2012 Seventh International Conference on P2P, Parallel, Grid, Cloud and Internet Computing

Performance Evaluation of Yahoo! S4: A First Look

Jagmohan Chauhan, Shaiful Alam Chowdhury and Dwight Makaroff
Department of Computer Science
University of Saskatchewan
Saskatoon, SK, CANADA S7N 3C9
(jac735, sbc882, makaroff)@cs.usask.ca

***Abstract*—Processing large data sets has been dominated recently by the map/reduce programming model [1], originally proposed by Google and widely adopted through the Apache Hadoop[1] implementation. Over the years, developers have identified weaknesses of processing data sets in batches as in MapReduce and have proposed alternatives. One such alternative is continuous processing of data streams. This is particularly suitable for applications in online analytics, monitoring, financial data processing and fraud detection that require timely processing of data, making the delay introduced by batch processing highly undesirable. This processing paradigm has led to the development of systems such as Yahoo! S4 [2] and Twitter Storm.[2] Yahoo! S4 is a general-purpose, distributed and scalable platform that allows programmers to easily develop applications for processing continuous unbounded streams of data. As these frameworks are quite young and new, there is a need to understand their performance for real time applications and find out the existing issues in terms of scalability, execution time and fault tolerance.**

We did an empirical evaluation of one application on Yahoo! S4 and focused on the performance in terms of scalability, lost events and fault tolerance. Findings of our analyses can be helpful towards understanding the challenges in developing stream-based data intensive computing tools and thus providing a guideline for the future development.

***Keywords*-Yahoo! S4; Performance; Stream-based Data Intensive computing**

I. Introduction

The cost-per-click billing model has become common for online advertisement, and this model is used by various commercial search engines like Google, Yahoo!, and Bing [3]. In this model, the advertisers pay the owner of the website (publisher) according to the number of clicks a specific ad gains by the users. However, it is a very challenging task to place the most relevant advertisements in the appropriate positions on a page. That is why algorithms were developed to estimate the time varying probability of clicking on a specific advertisement, considering user access history, geographic locations etc. An efficient search engine is capable of processing thousands of such queries per second, thus ensuring that a significant number of appropriate advertisements can be selected within a very short period of time. Inspired by these kinds of applications, especially to process user feedback, Simple Scalable Streaming System (S4) was developed by Yahoo! [2] to address the shortcomings of MapReduce [1], is a tool for distributed stream processing that applies data mining and machine learning techniques with the goal of ensuring low latency and scalability.

Disregarding fault tolerance issues, the MapReduce programming model enables parallelization of number of batch data processing tasks that consist of two different functions: 'map' and 'reduce' [1]. Not surprisingly, because of its efficiency in producing results, MapReduce has become a common technique in different kinds of real-world applications. However, with the proliferation of high frequency trading and real-time search, batch data processing systems, such as MapReduce, are becoming less satisfactory, indicating the necessity of significantly scalable software infrastructure for stream computing. The developers of Yahoo! S4 claim that the reason it was created was to overcome the shortcomings encountered by MapReduce in the case of massive amounts of real-time data processing [2]. In this paper, several performance issues of S4 are investigated that can be helpful in designing a more efficient and robust programming model for future data intensive applications.

In this study, our main focus is on the scalability, lost events and fault tolerance aspects of Yahoo! S4. The rest of the paper is organized as follows. Section II discusses the background and concepts related to Yahoo! S4. Related work is discussed in Section III. Section IV explains the experimental testbed and our evaluation methodology. Section V presents our experimental results and analysis. Finally, the conclusions are drawn in Section VI, with a brief description of future work.

II. Background

In this section, we discuss the design and architecture concepts related to Yahoo! S4.[3] This design and architecture distinguishes the deployment context and data processing philosophy from MapReduce. In S4, data events are routed to Processing Elements (PEs) on the basis of keys, which consume the events and do one or both of the following: 1) produce one or more events which may be consumed by

[1]http://hadoop.apache.org/

[2]http://engineering.twitter.com/2011/08/storm-is-coming-more-details-and-plans.html

[3]http://docs.s4.io/manual/overview.html

other PEs, or 2) publish results, either to an external database or consumer. More details of each of these operations are provided in the rest of this section.

S4 has been designed to achieve a number of specific goals that make it usable and efficient. Firstly, language neutrality was important for developers. Thus an application in Yahoo! S4 can be written using different languages such as Java, Python, C++, etc. Secondly, the framework can be implemented on commodity hardware to enable high availability and scalability. Thirdly, the processing architecture is designed to avoid disk access performance bottlenecks. Latency is minimized for application programs by making efficient and greater use of memory for processing events. Finally, a decentralized and symmetric architecture makes the system more resilient and avoids a single point of failure.

S4 is logically a message-passing system. Figure 1 shows the Yahoo! S4 architecture.[4] Events and the Processing elements form the core of the Yahoo! S4 framework. Java objects called *Events* are the only mode of communication between the Processing Elements. The external client application acts as the source of data which is fed into the client adapter. The Client adapter converts the incoming input data into events which are then sent to the Yahoo! S4 cluster. Every event is associated with a named stream. The processing elements identify the events with the help of stream names.

Processing Elements (PEs) are the basic computational units in S4. Some of the major tasks performed by PEs are consumption of events, emitting new events and changing the state of the events. A PE is instantiated for each unique value of the key attribute. This instantiation is performed by the S4 framework. Instantiating a new PE can have a significant overhead and hence the number of values and keys plays an important role in the performance of Yahoo! S4. Determining the number of PEs is a challenging task and should be done carefully to avoid overhead and gain real performance benefits. Every PE consumes events which contain the value that is assigned to the PE. As an example, if we consider a WordCount application, then there can be a PE named WordCountPE which is instantiated for each word in the input.

There are two types of PEs: Keyless PEs and PE Prototypes.

- A Keyless PE has no keyed attribute or value and consumes all events on the stream with which it is associated through a stream name . Keyless PEs are generally used at the entry point of Yahoo! S4 cluster because events are not assigned any keys. The assignment of keys to the events is done later. Keyless PE's are helpful when every input data is to be consumed without any distinction.
- PE Prototypes serve as a skeleton for PEs with only the first three components of a PE's identity assigned (functionality, stream name, and keyed attribute). As it is not a fully formed PE, the attribute value is unassigned. This object is configured upon initialization. Later, when a fully qualified PE with a value V is needed, the object is cloned to create fully qualified PEs of that class with identical configuration and value V for the keyed attribute.

[4]http://database51cto.com/art/201110/297784.htm

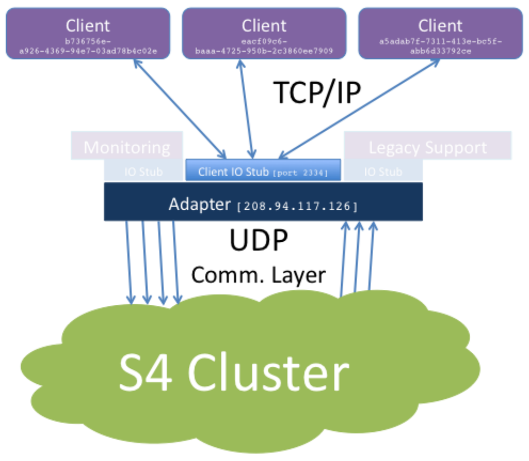

Client
b736756e-a926-4369-94e7-03ad78b4c02e
Client
eacf09c6-baaa-4725-950b-2c3860ee7909
Client
a5adab7f-7311-413e-bc5f-abb6d33792ce
TCP/IP
Monitoring
IO Stub
Client IO Stub [port 2334]
Legacy Support
IO Stub
Adapter [208.94.117.126]
UDP
Comm. Layer
S4 Cluster

Figure 1. Yahoo! S4 Framework

PEs are executed on Processing Nodes (PN), whose major tasks involve listening to events, executing operations on incoming events, dispatching events and generating output events. To send an event to an appropriate PE, Yahoo! S4 initially routes the event to PNs based on a hash function of the values of all known keyed attributes in that event. Once the event reaches the appropriate PN, an event listener in the PN passes incoming event to the processing element container (PEC) which invokes the appropriate PEs in the appropriate order.

The client adapter enables the Yahoo! S4 cluster to interact with the outside world, implementing a JSON-based API, for which client drivers are written. The external clients to Yahoo! S4, injects the events as and when they are generated into the Yahoo! S4 cluster with the help of adapter. Similarly, the external clients can receive events from S4 cluster through the adapter via the Communication Layer.

III. Related Work

The work on stream-based data intensive computing can be divided into two broad categories: developing new platforms and performance evaluations of those frameworks. These will be discussed in the rest of this section.

With respect to developing new and efficient stream-based

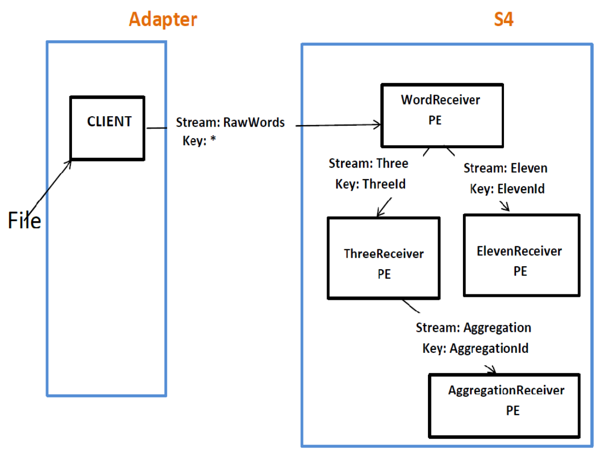


Figure 2. Event Flow

data intensive computing platforms, Gulisano, Jiminez-Peris, Patino-Martinez, Soriente and Valduriez [4] proposed StreamCloud, a scalable and elastic data streaming system for processing large data stream volumes. StreamCloud splits queries into subqueries for parallelization and dispatches them to independent sets of nodes for minimal overhead of distribution. It also provides a parallelization strategy that is fully transparent, producing executions equivalent to non-parallel Stream Processing Engine (SPEs).

Borealis [5] is a second-generation distributed stream processing engine. It inherits core stream processing functionality from Aurora [6] and distribution functionality from Medusa [7], but has enhanced functionality that has evolved as a result of new requirements from the applications. Aurora is a high performance stream processing engine, which has a collection of operators, workflow orientation, and strives to maximize quality of service for connecting applications. In contrast, Medusa was introduced to provide networking infrastructure for Aurora operations. It does some of the tasks including supporting a distributed naming scheme, collection of distributed catalogs that store the pieces of a distributed workflow and their interconnections, and transportation of message between components. Borealis modifies and extends both systems in non-trivial and critical ways to provide advanced capabilities that are commonly required by newly-emerging stream processing applications.

Some research efforts have been directed towards making MapReduce a real-time stream processing engine. One such effort is by Condie *et al.* [8] named Hadoop Online Prototype (HOP). They modified Hadoop framework to support online aggregation and continuous queries. HOP is backwards compatible with Hadoop, with the property that existing applications do not need to be modified. Other research efforts include DEDUCE [9]. It extends IBM's System 'S' stream processing middleware with support for MapReduce to combine the benefits of MapReduce and stream processing to address the data processing needs of some applications like market data analysis.

All the works described earlier are efforts done by the researchers and evaluated by themselves. System 'S' from IBM, Yahoo! S4 and TwitterStorm are some of commercial stream processing distributed frameworks. From the analysis of existing literature, we found out that although a lot of work has been done on developing new platforms, there exists a little work on the performance evaluation of these platforms. In fact, we found only one existing work doing such an evaluation.

Dayarathna, Takeno and Suzumura [10] compared the performance of Yahoo! S4 and System 'S'. The major aim of their work was to investigate and characterize which architecture is better for handling which type of stream processing workloads and observe the reasons for such characteristics. They observed that properly designed stream applications result in high throughput. Another conclusion they arrived at is that choice of a stream processing system should be done carefully considering factors such as performance, platform independence, and size of the jobs. Our work can be considered as an extension to their work. They focused only on the scalability aspects of Yahoo! S4, while we have also focused on other aspects like resource usage and fault tolerance. Some of our findings like heavy network usage in Yahoo! S4 confirms their results. However, our work also presents some new insights about Yahoo! S4 performance aspects.

IV. Experimental Setup and Methodology

In order to evaluate the performance of Yahoo! S4, a 4-node cluster is used in this work. All the nodes are homogeneous and have the same configuration. Each node consists of a Core 2 duo 2.4 Ghz processor with 2 GB of memory and 1 GBps network card. All the machines are part of the same rack. We designated one of the nodes as a Client adapter to pump the events into the Yahoo! S4 cluster. The other 3 nodes form the Yahoo! S4 cluster. For the experiments, we used Yahoo! S4 code base version 0.3.0.

Our experimental evaluation used a benchmark kernel application that we developed that stressed the scalability of the Yahoo! S4 architecture/infrastructure in a simple, predictable manner. This application reads "words" from a file through the client adapter. The number of words depends on the data size in our experiments. For example, in case of 10 MB of data size, the number of words were 200,000 and each of the words was 100 characters long. Most of the words in the file are actually numbers as they don't contain any non-digit characters. The client adapter converts them into events and pumps them to the Yahoo! S4 cluster. Figure 2 shows the logical ordering of different PE's and the execution flow of events for our application in Yahoo! S4. Our application works in the following way:

1) The client application reads from the file which contain 100 character long words line by line and send them to client adapter.
2) The client adapter gets each word which is 100 characters long, converts it into an event and sends it to the WordReceiver PE.
3) WordReceiver PE gets the event RawWords. It checks if the word is a number, create events named Eleven and Three and send them for further processing.
4) Both ElevenReceiver and ThreeReceiver PE get events from WordReceiver PE. ElevenReceiver PE checks if the number is divisible by 11. Similarly, ThreeReceiver checks if the number is divisible by 3.
5) AggregationReceiver gets the number of events processed by ThreeReceiver PE and counts them and puts it in a output file.

The size of the numbers make the work of ElevenReceiver and ThreeReceiver significantly more computationally expensive as simple division operation can not be applied. In order to check if a number is divisible by 3 or 11, appropriate string operations were applied. Although, not a typical benchmark, the size of the words make the processing computationally expensive enough to evaluate Yahoo! S4 in our case. Each experiment was repeated 5 times and the average results are shown here. We did not observe any significant variance in any case.

We focused on three different performance metrics in our study:

- Scalability and lost events: We checked how increasing the number of nodes affects the execution time and lost events rate.
- Resource usage: We measured the CPU, memory, disk and network usage at each node to see how resources are used in Yahoo! S4.
- Fault tolerance: We looked for the events handled when some nodes goes down in cluster. We also checked how many events were missed after the node goes down in cluster.

V. Analysis

A. Scalability

In this scenario we evaluated the scalability of the Yahoo! S4 cluster. We used two different data sizes in these experiments. The size of one data input was 10 MB and size of second data set was 100 MB. The other parameter we changed was the number of keys. The number of keys defines the number of PEs, which can be instantiated in the cluster. We also controlled the rate of event injection into the Yahoo! S4 cluster. It was done at two places. The first place was the client adapter where in one case we pumped data at the rate of 500 events/second, while in the other case the rate was 1000 events/second. The second place was at WordReceiver PE. Depending on the number of keys, we multiplied the number of events. For example, if the number of keys we instantiated is 4, then the number of new events created in WordReceiver PE for ElevenReceiver and ThreeReceiver PE was multiplied by 4.

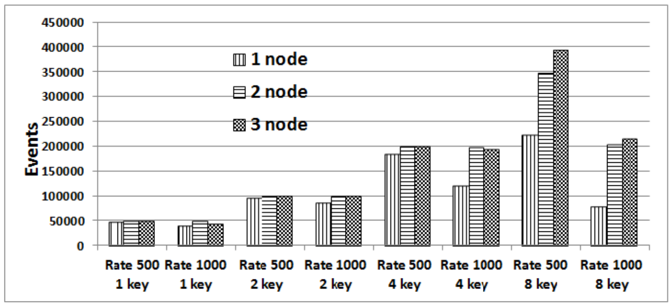
The first result is shown in Figure 3(a) for 10 MB of streaming data. The AggregateReceiver PE gives the count of the events processed by the Yahoo! S4 cluster for events of type Three. In the input data file, 50000 words were numbers divisible by three out of 200,000 words. Therefore, in an ideal scenario the number of processed events should be 50,000 when number of keys is 1. With increasing number of keys, the events are multiplied at WordReceiver as explained before and hence the number of events are increased by factor of 2, 4 and 8, respectively. We can see from the graph that the number of events processed by Yahoo! S4 increases when the number of nodes goes from 1 to 2. This is expected because with increasing nodes the scalability will increase. With increasing number of keys (i.e. PEs), the effect becomes more pronounced. As the amount of data is huge, it become difficult for just one node to process and the result is the loss of many events.

The most important observation which can be seen from the graph is that when the number of nodes is increased from 2 to 3, the number of processed events goes down or remains similar when the number of key is 1, 2 and 4. This emphasizes that increasing the number of nodes does not always increase throughput. The overhead of communication, and distribution can be attributed for this phenomenon. We can see that when the number of keys is 8 (which means a deluge of events), the extra third node helps as it overcomes the overhead we observed with fewer keys. The graph in Figure 3(b) shows the results when the data size is 100 MB, showing the same trends as the 10 MB case.

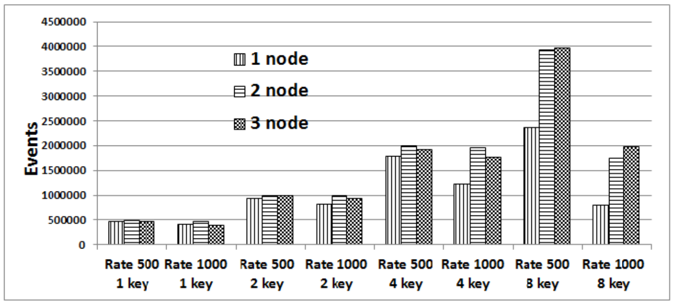
Figure 4 shows the event loss percentage for both the data sizes. The number of events lost is highest for single node scenario. The number of events lost is lower for 2-node scenario than 3-node scenario, except the case where the number of keys is 8.

We initially were keen to measure the execution time and we expected that with increasing number of nodes the time shall decrease. We did not set any event injection rate in such case. However, what we observed was that with increasing number of nodes the time for execution was increasing. After deep investigation, we found out that the shorter execution time for a single node was because more events were dropped. Thus, when the number of nodes was increased, the number of events successfully processed increased and hence execution time increased.

Sending and processing are asynchronous (incoming events are queued and sequentially processed, like in the actors model [11]) in Yahoo! S4. However, in the current version of S4, there is only 1 thread that processes events at the application level.

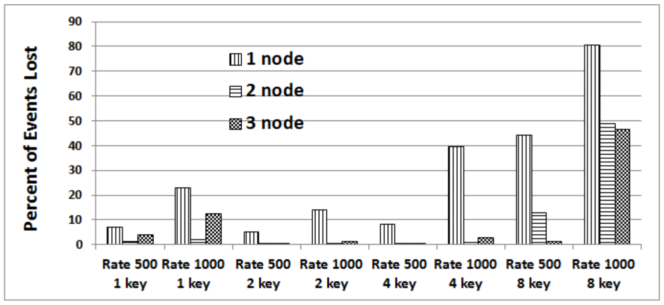


(a) Scalability for data size of 10 MB

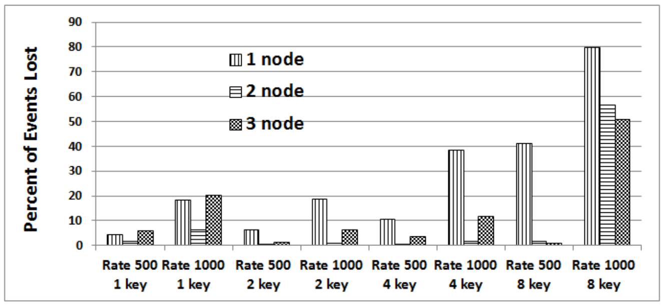


(b) Scalability for data size of 100 MB

Figure 3. Scalability results



(a) Event Loss percentage for data size of 10 MB



(b) Event Loss percentage for data size of 100 MB

Figure 4. Event Loss percentage results

B. Resource usage

The resource usage of all the nodes for 10 MB and 100 MB of input data is shown in various tables and graphs in this section. Table I and Figure 5 show the resource usage stats when only one node is present in Yahoo! S4 cluster. The usage of resources increase with more events and a larger number of keys. We did not show the disk usage as it was not more than 5% in any of the executed scenarios. This shows that the Yahoo! S4 architecture fulfills the design goal of removing the I/O bottleneck and performing most of the execution in memory.

We can see how memory is used heavily in Table II. We can also observe an interesting fact that the network transmitted packets are less than network received packets for node 2, which is quite opposite in all other cases. The same trend can be seen in Table III. The reason is the unequal sharing of work between the nodes by Yahoo! S4. So, one of the nodes may get less work compared to others and hence their resources are not used as much, compared to the other nodes. From the logs we saw that in the 3-node case, one of the nodes was getting twice the amount of work in terms of event processing than the other 2 nodes. Node 2 in Table II and Table III gets most of the work and hence its resource usage is highest and as number of events coming for processing are more than other nodes. In Figures 6 and 7, the stats for network reception are shown to be higher than network transmission for node 2.

			Keys			
			1	2	4	8
Rate	500	CPU %	17.5	25	50	75
		Memory %	65	65	65	78
	1000	CPU %	22.5	47.5	70	80
		Memory %	65	65	74	72

Table I
RESOURCE USAGE FOR 1-NODE CLUSTER (10 MB OF DATA)

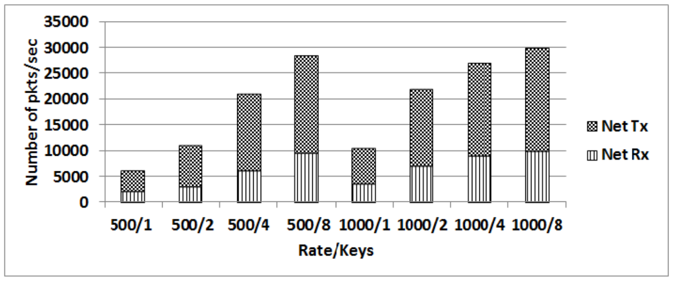


Figure 5. Network stats for data size of 10 MB (1-node cluster)

			Keys			
Node	Rate	Resource	1	2	4	8
1	500	CPU %	5	15	30	50
		Memory %	71	71	71	71
	1000	CPU %	10	25	55	80
		Memory %	71	71	71	78
2	500	CPU %	15	20	40	60
		Memory %	88	82	92	92
	1000	CPU %	20	35	70	85
		Memory %	96	96	96	96

Table II
RESOURCE USAGE FOR 2-NODE CLUSTER (10 MB OF DATA)

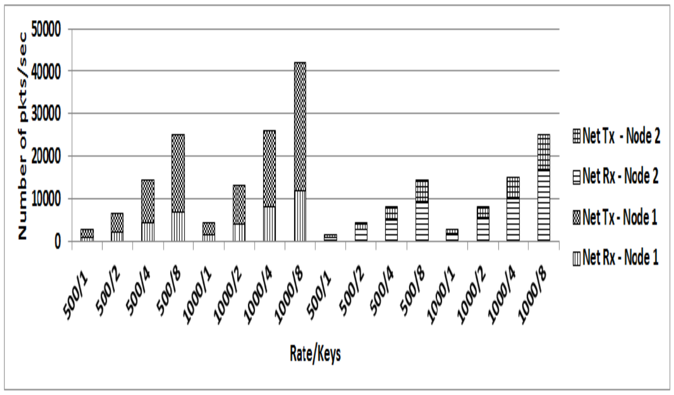


Figure 6. Network stats for data size of 10 MB (2-node cluster)

			Keys			
Node	Rate	Resource	1	2	4	8
1	500	CPU	5	12	18	40
		Memory	65	65	65	65
	1000	CPU	7	25	30	70
		Memory	65	65	65	68
2	500	CPU	8	18	35	60
		Memory	88	96	90	96
	1000	CPU	15	35	70	85
		Memory	95	96	96	96
3	500	CPU	17	15	25	40
		Memory	61	64	64	64
	1000	CPU	23	20	40	60
		Memory	61	62	66	66

Table III
RESOURCE USAGE FOR 3-NODE CLUSTER (10 MB OF DATA)

Table IV and Figure 8 show the resource usage statistics for data size of 100 MB and a single node cluster. Due to high event loss, the resource usage goes down for data rate of 1000 and 8 keys. Tables V and VI show similar trends with 100 MB data size as the results for 10 MB data size, for the CPU and Memory usage. This was expected as the data injection rate is the same; the only difference is the data size. The network statistics are shown in Figures 9 and 10. Node 2 gets the bulk of work and hence its resource consumption is the highest.

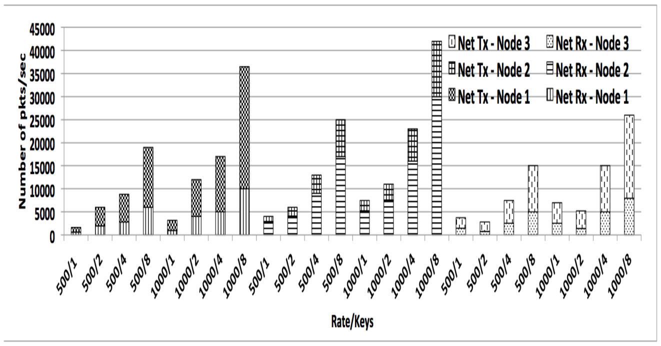


Figure 7. Network stats for data size of 10 MB - 3-node cluster

			Keys			
			1	2	4	8
Rate	500	CPU	15	25	45	70
		Memory	56	57	62	87
	1000	CPU	25	45	70	70
		Memory	56	60	96	70

Table IV
RESOURCE USAGE FOR 1-NODE CLUSTER (100 MB OF DATA)

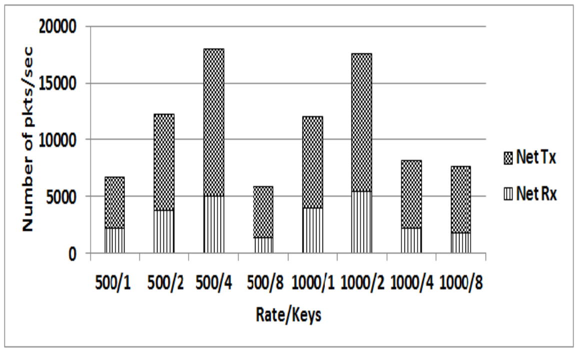


Figure 8. Network stats for data size of 100 MB (1-node cluster)

			Keys			
Node	Rate	Resource	1	2	4	8
1	500	CPU	5	15	28	52
		Memory	51	50	48	50
	1000	CPU	8	28	55	82
		Memory	53	50	52	72
2	500	CPU	14	20	38	75
		Memory	96	96	96	96
	1000	CPU	22	35	72	85
		Memory	96	96	96	96

Table V
RESOURCE USAGE FOR 2-NODE CLUSTER (100 MB OF DATA)

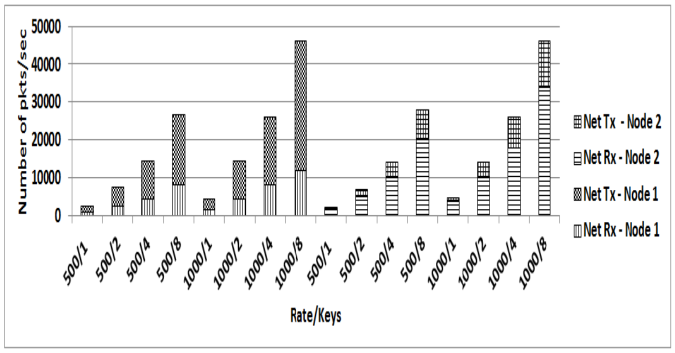


Figure 9. Network stats for data size of 100 MB for 2-node cluster

			Keys			
Node	Rate	Resource	1	2	4	8
1	500	CPU	5	12	18	40
		Memory	57	58	68	70
	1000	CPU	6	25	30	70
		Memory	57	68	69	75
2	500	CPU	8	18	35	65
		Memory	96	96	96	96
	1000	CPU	12	35	60	85
		Memory	96	96	96	96
3	500	CPU	10	8	18	35
		Memory	58	60	61	62
	1000	CPU	18	12	35	60
		Memory	60	63	62	66

Table VI
RESOURCE USAGE FOR 3-NODE CLUSTER (100 MB OF DATA)

C. Fault Tolerance

In this section, we measured how throughput is affected when nodes in the cluster go down and how this affects the remaining standing nodes. We only did experiments on the 10 MB input data size. We examined a few selected scenarios and results are shown in Figure 11. The stacked column shows the number of events processed by each node. For each experiment, we first measured the total execution time. After measuring it, we did the same experiment but killed the nodes when the execution was half way through. In the 2-node scenario, we only killed a single node. In the 3-node scenario, we killed 1 or 2 nodes in different runs. We can see that as expected the number of events processed by Yahoo! S4 cluster goes down substantially. We can check this by comparing Figure 11 with Figure 3. In Figure 11, the first two columns show the 2-node scenario with one node going down, while the other columns shows the 3-node scenario with 1 or 2 nodes failed as indicated. The throughput dropped to 55% in the 2 node scenario. The throughput dropped to 68% in the 3-node scenario with a single node failure and when 2 nodes failed, it dropped to around 50%. We expected that when one node goes down, the other node shall take the responsibility and shall process more events than it was processing earlier in the normal scenario. This was not the case and we obtained a different outcome. With one or more nodes down, the number of events being sent to the remaining nodes become very high and overwhelms them, leading to much higher event loss. Due to this sudden unexpected data rate, the remaining nodes could not even keep up their earlier processing rate and they perform even worse (i.e. processing fewer events than they were doing under normal conditions).

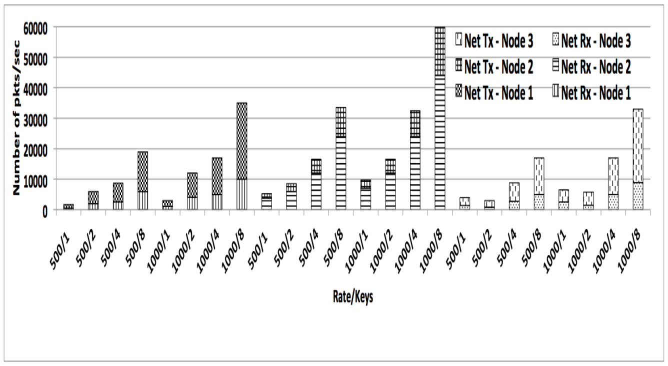


Figure 10. Network stats for data size of 100 MB for 3-node cluster

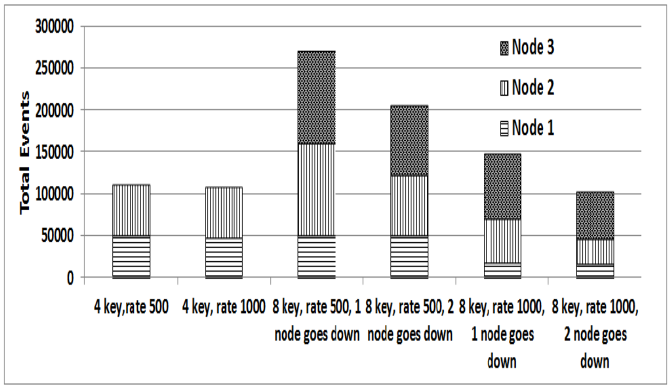


Figure 11. Total Events Processed for data size of 10 MB under node failures.

VI. CONCLUSIONS AND FURTHER WORK

We did an empirical evaluation on a Yahoo! S4 cluster to measure its scalability, resource usage patterns and fault tolerance. Based on our empirical study we can conclude a few key points:

- The throughout in terms of number of events processed does not necessarily increase with increasing number of nodes. The initial overhead can be sometimes high to subdue the advantage of adding more nodes. However, once the overhead is diminished the throughput is increased. We can see that in our experiment, increasing the number of nodes from 2 to 3 helps only when the number of keys was 8. This articulates that appropriate understanding of the actual workload is necessary before deciding the number of nodes in Yahoo! S4 and is not an easy task.

- Yahoo! S4 distributes the events unequally which leads to more resource consumption on some nodes than others. We think that event distribution algorithm should be improved in Yahoo! S4 and the event distribution should be equal to have better load balancing among the nodes.
- Yahoo! S4 fault tolerance is not up to the expectation. When a node goes down, Yahoo! S4 tries to send the events to the remaining nodes but the rate of events overwhelm the remaining nodes and leads them to under-perform. This means the remaining nodes must have extra processing capabilities compared to failed nodes to handle unexpected conditions but it is not possible in a homogeneous environment. The experimental results also suggests that these nodes have to be over-provisioned to meet performance requirements, which is not a good way to manage the resources in a cluster.
- For platforms like Yahoo! S4, the rate of event loss, not the execution time, is the correct performance metric to investigate for the type of benchmark we evaluated.
- The network is used heavily in Yahoo! S4 and the availability can be an issue for high network bandwidth demanding applications. Most of the processing of events is done in memory, which totally eliminates the disk bottleneck.
- The events are passed from one PE to another one using UDP. So, in case UDP datagrams get lost because of network issues, there is no acknowledgement technique currently available at application level.

We performed experiments at two input data rates: 500 and 1000 events/second. Un-throttled event generation over-saturated the system, providing unacceptable performance. Rates much lower than 500 did not stress the system at all. A systematic evaluation of the effect of input data rate on the resource usage, event loss and packet processing rates is planned as future work.

TwitterStorm is yet another platform similar to Yahoo! S4. As these frameworks are young and new, there is need to understand their performance and this motivates us to do this work. As future work, we plan to compare the performance of Yahoo! S4 and TwitterStorm. We are especially interested to find out the scenarios where one of them outperforms another. This evaluation will help us to identify the good features from both of the tools, which might be instrumental for future development of stream-based, data intensive tools.

REFERENCES

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[2] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *2010 IEEE Data Mining Workshops (ICDMW)*, Sydney, Australia, Dec. 2010, pp. 170 –177.

[3] B. Edelman, M. Ostrovsky, M. Schwarz, T. D. Fudenberg, L. Kaplow, R. Lee, P. Milgrom, M. Niederle, and A. Pakes, "Internet advertising and the generalized second price auction: Selling billions of dollars worth of keywords," *American Economic Review*, vol. 97, 2005.

[4] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez, "StreamCloud: An elastic and scalable data streaming system," *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, no. 99, p. 1, Jan. 2012.

[5] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. hyon Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik, "The design of the Borealis stream processing engine," in *In CIDR*, Asilomar, CA, Jan. 2005, pp. 277–289.

[6] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik, "Monitoring streams: a new class of data management applications," in *Proceedings of the 28th VLDB Conference*, Hong Kong, China, Aug. 2002, pp. 215–226.

[7] S. Zdonik, M. Stonebraker, M. Cherniack, U. Çetintemel, M. Balazinska, and H. Balakrishnan, "The Aurora and Medusa projects," *IEEE Data Engineering Bulletin*, vol. 26, 2003.

[8] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, J. Gerth, J. Talbot, K. Elmeleegy, and R. Sears, "Online aggregation and continuous query support in MapReduce," in *Proceedings of the 2010 SIGMOD Conference*, Indianapolis, Indiana, Jun. 2010, pp. 1115–1118.

[9] V. Kumar, H. Andrade, B. Gedik, and K.-L. Wu, "DEDUCE: at the intersection of MapReduce and stream processing," in *Proceedings of the 13th EDBT Conference*, Lausanne, Switzerland, Mar. 2010, pp. 657–662.

[10] M. Dayarathna, S. Takeno, and T. Suzumura, "A performance study on operator-based stream processing systems," in *2011 IEEE International Symposium on Workload Characterization (IISWC)*, Austin, TX, Nov. 2011, p. 79.

[11] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA: MIT Press, 1986.