In-network event processing on IoT

Krishna Prasad Anna Ramesh Kumar
College of Computing and Information Sciences
Rochester Institute of Technology

Abstract—Internet of Things (IoT) is becoming a part of our day to day lives and as the number of devices grow exponentially, the need for processing the data produced by these devices grow as well. Currently, there is a high demand to process streams of data from these devices and react to events in real-time but there is no concrete solution that does this without making use of cloud services. This project is aimed at solving this issue with processing most of the data in real-time within the network in order to locally react faster to events instead of waiting for a response from cloud services to take further action. Distributed stream processing engines like Apache Flink provides the necessary features to enable processing of events in real-time as well as the capability to express complex events easily. Adapting complex event processing ideas into our own Software Defined Networking - inspired framework to enable event detection is the vision we have for our research project.

Keywords—Internet of Things, Stream Processing, Complex Event Processing

I. INTRODUCTION

IoT devices are everywhere and most of the equipment we currently use are controlled by microprocessors built into these devices. All these devices continuously produce data and sometimes a system that monitors these devices needs to perform computations on this data to take an action immediately. At present, a system like this would upload all the data to a cloud service so that the computations can be done over the cloud. Meanwhile, the local network would wait for the cloud service to respond until it is finished with the computation.

There are two disadvantages to this approach: Uploading a stream of data to a cloud service continuously is an energy and time consuming process and the delay to transmit data from these IoT devices to a gateway, upload the data to a cloud service and wait for response is considerably high.

The need for reacting to events in real-time is present in a lot of use-cases. An example can be an alarm that needs to be triggered when there is a fire inside a temperature maintained data-center. The temperature and smoke sensors inside the room would be producing streams of data. The system that is receiving this data needs to react in real-time to trigger the alarm when there is a continuous increase in temperature over a period of time or when a smoke sensor is detecting high levels of carbon mono-oxide or when both of these events happen together.

Writing complex rules for a system can be cumbersome especially when the data needs to be processed as soon as it arrives. In order to facilitate writing complex rules with ease, the domain of complex event processing (CEP) comes in. Writing aggregation or average functions for stream processing and defining event patterns with the help of a programming language can turn out to be really messy. Therefore, an SQL-like language is used in the complex event processing domain. These languages are developed for very low latency and can process a huge number of events per second.

The need for both complex event processing and stream processing of data in real-time is highly evident from the description above and both these domains are at the verge of merging together so that streams of data can be processed in real-time and complex events can be defined for this processing as well.

In this project, based on the related and previous work, the complex events required for IoT networks have been modeled and these models have been adapted to a Software Defined Networking (SDN) -inspired iQdep [1] architecture. The event detection models have been simulated on the Mininet[2] platform as a proof of implementing in a SDN controlled IoT network.

II. RELATED WORK

The following sections discuss the background of work done in the research community on stream processing, complex event processing and distributed dataflow models.

A. Stream Processing - Apache Flink

Apache Flink [3] is a real-time streaming processing engine that offers distributed stream processing or computation at real-time. It has been built with the goal of providing very low latency, high performance, event time processing and exactly-once semantics. A recent release also provides a complex event processing library to detect complex events at real-time. The CEP library provides APIs in Java or Scala right now and does not currently offer a SQL-like language. A language called Stream SQL is being currently developed to integrate complex event processing with the existing stream processing capabilities of Flink.

B. Actor System

Apache Flink makes use of Akka model [4] for distributed communication. It is a very popular communication toolkit that has been used to build Apache Spark as well. Akka implements the actor system. For Apache Flink, the actors are JobManagers, TaskManagers and the JobClient. Actors have the following characteristics.

- Actors are isolated from each other and each of them run of separates threads even though resources are shared for scheduling, configuration and logging.
- An actor object maintains information about the state and behavior of an actor. An actor can be in any
one of the states that have been defined by the user. The behavior of an actor when it receives a request or message is dependent on the current state of the behavior and based on the behavior of an actor, the subsequent state is also set by this actors thread.

- Actors communicate between each other asynchronously and receive messages in a place called as mailbox. It processes messages sequentially and therefore there is no necessity for locks.

C. Job Execution Process

In a job execution process [5], the Job Manager is started first and then the Task Manager registers itself with the Job Manager. JobClient, which is the data stream source will supply the data in the form of a job graph to the Job Manager. A job graph is the complete job that Flink will run for this execution. From the job graph, an execution graph is created and based on the execution graph the job manager schedules jobs among available task managers and also allocated execution slots on them.

A sample job execution process is shown here [5]. First, a FLink program submits a job to the job client and that forwards it to the job manager. The job manager returns a success message on receiving the job. The job manager submits tasks to available task managers in the actor system. Once the task manager completes its task, it returns a UpdateTaskExecutionState result. Based on the results from the task managers, the job manager updates the job client with a JobResultSuccess parameters to indicate whether the job was completed successfully.

D. Correctness of Data

In a distributed system, consistency or correctness of data [6] is highly critical for a reliable system. Consistency can fall under the following three categories.

- **At most once** - The data that is being streamed from the source may or may not reach its destination. There is no guarantee if the data that reaches the destination is genuine or is in order in which it was streamed from the source.

- **At least once** - This method guarantees that the data that has been streamed from the source will definitely reach its destination and that the accumulated data can never be less than what was transmitted. Duplicates may be present in the data that was received but it cannot be lost in transmission.

- **Exactly once** - In a scenario where there has been no failure, the data received at the destination will definitely be the same as how it was transmitted. There will be no less or more data than that was sent from the source. Implementing a mechanism like this is expensive but Flinks checkpoint [7] mechanism is built on top this and therefore, even during failures the state of the actors is maintained.

E. Event Time

An event time in a distributed system refers to the exact time at which any particular event has happened. Flink programs which are based on event time specify something called as watermarks[8]. These are nothing but markers based on timestamps which are embedded in the data that is streamed from the data source. Watermarks indicate periodic sessions based on a time window and are in order, even if the data between watermarks are not in order. Watermarks can be embedded in a streaming source by making use of timestamp assigners or watermark generators which Flink provides.

III. COMPLEX EVENT PROCESSING

A. Cayuga

The paper [9] is about a complex event processing system based on a SQL like query model. The query language described in the paper is as shown in listing 1.

Listing 1. CEP Query

```
SELECT attr FROM stream_exp PUBLISH out_stream
```

The language can handle input events by applying FILTER, NEXT or FOLD operations and it provides the capability of using decorators to avoid reference ambiguities in self-joins or when referencing different streams within the query.

Later, the paper talks about the system architecture where there are event receivers running in multiple threads, a query engine running in a single thread and client notifiers which notify the subscribers of the output stream or result.

B. Apache Flink - CEP Library

Stream processing [19] is used when there is a need to process data in real-time but providing very low latency is not the main goal whereas Complex Event Processing is targeted towards finding and matching patterns with very low latency. There are some use cases where we need both i.e. Stream processing complex events with very low latency. Some of them are stock market trading, real time in-network monitoring, credit card fraud detection and much more.

To address this exact need, Apache Flink was developed to process data in actual real-time with very low latency. At present it provides a Complex Event Processing (CEP) library [10] which has the following features built into something called as Pattern API. Exploring the capabilities of this library and integrating it into the iQdep system for in-network complex event processing is the goal of this project.

A pattern can be identified by naming each sub pattern as a stage. When each stage is satisfied, then next stage is awaited and so on. The list of functions supported by the API is described here [10].

The pattern API can be used to filter events from a stream, define patterns, sub-patterns or sequences (NEXT (strict contiguity), FOLLOWED BY (non-strict contiguity : any number of events can happen in-between the two events that are compared) and also grouping of events by a logical AND operator or by making use of the OR operator whenever the comparison is mutually exclusive. In addition to this, it
also provides temporal operators like WITHIN which is used to define a time window for a defined pattern. Using these operators, several levels or stages of patterns can be written and highly complex events can be defined and matched over any data stream.

The core component of this CEP library is a NFA compiler which is strongly based on this paper [16]. Contributors of this project claim to be currently working on developing a more efficient way to represent CEP patterns with an SQL-like language called as StreamSQL [17] which is to be built of top of Flink’s table API.

The CEP library is designed in a way that lambda expressions can be used to define patterns, write customized filter conditions and also specify how the output of the pattern-matched event stream should be handled. If some patterns are not matched within a given time windows, then timed out patterns can also be handled. The output of the pattern matching can be another data stream which can act as an event source or it can be written to a file. Some complex event models have been discussed in section VI.

IV. DISTRIBUTED DATAFLOW

The paper [11] is about distributed data flow in data-parallel applications. The distributed data flow is in the form of a directed acyclic graph (DAG) which contains vertices that are to be computed on available resources. The main problem Dryad is aiming to solve is the difficulty of developing distributed and concurrent applications. Dryad is aimed at helping developers concentrate more on the dataflow and computation rather than developing the internal working of the system itself. It gives them the flexibility to finely control the applications dataflow graph and also helps them to optimize their design and computation.

Dryad discusses the various operations or transformations that can be done on the data during the dataflow and the authors demonstrate this with help of two experiments, namely optimizing execution of SQL query and also a data mining task on a large set of data. Dryad provides a great infrastructure in that sense and a lot applications have been developed on top of this. Some of those described at the end of the paper are:

A scripting language called Nebula, SQL Server Integration Services and distributed SQL queries.

V. ADAPTING CEP TO iQDEP

Complex event processing (CEP) can defined as

- Event
- Condition
- Action

Given an occurrence of an event, when a condition is satisfied then an action can be taken.

Based on the related work described above and the previous work done [12] for iQdep, the CEP operators that can be identified overall is as follows. Based on the operators, a query language can be developed to satisfy iQdeps requirements.

Complex events can be formed using event based operators shown below.

V. COMPLEX EVENT MODELS

The queries will be eventually converted in to directed acyclic graphs (DAG) and these have been modeled as discussed in the scenarios described in this section. These models form the basis for expressing complex events and can be visualized as shown below.

- **Scenario 1** - A user needs to check whether temperature sensor readings or smoke sensor readings are greater than some threshold value and if so, trigger a fire alarm. This can be modeled as an event graph shown below.

![Scenario 1](image1)

ES1 represents the temperature event stream and ES2 is the smoke event stream. If ES1s event is greater than threshold value of 30 or if ES2s event is greater than threshold value of 10, then trigger the fire alarm.
- **Scenario 2** Suppose a user needs to check whether both temperature sensor readings and the smoke sensor readings are above some threshold value within a time window of $T$ seconds, which is based on the event-time, then trigger the fire alarm.

![Scenario 2](image)

ES1 represents the temperature event stream and ES2 is the smoke event stream. If ES1's event is greater than threshold value of 30 and if ES2's event is greater than threshold value of 10 within a time window of 30 seconds, then trigger the fire alarm.

- **Scenario 3** Suppose a user needs to check if there are two consecutive temperature events above a threshold value which is followed by a smoke event which is also above a threshold value, the trigger the fire alarm.

![Scenario 3](image)

ES1 represents the temperature event stream and ES2 is the smoke event stream. If ES1's event E1 happened right before event E2 and then matched pattern T1 is FOLLOWED BY smoke event matched pattern S1, then trigger a fire alarm.

Similar to the above scenarios, more complex events can be created and the feasibility of mapping these models to a SDN-inspired architecture is discussed in the next section.

VII. SOFTWARE DEFINED NETWORKING

Software Defined Networking (SDN) [13] is used to maintain and inject policies easily across a network of switches and also efficiently manage data flow across nodes in a network. The reasons behind making use of SDN for our use case are

- Dynamically control dataflow based on the application’s requirements
- Re-use of existing framework to monitor and maintain network topology
- A clear separation between control plane and data plane
- It can support heterogeneous devices present in the network unlike frameworks like SensorML [18]

As we can realize the critical features it provides, we will discuss how we plan to use a SDN-based framework for iQdeps complex event processing needs.

A. Execution Graph

A complex event query, as we have seen in the earlier sections, is converted to a DAG. This is then converted into an execution graph based on the available resources. The execution graph is created based on the where each vertex in the DAG has to run on a node in the actual network. For example, two vertices of the DAG can also run on the same node.

The complex event models that were discussed in section VI has been mapped on to a network of switches and hosts in a virtual SDN like environment. This virtual environment has created with the help of mininet. The simulation of the topology and mapping the execution graph is discussed in the next two sections.

B. SDN topology simulation on Mininet

Using mininet, a custom topology of 5 switches and 5 hosts were initially set up. The type of SDN controller used is the Floodlight controller [20] and type of switch is Open vSwitch (OVS). The links between the switches and the hosts are bi-directional links. After the topology is created, the floodlight controller identifies the switches present in the network automatically by sending LLDP messages using OpenFlow protocol. And when the hosts ping each other, we can notice that the floodlight controller starts keeping track of the hosts as well. This can be visualized with the floodlight controller’s Web GUI.

C. Mapping the Execution Graph

Once the topology is set up, based on the CEP model shown earlier, the dataflow between the nodes need to configured. The CEP model that we will be discussing will be based on Scenario 1 presented in section VI.

The overall dataflow is modeled in a way that the host connected to the switches is not aware of the where the data stream needs to be sent but the switch is configured with flow rules injected from the SDN controller. The flow rules are used to guide the packets to their intended destination. The compute
functions that need to be on the nodes is set manually for now. Going forward, this will be injected into the hosts using the SDN controller by modifying the OpenFlow messages if viable or by creating a new protocol between the controller and switches that help inject the function into the hosts.

For scenario 1, the topology that was created using mininet is show in figure 4. The Host 1 (IP - 10.0.0.1) and Host 3 (IP: 10.0.0.3) in the topology can be made the event sources with Host 1 generating the temperature event stream (ES1) and Host 2 generating the sensor event stream (ES2). Host 2 and Host 4 can act as compute nodes which check if the event stream values are above the set threshold values. Host 5 will act as the OR node which triggers the actuator, which is a fire alarm in this scenario. All the host are independent actors in the computation. After injecting the switches with flow rules shown in table III, they will be able to forward packets based on the dataflow graph.

The hosts are connected to their respective switches on OpenFlow (OF) port 1. Other OF port numbers are used to connect to other switches only. Each flow rule for a switch can be read as, for any packet that is received at IN_PORT, it needs to send it out on the output OF port with a new IPv4 destination, if needed. The actual flow rules in the switches is shown in table III.

![Floodlight Controller Topology for Scenario 1](image)

**TABLE III. ACTUAL OPEN vSWITCH FLOW RULES**

<table>
<thead>
<tr>
<th>SWITCH</th>
<th>FLOW RULE</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>n_packets=0, n_bytes=0, idle_age=4, ip.in_port=1 actions=output:2</td>
</tr>
<tr>
<td>S2</td>
<td>n_packets=0, n_bytes=0, idle_age=107, ip.in_port=2 actions=mod_nw_dst:10.0.0.2, output:4</td>
</tr>
<tr>
<td>S3</td>
<td>n_packets=0, n_bytes=0, idle_age=107, ip.in_port=1 actions=output:3</td>
</tr>
<tr>
<td>S3</td>
<td>n_packets=0, n_bytes=0, idle_age=338, ip.in_port=2 actions=mod_nw_dst:10.0.0.3, output:12</td>
</tr>
</tbody>
</table>

For scenario 1, the data from Host 1 and Host 3 will be sent to their respective switches. This is done by making the host to broadcast its data packets to the switch. As soon as the switches S1 and S3 receive the packet, it matches the corresponding the flow rule and forward the packet out of the given OpenFlow port connecting to switches S2 and S4 respectively. The flow rules on S2 and S4 modify the destination of the packets to the hosts that it is connected to and forwards the packets out. The compute functions on H2 and H4 receive the packet from H1 and H3 correspondingly. If the condition is satisfied at the nodes, then H2 and H4 will send another packet to S3 and S4, from which the packets are sent to switch S5. All packets received at S5 will be sent to host H5 which can act as an actuator that triggers the fire alarm.

Therefore, similar to above scenario which has been modeled using SDN, other complex scenarios can be set up and the dataflow can be controlled using flow rules which are injected into the switches. Flow rules can get more complex based on the graphs being mapped. So far, we have identified that flow rules can modify packet IP headers and also MAC headers. There can be changes or modifications based on the OpenFlow standard specification [14].

**VIII. CONCLUSION**

Adapting event detection ideas from various frameworks can provide us with a broader perspective on the complex event processing needs of IoT networks. The complex event models can be explored further and this can help achieve in-network processing of events and thereby enable complex event detection. The feasibility of developing a framework for in-network processing has been well established with this research and the scope for future work is immense.

**IX. FUTURE WORK**

Some of the expected future work is listed below:

- Complex event processing language can be developed with inspiration from Flink’s CEP library.
- Software Agents can be used to inject the compute functions using protocols integrated with SDN.
- An inventory service can be developed to maintain the list of available nodes in the network along with their specification. This information can be retrieved based on the SDN controller’s topology.
- Job graphs and execution graphs can be created based on how different stream processing engines are designed.
- Based on the execution graphs, the dataflow between nodes can be configured through a REST client which interacts with the SDN controller.
- Spatial operators can be integrated into the CEP language as required.

**ACKNOWLEDGMENT**

I would like to thank Dr. Peizhao Hu [15] of Rochester Institute of Technology for giving me the opportunity to work on his grand vision of enabling in-network event processing in IoT devices.
X. REFERENCES


