId);
}

@Override
public void fail(Object id) {
    // Send again the number associated with this msgId
    Integer val = sentTuples.get(id);
    collector.emit(new Values(val), msgId);
}
```

Bolts emitting tuples: Example – 
x2Bolt

```java
package ....
import......

@SuppressWarnings("serial")
public class MultiplyBy2Bolt extends BaseRichBolt {

    private OutputCollector collector;

    @Override
    public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
        this.collector = collector;
    }
```
Bolts emitting tuples: Example – x2Bolt

@Override
public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("time1Value");
}

@Override
public void execute(Tupletuple) {
    // Print the computed value on the standard output
    // Multiply by 2 the value of the tuple and emit it on the output stream
    collector.emit(tuplet, new Values(tuplet.getIntegerByField("intValue")* 2));
    collector.ack(tuplet);
}

Declare the schema of the emitted tuples

Bolts emitting tuples: Example – x2Bolt

@Override
public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("time2Value");
}

@Override
public void execute(Tupletuple) {
    // Print the computed value on the standard output
    // Multiply by 2 the value of the tuple and emit it on the output stream
    collector.emit(tuplet, new Values(tuplet.getIntegerByField("intValue")* 2));
    collector.ack(tuplet);
}

Emit a new tuple on the output stream.
- The first parameter is the original tuple
- The second one is the new tuple

The first tuple is used to create a link between the original tuple and the generated ones for managing reliability
```java
package ....
import......

@SuppressWarnings("serial")
public class Sum1Bolt extends BaseRichBolt {

    private OutputCollector collector;

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

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
    }

    @Override
    public void execute(Tuple tuple) {
        // Print the computed value on the standard output
        // Sum 1 to the value of the tuple
        System.out.println(tuple.getInteger(0) + 1);
        collector.ack(tuple);
    }

    }
```
Spouts and Bolt: other methods

Spouts: other methods

- BaseRichSpout has also the following methods
  - void close()
    - Called when a spout is going to be shutdown
      - There is no guarantee that cleanup will be called
  - void activate()
    - Called when a spout has been activated out of a deactivated mode
  - void deactivate()
    - Called when a spout has been deactivated
Bolts: other methods

- BaseRichBolt has also the following methods
  - void cleanup()
    - It is called when a Bolt is going to be shutdown
    - There is no guarantee that cleanup will be called

Multiple input and output streams
Multiple input streams

- Each bolt can subscribe multiple input streams/the output of multiple components to
  - Implement join operations
  - Receive data and signals
  - ...
- For each stream, the most appropriate stream grouping technique is specified
- In the `nextTuple(..)` method a different operation is executed depending on the origin of the tuple (i.e., the input stream)

Multiple input streams

- A bolt can subscribe the streams of multiple components by means of a chain of calls to the stream grouping methods
  - One call for each subscribed component
- Example
  
  ```java
  builder.setBolt("merge",
      new ProcessMultipleStreamsBolt(), 2)
          .shuffleGrouping("firstSpout")
          .shuffleGrouping("secondSpout");
  ```
Multiple input streams: Example

- Run a topology with two spouts and one bolt
- The two spouts emit random integer numbers
- The bolt multiply by 2 the numbers emitted by the first spout and by 10 the numbers emitted by the second spout
- Print the results computed by the bolt on the standard output

```
(., 74, 10, 5, ..)
(., 14, 12, 2, ..)
```

```
(., 148, 20, 140, 10, 120, 20, ..)
```

These spouts emit streams of random integer numbers between 0 and 99.

This bolt multiplies each element emitted by firstSpout by 2 and each element emitted by secondSpout by 10.
package ....
import ......

public class MultipleInputStreamsTopology {
    public static void main(String[] args) throws Exception {
        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("firstSpout", new EmitRandomIntSpout(), 1);
        builder.setSpout("secondSpout", new EmitRandomIntSpout(), 1);
        builder.setBolt("merge", new ProcessMultipleStreamsBolt(), 2)
                        .shuffleGrouping("firstSpout")
                        .shuffleGrouping("secondSpout"),

        Config conf = new Config();
        conf.setDebug(false);
        conf.setNumWorkers(3);

        merge subscribes the streams emitted by firstSpout and secondSpout
    }
}

Multiple input streams: Example - Topology

if (args != null && args.length > 0) {
    String topologyName = arg[0];
    StormSubmitter.submitTopology(topologyName, conf, builder.createTopology());
} else {
    System.out.println("storm [a target/example-1.0.0.jar
storm_example.multiple_input_streams.MultipleInputStreamsTopology <topology name>");
}
}
package ...
import ...
@SuppressWarning("serial")
public class EmitRandomIntSpout extends BaseRichSpout {
    private SpoutOutputCollector collector;
    private Random rand;
    private Integer msgId;
    HashMap<Integer, Integer> sentTuples;

    @Override
    public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
        this.collector = collector;
        this.msgId = 0;
        this.rand = new Random();
        this.sentTuples = new HashMap<Integer, Integer>();
    }

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

@override
public void nextTuple() {
    Util.sleep(1000);
    Integer val = rand.nextInt(100);
    msgId++;
    collector.emit(new Values(val), msgId);
    // Store the sent tuple until the ack is received
    sentTuples.put(msgId, val);
}

@override
public void ack(Object id) {
    sentTuples.remove(msgId);
}

@override
public void fail(Object id) {
    // Send again the number associated with this msgId
    Integer val = sentTuples.get(id);
    collector.emit(new Values(val), msgId);
}
}
package ....
import ....

@SuppressWarning("serial")
public class ProcessMultipleStreamsBolt extends BaseRichBolt {

    private OutputCollector collector;

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

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {

    }

    @Override
    public void execute(Tuple tuple) {
        // Print the computed value on the standard output
        // Multiply by 2 the numbers of the stream collected from the firstSpout
        // and by 10 the numbers of the stream of the secondSpout
        if (tuple.getSourceComponent().equals("firstSpout")) {
            System.out.println("firstSpout: \" + tuple.getIntegerByField("intValue") + ":\" +
                                + tuple.getIntegerByField("intValue") * 2);
        } else {
            System.out.println("secondSpout: \" + tuple.getIntegerByField("intValue") + ":\" +
                                + tuple.getIntegerByField("intValue") * 10);
        }

        collector.ack(tuple);
    }

    }

Check which spout (component) emitted the tuple
Multiple output streams

- Each spout can emit multiple output streams
  - The emitted streams are usually used by different paths of the topology to perform different analysis in parallel
- Each output stream must be associated with a unique name
- The `emit(..)` method must be called specifying the name of the emitting stream for every emitted tuple

Multiple output streams

- Names and schemas of the emitted streams are defined in the `declareOutputFields(...)` method of the spout by using the `declareStream(name, schema)` method
- Example

```java
public void declareOutputFields(OutputFieldsDeclarer declarer){
    declarer.declareStream("firstStream",
        new Fields("firstAttr", "secondAttr"));
    declarer.declareStream("secondStream",
        new Fields("attr1"));
}
```
Multiple output streams

- In the nextTuple(..) method the emit(..) method must be called by specifying the stream name
- Example
  
  ```java
  public void nextTuple() {
      ...
      if (test) {
          collector.emit("firstStream", new Values(vals, val2), msgId);
      } else {
          collector.emit("secondStream", new Values(val1), msgId);
      }
      // Store the sent tuple until the ack is received
      sentTuples.put(msgId, val);
  }
  ```

Multiple output streams

- Bolts must specify which emitted stream want to subscribe by specifying the name of the spout and the name of the stream
  - Each bolt can subscribe multiple streams of the same spout by means of multiple calls to the grouping methods
    - One different call for each subscribed stream
- Example
  ```java
  builder.setBolt("myBolt", new MyBolt(), 2)
      .shuffleGrouping("spout", "firstStream");
  ```
Multiple output streams: Example

- Run a topology with one spout and two bolts
- The spout emits two streams of random integer numbers
  - The first stream (called evenStream) contains even numbers
  - The second stream (called oddStream) contains odd numbers
- One bolt subscribes the evenStream and multiplies each value by 2
- The other bolt subscribes the oddStream and sums 1 to each value
- Print the results computed by the two bolts on the standard output

This spout emits two streams of random integer numbers between 0 and 99. evenStream emits even numbers and oddStream emits odd numbers

(\ldots, 1, 10, 5, \ldots) \rightarrow (\ldots, 2, 20, \ldots) \rightarrow (\ldots, 2, 6, \ldots)
package ....
import ....

public class MultipleOutPutStreamsTopology{
    public static void main(String[] args) throws Exception {
        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("spout", new EmitMultipleRandomIntSpout(), 1);
        builder.setBolt("processEven", new MultiplyBy2Bolt(), 2)
            .shuffleGrouping("spout", "evenStream");
        builder.setBolt("processOdd", new Sum2Bolt(), 2)
            .shuffleGrouping("spout", "oddStream");

        Config conf = new Config();
        conf.setDebug(false);
        conf.setNumWorkers(3);

        if (args != null && args.length > 0) {
            String topologyName = args[0];

            StormSubmitter.submitTopology(topologyName, conf,
                builder.createTopology());
        } else {
            System.out.println("storm jar target/example-1.0.0.jar
                storm_example.multiple_output_streams.MultipleOutPutStreamsTopology <topology
                name>");
        }
    }
}
package ....
import .......

@SuppressWarnings("serial")
public class EmitMultipleRandomIntSpout extends BaseRichSpout{

    private SpoutOutputCollector collector;
    private Random rand;
    private Integer msgId;
    HashMap<Integer, Integer> sentTuples;

    @Override
    public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
        this.collector = collector;
        this.msgId = 0;
        this.rand = new Random();
        this.sentTuples = new HashMap<Integer, Integer>();
    }
}

@Override
public void declareOutputFields(OutputFieldsDeclarer declarer) {

    declarer.declareStream("oddStream", new Fields("intValue"));
    declarer.declareStream("evenStream", new Fields("intValue"));
}

@Override
public void nextTuple() {
    Utils.sleep(100);
    Integer val = rand.nextInt(100);
    msgId++;
    if (val % 2 == 0) {
        collector.emit("evenStream", new Values(val), msgId);
    } else {
        collector.emit("oddStream", new Values(val), msgId);
    }
    // Store the sent tuple until the ack is received
    sentTuples.put(msgId, val);
    }

Declare the output streams:
- Name
- Schema
Multiple output streams: Example - Spout

```java
@Override
public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declareStream("oddStream", new Fields("intValue"));
    declarer.declareStream("evenStream", new Fields("intValue"));
}

@Override
public void nextTuple() {
    Utils.sleep(100);
    Integer val = rand.nextInt(100);
    msgId++;
    if (val % 2 == 0) {
        collector.emit("evenStream", new Values(val), msgId);
    } else {
        collector.emit("oddStream", new Values(val), msgId);
    }
    // Store the sent tuple until the ack is received
    sentTuples.put(msgId, val);
}
```

Specify the name of the output stream

---

Multiple output streams: Example - Spout

```java
@Override
public void ack(Object id) {
    sentTuples.remove(msgId);
}

@Override
public void fail(Object id) {
    // Send again the number associated with this msgId
    Integer val = sentTuples.get(id);
    if (val % 2 == 0) {
        collector.emit("evenStream", new Values(val), msgId);
    } else {
        collector.emit("oddStream", new Values(val), msgId);
    }
}
```

Remove the tuple from the queue when received an ack
Multiple output streams: Example - Spout

```java
@Override
public void ack(Object id) {
    sentTuples.remove(msgId);
}

@Override
public void fail(Object id) {
    // Send again the number associated with this msgId
    Integer val = sentTuples.get(id);
    if (val % 2 == 0) {
        collector.emit("evenStream", new Values(val), msgId);
    } else {
        collector.emit("oddStream", new Values(val), msgId);
    }
}
```

Send again the tuple in case of failure

Multiple output streams: Example – MultiplyBy2Bolt

```java
package ...
import ...
@SuppressWarnings("serial")
public class MultiplyBy2Bolt extends BaseRichBolt
```

```java
private OutputCollector collector;

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

@Override
public void declareOutputFields(OutputFieldsDeclarer declarer) {
}
```
Multiple output streams: Example – MultiplyBy2Bolt

```java
@override
public void execute(Tuple tuple) {
    // Print the computed value on the standard output
    // Multiply by 2 the value of the tuple
    System.out.println(tuple.getSourceStreamId());
    System.out.println("Even" + tuple.getIntegerByField("intValue") + ">" +
    (tuple.getIntegerByField("intValue") * 2));
    collector.ack(tuple);
}
```

Multiple output streams: Example – Sum1Bolt

```java
package ...
import ...
@SuppressWarnings("serial")
public class Sum1Bolt extends BaseRichBolt {

    private OutputCollector collector;

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

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
    }
```
Multiple output streams: Example – Sum1Bolt

```java
@Override
public void execute(Tuple tuple) {
    // Print the computed value on the standard output
    // Sum 1 to the value of the tuple
    System.out.println(tuple.getSourceStreamId());
    System.out.println("Odd":+tuple.getIntegerByField("intValue") + ";" +
    (tuple.getIntegerByField("intValue") + 1));
    collector.ack(tuple);
}
}
```

Multiple output streams

- Also bolts can emit multiple streams
- The approach is the same used for spouts
Reliability with complex topologies

Reliable Topologies

- Storm offers several different levels of guaranteed message (tuple) processing
  - Best effort
    - No reliable spouts
      - Acks and fails are not managed
  - At least once
    - Reliable spouts
      - Acks and fails are managed and non-processed tuples are sent again in order to be processed
  - Exactly once through Trident
    - We will see it later
Reliable Topologies

- At least once
  - We already discuss how to implement simple reliable topologies
    - A topology with one spout and one bolt
    - A topology with a single path
  - The next slides discuss how to manage reliability with more complex topologies

Tuple trees

- A tuple coming off a spout can trigger thousands of tuples to be created based on it
- For each tuple emitted by a spout, storm can build a tuple tree
  - It represents the dependencies among the original tuples and its “descendants”
- Storm considers a tuple coming off a spout “fully processed” when the tuple tree has been exhausted and every message in the tree has been processed
- A tuple is considered “failed” when its tree of messages fails to be fully processed within a specified timeout or when at least one failure appends
Storm's reliability

- To benefit from Storm's reliability capabilities you must
  - Tell Storm whenever you're creating a new link in the tree of tuples
    - Anchoring the new tuples to the original ones
  - Tell Storm when you have finished processing an individual tuple
    - By calling the ack method on the processed tuples
- By doing both these things, Storm can detect when the tree of tuples is fully processed and can ack or fail the spout tuple appropriately

Anchoring

- Specifying a link in the tuple tree is called anchoring
- Anchoring is done at the same time you emit a new tuple by specifying also the original tuple in the emit(…) method
  - collector.emit(tuple, emitted tuple)
Multiple-Anchoring

- An output tuple can be anchored to more than one input tuple
  - This is useful when doing streaming joins or aggregations
- A multi-anchored tuple failing to be processed will cause multiple tuples to be replayed from the spouts

Multiple-Anchoring

- Multi-anchoring is done by specifying a list of tuples rather than just a single tuple when calling the emit() method
- Example
  ```java
  List<Tuple> anchors = new ArrayList<Tuple>();
  anchors.add(tuple1);
  anchors.add(tuple2);
  collector.emit(anchors, new Values(1, 2, 3));
  ```
Aggregations and joins

- Bolts that do aggregations or joins may delay acking a tuple until after it has computed a result based on a bunch of tuples
- Aggregations and joins will commonly multi-anchor their output tuples as well
- We will see an example later

Common Topology Patterns
Streaming joins

- A streaming join combines two or more data streams together based on some common fields
- There are several definitions/types of “streaming join”
  - Some applications join all tuples for two streams over a finite window of time
  - Other applications expect exactly one tuple for each stream involved in the join
- The join type is usually application-dependent
Streaming joins

- The common pattern among all these join types consists of the following steps
  - Send the tuples of the multiple input streams with the same values of the join fields to the same task of the joining bolt
    - This is accomplished by using a fields grouping on the join fields for the input streams to the join bolt
  - Temporarily store the tuples in an instance variable of the task
  - Perform the join operation inside the task
  - Remove the tuples from the instance variable as soon as they are not more needed

In-memory caching + fields grouping combo
In-memory caching + fields grouping combo

- It is common to keep caches in-memory in Storm bolts
  - For example to avoid invoking multiple times an external service through HTTP requests
- Caching becomes particularly powerful when you combine it with a fields grouping
  - Each task keeps only the subset of cache used to process the values sent to it
    - No useless overlapping among the caches of the bolt’s tasks

In-memory caching + fields grouping combo

- Suppose you have a bolt that expands short URLs into long URLs
  - Given a short URL, an HTTP request to an external service is invoked to obtain the long URL
  - Keep an LRU cache of short URL to long URL to avoid doing the same HTTP requests multiple times
  - To improve the efficiency and reduce multiple requests for the same short URL, fields grouping on the short URL field must be specified
    - Each task of the bolt manages a subset mapping short URL -> long URL
BasicBolt

- Many bolts follow a similar pattern of
  - Reading an input tuple
  - Emitting zero or more tuples based on that input tuple
  - And then acking that input tuple immediately at the end of the execute method
- Bolts that match this pattern are things like functions and filters
**BasicBolt**

- This is such a common pattern that Storm exposes an abstract class called BaseBasicBolt that automates this pattern for you
  - All acking is managed for you
  - Throw a FailedException if you want to fail the tuple

**Periodic statistics/output**
Periodic statistics/output

- Many applications emit a statistic of interest, based on the analysis of the input stream, every $t$ seconds
- For example, suppose you have a bolt that every $t$ seconds emits the number of analyzed input tuples

Periodic statistics/output: Sol #1

- This problem can be solved by using a spout generating a “signal” every $t$ seconds
  - The bolt emits the current value of the statistic every time it receives the “signal” tuple
  - The bolt subscribes both the signal stream and the stream of data to analyze
Periodic statistics/output: Sol #2

- Storm provides a special type of tuples called Tick tuples
- They are configured per-component, i.e. per bolt
  - One Tick tuple is sent to each component every
    Config.TOPLOGY_TICK_TUPLE_FREQ_SECS seconds
- We can use this special type of tuples to decide when to emit the statistic of interest

Tick tuples

- The frequency of the tick tuples for each bolt is set in the getComponentConfiguration method of the bolt
  @Override
  public Map<String, Object> getComponentConfiguration() {
      Map<String, Object> conf = new HashMap<String, Object>();
      conf.put(Config.TOPLOGY_TICK_TUPLE_FREQ_SECS, emitFrequencyInSeconds);
      return conf;
  }
Tick tuples

- The method TupleUtils.isTick(tuple) can be used in the execute(..) method of a bolt to check if the current tuple is a Tick tuple

Tick tuples

- Tick tuples are not 100% guaranteed to arrive in time
  - They are sent to a bolt just like any other tuples, and will enter the same queues and buffers
    - Congestion, for example, may cause tick tuples to arrive too late.
  - Across different bolts, tick tuples are not guaranteed to arrive at the same time
    - Even if the bolts are configured to use the same tick tuple frequency
  - Currently, tick tuples for the same bolt will arrive at the same time at the bolt's various task instances
    - However, this property is not guaranteed for the future
- Tick tuples must be acked like any other tuple
Periodic statistics/output: Example

- Run a topology that every $t$ seconds emits the number of tuples emitted by a spout that emits a stream of random integers

Streaming top N
Streaming top N

- A common continuous computation done on Storm is "streaming/selecting top N" elements
- For example, suppose you have a spout that emits tuples of the form ["value", "count"] and you want a bolt that emits, every $t$ seconds, the top N tuples based on count

Streaming top N: Solution #1

- The simplest way to implement streaming top N is based on one single bolt
- The bolt
  - Does a global grouping on the stream
    - i.e., all tuples are sent to one single task of the bolt
  - Maintains a list in memory of the top N items
    - In the only task executing the bolt
  - Emits the top-N list every $t$ seconds
- This approach does not scale to large streams since the entire stream has to go through one single task
Streaming top N: Solution #2

- A more scalable solution is based on two bolts
- The first bolt computes local top-N lists in parallel on the input stream
  - One top-N list in each task of the first bolt
  - Each task emits its local top-N list every $t$ seconds
- The second bolt computes the global top-N list merging the local ones
  - This bolt does a global grouping on the output of the first bolt and emits the global top-N list every $t$ seconds

Streaming top N

- The differences between Solution #1 and Solution #2 is highly related to $t$ (the frequency of emission of the global top-N list)
  - The higher $t$, the higher the difference between Sol. #1 and Sol. #2
Some applications need to process a group of tuples in batch rather than individually
- You may want to batch updates to a database for efficiency reasons
- You may need to do a streaming aggregation
**Batching**

- If you want reliability in your batching data processing
  - You must hold on the tuples in an instance variable while the bolt waits to do the batching
  - Once you complete the batch operation, ack all the tuples you were holding
  - If the bolt emits tuples, then you may want to use multi-anchoring to ensure reliability

**Batching**

- This pattern can be implemented by using
  - The standard classes
  - Or transactional topologies
    - There are specifically designed for processing batch of tuples