E6889 Large-Scale Stream Processing Project Report

Implement the query-based network monitoring application (from Section 4.5, Exercise 2.)

Chenye Yang cy2540, Bingzhuo Wang bw2632, Zhuoyue Xing zx2269

1. Introduction

Generally, a typical network service provider can employ several techniques to track and identify the individual flow associated with the protocol, the source and destination IP addresses, as well as a set of packets and their sizes. With that information available, it is possible to provide live and continuous queries to keep monitoring the overall network resources. In light of that, this project gives attempts to generate individual flows, use a streaming system to process data and return some statistical analysis based on queries.

Specifically, we set up six queries of the following types and implement them in the streaming system:

- List protocols that are consuming more than H percent of the total external bandwidth over the last T time units.
- List the top-k most resource intensive protocols over the last T time units.
- List all protocols that are consuming more than X times the standard deviation of the average traffic consumption of all protocols over the last T time units.
- List IP addresses that are consuming more than H percent of the total external bandwidth over the last T time units.
- List the top-k most resource intensive IP addresses over the last T time units.
- List all IP addresses that are consuming more than X times the standard deviation of the average traffic consumption of all IP addresses over the last T time units.

According to the six types above, the queries are decided by the raw streaming data and four parameters: The percentage H of the total external bandwidth; the time duration of the last T time units; the number k of which it shows how many results should return; the multiplier X of the standard deviation of the average traffic consumption of all IP addresses or protocols. To change these four independent parameters, we are able to monitor the network condition and get statistical results dynamically.

2. System Architecture

2.1. Overview

The system is composed of four parts: Data Generator, Stream Processor, Database and Website UI. The architecture is shown in Figure 1.

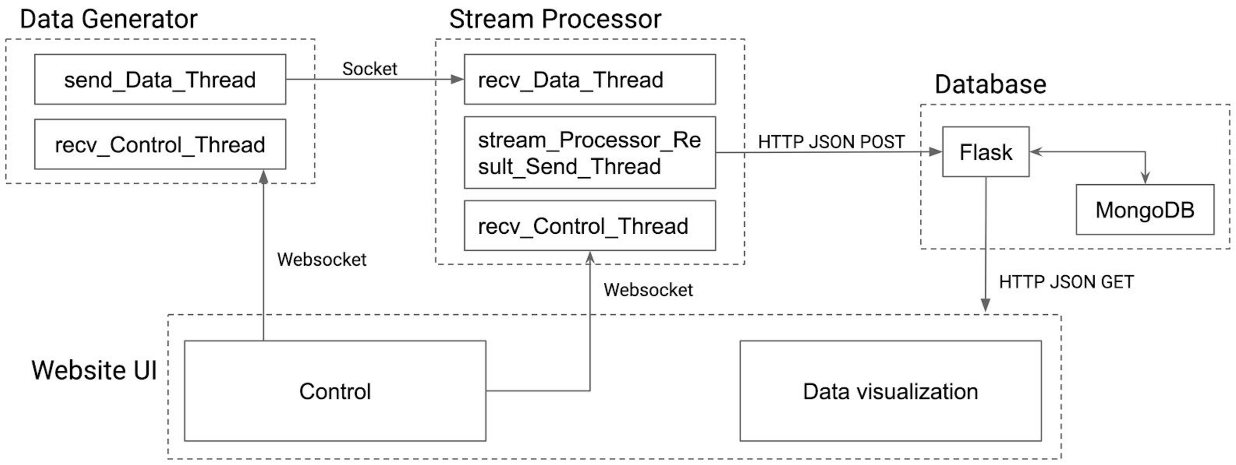


Figure 1 System architecture

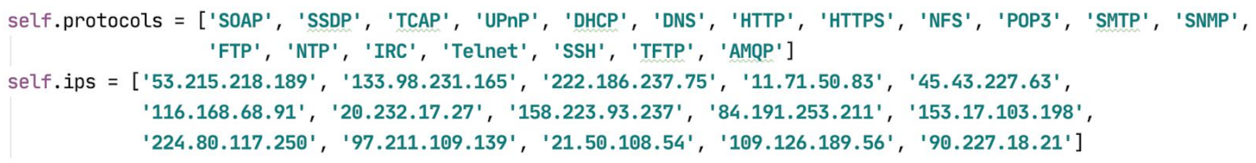
To fulfil the demand of sending data and receiving control simultaneously, Data Generator is designed as a multi-thread program, containing *send_Data_Thread* and *recv_Control_Thread*. Similarly, Stream Processor program contains *recv_Data_Thread*, *stream_Processor_Thread* and *recv_Control_Thread* (in our final implementation, *recv_Data* and *stream_Processor* are combined together). The communication between Data Generator and Stream Processor is based on Socket protocol.

Different from the raw stream data generated by Data Generator, which is processed directly by Stream Processor on arrival, the result processed by Stream Processor needs to be stored in Database. We use MongoDB as the database and Flask app as the interface handling the HTTP requests to the database. After the result is calculated, it is sent to the database through HTTP JSON POST. Meanwhile, every time the Website UI requests results for visualization, it sends HTTP JSON GET to the database.

Website UI handles the work of controlling Data Generator and Stream Processor, and visualizing results. The control signal is sent through Websocket protocol (UI implementation) or Socket protocol (system function testing).

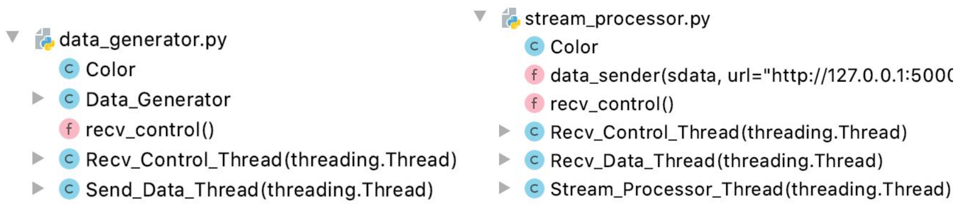
2.2. Data Generator

Data Generator has 19 predefined protocols and 15 ip addresses. Users need to specify the number and percentage of protocol and ip, then random data items are automatically generated and sent at a variable rate.



```
self.protocols = ['SOAP', 'SSDP', 'TCAP', 'UPnP', 'DHCP', 'DNS', 'HTTP', 'HTTPS', 'NFS', 'POP3', 'SMTP', 'SNMP',
                  'FTP', 'NTP', 'IRC', 'Telnet', 'SSH', 'TFTP', 'AMQP']
self.ips = ['53.215.218.189', '133.98.231.165', '222.186.237.75', '11.71.50.83', '45.43.227.63',
            '116.168.68.91', '20.232.17.27', '158.223.93.237', '84.191.253.211', '153.17.103.198',
            '224.80.117.250', '97.211.109.139', '21.50.108.54', '109.126.189.56', '90.227.18.21']
```

Figure 2 Predefined list



```
data_generator.py
    Color
    Data_Generator
    recv_control()
    Recv_Control_Thread(threading.Thread)
    Send_Data_Thread(threading.Thread)
```

```
stream_processor.py
    Color
    data_sender(sdata, url="http://127.0.0.1:500
    recv_control()
    Recv_Control_Thread(threading.Thread)
    Recv_Data_Thread(threading.Thread)
    Stream_Processor_Thread(threading.Thread)
```

Figure 3 Details of Data Generator and Stream Processor

To let generator and processor work collaboratively, we made many attempts like:

Table 1 Communication between generator and processor

Type	Basic socket communication		Build-in socket func of Spark	
Implementation	generator	client	generator	server
	processor	server	processor	client
Describe	Time consuming. Needs hard drive storage.		Time saving. New knowledge.	

The above two types of communication are implemented and the build-in socket function of Spark is chosen as the solution (will further explain in following), which are Send_Data_Thread() and Stream_Processor_Thread() in the code.

As for the control signal, both Socket and Websocket protocol are implemented. The Socket is used for control function testing and the signal is sent by a python program. The Websocket is used for website control and the signal is sent by the Website UI, because javascript doesn’t support basic socket communication.

When it receives a control signal like:

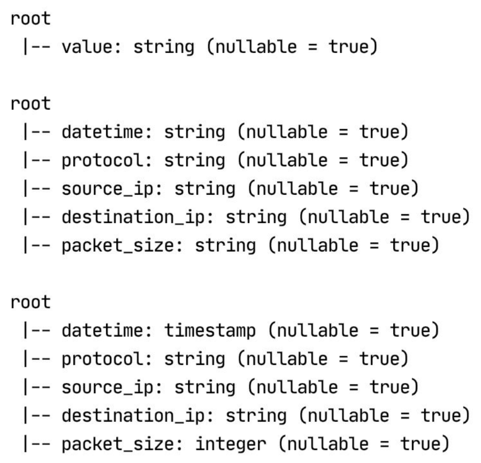
change_rate_ipNum_protocolNum_ipPercent_protocolPercent

It will kill the current send_Data_Thread and restart a new one with the parameters in the command. Apart from “change”, users can also make it “start” or “stop”.

2.3. Stream Processor

The data from the generator can be regarded as a real-time stream. To deal with the theoretically late data and avoid using hard drive storage, we utilize the Spark Structured Streaming.

At the beginning, we tried to use the socketTextStream() function in StreamingContext() to read the stream data. However, this function inputs the data as DStream, which can not be transformed to Dataframe. Moreover, the DStream lacks lots of functions like sort(), because Spark regards it as an infinite stream. Therefore, we use the readStream function in SparkSession. Because this function will directly input the data as DataFrame and we can use the watermark and window to process data.



```
root
 |-- value: string (nullable = true)

root
 |-- datetime: string (nullable = true)
 |-- protocol: string (nullable = true)
 |-- source_ip: string (nullable = true)
 |-- destination_ip: string (nullable = true)
 |-- packet_size: string (nullable = true)

root
 |-- datetime: timestamp (nullable = true)
 |-- protocol: string (nullable = true)
 |-- source_ip: string (nullable = true)
 |-- destination_ip: string (nullable = true)
 |-- packet_size: integer (nullable = true)
```

Figure 4 Formating

The schema of data inputted by readStream is one string and it needs to be splitted, which can be difficult for readStream because it only supports two kinds of format. We use regular expressions and then change the datatype to do the separation (the following code), and the changes of schema is shown in Figure 4.

```
fields = partial(
  regexp_extract, str='value',
pattern='^([0-9]*-[0-9]*-[0-9]*_[0-9]*:[0-9]*:[0-9]*.[0-9]*)\s*(\w*)\s*([0-9]*.[0-9]*.[0-9]*.[0-9]*)\s*([0-
9]*.[0-9]*.[0-9]*.[0-9]*)\s*([0-9]*)$'
)
self.log_tuples_df = self.log_lines_df.select(
  fields(idx=1).alias('datetime'),
  fields(idx=2).alias('protocol'),
  fields(idx=3).alias('source_ip'),
  fields(idx=4).alias('destination_ip'),
  fields(idx=5).alias('packet_size')
) # split the string in Structured Streaming Dataframe
# change the data type of the dataframe
self.log_tuples_df = self.log_tuples_df.withColumn('datetime', to_timestamp('datetime',
'yyyy-MM-dd_HH:mm:ss.SSS'))
self.log_tuples_df = self.log_tuples_df.withColumn('packet_size',
self.log_tuples_df['packet_size'].cast(IntegerType()))
self.log_tuples_df.printSchema()
```

After the transformation, the input dataframe is like a normal dataframe and will be very easy to use. Then, the specific process functions are realized with watermark, window, groupby, agg or sort.

As for the control signal, it is similar to Data Generator. When it receives a control signal like:

change_H_T_k_X

It will kill the current stream_Processor_Thread and restart a new one with the parameters in the command. Apart from "change", users can also make it "start" or "stop".

2.4. Database

We built our database on the MongoDB database program and use Python Flask frame to connect the database with our Stream Processor part and our Website UI.

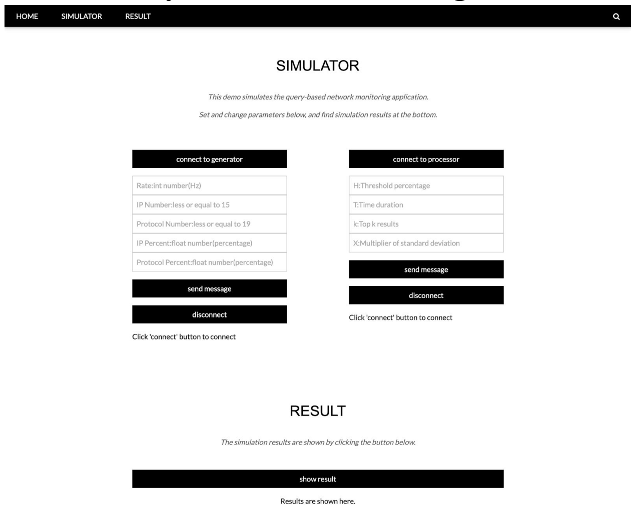
In our Flask program, we implemented the following fout functions using HTTP requests:

- Get all data from a table in the database.
- Get the data that is most recently inserted into a table.
- Insert the data from the Stream Processor to a table in the database.
- Delete all the data in a table in the database.

Whenever the streaming processor has output data, it will send data to the database through function 3 above. Whenever the website UI needs the data from the database, it will get data from the database through function 2 above. We also use Postman software to test the functions in our Flask frame.

2.5. Website User Interface

The website performs as the interface to interact with the database, data generator and streaming processor. The simulator part uses socket connection to change parameters with the generator and the processor, the result part will return the result from the database. The layout is shown in Figure 5.



This demo simulates the query-based network monitoring application.

Set and change parameters below, and find simulation results at the bottom.

Click 'connect' button to connect

Click 'connect' button to connect

The simulation results are shown by clicking the button below.

Results are shown here.

Figure 5 Website UI

3. Results

3.1. Data Generator

When the Data Generator is started, it will listen to port 12301 and 12302, for data signal and control signal. After it gets a connection from Stream Processor, it will send the raw stream data of the format as shown in Figure 6.

```
************************************ Data Generator is starting ************************************
GOOD: Data sending thread started
GOOD: Control receiving thread started
GOOD: Connection complete. Data Generator is listening for control signal, from port 12302.
GOOD: Connection complete. Data Generator is listening for spark stream, from port 12301.
2020-05-09_20:41:28.119 UPnP 11.71.50.83 133.98.231.165 8216
2020-05-09_20:41:28.324 SOAP 133.98.231.165 53.215.218.189 3986
2020-05-09_20:41:28.528 SOAP 53.215.218.189 133.98.231.165 5594
2020-05-09_20:41:28.733 SSDP 45.43.227.63 45.43.227.63 2824
2020-05-09_20:41:28.937 SOAP 11.71.50.83 222.186.237.75 10003
2020-05-09_20:41:29.141 TCAP 11.71.50.83 53.215.218.189 11980
```

Figure 6 Raw stream data

3.2. Stream Processor

Figure 7 shows the formatted dataframe. It has 5 attributes of 3 different types, which are timestamp, string and integer.

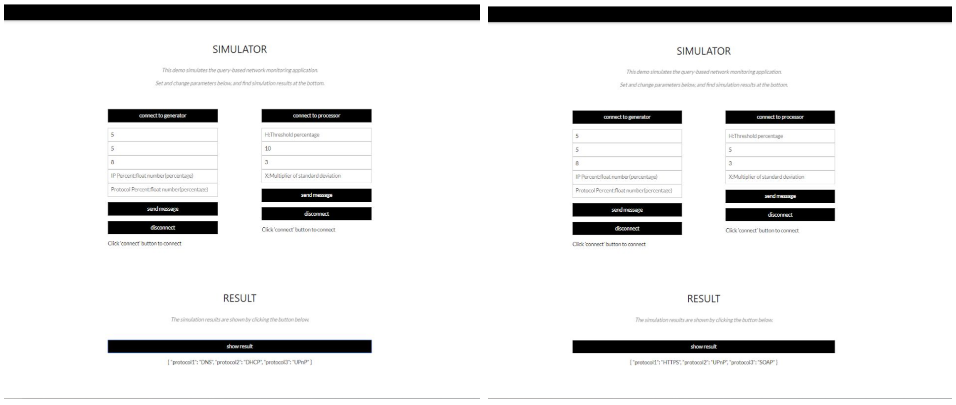
```
+-------------------+--------+--------------+--------------+-----------+
|           datetime|protocol|     source_ip|destination_ip|packet_size|
+-------------------+--------+--------------+--------------+-----------+
|2020-05-09 20:41:37|    SOAP|222.186.237.75|53.215.218.189|       6772|
+-------------------+--------+--------------+--------------+-----------+
```

Figure 7 Formatted dataframe

As an example, we use the second query "List the top-k most resource intensive protocols over the last T time units" to share how different parameters setup affects the final output. As Figure 9 shown below, when the data generator and the stream processor are running, we can send control signals to them through the website. Here to fully illustrate the difference between two sets of parameters, we differentiate the time duration T in two cases. We set T=10 in the left case while setting T=5 in the other (where the protocol size also gives implication), the final output protocols are totally different which results from both the data randomness and the changing of the setup.

```
+--------------------+--------+-------------+ +--------------------+--------+-------------+
|              window|protocol|protocol_size| |              window|protocol|protocol_size|
+--------------------+--------+-------------+ +--------------------+--------+-------------+
|[2020-05-09 22:46...|     DNS|         6140| |[2020-05-09 22:46...|   HTTPS|         7098|
|[2020-05-09 22:46...|    DHCP|        11002| |[2020-05-09 22:46...|    UPnP|         3517|
|[2020-05-09 22:46...|    UPnP|         9828| |[2020-05-09 22:46...|    SOAP|         2398|
+--------------------+--------+-------------+ +--------------------+--------+-------------+
```

Figure 8 Output dataframe with different parameter setup



This demo simulates the query-based network monitoring application.
Set and change parameters below, and find simulation results at the bottom.

5
5
8
IP Percent:float number(percentage)
Protocol Percent:float number(percentage)

H:Threshold percentage
10
3
X:Multiplier of standard deviation

Click 'connect' button to connect

The simulation results are shown by clicking the button below.

{ "protocol1": "DNS", "protocol2": "DHCP", "protocol3": "UPnP" }

5
5
8
IP Percent:float number(percentage)
Protocol Percent:float number(percentage)

H:Threshold percentage
5
3
X:Multiplier of standard deviation

Click 'connect' button to connect

The simulation results are shown by clicking the button below.

{ "protocol1": "HTTPS", "protocol2": "UPnP", "protocol3": "SOAP" }

Figure 9 Different parameter setup through website

4. Problems & Future work

4.1. Scaling to high-frequency data

We can reconstruct batches based on the original time by Spark streaming micro-batch and Spark streaming window.

4.2. Changing dynamically

We change parameters and make comparisons when reaching results. Using WebSocket to send control signals from the website to the data generator and the stream processor ensures the communication of dynamic parameter settings securely since WebSocket is based on TCP. With multi-threading, The system kills the current thread and restarts a new one when the new connection is established.

4.3. Streaming algorithm optimization

Operator Separation and Fission optimization methods can be used in our project because it is easy for the filters in our project to be divided.

4.4. Using Spark streaming for data generator

We found a method[1] that will generate random values based on the data type of a specific schema field. It will generate value from a list of words first. Then it will generate a row of data from value. Next, it will generate a batch of records from data rows. Finally, it will generate a RDD dataframe from all batches of records.

4.5. How to change parameters

As written in part 2.2 and 2.3, Data Generator and Stream Processor will receive a message from Website UI (final implementation) of Websocket protocol, or from python code (testing) of Socket protocol. Then the old send_Data_Thread or the stream_Processor_Thread will be killed and a new thread will be started with the changed parameters.

[1] https://github.com/glebkorolkov/datagen/blob/master/datagen/data_generator.py