CS 237 Project Report :
Event Framework for TIPPERS

Group #15:
Quishi Bai (95722145)
Avinash Kumar (68722842)
Jonathan Harijanto (37128708)

June 12, 2018

1 Introduction

In recent years, we have witnessed a rapid growth in the development of Big Data applications. This partly has been because of the rise in the data available to us and its ease of access through the internet. With the advent of cheap micro-sensor technology, data from various sensors such as Bluetooth beacons or thermometers is flowing in at a rate of thousands of readings per second. Therefore, data from IoT devices requires specific handling and usage. One notable difference between IoT's sensor data and typical data is that these data don't sit on persistent storage because they have to be processed and directed to designated applications in a real-time manner. Due to this reason, there is a need to build distributed applications which can handle the velocity of the incoming data and be able to direct queries to it without the need for disk storage.

In our Informatics and Computer Science department at the University of California, Irvine, there is an ongoing project called TIPPERS (Testbed for IoT-based Privacy-Preserving pERvasive Spaces) that revolves around the field IoT. Based on the description from the authors [9], TIPPERS is a data management middleware that supports: (1) a dynamic representation of a smart space by integrating diverse sensors and cameras to create a semantic abstraction of sensor data and represent the data at the conceptual level, and (2) policies for capturing, analyzing, sharing, and retaining data from Plug-n-Play architecture to support integration of different privacy technologies encrypted data representation, differential privacy, and CMU's IoT and IRR technologies. One observation that we noticed inside TIPPERS environment is that the lacks an eventing framework. In other words, TIPPERS has all the data sensors from all the IoT devices inside a large school building but doesn't allow the users to utilize the data to receive an update on a particular event that happens in the building automatedly. Currently, there is a similar application in TIPPERS called Concierge that allows user to receive a notification about a room, locate a person or a room, and search for directories. However, this app has a backend which does all the work of filtering data and finding the result. Motivated by this issue, we intend to make this process more declarative and abstract, such that users need not write custom solutions for every need.

For our class project, we develop an event framework, where users can subscribe to a specific event such as “temperature is above X degrees” and receive a notification everytime that event happens. From a high-level perspective, our framework allows users to view all the accessible IoT's sensor data from TIPPERS and define specific conditions from these data according to their interest. Then, every time the conditions are satisfied, our module will send these users an event. From there, once the users receive the update, they can do a variety of tasks such as lowering the AC if the event is "temperature is above X degrees".

Our system consists of two essential pieces: a rules engine and a Pub/Sub framework. In section 2, we provide a literature survey as an effort to acquaint us with the various

state-of-the-art software solutions available for us to build the system. Section 3 provides a further in-depth discussion about the design of our framework prototype and how each piece synchronized to make eventing possible. Section 4 covers the issue we encountered during the development and the workaround we did to resolve the problem. Lastly, section 5 concludes everything that we did in this project and introduces a possible extension to it.

2 Research Approach

Our event framework consists of different components that required integration to operate precisely. Therefore, we spent some time researching on the various state-of-the-art software solutions that are available for free. Furthermore, we also investigated different properties that each component needs to have so that it could function maximally. Starting from a stream processing component, we extracted some useful information that discusses the requirements of a system to possess to excel at real-time stream processing applications. The requirements are as follows [1] :

1. The system must be able to process messages in-stream without the need to store them to perform any operation. Also, the system should follow an active (event-based model) rather than a passive (polling-based model) approach.
2. The system should support a high-level StreamSQL and have extensible stream-oriented operators.
3. It should provide resiliency against stream imperfections like missing or out-of-order data.
4. The system must guarantee predictable and repeatable outcomes. Time-ordered processing of messages should be maintained throughout the processing pipeline regardless of the order in which they arrive.
5. For stream processing applications, comparing present to past is a common task. So, the system must provide careful management of stored state and ability to combine it with live streaming data.
6. It should guarantee availability and data integrity despite failures.
7. It must be able to do distributed computing to achieve incremental scalability. Ideally, the distribution should be automatic and transparent.
8. It should have a highly-optimized, minimal-overhead execution engine to deliver real-time response for high-volume applications.

In our discovery, we found three examples of software systems which can handle high-volume low-latency streaming applications which are DBMS, rules engines and Stream Processing Engines (SPE) [1]. Thus, we decided to dive deeper into some of the stream processing software that are well-known, and, at the same time, peak our interest.

- **Apache Storm** – It focuses on extremely low latency and provides near real-time processing. It can handle vast quantities of data with and deliver results with lower latency than other solutions. However, Storm guarantees at-least-once handling. For

exactly once, an extension called Trident is available, but it tends to slow down Apache Storm due to the modifications were done to support exactly-once.

- **Apache Samza** – It is a stream processing system that is intricately tied to Apache Kafka. It partitions the stream based on a key and the messages in a partition are totally ordered. It also provides a straightforward and inexpensive multi-subscriber model to each data partition. All output (including intermediate results) is forwarded to Kafka and can be consumed by stages downstream.
- **Apache Spark** – It processes all data in-memory and interacts with the storage layer to initially load the data into memory and at the end to persist the final results. For stream processing capabilities, one must use Spark Streaming.
- **Apache Flink** – It can be used for Complex Event Processing. Complex event processing (CEP) addresses the problem of matching continuously incoming events against a pattern. In contrast to traditional DBMSs where a query is executed on stored data, CEP runs data on a stored query.

Once we gained a better understanding of real-time stream processing systems, we moved forward to analyze about rules engine for an event stream processing. Rule based systems (RBS) work as an observer on event streams and when a specific pattern is observed, it triggers actions. These systems generally compile large number of rules to a graph and use graph optimization algorithms to reduce workload of the system. Apache DROOLs is one of the famous centralized RBS. It is designed to run on a single machine as it uses shared memory model and all computation is done in the same memory. [2] and [3] discuss the research been done to distribute the computing nodes of RBS. However, the systems discussed still work on a centralized memory shared by all nodes. This approach again has two shortcomings - 1. Scalability i.e. data size and rule size can be a bottleneck, 2. Performance suffers because of I/O bottleneck. Therefore, there is a need to make a distributed RBS using stream processing engines discussed previously like Apache Spark Streaming, Apache Flink etc. [4] presents an approach to adapt RBS with Spark Streaming. It has majorly three components - Rule Base (which contains rules), Inference Engine (which infers facts) and Working Memory (which stores temporary assertions). The system matches rules against working memory and figures out what rules need to be activated. Rete algorithm is used to achieve this process efficiently. Rete algorithm compiles rule sets into data structures called Rete graph. The experiments on DRESS show it to be more efficient than Drools even in a single machine due to inflexible memory model in terms of garbage collection. On a cluster and large data sets, DRESS comprehensively performs better than Drools. There are commercial rules engine also out there for IOT. Losant has a rule engine which has a workflow based interface to build rules. Waylay is another rule engine which belongs to inference based rule engines and uses Bayesian networks for rule matching.

The last step we did was to investigate the Pub/Sub service. We discovered that there are different types of architecture from a centralized broker model, where multiple servers are connected to each other and form a network, and subscribers are connected to a broker that

could be transported messages from the publishers [5]. There is also a peer-to-peer architecture, in which there is no intermediary system exist between publishers and subscribers. Instead, each node can act as a publisher, subscriber, or a broker [6]. By research, we found that a broker model is more popular in the real world. Currently, numerous frameworks are optimized to offer a publish-subscribe service that implements centralized broker overlay architecture and has a minimum hardware resource requirement. The two most popular frameworks that are being widely used nowadays are:

- **Google Cloud Pub/Sub** – A publisher creates a topic and sends messages to forwarders where messages are stored in one of the clusters inside Google's data center. A subscriber connects to Cloud Pub/Sub, and a router will direct the subscriber to a data center that has the lowest latency [7]. Once the connection between a subscriber and a data center is set, the router provides a list of recommended subscribing forwarders to connect. A subscriber has the options to have a pull or push subscription model. In a pull method, a subscriber is connected to subscribing forwarder and request one or more publishing forwarders that have the messages for a relevant topic. When a publishing forwarder receives a request from a subscribing forwarder, it will send the message and wait for a reply signal. Once the subscriber has received the message, it will send a reply signal to indicate the forwarder that a message can be removed from the cluster storage. On the other hand, in a push method, a subscriber directly received messages from publishing forwarders that have the messages for a relevant topic through an endpoint URL that supports HTTP POST request [7]. Similarly, the forwarder is waiting for a reply signal, and once the subscriber has received the message, it will send a reply signal.
- **Apache Kafka** – At its core, it implements centralized brokers overlay architecture. Inside a large cluster, multiple brokers are connected to each other and they are all running Kafka. Each broker contains multiple topics, and some topics have multiple partitions. Each partition in different brokers may contain replicated message like from the other brokers. The purpose of having multiple partitions on the same topic in a broker is for load balancing. Furthermore, the partitions allow multiple subscribers to read from a topic in parallel which results in very high message processing throughput [8]. A publisher creates a topic and pushes messages to that topic in each broker. A subscriber registers to one or more topics from the brokers. After that, Kafka will assign that subscriber to a set of partitions (in every topic) to pull the messages from for every topic. Another interesting feature from the framework is that a subscriber can join a group called a consumer group which is a set of subscribers that are registering to a specific topic [8]. Every message that just entered a partition in a particular cluster will be set with an expiration time. Once the time expires, the message for a topic will be removed from every partition at every broker [8].

3 Architecture

The architecture of our event framework prototype looks as follows:

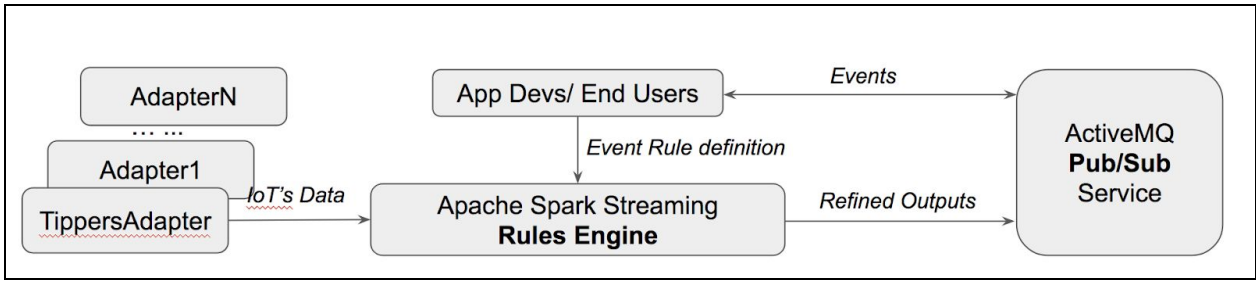

Figure 1. Prototype of Architecture

The core of our system is at “Rules Engine” + “Pub/Sub Service” component; however, to make our project more complete and also for the convenience of purpose, we also developed one Stream Adapter for TIPPERS system “TippersAdapter”. We use “TippersAdapter” to simulate the stream data coming into our Rules Engine in a real-time manner. In a real world scenario, if the IoT sensor network supports pushing stream data real time into some output port, this adapter module will be very simple with just some formatting or redirection work only.

The core module of our system is the Rules Engine, which processes the input stream data in a distributed environment and apply UDF Event Rules to data records and finally triggers events to be published to the next Pub/Sub Service module. And our Rules Engine logic runs on Apache Spark Streaming platform who provides us the built-in distributed environment and scales our framework greatly.

Once the Rules Engine finds data records that match the Event Rules defined by users’ UDFs, it will publish the events to a Pub/Sub Service by RESTful API. And the Pub/Sub Service will push the events in a predefined Topic channel to the Subscriber client who acts as an agent and triggers the application logic implemented in users’ applications.

Next, we will introduce each module.

3.1 TippersAdapter

TippersAdapter will pull the real time IoT sensor data from TIPPERS’ database and serve as Socket server for clients to connect to and stream this data.

As stated above, for experiments purpose, this adapter can also plays a role like a simulator who simulates every ***tickFrequency*** (e.g. 5) seconds and pulls the Tippers data recent ***tickFrequency*** * ***frequencyScale*** (e.g. 5*12) seconds. Of course the start date of this simulation data is not today, it's Nov 08, 2017.

In another word, once the TippersAdapter starts, it will start a Socket server at localhost:9999 port. Any client connects to it can get the streaming data from Tippers database since Nov 08, 2017, and every 5 seconds, it will get the next 1 minute new data.

Current TippersAdapter supports all 5 types of sensor data streams as following:

- Physical sensor type: WeMo, Thermometer, and WifiAP
- Semantic sensor type: Presence and Occupancy

The output example data stream looks like following:

```
ThermometerObservation|533528fb-d4d8-4fdb-84a2-97d37e3874b8|53|2017-11-08
07:50:00|af3d5dfd_fdb7_4d47_9baa_ee2254a27c02
… …
PRESENCE|10f92ff6-d07e-4acc-835d-ac45745331e3|9304bdc5_e006_41c2_8f80_97d8c0422e19|1
100|2017-11-08 07:48:00|vSensor1
… …
OCCUPANCY|d072b58d-6b24-43d9-8b33-012ec9972577|4100_5|45|2017-11-08
07:48:00|vSensor2
… …
WiFiAPObservation|8751aa81-1e07-4d5f-bb0e-5969d5896916|b38bd906-8fc1-4171-9d45-e16df
665f189|2017-11-08 07:50:00|770322d8_7899_4052_917e_a8ba7a5fec0f
… …
```

Table 1. Sample output data from TippersAdapter

3.2 Rules Engine

Rules Engine is the core module of this project and also runs the main logic of our whole event framework. Rules Engine is based on Apache Spark Streaming who will listen to the localhost:9999 port and process all the streaming data received from this port.

Once started, the Rules Engine will process the incoming IoT data stream in a real-time manner. It loads the Event Rules defined by user-defined function (UDF) applies the event rules by filtering the IoT data. And finally publishes a list of records that satisfy the event rules to Pub/Sub Service as corresponding Topics as the rule id or rule name. Then the user defined application Subscribers will get the event notices from the Pub/Sub service.

The Rules Engine supports both the combination of predicates among streams and also conjunction predicates within a record.

3.2.1 Combination of predicates among streams

In terms of combination of predicates among streams, here's an example, some user wants to define an event rule as following:

presence.userId = "X"

AND ***occupancy.location = “2065”*** ***AND*** ***occupancy.NumberOfPeople = 0***

Table 2. Example 1, combination of predicates among streams

The meaning of this event rule is “Person X is in building AND Room 2065 is empty”.

3.2.2 Conjunction predicates within a record

In terms of conjunction predicates within a record, here’s an example, some user wants to define an event rule as following:

presence.userId = “X” AND presence.location like “20%”

Table 3. Example 2, conjunction predicates within a record

The meaning of this event rule is “Person X is present on the 2nd floor”.

3.2.3 Workflow of Rules Engine

Rules Engine processes the input data streams by the following 6 steps:

1. Load the Event Rules defined by the user applications.
2. Map the whole stream to different data source streams.
3. Apply the Event Rules' predicates to each record in each stream.
4. Reduce the predicate-mapping-to-records of each Rule.
5. Apply the Merger functions defined by Event Rules to all the predicate-mapping-to-records for each Rule.
6. Publish the records to Apache ActiveMQ with topicId = RuleId.

To be more specific, the following figure shows the data flow inside the Rules Engine workflow:

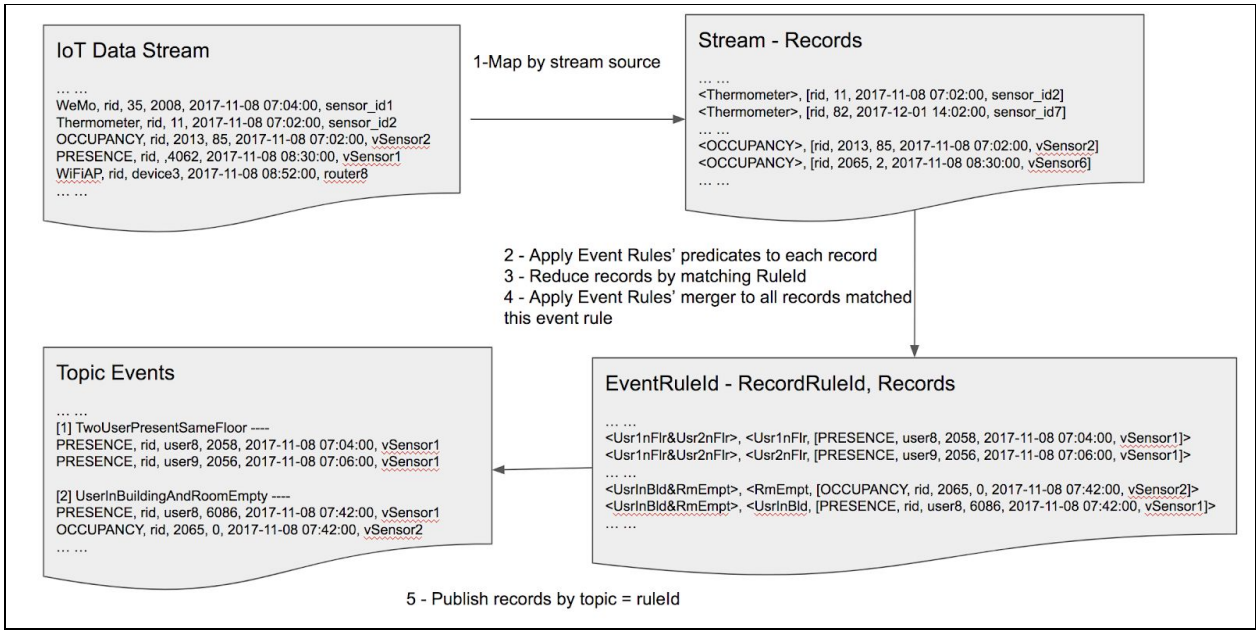

Figure 2. Data flow of Rules Engine

3.2.4 User Defined Function - UDF

Another important design of this Rules Engine is the UDF interfaces. There are two principles that need to be followed when designing the UDF interfaces are: (1) The interfaces need to be simple for users to implement, the best way is to provide a declarative language such as SQL, JSON, etc. (2) The interfaces need to support complex semantic flexibility that allow users to define "Combination of predicates among streams" and "Conjunction predicates within a record".

To achieve the above goals, we designed a hierarchical set of interfaces for users easy to define event rules with the above advantages.

(1) The root of an event rule is a "Event Rule" instance.
(2) Each "Event Rule" instance can contain a list of arbitrary number of "Record Rule" instances and one instance of "Rule Merger".
(3) Each "Record Rule" instance can contain a list of arbitrary number of "Predicate" instances.

Take "Table 2." Event Rule example as illustration, the following figure gives the definition of a UDF event rule using our interfaces.

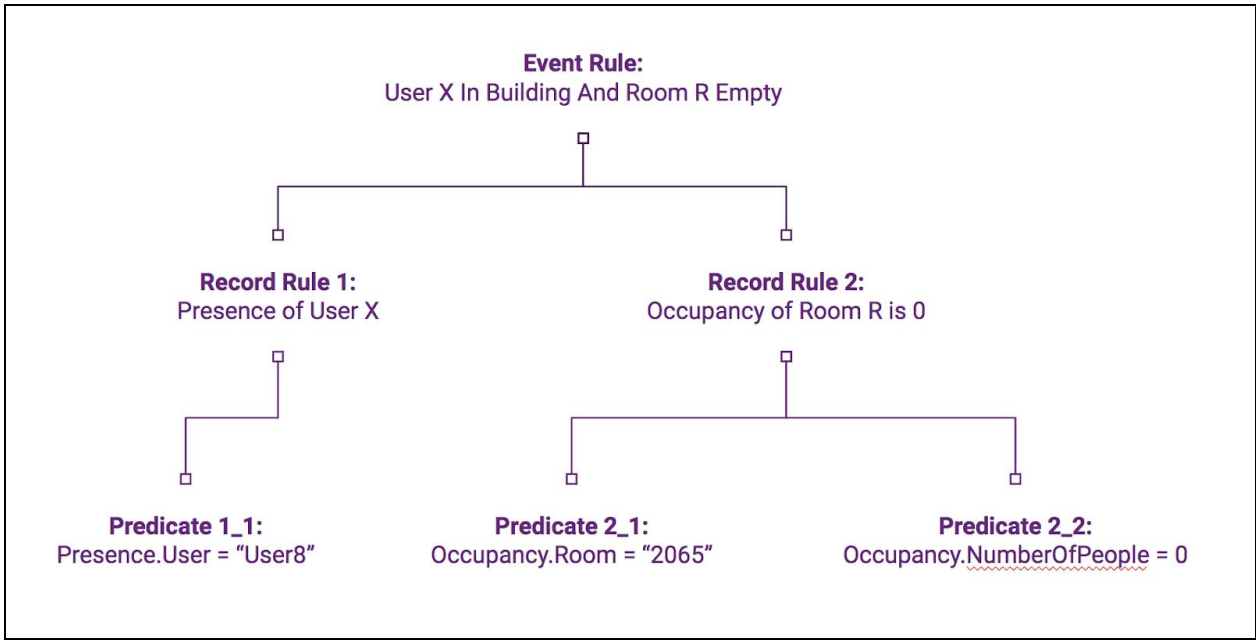

Figure 3. Event Rule definition using UDF interfaces

The UDF code for this example Event Rule definition looks like as following:

```
public class UserInBuildingAndRoomEmpty extends IEventRule {

   String user, room;

   UserInBuildingAndRoomEmpty(String user, String room) {
      this.user = user;
      this.room = room;
   }

   @Override
   String ruleId() {
      return userName() + "-" + topicName();
   }

   @Override
   String userName() {
      return "test";
   }

   @Override
   String topicName() {
      return "UserInBuildingAndRoomEmpty";
   }
```

```
@Override
List<IRecordRule> recordRuleList() {

    List<IRecordRule> recordRuleList = new ArrayList<>();

    RecordRule r_user_in_building = new RecordRule(this);
    r_user_in_building.id = "r_user_in_building";
    r_user_in_building.stream = "PRESENCE";

    Predicate p_user = new Predicate(r_user_in_building);
    p_user.id = "p_user";
    p_user.attribute = "semantic_entity_id";
    p_user.attributeType = AttributeType.STRING;
    p_user.operator = Operators.EQUAL;
    p_user.valueString = this.user;

    r_user_in_building.predicateList.add(p_user);

    RecordRule r_room_empty = new RecordRule(this);
    r_room_empty.id = "r_room_empty";
    r_room_empty.stream = "OCCUPANCY";

    Predicate p_room = new Predicate(r_room_empty);
    p_room.id = "p_room";
    p_room.attribute = "semantic_entity_id";
    p_room.attributeType = AttributeType.STRING;
    p_room.operator = Operators.EQUAL;
    p_room.valueString = this.room;

    Predicate p_empty = new Predicate(r_room_empty);
    p_empty.id = "p_empty";
    p_empty.attribute = "occupancy";
    p_empty.attributeType = AttributeType.INT;
    p_empty.operator = Operators.EQUAL;
    p_empty.valueInt = 0;

    r_room_empty.predicateList.add(p_room);
    r_room_empty.predicateList.add(p_empty);

    recordRuleList.add(r_user_in_building);
    recordRuleList.add(r_room_empty);

    return recordRuleList;
}

@Override
IRuleMerger merger() {
```

```
        return new AndTwoMerger(this, "r_user_in_building", "r_room_empty");
    }
}
```

Table 4. Example code of Event Rule definition using UDF interfaces

3.3 Pub/Sub Service

Actually, in terms of Pub/Sub Service, our project doesn't not limit the choice based on our design with general purpose in mind. But for the purpose of prototype demo, we explored two types of services: Google Cloud Pub/Sub Service and Apache ActiveMQ JMS Pub/Sub Middleware. And for the convenience of demo and experiments, we used the Apache ActiveMQ as the prototype Pub/Sub Service provider.

To the respective of Apache ActiveMQ, our Rules Engine acts as the role a publisher who will publish events to different topics based on the Event Rules definitions and push the topic events with data records as message bodies to the ActiveMQ server.
ActiveMQ server receives a list of records that satisfies a specific rule from our Rules Engine and then publish to the topics' Subscribers.

Also for the purpose of demo, we developed an example Subscriber application who listens to the example Event Rules' topics and output the events' data records to the console.

Still take the the example Event Rules as illustration, the following table shows the output results of our Demo Subscriber:

```
========== [Message Received] Topic : test-TwoUserPresentSameFloor -------->
Message: PRESENCE|9dc01361-d8e6-4562-83d0-f1872a441c41|user8|2058|2017-11-08
07:04:00|vSensor1
========== [Message Received] Topic : test-TwoUserPresentSameFloor -------->
Message: PRESENCE|a5905739-d2a8-4906-a0bb-2b320678a0d8|user9|2056|2017-11-08
07:06:00|vSensor1
========== [Message Received] Topic : test-UserInBuildingAndRoomEmpty
--------> Message:
OCCUPANCY|e4fa81e9-922c-43bf-9643-31ad955c43f1|2065|0|2017-11-08
07:42:00|vSensor2
========== [Message Received] Topic : test-UserInBuildingAndRoomEmpty
--------> Message:
PRESENCE|05b42de7-6403-4cf3-b08d-a8881445cc9b|user8|6086|2017-11-08
07:42:00|vSensor1
```

Table 5. Sample output of the example Subscriber

4 Challenges

We encountered multiple implementation issues with Cloud Pub/Sub throughout the development stage of the framework. The problems can be categorized into three classes: authentication, runtime, and dependency. The first issue is regarding authentication process with Google. In order to invoke Cloud Pub/Sub functionalities directly in our code, we need to install the API client library and set up authentication. The installation part was simple because we only need to include a snippet of dependency code into our pom.xml file. On the other hand, the authentication step was not as straightforward as what listed in the tutorial page. We were required to create a service account key in the form of JSON file and assign an environmental variable to it. After we followed all the steps, we tried to run our program and got an error message saying that the authentication is still failing. Thus, we had to spend many hours trying to debug and repeat the authentication process repeated times. This specific issue was resolved when we discovered that one must download Cloud SDK -- Command-line interfaces for Google Cloud Platform products and services -- into his/ her local machine, then install and login using Google account that is subscribed to the Pub/Sub framework. However, this solution leads to another issue that is one needs to login to his/ her teammates' local machine so that each of the members could try to run the code.

The next problem that we faced was associated with a runtime error. In our project survey document, we briefly mentioned that there is a tendency to use Cloud Pub/Sub instead of Apache Kafka due to its supports toward push subscription -- Pub/Sub framework initiates a request to the subscriber's application to deliver messages. We noticed that there is one crucial requirement to push messages, which is a publicly accessible HTTPS server that handles POST requests and has both a validated domain and a validated SSL certificate. Once the requirement is fulfilled, then we are allowed to register the endpoint domain with the GCP project. Fortunately, we were able to find an alternative solution that could bypass all the requirement by using App Engine -- Google's web framework platform for hosting web applications through Cloud [10]. However, we didn't expect that App Engine has its own problem. There are various ways to run the web server using different programming language. In our first trial, we tried to run the App Engine using a sample code that is written in Java. Unfortunately, we consistently failed in building a project due to an issue caused by maven every time we tried to deploy the website. After repeated trial and error, we decided to workaround the problem by using an alternative sample source code that is written in Python. We were perplexed that the code instantly compiled and ran. Regardless, we were relieved that finally we had a running HTTPS server that could receive a sample push-based message.

After all the hard work we did to run the Pub/Sub feature correctly, where publishers, subscribers, and topics could be configured remotely from our code, a new problem arose when we integrated the rules engine code with the Pub/Sub code. We were faced with a situation where Cloud API Client Library requires too many dependencies that we were not familiar with.

As a result, we had compilation errors many times and needed to continually add a new dependency for every new error introduced. Furthermore, we also discovered that some dependencies that were being changed to make Cloud Pub/Sub ran correctly lead to different errors on our Spark Streaming (Figure 2) because both frameworks required a different version of library dependencies. Due to the library dependencies issue that kept occurring, we took a bold decision to use an entirely new Pub/Sub framework called Apache ActiveMQ. This new framework works the same way as Cloud Pub/Sub, but it is much easier to configure because we didn't face any authentication and dependency problems at all. However, one disadvantage of ActiveMQ that could not be overcome is that the framework uses pull subscription method, where subscribers need to perform some actions to receive updates.

```
Exception in thread "main" java.lang.VerifyError: Bad type on operand stack
Exception Details:
  Location:
    com/google/api/AnnotationsProto.registerAllExtensions(Lcom/google/protobuf/ExtensionRegistryLite;)V @4: invokevirtual
  Reason:
    Type 'com/google/protobuf/GeneratedMessage$GeneratedExtension' (current frame, stack[1]) is not assignable to 'com/google/protobuf/ExtensionLite'
  Current Frame:
    bci: @4
    flags: { }
    locals: { 'com/google/protobuf/ExtensionRegistryLite' }
    stack: { 'com/google/protobuf/ExtensionRegistryLite', 'com/google/protobuf/GeneratedMessage$GeneratedExtension' }
  Bytecode:
    0x0000000: 2ab2 0003 b600 04b1

        at com.google.pubsub.v1.PubsubProto.<clinit>(PubsubProto.java:404)
        at com.google.pubsub.v1.PubsubMessage$AttributesDefaultEntryHolder.<clinit>(PubsubMessage.java:148)
        at com.google.pubsub.v1.PubsubMessage$Builder.internalGetAttributes(PubsubMessage.java:758)
        at com.google.pubsub.v1.PubsubMessage$Builder.buildPartial(PubsubMessage.java:622)
        at com.google.pubsub.v1.PubsubMessage$Builder.build(PubsubMessage.java:610)
        at cs237.PubSubFramework.publishMessages(PubSubFramework.java:59)
        at cs237.SparkAppJava.main(SparkAppJava.java:443)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:498)
        at org.apache.spark.deploy.JavaMainApplication.start(SparkApplication.scala:52)
        at org.apache.spark.deploy.SparkSubmit$.org$apache$spark$deploy$SparkSubmit$$runMain(SparkSubmit.scala:879)
        at org.apache.spark.deploy.SparkSubmit$.doRunMain$1(SparkSubmit.scala:197)
        at org.apache.spark.deploy.SparkSubmit$.submit(SparkSubmit.scala:227)
```

Figure 3: One of the dependency issues between Cloud Pub/Sub and Spark Stream

5 Conclusion

The report sums up the stages we went to develop the project. First, we analyzed the problem that occurs around campus and figured out how a middleware project could bring a positive impact to such issue. In our case, we discovered TIPPERS which is an ongoing middleware project in the Informatics and Computer Science department at the University of California, Irvine. TIPPERS is a great application that allows the users to interact with the Donald Bren Hall building through a web UI. However, it still missing an eventing application that enables users to request for a specific event in the building and to receive a continuous update everytime that particular event occurs. Therefore, we are motivated to develop an eventing framework that could solve the issue and submitted this idea as our term project. The development process started by designing the architecture of our framework, doing thorough research about each

component (streaming services, rules engine, and Pub/Sub service), and publishing a literature review about the research discoveries. Then, we started to code each component and integrate them at the end of the process. Few challenges were encountered throughout the whole process such as authentication issue with Google Pub/Sub service. Fortunately, we were able to work around every problem we faced accordingly.

TIPPERS is an ongoing project that still has lots of room for improvement. With further refinement, we believe that our event framework prototype could turn into a working, scalable application that could provide benefit to all the users. Thus, we are hoping to discuss and collaborate with the TIPPERS design team so that our application could be integrated into TIPPERS environment.

Several potential improvements of this project are as following:

(1) Provide more convenient user interface (Command Line / Web UI) to upload their UDF compiled jars to the system.
(2) Import some SQL parser or JSON parser to wrap current UDF interfaces as more standard declarative language.
(3) Research on how to support join operations among streams.

References

1. Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. 2005. The 8 requirements of real-time stream processing. SIGMOD Rec. 34, 4 (December 2005), 42-47. DOI: https://doi.org/10.1145/1107499.1107504
2. Christoph Nagl, Florian Rosenberg, and Schahram Dustdar. 2006. VIDRE--A Distributed Service-Oriented Business Rule Engine based on RuleML. In Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC '06). IEEE Computer Society, Washington, DC, USA, 35-44. DOI=http://dx.doi.org/10.1109/EDOC.2006.67
3. Jinghan Wang, Zhou Rui, Jing Li, and Guowei Wang. "A distributed rule engine based on message-passing model to deal with big data." *Lecture Notes on Software Engineering* 2, no. 3 (2014): 275.
4. Yi Chen and Behzad Bordbar. 2016. DRESS: a rule engine on spark for event stream processing. In Proceedings of the 3rd IEEE/ACM International Conference on Big Data Computing, Applications and Technologies (BDCAT '16). ACM, New York, NY, USA, 46-51. DOI: https://doi.org/10.1145/3006299.3006326
5. P.Th. Eugster, P.A. Felber, R. Guerraoui, and A.M. Kermarrec. The Many Faces of Publish/Subscribe. http://members.unine.ch/pascal.felber/publications/CS-03.pdf
6. Ying Liu and Beth Plale. Survey of Publish Subscribe Event Systems. https://www.cs.indiana.edu/pub/techreports/TR574.pdf
7. Anon. What Is Cloud Pub/Sub? Retrieved May 11, 2018 from https://cloud.google.com/pubsub/docs/overview
8. Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: a Distributed Messaging System for Log Processing. https://www.ece.cmu.edu/~ece845/docs/kafka.pdf
9. Mehrotra, S., Kobsa, A., Venkatasubramanian, N. and Rajagopalan, S. (2016). TIPPERS: A Privacy Cognizant IoT Environment. Pp.1-6.
10. Google. (2018). Google Cloud App Engine. [online] Available at: https://cloud.google.com/appengine/docs/flexible/java/concepts [Accessed 12 Jun. 2018].