Data stream processing is an important aspect of distributed systems. A data stream represents a sequence of values or tuples from a sensor, a device, or a code module. Data streams typically have a schema describing the format of the stream elements. We often model streams as being infinite or unbounded, and process input items one at a time.

Data stream processing platforms often operate at one or both of two layers: distributed message queuing services via platforms such as Apache Kafka, Amazon SQS, and RabbitMQ; and distributed real-time data processing engines such as Apache Storm and Samza, Spark Streaming, Flink, and other competitors.

For this class we will adopt the programming model of Storm. Since Storm requires substantial setup that we would prefer to avoid, we have built StormLite, a slightly simplified (non-distributed) implementation of the Apache Storm-style interfaces. Where possible, we attempted to follow the Storm APIs to make it easier for you to learn the programming model, e.g., from Stack Overflow and documents on the Web. The major exception is that package names for the basic constructs have been changed. The basic constructs in Storm and StormLite are as follows.

Core Components

Streams are named sequences of data, arriving in some order and satisfying a particular schema (set of named fields). Streams may be processed or created in parallel by multiple threaded operations1 (spouts and bolts). A Stream is in essence a sequence of Tuples sent to an IOutputCollector, which in turn uses a “router” to send to a destination.

A Topology is a logical description of the operations (spouts and bolts) as a conceptual “graph” linking bolts to spouts, to other spouts. See the figure to the right.

Spouts are instances of programmer-created classes that supply data to a stream (via an OutputCollector). These implement an interface called IRichSpout. For instance, a spout may read records from a text file, or it may be a web crawler that emits documents.

---

1 And, in the real Apache Storm, the operators may be distributed.
Bolts are instances of programmer-created classes that process data (possibly from multiple input streams) and produce an optional output stream (via a SpoutOutputCollector). These implement IRichBolt. An example bolt might maintain a running count of specific words encountered.

Each spout or bolt may actually be run in multiple parallel (or even distributed) instances called executors, but all of these instances conceptually send results to the same named output stream, and ultimately to one or more executors of the “downstream” operator. In reality, when a tuple is sent to an OutputCollector, it first goes to a IStreamRouter, which determines which executor to use according to a grouping policy. Executors can be chosen based on field values (equivalent to sharding by a particular key) or in round-robin fashion.

A Dataflow Program in StormLite

A “program” in Storm (and StormLite) really consists of a group of objects (instances of spouts and bolts) strung together, and communicating through in-order messages. This is achieved by using the TopologyBuilder to create a set of named spouts and bolts, to suggest a number of parallel instances, and to define how the output is handled. An example from http://storm.apache.org/releases/0.10.0/Tutorial.html is shown below:

```
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");
```

Notice here that we have a Spout operator – TestWordSpout – that produces words in a stream called words. There are up to 10 parallel instances (executors) of TestWordSpout, as indicated by the 3rd parameter.

Next we see two ExclamationBolts, the first with 3 parallel executors and the second with 2 executors. The first one receives the data from the words stream, and uses a shuffle grouping (round-robin) to route each word to a different executor for the ExclamationBolt. The output is a stream called exclaim1. The second ExclamationBolt receives data from the exclaim1 stream, again in round-robin order across its executors, and produces the exclaim2 stream.

An alternate kind of grouping is the FieldGrouping, which lets us partition the data according to certain attributes’ values – i.e., sharding the data to different executors.
Our sample code includes **TestWordCount**, which generates a spout (WordSpout, which reads a file called words.txt and then starts generating random words) and two bolts (WordCounter, which keeps a running count of the number of occurrences of each received word; and PrintBolt, which simply outputs the values it receives). To show the power of the model, we use multiple executors of the **WordCounter**, and use a **fieldGrouping** to ensure that the same words always get mapped to the same executor.

If you look “under the hood” you will see that StormLite has a basic notion of a “runnable task” – this involves creating a single tuple from a Spout, or handling a single tuple in a Bolt. We use a thread pool to process the tasks.

As part of Homework 3 you’ll extend StormLite to support distributed computation.