Storm with Varying Timeouts for a new Fault Tolerance mechanism.

Ravali Kandur
University of Minnesota, MN
kandu009@umn.edu

Abhishek Chandra
University of Minnesota, MN
chandra@cs.umn.edu

ABSTRACT

This document describes limitations of Storm when it is applied to a stream processing application that might have some intentional delays added by the process workflow itself. Some examples of this kind include data processing over wide area network, interacting with a third party service during the data processing phase which might add up some expected delays. We also propose some variants of Storm's implementation which could potentially perform better than the current implementation for data processing which involves adding intentional delays.

1. INTRODUCTION

Storm provides data processing semantics like *Exactly Once*, which relies on the mechanism of tracking (by hidden acker bolts added to each topology by storm) the tuples emitted by each component based on the tuple ID associated with it. If a certain tuple emitted by a Spout is not acknowledged within a configured timeout, i.e., explicitly told that it is processed successfully by acknowledging the tuple at each stage, Storm assumes that it has failed and replays the data again.

In applications where intentional delays are added while processing data, since we intentionally wait before we actually push data further, even the acknowledgements sent to acker bolts will be delayed accordingly. This might lead to a failure hit at acker bolt which might lead to data replay. This is acceptable if the failure is genuine, but how feasible is it to consider even the delayed tuples which might soon be processed accounted as failures? Would this make any negative impact on the performance? Is there a better way to see if the tuples are actually failed or just delayed? Finding answers to such questions is the purpose of our current work.

To analyze this more, we have considered two metrics, *timeout (topology level message timeout which is used to treat tuples as failures)* and *data replay (when and how should the data be replayed in case of a failure).*

1.1. Timeout

As current fault tolerance mechanism in Storm supports only a topology level timeout, we can only choose one timeout value for the entire topology. In applications with intentional delays, our intuitive choice of choosing this timeout would be to sum up all the maximum possible delays expected in the topology. But, wouldn't this be too high? Because this is the worst case scenario which may not occur too often, we would end up waiting too long even if there is a genuine failure, which might impact the data latency. On the other hand, if we choose a timeout too low based on the lower bounds of the expected delays, we might end up with some premature and unwanted timeouts in turn causing unwanted data replays. At this point it is intuitive to guess that this notion of a per-topology timeout is not a good fit for these applications. Therefore an intelligent choice should be made on the timeout that we chose in these applications.

One better approach in this respect would be to support a notion of varying timeouts in Storm. This way, we can decide if the failure is spurious or a genuine failure. As a part of this, we propose an idea of changing Storm to have a per Edge timeout along with the existing optional per topology timeout (strictly speaking per stream id timeout which is explained in detail later).

Since we can estimate the maximum delay that can occur in a particular link in the given topology, we can only use that maximum bound as the timeout for that link in the topology. If we don't see an acknowledgement from the next node in the topology within this bound, then we can definitely assume that it is a failure. Unlike the earlier scenario, where it takes sum of all possible maximum delays to decide if the tuple is actually a failure or not, here we only take the maximum delay of that particular link before we flag the tuple as failed. This way, data can quickly be requested for processing again and hence a reduction in the latency of the data being processed can also be possible.

1.2. Data replay

As mentioned earlier, per edge timeouts could potentially benefit us in reducing the latency of the data being processed. But in spite of all this, if the tuples fail, it takes a really a long time for the tuple to be processed again which might lead to high data processing latency. In order to minimize this, another different aspect comes to light that can be changed in order to improve the latency of the data by changing the replay semantics that Storm uses.

Current implementation of Storm only relies on replaying failed data all the way back from the source (which is spout) rather than any preceding intermediate bolts that have successfully processed that tuple. Intuitively, we can say that, if we could somehow replay failed data from one of the closest preceding nodes (that has successfully processed the tuple) rather than all the way back from source, we can avoid a lot of re-computation in all the intermediate nodes which have successfully processed the tuple earlier. This could potentially lead to an improvement in the performance of the system in terms of process latency.

2. IMPLEMENTATION

Work done so far is only on adding the support of per edge timeout to storm and also check the impact of varying timeouts in wide range of applications. The implementation section talks more about only the timeout aspect among the two aspects mentioned in the introduction section.

2.1. Adding Acker Bolt for each Spout/Bolt.

Currently, Storm uses a hidden bolt called *Acker Bolt* which tracks the tuples based on the tuple ID and the corresponding spout which has emitted it. Once a spout emits a tuple (T), it adds that tuple ID to the acker bolt list of monitored tuples, in the subsequent computation nodes in the topology, whenever a new tuple (T') is emitted based on the consumption of this tuple (T), an acknowledgement is sent to acker bolt which is used to track the status of the tuple in the tuple tree. When the tuple ID is not acked by all the nodes in its tuple tree within the specified timeout, Acker Bolt **(*Figure1*)** notifies the Spout to replay it.

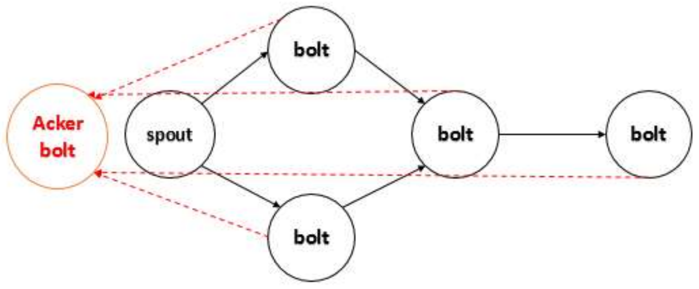

Figure 1

Using the same approach on a per node bases, i.e., there will be a unique AckerBolt per node **(*Figure2*)** which will only track its own messages using a timeout value equal to the timeout of the respective edge in the topology, whenever a timeout occurs, we could either use current Storm replay semantics or go with the new approach mentioned under introduction.

Drawback of this approach is that, we double the number of bolts in the topology by approximately 2 times as we are almost adding a new bolt (Acker) for every bolt in the topology. This might increase the overhead because of the node management i.e., distributing the resources (to be precise tasks and executors) among the nodes. On the other hand, the advantage would be to reuse the existing Storm's timeout mechanism for establishing the per edge timeout mechanism.

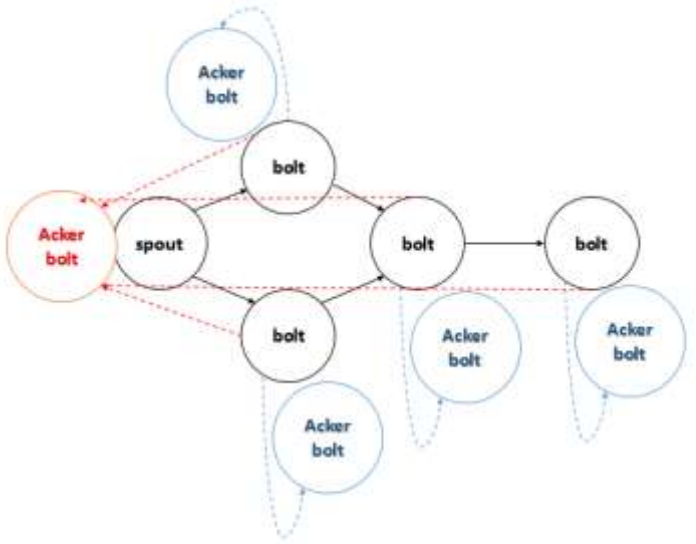

Figure 2

2.2. Adding Tuple Acking streams

Storm allows us to create different streams in the same topology which can have different timeouts even if they exist in the same topology. We can leverage this per stream timeout in order to establish a time bound communication link between the source and destination nodes of an edge i.e., if we see any bolt which wants to participate in the per edge timeout, we create a hidden edge with a new stream id which is used to pass acknowledgements from the destination node of the edge to its source node. This new hidden stream will be associated with a timeout (the maximum delay possible for the tuple to pass this current node) which will actually be the edge timeout ***(Figure 3)***.

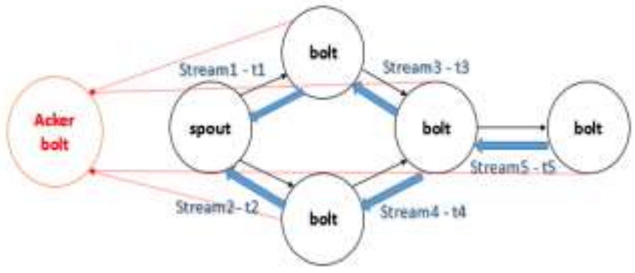

Figure 3

Drawback of this approach is that we need to implement the way tuples are tracked in order to see which of them have been timed out or which of them have been successfully processed within timeout. On the other hand, the advantage is that we will not increase the number of nodes in the topology.

Because of the same reasons, we have currently explored only approach2 for implementing the per edge timeout. Experiments section will give more insights on the results obtained using approach2.

3. DATA REPLAY SEMANTICS

As mentioned earlier, data replay semantics in current storm might lead to a lot of data replays especially when we are aggregating the data at each node before we flush out the data to the next node. This is because, data is compressed/aggregated significantly at each stage and even in case of a single failure in the downstream, it might lead to replaying a huge amount of data from upstream and also all the way back from source. This might lead to an overhead in terms and could lead significantly impact performance, resource consumption and ultimately staleness.

To avoid this, we can modify storm in a way that each node in the topology will be aware of and responsible for the traffic emitted by itself i.e., each node holds and tracks the tuples emitted by itself until a point where we can safely assume that it is no longer needed for processing downstream. Whenever a node notices a timeout using any of the earlier approaches, it can replay the failed tuples right from the immediate source of the node rather than right from the spout (***Figure4)***. This could save us a lot of computation at intermediate nodes and also help in minimizing the staleness of the data being processed.

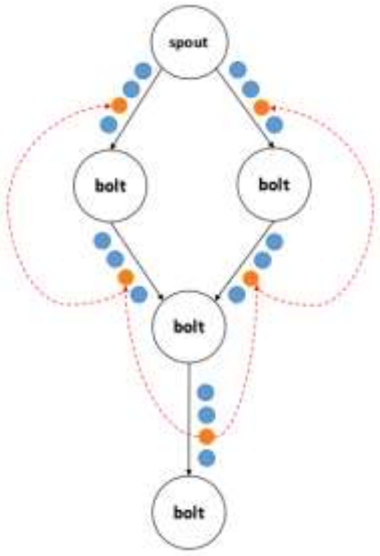

Figure 4

4. EXPERIMENTS

Experimental setup includes a 3 node storm cluster set up on the zen cluster (zen13, 14 and 15), testing this out using a more realistic test bed like planetLab is a part of Future work. We have used Word Count Topology for the experiments (shown below). Functionality of each component is described in detail below ***(Figure5)***.

Spout: Emits Sentences (group of words) in each Tuple.

Splitter: Splits the words in each sentence Tuple received as input and emit N Tuples where N is the number of words in the input sentence Tuple i.e., one input Tuple leads to one or more output Tuples.

Edge Aggregator: Aggregates similar words in the current window and emits a Tuple <word, count> to the Centre Aggregator. Where count means the count of number of times that word has been received in the current window.

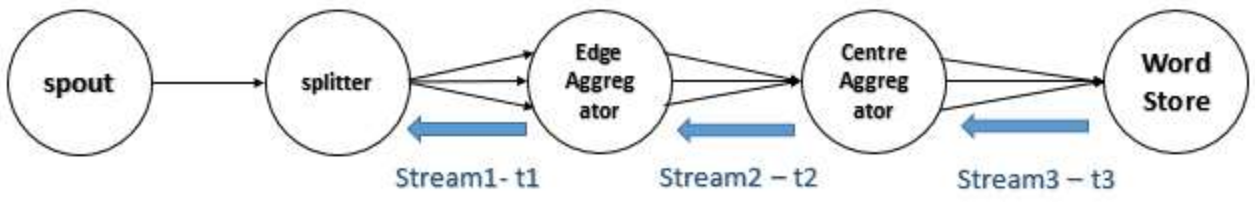

Figure5

This is an example of windowed aggregation where a set of one or more input Tuples contribute to one output Tuple.

Centre Aggregator: Aggregates words in the current window which start with the same character. This emits Tuple <character, count> where character means the character with which 'count' number of tuples have been received in the current window. This is an example of windowed aggregation where a set of one or more input Tuples contribute to one output Tuple.

Word Store: This will just store and print the character vs the aggregate count received so far from the start of the Topology.

4.1. Standard Storm with acking mechanism enabled

As the overall timeout increases, if there is a true failure, we end up waiting for a longer time before we actually announce it as a failure and reprocess the data. Therefore, in theory we should an increase in latency as timeout increases but experiments show it the other way, i.e., latency is decreasing as timeout increases as shown below. We need to come up with a custom metric calculation to verify if the latency estimated by storm is what we want to be considered as latency in our experiments ***(Figure6)***.

As the overall timeout increases, since we have more time for hearing back an acknowledgement from the internal nodes, there are less failures that occur due to missing acknowledgements on time ***(Figure7)***.

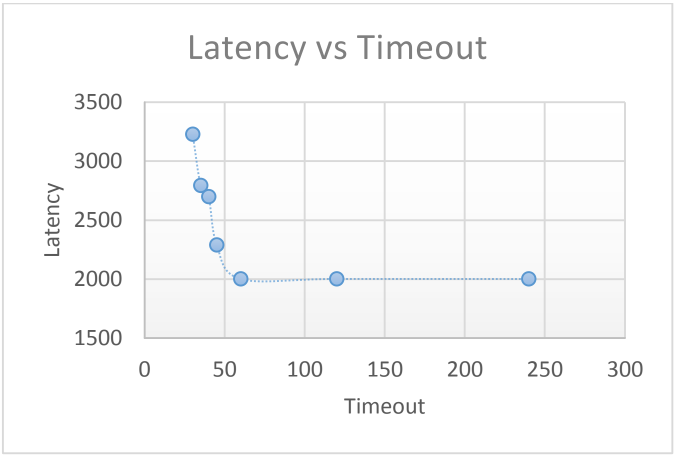

Figure 6

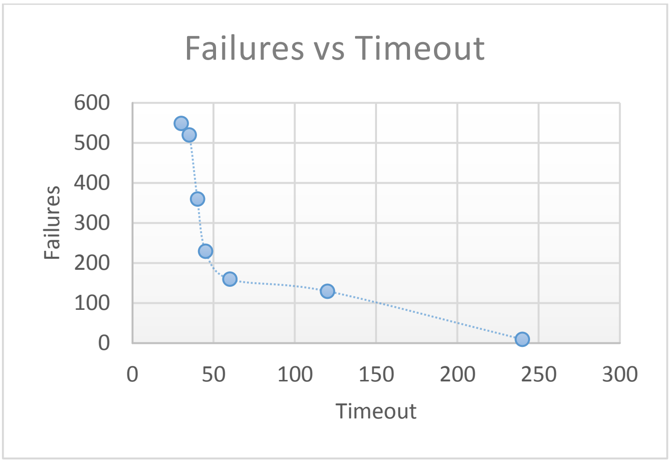

Figure7

4.2. PerEdge Vs Storm+PerEdge Timeout enabled

As the overall timeout increases, as per our theory and experiments, we see that number of failures come down. Also, experiments show that number of failures are less when we use only per edge acknowledgement when compared to using both storm acking mechanism and per edge acking mechanism. This can be attributed to additional work we need to do in order to help storm in tracking the tuples ***(Figure8).***

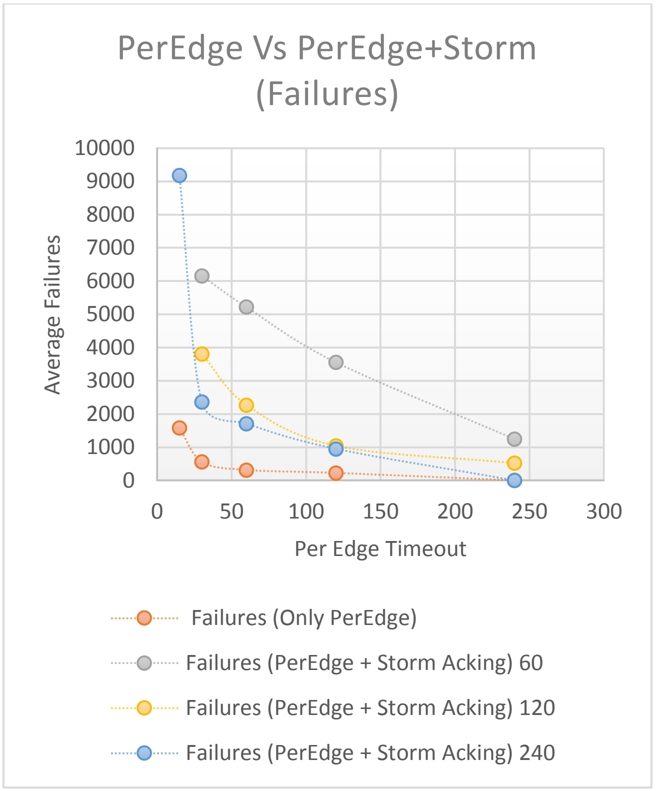

Figure8

Since current storm metrics calculation logic does not provide a measure to determine the latency directly when storm timeout is not used. I have added some logs to estimate this latency but looks like there is some variation in what storm calculates and what I calculate, need to look into it more as a part of future work and then we can confidently talk about the latency comparison (***Figure9)***.

4.3. Storm vs PerEdge

In the following 3, 4 and 5 experiments, we are comparing the sum of per edge timeouts vs the overall timeout. Experiments show very orthogonal results than we have theoretically estimated. Even though tuples are successfully carried forward with our per edge timeout mechanism, there is a very high delay in receiving the acknowledgements from the downstream. This is leading to false alerts when we have a per edge timeout set to the actual estimated value. Only if this value is raised to a very high exponent of about 10 times the actual value, we are able to see the results closer to what we theoretically expected.

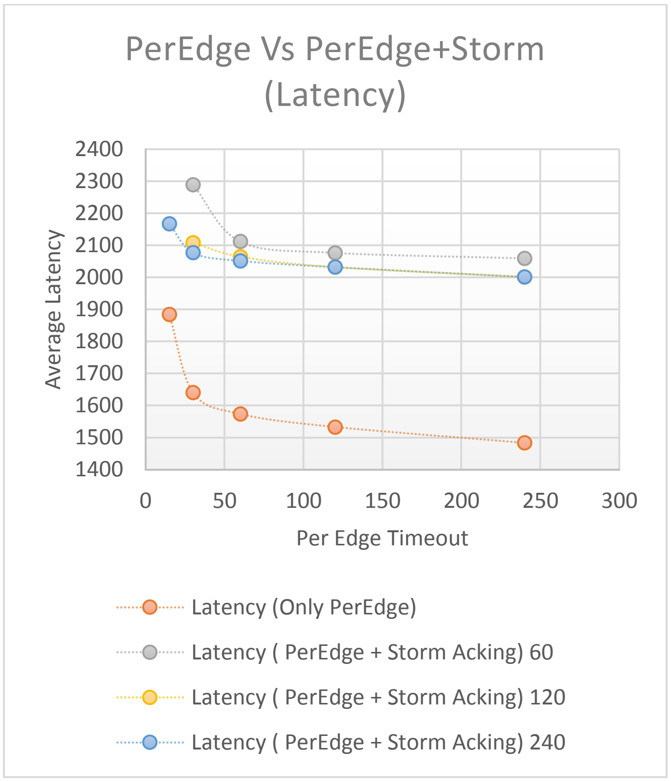

Figure9

Experiments are run for combinations where sum of timeouts is ranging from a value less than overall timeout to a sum of timeouts 10 times higher than overall timeout in each of the three cases where overall storm timeout is 240 (***Figure10, Figure11***), 120 (***Figure12, Figure13***) and 60 (***Figure14, Figure15***) respectively.

One possibility for this behavior could be that, during the windowed aggregation, since one tuple might lead to acknowledging thousands of tuples upstream, acknowledgement traffic is a lot higher than the actual data which might be causing a lot of buffering in the queues of the bolts and hence delaying the processing of acknowledgements. As a part of future work, I need to work on coming up with a solution which can only communicate failures rather than communicating acknowledgement for every tuple thus reducing the traffic which might lead to receiving acknowledgements in a more realistic time bound.

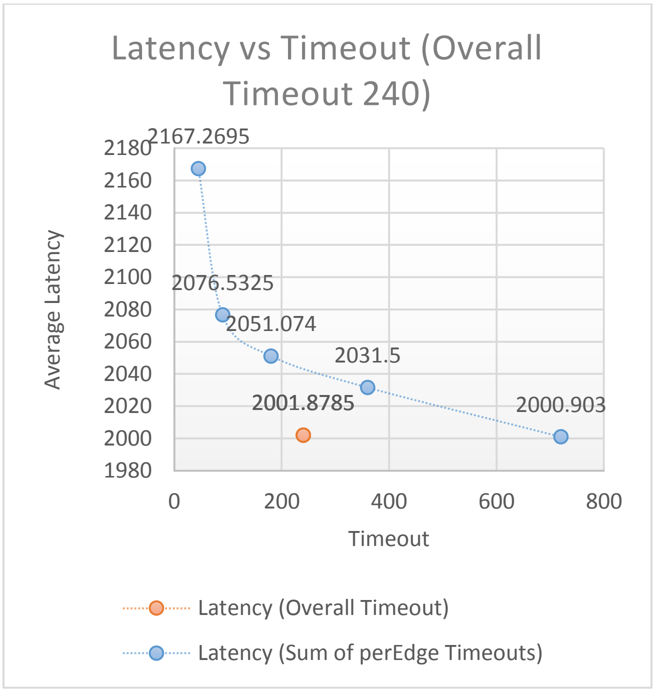

Figure10

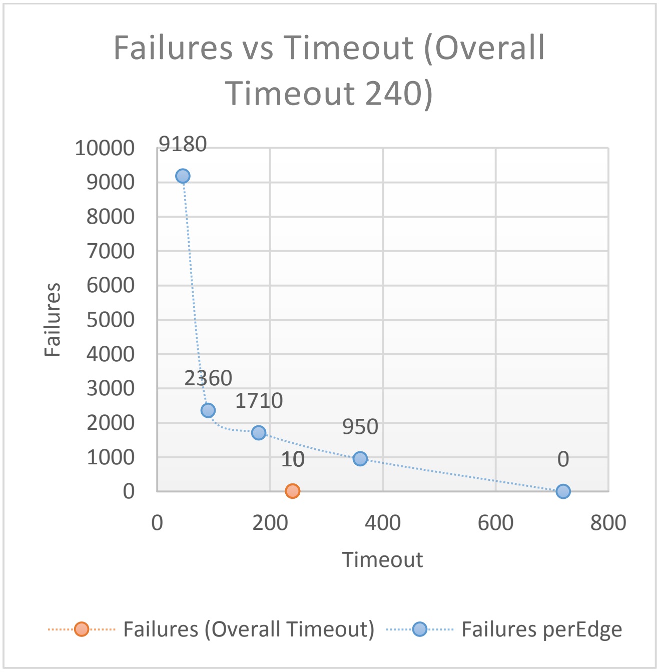

Figure11

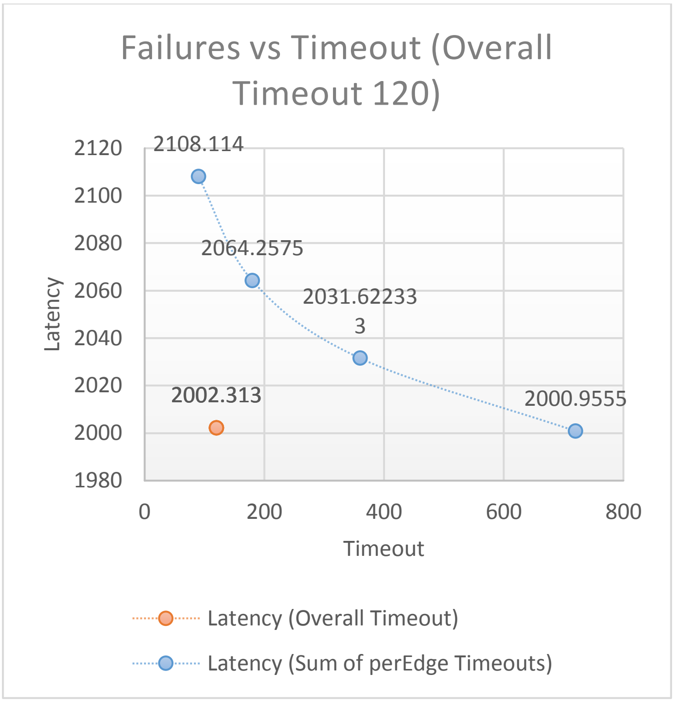

Figure12

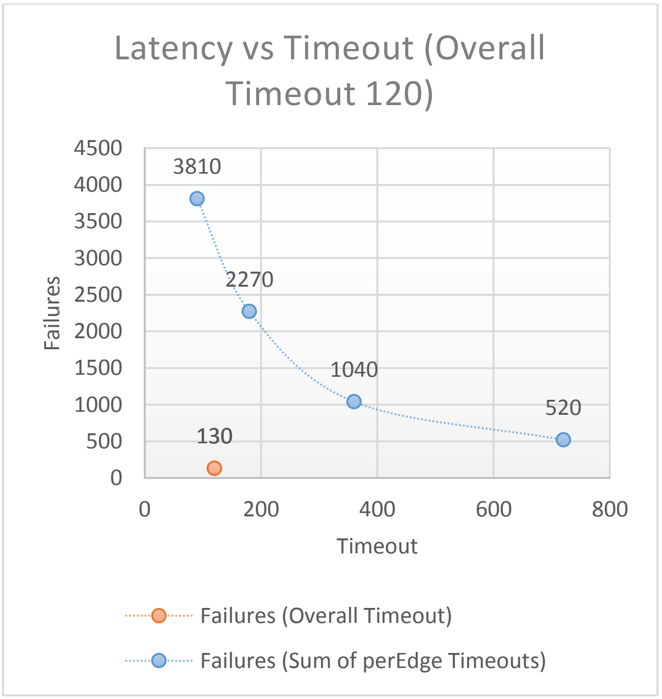

Figure13

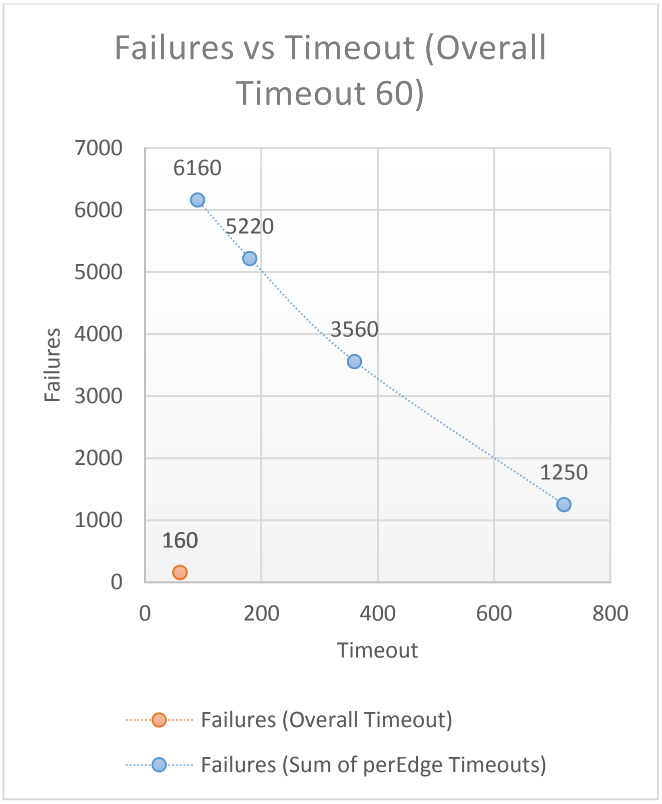

Figure14

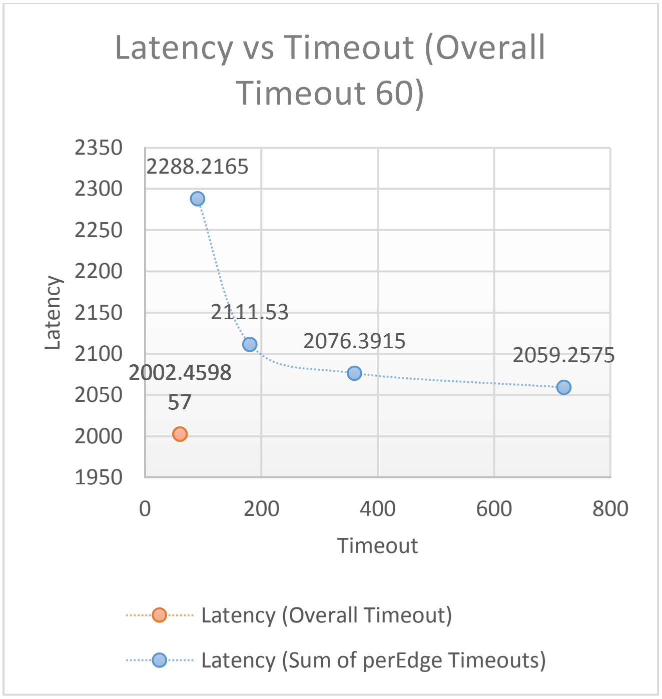

Figure15

5. CONCLUSION

Although per edge timeout mechanism gives us a better way to deal with the applications with intentional delays added during data processing, as learnt from the experiments each per edge timeout needs to be set to a very high value when compared to the actual values for avoiding false failure alerts. Only then, our results are in comparison with the standard storm. Therefore we need to come up with a different approach or improvise the current approach to scale the per edge timeout to a more realistic value.

Few approaches that I would like to test out as a part of future work are as follows

Negative Acknowledgements: The main overhead with the current implementation is that, every Tuple's acknowledgement is carried back to its source. In aggregated data processing systems, since one tuple might lead to thousands of Tuples downstream, these thousands of Tuples needs acknowledgements, which leads to a high number of acknowledgement Tuples when compared to the actual data that is pushed downstream. Therefore, we can think of a design aspect which only sends negative acknowledgement indicating a failure rather than acknowledging every processed Tuple.

Grouped Acknowledgements: Instead of sending an acknowledgement for each and every Tuple that is pushed downstream, we could think of a possibility to send a grouped acknowledgement which indicates that all Tuples within the given range are acknowledged. This way we can avoid heavy traffic just for acknowledgements.

6. RELATED WORK

We have come across Discretized streams [5], D-Streams which focusses on automatically handling failures in a stream computing system using RDD [18] which enables a parallel recovery mechanism. Although this system improves efficiency over the traditional replication and backup schemes, these systems do not provide the flexibility to handle data processing with intentional delays. Therefore such systems cannot be leveraged for data processing where intentional delays are incorporated.

There is another system TimeStream which uses a mechanism that tracks data dependencies between the output and input streams to enable efficient re-computation based failure recovery that achieves strong exactly-once semantics. This is very close to what we have described under introduction section 3. In fact this has been the inspiration for incorporating such an idea into storm. Now the question is why not use Time Stream instead of storm as it has all the required fault tolerance and data tuple tracking mechanism in place? The answer to the question is that TimeStream uses micro batching in order to achieve the continuous streaming kind of semantics. But in case of windowed aggregation we need more control on when and how often data needs to be sent. Frequency at which data should be sent out might also dynamically change based on the type of data that is being processed. As TimeStream cannot cater for such requirements, we cannot use it for data processing with intentional delays.

There are other stream computing systems like spark which provide a better fault tolerance mechanism when compared to storm by using RDD's notion of lineage i.e., if some data is lost, then it can be recomputed using other RDD from which the current RDD is derived from. Even though RDD provides good abstraction for efficiently handling failures, it does not provide enough control on the internals of data processing/execution which does not let us work with applications involving intentional delays during data processing.

Thus, though there are systems which handle fault tolerance in a better way when compared to Storm, they cannot be used by themselves for supporting data processing with intentional delays. This strengthens our motive of improving the fault tolerance mechanism in Storm such that storm could be used as a one stop for applications which involve intentional delays in data processing.

7. FUTURE WORK

Since the current per edge timeout implementation added to storm basic mechanism is not exactly as expected, need to investigate if the implementation can be improved in a way that we can only communicate failures rather than every acknowledgement or send some kind of grouped acknowledgement upstream which minimizes the upstream traffic and improve the performance.

If the results with new implementation changes are as theoretically expected, then we can analyze the behavior of new fault tolerance mechanism in storm for more realtime scenarios, compare the current statistics with the new version of storm statistics in various scenarios as mentioned below.

Additional resource utilization: Since we track tuples and wait until previous k-windows (k is typically 2 or 3) at each edge, resource consumption is expected to rise slightly, should not be too high though. Also, average processing time per component with and without per edge timeouts should remain almost the same.

Lost/Killed workers & Topology rebalance: Since Storm does not transfer state information when it restarts jobs from failed workers to other worker nodes, this event might lead to a surge of failures which will also happen in case of basic storm. But, since the original storm will wait until the end of overall time window to replay data unlike the new storm which only waits until the timeout of the corresponding edge, latency is expected to go down.

Where failures are detected: The earlier the failures happen the better improvement we achieve in per edge timeout as we report a failure soon and replay the data soon which leads to re-processing the failed data soon and therefore latency should come down.

8. ACKNOWLEDGEMENTS

I firstly thank Professor Abhishek Chandra for guiding me through this project. I would also thank Benjamin Heintz for sharing his valuable feedback on the design and implementation related to per edge timeouts added to storm.

9. REFERENCES

[1] Apache Storm, https://github.com/apache/storm, 2015.

[2] Apache Storm Documentation, https://storm.apache.org/documentation/Home.html, 2015.

[3] Storm Applied, MEAP Edition, Version 10, Sean T.Allen, Peter Pathirana, Matthew Jankowski.

[4] Spark: Cluster Computing with Working Sets, Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica, University of California, Berkeley.

[5] Discretized Streams: Fault-Tolerant Streaming Computation at Scale. Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, Ion Stoica. SOSP'13.

[6] Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker and Ion Stoica. NSDI 2012.

[7] Storm @Twitter, Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel*, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong

Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, Dmitriy Ryaboy.

[8] TimeStream: reliable stream computation in the cloud. Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. EuroSys '13.

[9] MillWheel: Fault-Tolerant Stream Processing at Internet Scale. Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, Sam Whittle. VLDB'13.

[10] Apache Zookeeper, https://zookeeper.apache.org/

[11] MillWheel: Fault-Tolerant Stream Processing at Internet Scale. Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, Sam Whittle. VLDB'13.

[12] Apache Kafka, http://kafka.apache.org/

[13] Apache Trident, https://storm.apache.org/documentation/Trident-API-Overview.html

[14] J.-H. Hwang, U. Cetintemel, and S. Zdonik. Fast and highly-available stream processing over wide area networks. In Proc. of ICDE, pages 804–813, 2008.

[15] Optimizing Grouped Aggregation in Geo-Distributed Streaming Analytics, Benjamin Heintz, Abhishek Chandra, Ramesh K Sitaraman

[16] Balazinska, M., Balakrishnan, Madden, S., Stonebraker, M., Fault-tolerance in the Borealis distributed stream processing system. In SIGMOD 2005.

[17] ZeroMQ, http://zguide.zeromq.org/page:all

[18] Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker and Ion Stoica. NSDI 2012.