1 Complex Event Processor

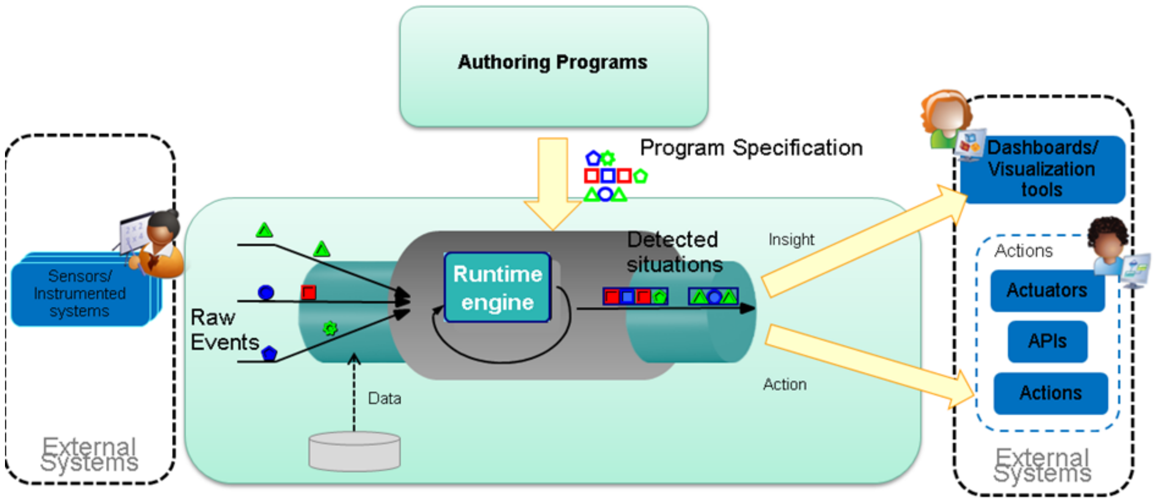


Figure 1 - Proton Authoring Tool and Runtime Engine

Proton—**IBM Proactive Technology On-Line**—is an open source IBM research asset for complex event processing extended to support the development, deployment, and maintenance of proactive event-driven applications. **Proactive event-driven computing** (Engel and Etzion 2011) is the ability to mitigate or eliminate undesired states, or capitalize on predicted opportunities—in advance. This is accomplished through the online forecasting of future events, the analysis of events coming from many sources, and the enabling of online decision-making processes.

Proton receives **raw** events, and by applying **patterns** defined within a **context** on those events, computes and emits **derived** events (see Figure).

1.1 Functional Highlights

Proton's generic application development tool includes the following features:

- Enables fast development of proactive applications.
- Entails a simple, unified high-level programming model and tools for creating a proactive application.
- Resolves a major problem—the gap that exists between events reported by various channels and the reactive situations that are the cases to which the system should react. These situations are a composition of events or other situations (e.g., "when at least four events of the same type occur"), or content filtering on events (e.g., "only events that relate to IBM stocks"), or both ("when at least four purchases of more than 50,000 shares were performed on IBM stocks in a single week").
- Enables an application to detect and react to customized situations without having to be aware of the occurrence of the basic events.
- Supports various types of contexts (and combinations of them): fixed-time context, event-based context, location-based context, and even detected situation-based context. In addition, more than one context may be available and relevant for a specific event-processing agent evaluation at the same time.
- Offers easy development using web-based user interface, point-and-click editors, list selections, etc. Rules can be written by non-programmer users.

- Receives events from various external sources entailing different types of incoming and reported (outgoing) events and actions.
- Offers a comprehensive event-processing operator set, including joining operators, absence operators, and aggregation operators.

1.2 Technical Highlights

- Is platform-independent, uses Java throughout the system.
- Comes as a J2EE (Java to Enterprise Edition) application or as a J2SE (Java to Standard Edition) application.
- Based on a modular architecture.

1.3 Proton High-level Architecture

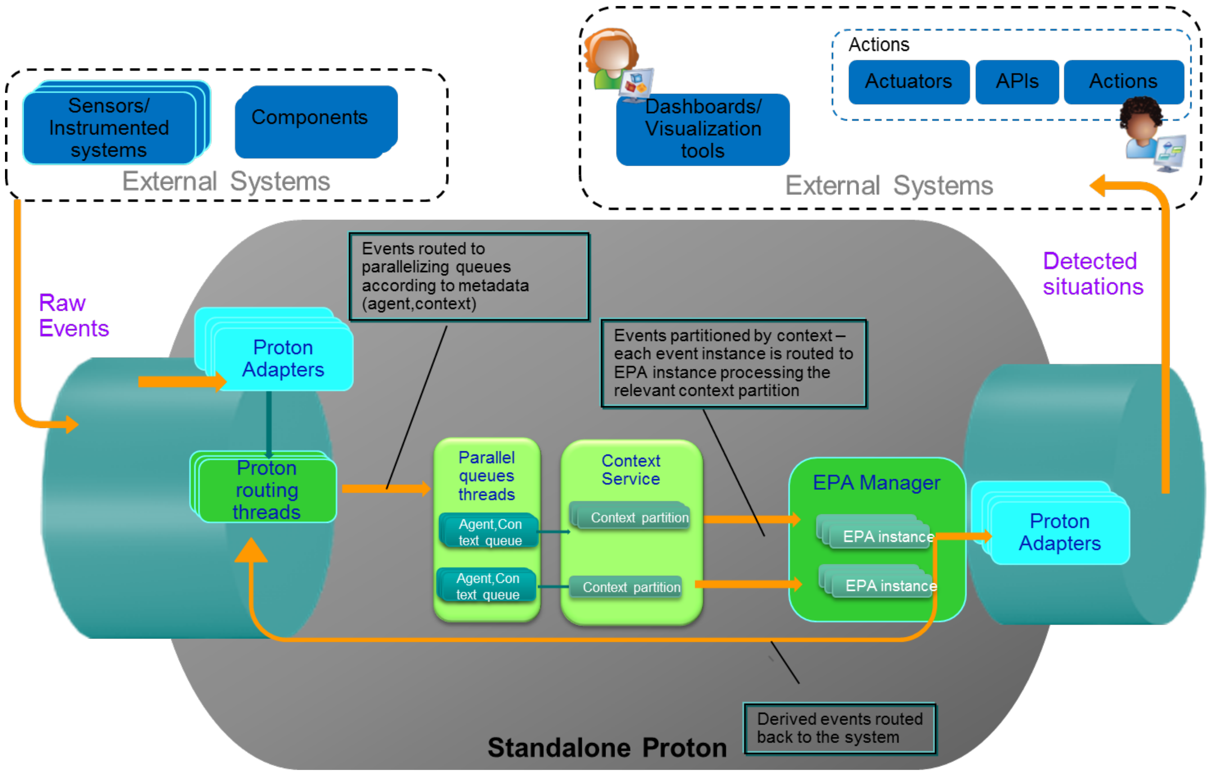


Figure 2 - Proton Runtime and external systems

Proton architecture consists of a number of functional components and interaction among them, the main of which are (see Figure):

- Adapters – communication of Proton with external systems
- Parallelizing agent-context queues – for parallelization of processing of single event instance, participating in multiple patterns/contexts, and parallelization of processing among multiple event instances
- Context service – for managing of context's lifecycle –initiation of new context partitions, termination of partitions based on events/timers, segmenting incoming events into context groups which should be processed together.
- EPA manager –for managing Event Processing Agent (EPA) instances per context partition, managing its state, pattern matching and event derivation based on that state.

When receiving a raw event, the following actions are performed:

1. Look up within the **metadata**, to see which context effect this event might have (context initiator, context terminator) and which pattern this event might be a participant of
2. If the event can be processed in parallel within multiple contexts/patterns (based on the EPN definitions), the event is passed to **parallelization queues**. The purpose of the queues:
    a. Parallelize processing of the same event by multiple unrelated patterns/contexts at the same time keeping the order for events of the same context/pattern where order is important
    b. Solve out-of-order problems – can buffer for a specified amount of time
3. The event is passed to **context service**, where it is determined:
    a. If the context is an initiator or a terminator, new contexts might be initiated and or terminated, according to relevant policies.
    b. Which context partition/partitions this event should be grouped under
4. The event is passed to **EPA manager:**
    a. Where it is passed to the specific EPA instance for the relevant context partition,
    b. Added to state of the instance
    c. And invokes pattern processing
    d. If relevant, a derived event is created and emitted

1.4 Proton component architecture

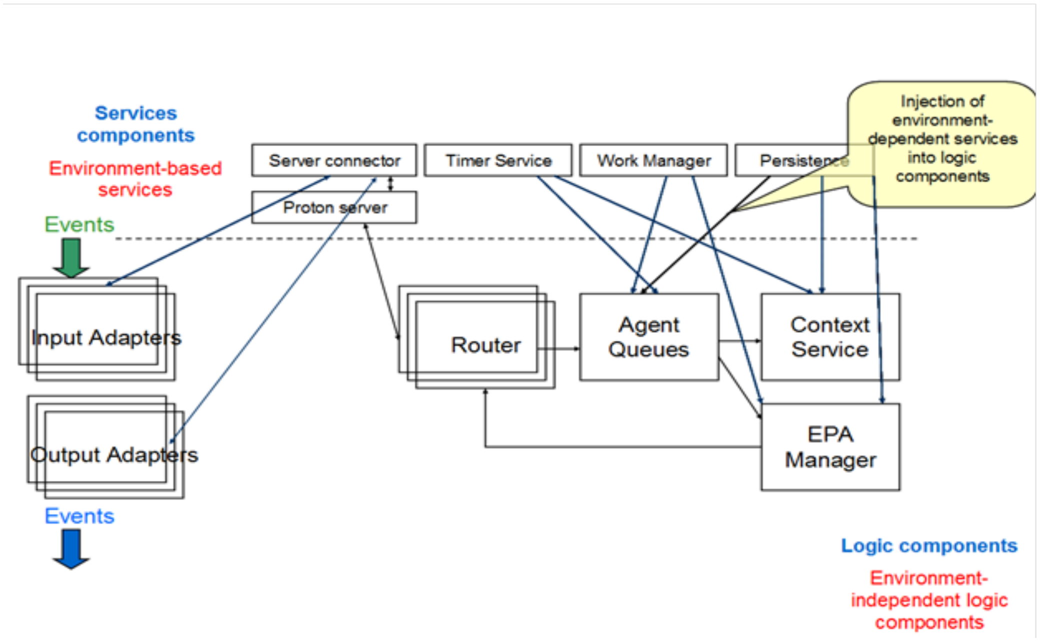


Figure 3 - Proton components

Proton's logical components are illustrated in Figure . The queues, the context service, the EPA manager are purely java-based. They utilize dependency injection to make use of the infrastructure services they require, e.g. work manager, timer services, communication services. These services are implemented differently for the J2SE and J2EE versions.

1.5 Distributed Architecture on top of STORM

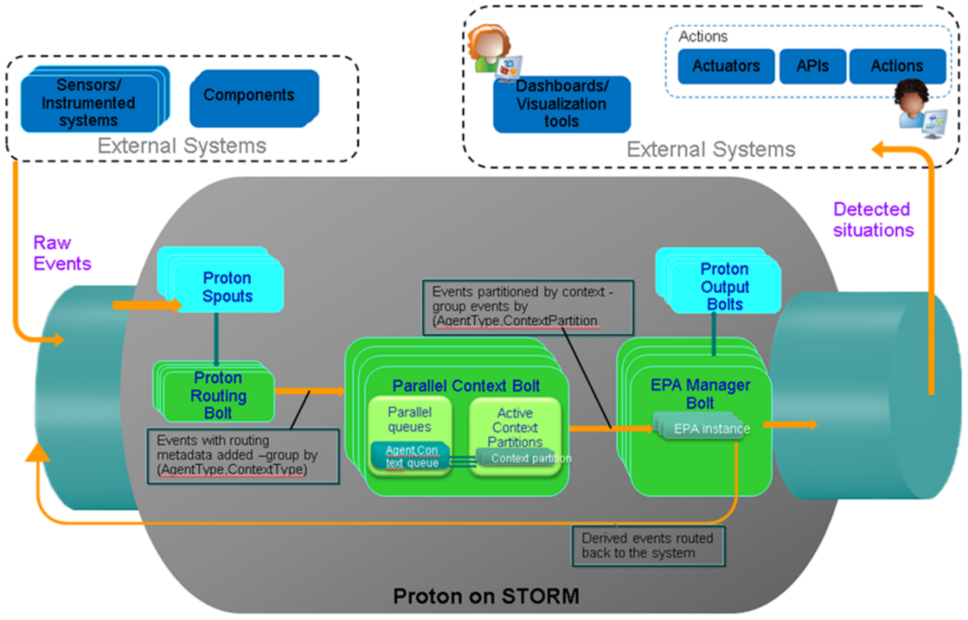

Sensors/ Instrumented systems
Components
External Systems
Actions
Actuators
APIs
Actions
Dashboards/ Visualization tools
External Systems
Detected situations
Raw Events
Proton Spouts
Events partitioned by context - group events by (AgentType.ContextPartition
Proton Output Bolts
Proton Routing Bolt
Parallel Context Bolt
Parallel queues
Active Context Partitions
Agent Context queue
Context partition
EPA Manager Bolt
EPA instance
Events with routing metadata added –group by (AgentType.ContextType)
Derived events routed back to the system
Proton on STORM

Figure 4 - Architecture of Proton on Storm

The Proton architecture on top of Storm[1] (see Figure) preserves the same logical components as are present in the standalone architecture: the queues, the context service and the EPA manager, which constitutes the heart of the event processing system. However the orchestration of the flow between the components is a bit different, and utilizes existing Storm primitives for streaming the events to/from external systems, and for segmenting the event stream.

After the routing metadata of an incoming event is determined by the routing bolt (which has multiple independent parallel instances running), the metadata – the agent name and the context name – is added to the event tuple.

We use the Storm field grouping option on the metadata routing fields – the agent name and the context name – to route the information to the next Proton bolt – the context processing. Therefore all events which should be processed together – relating to the same context and agent – will be sent to the same instance of the bolt.

After queueing the event instance in the relevant queues (in order to solve out of order, if needed and parallelize event processing of the same instance where possible by different EPAs in the same EPN) and after processing by context service, the relevant context partition id is added to the tuple.

Here again we use the field grouping on context partition and agent name fields to route the event to specific instances of the relevant EPA, this way performing data segmentation – the event will be routed to the agent instance which manages the state for a specific agent on a specific partition.

If the pattern matching is done and we have a derived event, it will be routed back into the system, and passed through the same channels as the raw event.

Work on implementing Proton on Storm is being performed as part of the FERARI EU project (http://www.ferari-project.eu).

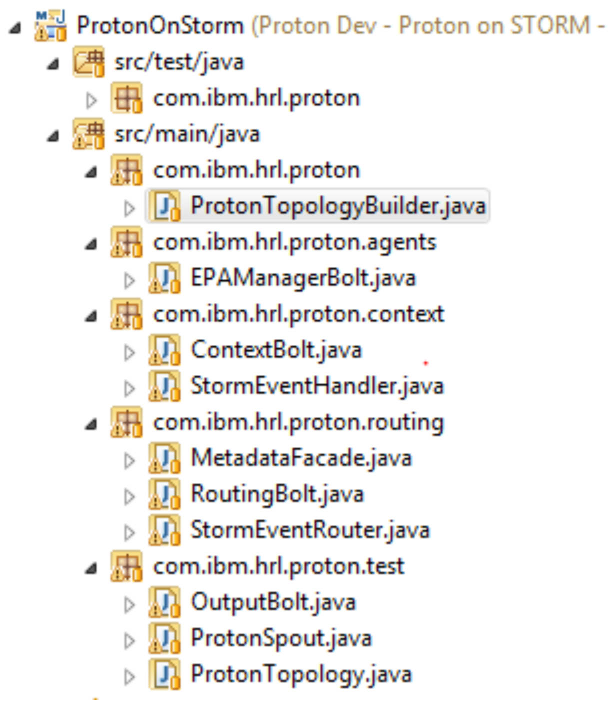
1.6 Project structure for Proton on STORM

Proton on STORM is a wrapper project using both the Proton standalone version and the STORM infrastructure, as presented in previous section.

Using STORM primitives, specifically Proton topology , we allow to build a STORM application, provide it with a spout implementation which will act as the source of Proton events, provide it a bolt implementation, which will acts as consumer of Proton situations, aka derived events , and we can run this topology in a distributed manner on top of STORM cluster.

1.6.1 Understanding the structure and source code of ProtonOnSTORM:

The structure of the Proton on STORM project is as follows (see Figure 5):



```
ProtonOnStorm (Proton Dev - Proton on STORM -
  src/test/java
    com.ibm.hrl.proton
  src/main/java
    com.ibm.hrl.proton
      ProtonTopologyBuilder.java
    com.ibm.hrl.proton.agents
      EPAManagerBolt.java
    com.ibm.hrl.proton.context
      ContextBolt.java
      StormEventHandler.java
    com.ibm.hrl.proton.routing
      MetadataFacade.java
      RoutingBolt.java
      StormEventRouter.java
    com.ibm.hrl.proton.test
      OutputBolt.java
      ProtonSpout.java
      ProtonTopology.java
```

Figure5 – ProtonOnSTORM project structure

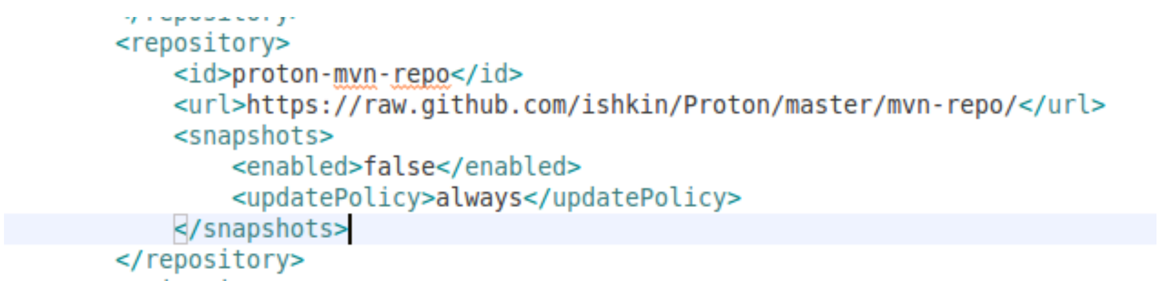
The packages **com.ibm.hrl.proton.context, com.ibm.hrl.proton.context, com.ibm.hrl.proton.agents** consist of STORM wrappers around Proton logical components (such as context service, EPA manager and metadata and routing components) to allow to run them in distributed manner on top of STORM.

1.6.2 To include ProtonOnSTORM project in your maven build for your topology

There is a maven repository including the ProtonOnSTORM artifact which can be included as is in your project.

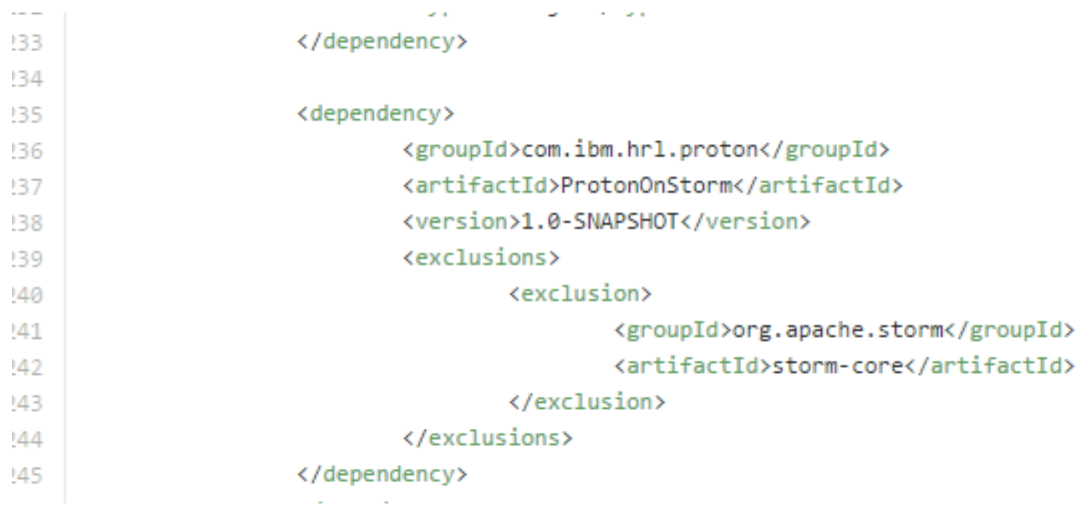
To include the ProtonOnSTORM dependency in your project, please add the following to your pom.xml buildfile:

1. The maven repository on git:



```
        <repository>
            <id>proton-mvn-repo</id>
            <url>https://raw.github.com/ishkin/Proton/master/mvn-repo/</url>
            <snapshots>
                <enabled>false</enabled>
                <updatePolicy>always</updatePolicy>
            </snapshots>
        </repository>
```

2. The ProtonOnSTORM dependency



```
                </dependency>

                <dependency>
                        <groupId>com.ibm.hrl.proton</groupId>
                        <artifactId>ProtonOnStorm</artifactId>
                        <version>1.0-SNAPSHOT</version>
                        <exclusions>
                                <exclusion>
                                        <groupId>org.apache.storm</groupId>
                                        <artifactId>storm-core</artifactId>
                                </exclusion>
                        </exclusions>
                </dependency>
```

1.6.3 To use Proton in your topology

The package **com.ibm.hrl.proton** contains one class, **ProtonTopologyBuilder**, which is a helper class. When building a STORM topology with Proton, the developer will have to instantiate a **ProtonTopologyBuilder** class instance, and invoke a ***buildProtonTopology()*** method, while providing it with the following information:

- **TopologyBuilder topologyBuilder** – a STORM topology builder which should be created using STORM APIs. When the Proton topology is created, this builder is used to concatenate the Proton topology to the general topology
- **BaseRichSpout inputSpout** – an implementation of STORM spout interface which will inject events into Proton engine
- **BaseRichBolt outputBolt –** an implementation of STORM bolt, which will act as receiver of derived situations, represented as tuples, from the Proton engine
- **String outputBoltName –** the name of the output bolt, so when the topology is created the bolt is chained after the Proton bolts.
- **String jsonFileName –** an absolute path to a file containing Proton application definitions. This file should of course be located on the machine which is creating and submitting STORM topology. Additional method can be added to this class, which will receive a String representation of the JSON definition file and load it the same way. Please see explanation regarding Proton semantics and instructions on how to build Proton scenario file in Proton user guide.

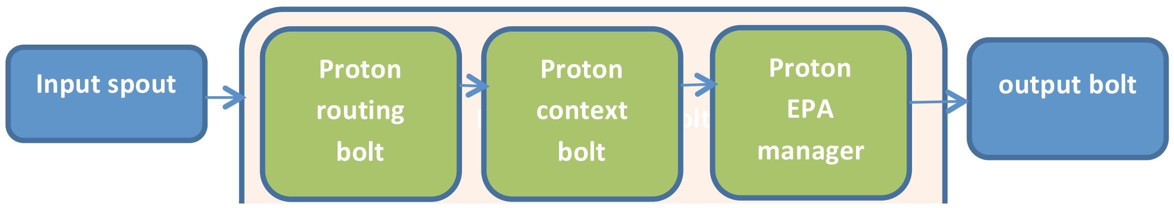
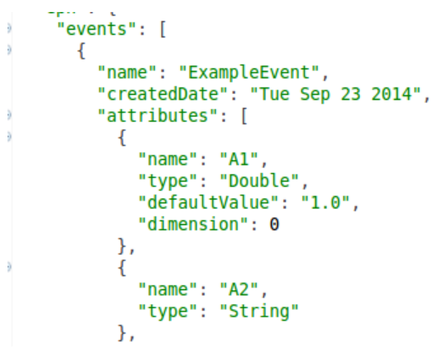


Figure6 – ProtonOnSTORM project structure

The input spout and output bolt are provided by the user and are concatenated into Proton topology by the ProtonTopologyBuilder (see Figure 6). The rest of the topology is static.

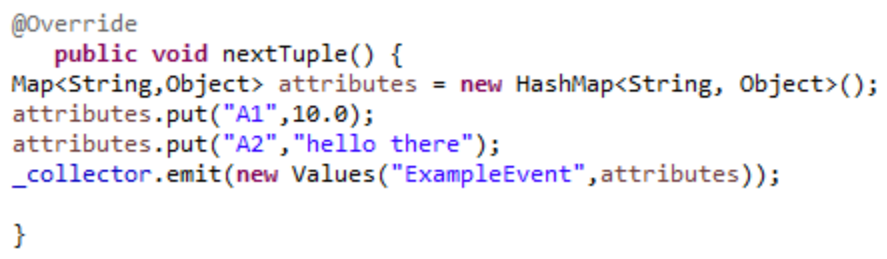
The input spout should create events which should match the Proton definition file. For example, in case we have an input even in Proton looking like this (this is an extract from Proton definition JSON file):



```
  "events": [
    {
      "name": "ExampleEvent",
      "createdDate": "Tue Sep 23 2014",
      "attributes": [
        {
          "name": "A1",
          "type": "Double",
          "defaultValue": "1.0",
          "dimension": 0
        },
        {
          "name": "A2",
          "type": "String"
        },
```

in that case, the input spout should emit tuples representing this event type, where the name of the tuple is the name of the event, and it includes a hashmap <String,Object> where the String is the event attribute name, and the Object is the attribute value.

So an implementation of input spout for this example would look like :

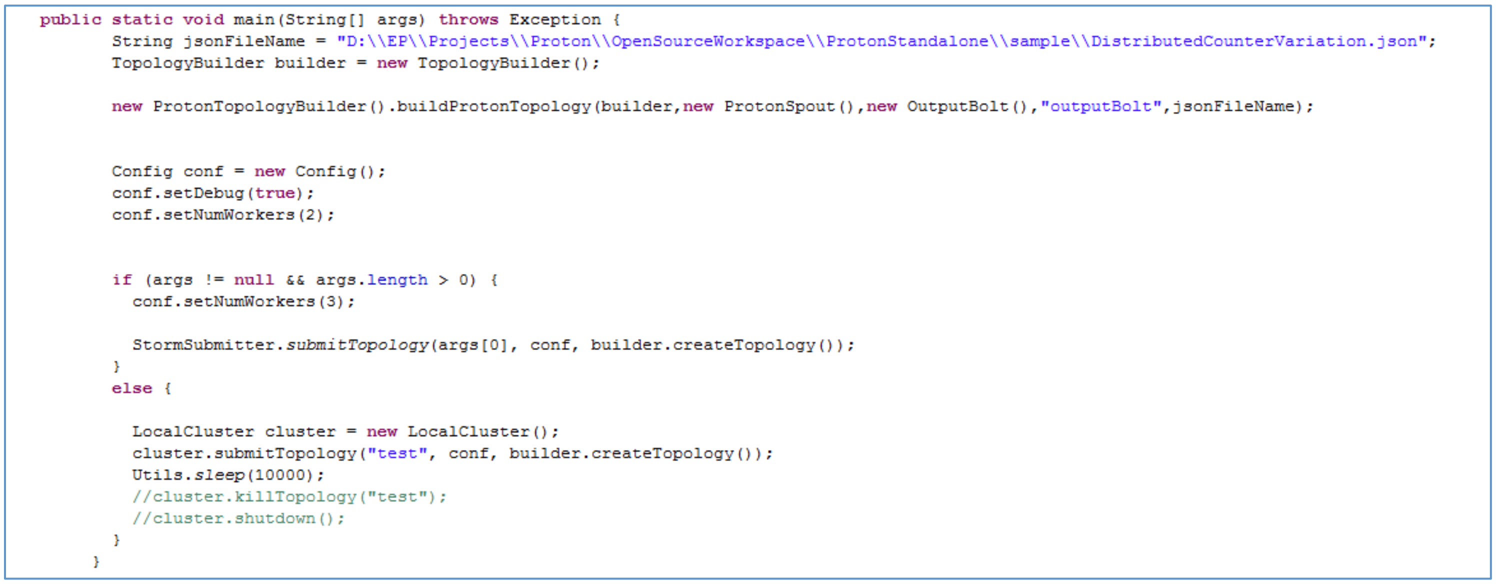


```
@Override
   public void nextTuple() {
Map<String,Object> attributes = new HashMap<String, Object>();
attributes.put("A1",10.0);
attributes.put("A2","hello there");
_collector.emit(new Values("ExampleEvent",attributes));

}
```

The same thing goes for output bolt, it will receive events from Proton built in this manner exactly – a tuple where the name matches the name of Proton output event, and a map of its attributes.

An example on how to incorporate Proton topology into your application on top of STORM can be found **in com.ibm.hrl.proton.test** package. It includes an implementation of test input spout, just producing tuples from hard-coded map, a test output bolt, writing the derived events to a file, and a **ProtonTopology** class, which orchestrates it all together – it instantiates the **ProtonTopologyBuilder** class , passes to it the input spout and output bolt and JSON configuration file. See Figure 7 for demonstration



```
public static void main(String[] args) throws Exception {
        String jsonFileName = "D:\\EP\\Projects\\Proton\\OpenSourceWorkspace\\ProtonStandalone\\sample\\DistributedCounterVariation.json";
        TopologyBuilder builder = new TopologyBuilder();

        new ProtonTopologyBuilder().buildProtonTopology(builder,new ProtonSpout(),new OutputBolt(),"outputBolt",jsonFileName);


        Config conf = new Config();
        conf.setDebug(true);
        conf.setNumWorkers(2);


        if (args != null && args.length > 0) {
          conf.setNumWorkers(3);

          StormSubmitter.submitTopology(args[0], conf, builder.createTopology());
        }
        else {

          LocalCluster cluster = new LocalCluster();
          cluster.submitTopology("test", conf, builder.createTopology());
          Utils.sleep(10000);
          //cluster.killTopology("test");
          //cluster.shutdown();
        }
      }
```

Figure7 –invocation of Proton topology builder

Take into account that this method is submitting the topology to run in test mode in local cluster, to see examples on how to submit a distributed topology please refer to STORM documentation.

Building and running Proton on STORM (without the encompassing STORM topology, just a test)

The project includes maven build file (pom.xml) which includes all required dependencies and plugins.

To build the Proton on STORM project, cd to the root directory of the project (the maven project repository) and run:

mvn clean install

Once the project is successfully built, you can run the whole topology by using :

mvn -f pom.xml compile exec:java -Dstorm.topology=<`com.ibm.hrl.proton.test.ProtonTopology>`

The class name to use is the class with the main method, which builds and submits STORM topology.