CS425, Distributed Systems: Fall 2024
Machine Programming 4:

Released Date: Nov 12, 2024
Due Date (Hard Deadline): Sunday, Dec 8, 2024 (Code+Report due at 11.59 PM)
Demos on Monday Dec 9, 2024

The is a very intense and time-consuming MP! So please start early! Start now!
(Disclaimer: Like all MPs and HWs, all references to all companies and people in this spec are purely fictitious and are intended to bear no resemblance to any persons or companies, living or dead.)

ExSpace (MP3) miraculously merged with two social media companies Qwitter and KitKat to form a new conglomerate called…MartWall Inc. MartWall loved your previous work at ExSpace (and they're also aware of your great work on the mission to Saturn in HW3), so they've hired you as a "MartWall Fellow". That's quite prestigious! Congratulations!

You must work in groups of two for this MP. Please stick with the groups you formed for MP1. (see end of document for expectations from group members.)

MartWall needs to fight off competition from their three biggest competitors: Pied Piper Inc., Hooli Inc., and Amazing.com. They also must fight the twin scourges of fake news and online shopping! So, they've decided to build a stream processing system that is faster than Storm. You have two tasks in this MP:

1. Your first task at your job is to use the HyDFS (from MP3) and the failure detector (MP2) to build a new stream-processing framework called RainStorm**,** which bears similarities to stream processing frameworks such as Storm, Samza, Spark Streaming, MillWheel (https://dl.acm.org/doi/pdf/10.14778/2536222.2536229).
2. Your second task is to **compare** RainStorm's performance against the latest version of Spark Streaming (this involves deploying and running Spark Streaming on the VMs).

These tasks are sequential, so please start early, and plan your progress with the deadline in mind. Please DO NOT start a week (or even two weeks) before the deadline – at that point you're already too late and will likely not be able to finish in time.

Below, MartWall has been very detailed about the design of RainStorm. However, they want you to fill in some gaps in the design, and of course to implement the system. Be prepared to improvise, be prepared to deploy new systems, and be prepared for the unknown. Remember – Move Fast and Break Things! (i.e., build some pieces of code that run, and incrementally grow them. Don't write big pieces of code and then compile them all and run them all!) RainStorm shares similarities with some of the above-mentioned stream processing systems except that it is simpler. You can look at the docs and code of these systems, but you cannot reuse any code from there (we will check using Moss. Also, it's faster to write your own RainStorm than borrow and throw out code.). In this MP you should look to use code from MP1, MP2, and MP3.

At its core, RainStorm is a stream-processing framework where users can specify a sequence of user-defined transformations on input data that produces one or more streams of output data. Unlike MapReduce, where users need to wait for entire computation to complete, RainStorm (like other stream processing frameworks) allows one to perform real-time analytics on a continuous stream of input data. Each of these transformations is parallel, and they are not separated by a barrier like MapReduce. RainStorm is intended to run on an arbitrary number of machines, but for most of your experiments you will be using up to the max number of VMs you have.

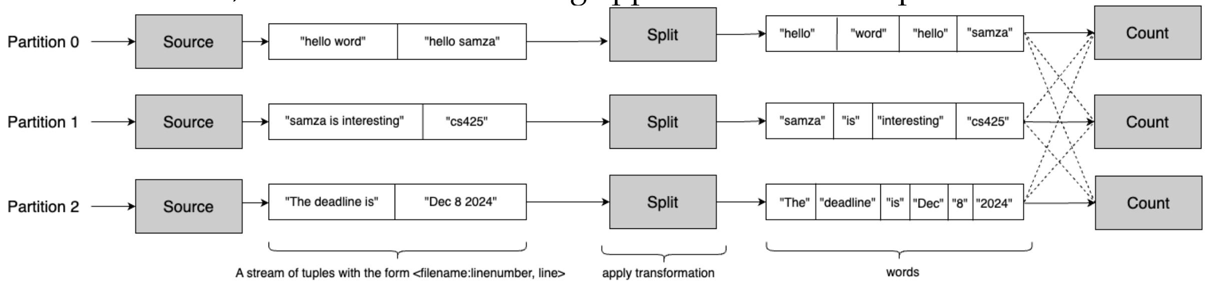
A stream is a sequence of tuples of form <key, value>. Real stream processing frameworks support arbitrary DAGs with many stages of transformations /operators. However, for your MP, RainStorm will need to support not more than two-three processing stages. Additionally, you will have a source of streams that will read input files from HyDFS and produce a stream of tuples of the form <filename:linenumber, line> for the first stage to consume.

Like MapReduce, each stage must have parallel tasks. Each task in a stage will process one or more input streams from the previous stage and produce one or more streams for tasks in the next stage. The final stage will stream outputs to the console continuously along with appending outputs to a single HyDFS file. Incoming streams are split among the tasks of a stage using a partitioning function on the key.

RainStorm tasks can perform three types of operators:

1. Transform: apply a user-defined function on records and pass on to the next stage
2. FilteredTransform: filter records which satisfy a condition and transform them; pass zero or more records to the next stage
3. AggregateByKey: maintains a running aggregate statistic of all records seen so far and passes the aggregate to the next stage. Note that this type of operator must maintain state across invocations unlike the previous two.

Given the above, a word count streaming application can be implemented as follows:



First, multiple Source processes continuously read from their partition on HyDFS and produce a stream of lines and pass it on the first stage. The first stage (Split) splits the incoming lines into words. Multiple Split tasks will apply this transformation in parallel on the stream of lines. The second stage is Count that maintains a running count for the different words and continuously outputs the total counts so far for the words to the console and to a file on HyDFS (every interval, say 100ms). Like MapReduce, each stage has multiple tasks. Each stream is partitioned by key. Streams can be repartitioned into a different key. For example, the key for the input to Split is file:linenumber, whereas the keys for the input to the Count are the words. You will run at least 2-3 parallel tasks per stage. Unlike MapReduce, there is no barrier between the Split and Count stages. Split continuously streams tuples to Count and Count continuously streams the running counts to the console and HyDFS.

We will ask you to write custom operators/transformations for the demo. The exact application will be revealed when we release the demo instructions.

We will invoke RainStorm as follows.

```
RainStorm <op1 _exe> <op2 _exe> <hydfs_src_file> <hydfs_dest_filename> <num_tasks>
```

The parameters op1_exe and op2_exe are user-specified executable that takes as input a small batch of <key, value> tuples and produces zero or more <key,value> pairs. op1_exe and op2_exe are the file names local to the file system of wherever the command is executed. The last series of parameters (hydfs_src_file) specifies the input file. You must *partition* the entire input data so that each of the source processes receives about the same amount of input. Hash partitioning is ok. Each task (at every stage) is responsible for a portion of the keys – each key is allotted to exactly one task in a stage (this is done by the Leader server). num_tasks is the number of tasks to run per stage of the application.

Design: The RainStorm cluster has N (up to max number of VMs) server machines. One of them is the leader server, and the remaining N-1 are worker servers. Similar to MapReduce, the tasks are scheduled on a worker machine. The leader is responsible for all critical functionalities: receiving commands, scheduling appropriate tasks, allocating keys to tasks, and dealing with failures of the other (worker) servers. In short, any coordination activity, other than failure detection/membership, can be done by the leader. Additionally, you may assume that the leader server/RM is fault-free.

Worker failures must be tolerated. When a worker fails, the leader must reschedule the task quickly. When a worker rejoins the system, the leader must consider it for new tasks. Worker failures must not result in incorrect outputs (more on this below).

Mandatory: Fault Tolerance and Exactly-once delivery semantics. One of the main features in RainStorm is its exactly-once delivery semantics in the presence of failures, i.e., each input record is processed only once even in the presence of failures. To provide this, each stage including the source stage, will keep track of what records have been processed by the tasks in the subsequent stages.

In RainStorm, once a record is processed by a task, it acknowledges the sender task; the sender then forgets the records. If the sender doesn't receive an ack on time, to handle failures, the sender will retry sending the record until it gets an ack. Now because retries might result in duplicates, any input record must be checked for duplicates from previous record deliveries; the framework should discard any duplicate records. Moreover, when rescheduling tasks during failures, the newly scheduled task shouldn't process any records that were processed and then delivered to the next stage by the failed task. What information does the framework need to maintain for duplicate detection? One way is to associate each tuple with a uniqueID and each task remembers the IDs processed so far. Each task will then discard tuples previously seen before without processing.

But how can you detect duplicates across failures? So, one needs to maintain the uniqueIDs a task has processed in persistent storage. RainStorm uses an append-only log on HyDFS for every task for this purpose. Every tuple is assigned a uniqueID and a task logs the uniqueIDs it has processed to the HyDFS file. Also, before producing outputs for the next stage, the outputs are first journaled to the HyDFS file. Only then the sender is acknowledged.

In case of failure, the journal is replayed, and the recovered task can use the IDs from the log file to filter duplicates and only process the previously unseen data. It can also resend the outputs (recovered from the log) for which it has not received acks. But logging when processing every record might be expensive. Can you batch multiple records and interact with HyDFS once for a batch of input records? Can you also batch writing the uniqueIDs of input records and the writing of output records in the same append to HyDFS?

Mandatory: State recovery. For stateful operators like the Count, even if there is a failure, the task must still produce correct counts. So, how does a task remember counts for all records previously seen across failures? For example, if the Count task failed after processing 10 counts of a word "xxx". The next time it sees "xxx" after a failure, it should output 11 as the count and not 1 (just because it forgot the previous count). So, stateful operators additionally persist its state to HyDFS (for Count task, the state is the list of keys and their counts). Do you need to persist the entire state? Or can you just log the changes to the state?

So, here is a starter design for the framework. When an operator receives an input tuple for a task, RainStorm first checks if the record is a duplicate for the task. If so, it discards it. If not, it runs the user-specified transformation on it. Any changes to the state (if state is maintained by the operator) due to the processing of the record, the output records, and the input recordIDs processed are all journaled to a file in HyDFS. The previous stage is acked. The outputs are sent to the next stage and buffered until the next stage acknowledges it, after which the outputs can be forgotten by the current stage. Improve on the design for performance and handling all failure cases. Note that you need to implement fault tolerance and exactly once semantics for both stateless and stateful operators, but state recovery is only needed for stateful ones.

You must use the code for MPs1-3 in the RainStorm system. Use MP1 to log, MP2 to detect failures, and MP3 to store the inputs, outputs, and intermediate data for RainStorm.

Create logs at each machine (queriable via MP1 or via grep). You can make your logs as verbose as you want them (for debugging purposes), but at the least a worker must log each time task is started locally, and the leader must log whenever a job is received, each time a task is scheduled or completed, and when the job is completed. We will request to see the log entries at demo time, either via local grep or the MP1's querier.

Other parts of the design are open, and you are free to choose. Design first, then implement. Keep your design (and implementation) as simple as possible. Use the adage "KISS: Keep It Simple Si…". Otherwise, MartWall Inc. may, in their anger at your complex design, fire you into space (or worse, into deep see near the Titanic), with only a book to read. That would be a very boring vacation, right?

We also recommend (but don't require) writing tests for basic scheduling operations. In any case, the next section tests some of the workings of your implementation.

Dataset information appears later in this document (you may use others). Try to use datasets that are at least 100s of MBs large (if possible, even larger). Where datasets may not be available, create (auto-generate) synthetic datasets that are non-trivial and use these in your experiments. Smaller is ok for tests, but ensure that the run time is at least a few tens of seconds, so that comparison makes sense.

Comparison against SparkStreaming: After you have your RainStorm working, make it more efficient. Can you make it faster than SparkStreaming? Download it from https://spark.apache.org/streaming/ and run it on your VMs (this step will take some effort, so give it enough time!).

Compare the performance of RainStorm with SparkStreaming**, and see if you can make RainStorm faster.** Most of the inefficiencies in RainStorm will be in accessing storage, so think of how your files are written and read.

Make sure that you're comparing RainStorm and SparkStreaming in the same cluster on the same dataset and for the same topology. To measure performance, use output records produced per time. Make the comparison *fair* -- Run the same job with the same settings for both systems (for example, you want to use the same batching interval or size). Are you able to beat SparkStreaming?

Datasets: Good places to look for datasets are the following (don't feel restricted by these):

- Champaign Map Databases are available at: https://gis-cityofchampaign.opendata.arcgis.com/search?collection=Dataset . Try queries like finding number of parking meters or number of apartment buildings with > 100 residents, etc. Look for other "GIS" databases.

- Stanford SNAP Repository: http://snap.stanford.edu/
- Web caching datasets: http://www.web-caching.com/traces-logs.html
- Amazon datasets: https://aws.amazon.com/datasets/
- Wikipedia Dataset: http://www.cs.upc.edu/~nlp/wikicorpus/

Machines: We will be using the CS VM Cluster machines. You will be using all your VMs for the demo. The VMs do not have persistent storage, so you are required to use git to manage your code. To access git from the VMs, use the same instructions as MP1.

Demo: Demos are usually scheduled on the Monday right after the MP is due. The demos will be on the CS VM Cluster machines. You must use up to the max VMs for your demo (details will be posted on Piazza closer to the demo date). Please make sure your code runs on the CS VM Cluster machines, especially if you've used your own machines/laptops to do most of your coding. Please make sure that any third party code you use is installable on CS VM Cluster. Further demo details and a signup sheet will be made available closer to the date.

Language: Choose your favorite language! We recommend C/C++/Java/Go/Rust/Python.

Report: Write a report of less than 3 pages (12 pt font, typed only - no handwritten reports please!). Briefly describe your design (including architecture and programming framework) for RainStorm, in less than 0.75 pages. Be concise and clear.

Show plots comparing RainStorm's performance to SparkStreaming for at least 2 different scenarios (involving datasets – you can pick from the above list or your own). For each dataset scenario, pick TWO examples of sequence of operators (one simple, one complex). So, there should be a total of FOUR scenarios.

For each data point on the plots, take at least 3 measurements, plot the average (or median) and standard deviation. Run each experiment (for each system) on all the VMs in your allocation. **Devote sufficient time for doing experiments** (this means finishing your system early!).

Discuss your plots, don't just put them on paper, i.e., discuss trends, and whether they are what you expect or not (why or why not). (Measurement numbers don't lie, but we need to make sense of them!) Stay within page limit. Make sure to plot average, standard deviations, etc.

Submission: (1) Submit your report to Gradescope BEFORE the deadline, and TAG your page(s) on Gradescope (there will be a penalty if you don't). Only one group member should submit – you can tag your partner.

(2) There will be also be a demo of each group's project code. Signup sheets will be posted on Piazza.
(3) Submit your working code via gitlab sharing (including your group number in your project name). Please include a README explaining how to compile and run your code. Default submission is via gitlab sharing – please include your group number in your gitlab share names! Further submission instructions will be posted on Piazza.
Other submission instructions are similar to previous MPs.

When should I start? Start NOW. You already know all the necessary class material to do this MP. Each MP involves a significant amount of planning, design, and implementation/debugging/experimentation work. **Do not** leave all the work for the days before the deadline – **there will be no extensions**.

Evaluation Break-up: Demo [60%], Report (including design and plots) [30%], Code readability and comments [10%].

Academic Integrity: You cannot look at others' solutions, whether from this year or past years. We will run Moss to check for copying within and outside this class – first offense results in a zero grade on the MP, and second offense results in an F in the course. There are past examples of students penalized in both those ways, so just don't cheat. You can only discuss the MP spec and lecture concepts with the class students and forum, but not solutions, ideas, or code (if we see you posting code on the forum, that's a zero on the MP). FakeNews Inc. is watching and will be very Sad!

We recommend you stick with the same group from one MP to the next (this helps keep the VM mapping sane on EngrIT's end), except for exceptional circumstances. We expect all group members to contribute about equivalently to the overall effort. If you believe your group members are not, please have "the talk" with them first, give them a second chance. If that doesn't work either, please approach Aishwarya/Indy.

Happy Streaming (from us and the fictitious MartWall Inc.)!