Updated on 11/17:

1. Clarification on INPUT_RATE parameter in page4

CS425, Distributed Systems: Fall 2026
Machine Programming 4: Stream Processing

Released Date: Nov 10, 2026
Due Date (Hard Deadline): Sunday, Dec 6, 2026 (Code+Report due at 11.59 PM)
Demos on Monday Dec 7, 2026

This is a very intense and time-consuming MP! So please start early!

LLM Usage: You are allowed to use LLMs for any part of your MP4, **including the actual coding** (which is a difference from MP3). In other words, you are free to use LLMs for any, all, or none of designing, writing test code, writing the actual code, testing, debugging, and running experiments in your MP. Do not "one-shot" the MP code; we will check the collected logs. If you choose to use LLMs, here are some recommended ways to use LLMs. (1) Design: Come up with a rough draft of the design, and then use your favourite LLM as a sounding board to iterate on and improve that design, and to break it down into modules. (2) Testing: You may then use LLMs to generate unit test cases for each module, and then eventually integration tests. (3) Break down your design into smaller components, and generate code for each component, and verify with the unit tests. (4) Put all the components together, and verify with your integration tests. (5) Debugging: You may use LLMs to run the tests against your code and help debug your code. (6) Experiments: you may use the LLMs to download and set up Spark, to run your experiments, collect data, and generate the graphs, **but the rest of the Report, including the analysis of the graphs/data must be authored by only you**.

Crucially, you must document your use and experience with the LLM in the Report. Additionally, you must submit logs that contain your prompts to the LLM and the LLM responses (please see instructions in the report section).

Given the increased use of LLMs to generate code in the industry, these rules for MP4 serve an educational purpose. However, note that, as in your future job, you and only you are responsible for the correctness and efficiency of any design & code that you submit. Additionally (as in your future job), you must understand and be able to explain your code to (your tech-lead or manager) the TA during the Q&A portion of your demo slot. If something fails in the demo, it is your responsibility, not the LLM's. If you're unable to explain your code, you lose points. Proceed cautiously and judiciously. You must work in groups of two for this MP. Please stick with the groups you formed for MP3. (See end of document for expectations from group members.)

For MP4, you will build a stream processing system. There are two parts to this MP.

1. Use DKVS (from MP3) and the failure detector (MP2) to build a new stream-processing framework called **RainStorm,** which bears similarities to other stream processing frameworks such as Storm, Spark Streaming, and MillWheel (https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/41378.pdf).

2. Compare RainStorm's performance against the latest version of Spark Streaming (this involves deploying and running Spark Streaming on the VMs).

Downloading and deploying Spark to run on the VMs will take some time, and can be done in parallel, so please start early, and plan your progress with the deadline in mind. Starting even 2 weeks before the deadline will be too late, and you're unlikely to finish in time.

We have provided detailed requirements for and some design elements of RainStorm. However, we want you to fill in the gaps in the design and to implement the system. RainStorm shares similarities with some of the aforementioned stream processing systems. You can look at the docs and code of those systems, but not reuse any code. In this MP, you should look to use code from your earlier MPs.

RainStorm is a stream-processing framework where users specify a sequence of transformations on input data to produce one or more streams of output data. Unlike MapReduce, where one waits for the entire computation to complete, stream processing frameworks like RainStorm allow one to perform real-time analytics on a continuous stream of input data. The user-defined transformations run in parallel, and are not separated by barriers like in MapReduce. RainStorm can run on an arbitrary number of machines, but most of your experiments will be limited by the maximum number of VMs you have.

A stream is a sequence of <key, value> tuples. Stream processing frameworks support many stages of transformations. However, RainStorm supports no more than three processing stages. Additionally, RainStorm must read an input file from the VM on which the "Source Stream" task (in the Figure below) runs, and use that to produce the source stream of <linenumber, line> tuples for the first stage to consume.

Like MapReduce, each stage must have parallel tasks. Each task in a stage must process one or more input streams from the previous stage and produce one or more streams for tasks in the next stage. The stream from the final stage must be output to the console (continuously) and be stored to DKVS. The stream from the final stage must also be eventually written out into a file at the end of the entire run (the `final_stage_output` command is explained later). Because MP3's DKVS supports a value size of at most 1MiB, the final stage must store the output stream as 1 MiB chunks, which can then be read and reconstituted into a single output file for test validation[1]. Data streaming into a stage is partitioned among the tasks of the stage uniformly (you could use hash partitioning).

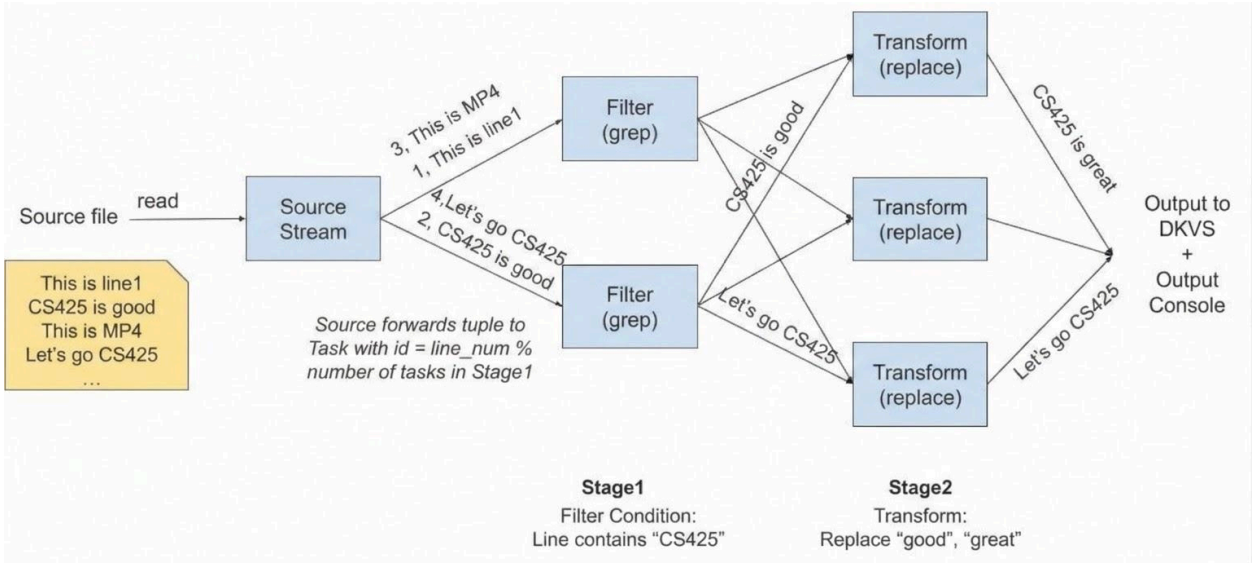
RainStorm tasks support three types of operators:

1. Transform: apply a user-defined function on each tuple, and send to the next stage (e.g., Replace)
2. Filter: filter out tuples that don't satisfy a condition, and send the remaining to the next stage (e.g., Grep)
3. AggregateByKey: maintain a running aggregate statistic of all records seen so far and pass the aggregate to the next stage (e.g., count grouped by a key). Note that this type of operator must remember the state (i.e., the aggregate statistic) across invocations, unlike

[1] If your MP3's DKVS has a higher max value size, you must modify it to allow no more than 1MiB sized values; max value sizes smaller than 1MiB are also perfectly acceptable. MP4 requires you to design per-task logs that are larger than a single DKVS entry, and therefore we impose a 1MiB limit.

the previous two.

Given the above, a grep+replace streaming application can be implemented as shown in this Figure.



First, a source task continuously reads from the source file stored locally in that VM, produces a stream of lines, and sends it to tasks in the first stage based on the line number. Each of multiple parallel tasks in the first stage (Grep) filters out lines that do not contain the user-specified grep pattern, and forwards only lines with the matching pattern to the second stage. Each of multiple parallel tasks in the second stage (Replace) replaces a user-specified grep pattern in each line with another user-specified replace pattern, and continuously outputs the lines to the local console and to DKVS.

Like in MapReduce, each stage must have multiple tasks. The stream between two stages is partitioned by key. For example, the key for the input to the Grep stage is the linenumber, whereas the key for the input to the Replace stage is the line. You must *partition* the incoming stream to a stage such that each task receives approximately the same number of tuples[2]. You could use hash partitioning on the key modulo the number of tasks in the next stage. Each key must be assigned to exactly one task in a stage. Unlike MapReduce, there is no barrier between stages. Each task forwards any tuple it has processed immediately to the next stage, and tasks in the last stage must output tuples to the console and store them in DKVS.

RainStorm is invoked as follows:

```
RainStorm  <Nstages> <Mtasks_per_stage> \
           <op1_exe>  <op1_args>  …  <opNstages_exe>  <opNstages_args>  \
           <input_src_file> <DKVS_dest_file> \
           <exactly_once:true|false>  <autoscale_enabled:true|false>  \
           <INPUT_RATE> <LW> <HW>
```

The first parameter specifies the number of stages in the application. The second specifies the number of tasks in each stage. The next is a series of parameter `op_exe` and `op_args` pairs, one per stage, which are the user-specified executable and arguments for that stage. Each `exe` takes

[2]You are not required to design load balancing guarantees. A best-effort approach will suffice.

as input a <key, value> tuple and produces zero or more <key,value> tuples. In our example application, `op1_args` can be the grep pattern to be used by `op1_exe` from stage1. The `input_src_file` parameter specifies the location/name of the input file stored locally in the VM running the Source task. The `DKVS_dest_file` must be used as a prefix for the keys to the output stream getting stored in DKVS, and is also the name of the output file that should be reconstituted using those DKVS entries. The next two parameters (`exactly_once` & `autoscale_enabled`) are booleans that control the toggling of two features. The Source process must sustain the stream at INPUT_RATE tuples(lines)/second. If INPUT_RATE < 0, then your source stream rate should run uncapped; this may be needed for experiments comparing RainStorm with Spark. And the last 2 ~~3~~ parameters need to be specified only if `autoscale_enabled` is true. More on them later.

Note that `op1_exe`, `op2_exe`, etc., are custom programs that you will write based on the stream processing application (similar to how one writes custom Map and Reduce functions). We will specify the exact stream processing application when we release the demo instructions. Thus, your framework must take arbitrary `op_exe` programs and `op_args` that filter/transform a tuple and produce one or more tuples for each stage. You can implement the operators to take input via stdin and produce output via stdout, and their RainStorm task can feed the input to and receive the output from the operator.

Design: The RainStorm cluster has N (up to max number of VMs) server machines. Similar to MapReduce, RainStorm worker tasks are scheduled on these machines. A leader task is responsible for all critical functionalities, including receiving commands, scheduling tasks, determining the hash partition functions, and handling failures of the tasks. In short, any coordination activity is done by the leader. The leader should distribute the RainStorm worker tasks *approximately evenly* among the VMs; the number of overall worker tasks in RainStorm should not be limited by N. Additionally, we assume in MP4 that **the leader task and the Source stage task will never fail**.

In MP4, we will **not crash or stop any of the VMs or the DKVS tasks**. Failures will be triggered by killing only RainStorm worker tasks. Therefore, you must implement each RainStorm task as a separate process, so it can be killed without impacting any VMs or DKVS tasks. RainStorm task failure must be detected within 3-4s; when a task fails, the leader must restart the task and add it to the appropriate stage. Task failures must not result in incorrect outputs (more below). RainStorm must tolerate **up to 2 concurrent task failures**.

Mandatory #1: Exactly-once semantics. RainStorm supports **exactly-once semantics even in the presence of failures**, i.e., each input tuple is processed exactly once by each stage. This feature is toggled by the `exactly_once` command-line parameter. A failure is triggered by killing a specific task of a stage. Note that only specific RainStorm tasks are failed, but not DKVS tasks. That is, **DKVS can be assumed to be always available.** To be clear, you will be held responsible for any DKVS task crashes, so you should test it thoroughly.

To provide this semantic, each stage must track which tuples it has processed and which tuples have been processed by the next stage; the Source "stage" tracks only the latter. Each stage also acknowledges the (sender task from the) previous stage once it has processed a tuple. If a sender task doesn't receive an ack on time, it re-sends the tuple until it receives an ack. A receiver task must check if a (re-sent) input tuple is a duplicate to avoid processing it a second time; it must discard duplicates but ack them. How can a task detect duplicate tuples? One

method is for a sender task to associate a unique ID with each outgoing tuple and for the receiver task to use that ID for such detection.

To support exactly-once semantics across task failures and restarts, each task must persist the aforementioned tracking; in other words, each task maintains this tracking as log entries in DKVS. Each task has an identity, and the leader must ensure that a restarted task inherits the corresponding failed task's identity, even when the task is restarted on a VM that is different from the failed task's VM. The restarted task can use that identity to locate and read its log entries in DKVS. A restarted task can look at (and replay) its log[3] to figure out which input tuples it has processed (and ack'ed), and which output tuples have been ack'ed by the next stage. If a task fails, the leader must restart the task to bring the total number of tasks back to the level it was before the failure.

Note that we will fail only tasks that implement filter or transform operators and **not fail stateful ones that implement aggregation**. You can also use `op_args` to pass in a boolean flag that indicates if this is a stateful operator.

Mandatory #2: Autoscaling. RainStorm supports dynamic adjustment of #tasks in each stage based on processing load. This feature is toggled by the `autoscale_enabled` command line parameter. It is not easy to design exactly-once semantics to interact accurately with dynamic changes in the number of tasks in a stage.

1. Therefore, to keep things simple, **we will disable exactly-once semantics when the autoscaling feature is enabled**.
2. However, we **expect at-least-once delivery semantic for tuples** even during any autoscale-triggered changes to #tasks in a stage. In other words, we expect that tuples are never dropped.
3. In addition, we **do not require RainStorm to autoscale a stateful stage**; i.e., the RM need not monitor and autoscale a stage implementing aggregation.
4. Note: you can also assume that no failures will be triggered (task or VMs) during any runs with `autoscale_enabled` set to true.

Rainstorm should support dynamic adjustment of tasks as follows. The Source process feeds into the tasks of the first stage. The Source process must sustain this stream at INPUT_RATE tuples(lines)/second. If the input load to a stage drops below an average low-watermark of LW tuples/second per task, the system must decrease the number of tasks in that stage by one. And, if the input load to a stage goes above an average high-watermark of HW tuples/second per task, the system must increase the number of tasks in that stage by one. Any change made by the system to #tasks in a stage must happen within 3 seconds of its input load crossing a watermark. Each experiment/demo will start with `Mtasks_per_stage` per stage, and the system will scale up or down the number of tasks in a stage based on the watermarks and the input rate to that stage. The system must detect the input rates of each stage and scale the stage automatically, and not just react to the INPUT_RATE parameter, which controls only the first stage's input rate. All three values, INPUT_RATE, LW, and HW, are specified as parameters to the RainStorm command line.

One way to design autoscaling is to have a central ResourceManager (RM) module running on the leader VM. RM communicates with each task, computes the total load and average per-task

[3] Since DKVS tasks never fail, a RainStorm task can always access its latest log.

load in each stage, and reacts by informing the leader to dynamically increase or decrease the number of tasks in a given stage.

General

You must use the code for MPs 1-3 in the RainStorm system. Use MP1 to log messages for debugging, MP2 to detect failures, and MP3 to store the output streams and intermediate data for RainStorm.

RainStorm tasks need to store to DKVS the aforementioned log entries and the output stream tuples from the final stage. If writing to DKVS one tuple at a time is slow, explore improving your design–you can use the LLM to explore alternatives. You may consider batching as an optimization. For example, you can batch multiple tuples and interact once with DKVS to read or write each batch of tuples. With batching, the task may also need a timeout to ensure it writes out a partial batch of tuples during a pause or at the end of a streaming run. By making batch size a configurable argument in your code, you can test with different batch size settings, including setting it to 1 (i.e., no batching). Maintaining a per-task log in DKVS requires some design because the log may grow past 1MiB, which is the max value size supported by DKVS. Here's one potential design: a given task stores a batch of tuples as one entry with the key <stage#, task#, batch#>. The task has another entry with the key <stage#, task#, received> with the value (start_batch#, end_batch#), which points to a sequence of batched entries that have been received from tasks of the previous stage. Similarly, the task can use another entry to track batches that have been sent to tasks in the next stage. **This is just a suggestion, so feel free to come up with your own design.** Because DKVS tasks will never be failed in all of MP4, you are free to run DKVS in any consistency mode. However, please document your choice in your report. In MP4, you must run DKVS with replication factor N=3. If you choose to run DKVS in an Eventual consistency mode, then set R=2 and W=2.

Once you have your code working, your first test must confirm that the flow-rates measured by each task and reported to the RM accurately track the INPUT_RATE from the source. In other words, create 2 tasks per stage, have each stage simply pass through each tuple (i.e., make `op_exe` an identity function), and ensure that the measured and reported (by RM) tuple/sec rate across all tasks in each stage is equal to the INPUT_RATE. You should verify with testing that your MP4 can handle an INPUT_RATE of up to 250 tuples/sec, and use this test to also tweak the aforementioned batching to ensure that DKVS can sustain that write throughput.

Audit Trail

Create an audit trail file (of debug messages) on the local VM—one per worker task and one for the leader (including the RM) task—that can be queried (via MP1 or via grep). Each invocation of RainStorm and creation of a RainStorm task must use a new audit trail file, so it's easy to find and grep through during the demo. Do not discard or overwrite audit trail files in between tests. Each task appends debug messages to its local audit trail file. We will request to see the audit trail entries at demo time, either via local grep or using the MP1's querier. You can make your debug messages as verbose as you wish, but at the minimum, each worker task must record the following events (each recorded event must also include an **ISO timestamp** using the VM's local clock):

1. Starts/restarts of the task.

2. Tuples output by the task.
3. For exactly-once, any rejected duplicate tuples (we may grep for them during the demo)

The leader's audit trail must record the following, at the minimum (each recorded event must include an **ISO timestamp** using the VM's local clock):

1. Start and end of each run of RainStorm. Each of these events should also record the `RainStorm` command invocation that started this run, including the exact parameters.
2. The start and end of each task.
3. Detection of a failed task.
4. Restart of a task after a failure.
5. When recording task start/restart, include the task's VM, PID, op_exe, and the local audit trail filename.
6. Autoscale-triggered event: upscale or downscale, which stage, the measured tuples/sec and the watermark that was breached, and the resultant starting of a new task or the stopping of an existing task.
7. Every second, record the tuples/sec processing rate of each task.
8. Any rejected tuples (for exactly-once), together with the VM-id + task-id that rejected it.

For demo purposes, you must implement a few extra commands, and take care of a few other things. For each command, please print messages on the console that show when the command has been completed.

1) All commands listed in the MP3 specification must work.
2) Passwordless SSH access to all VMs must be set up and available before the demo, which allows the TA to run a command directly on any VM.
3) `list_tasks`: query the leader process and output info about all streaming tasks. For each task process, output its VM, PID, op_exe, and local audit trail filename.
4) `kill_task <VM-id> <PID>`: given a streaming task's VM and PID, abruptly kill the task process. Send a "kill -9" or similar signal; **it should not be a graceful shutdown**.
5) `dump_log <VM-id> <PID> <filename>`: given a streaming task's VM and PID, generate the task's log by concatenating the corresponding DKVS entries, and write it to `<filename>`.
6) `final_stage_output <DKVS_dest_file>`: Generate the final output file by concatenating the values from the DKVS entries output by tasks of the final stage.
7) `terminate_run`: Since streaming applications don't have a termination condition, RainStorm tasks will run forever. This command either explicitly kills or gracefully shuts down MP4 processes on all VMs after a run completes. You will need this command for your own experiments, and when we switch between tests of the demo.
8) Please ensure that the per-task and leader audit trail files contain the events described in the "Audit Trail" section above.

Spark: Download Spark Streaming from https://spark.apache.org/streaming/ and run it on your VMs (this step will take some effort, so start early!). You will need to install/deploy any necessary dependencies. For e.g., Spark requires all VMs to have access to a shared storage system; one option could be to deploy HDFS. Compare the performance of RainStorm with Spark Streaming, and **see if you can make RainStorm faster.** Make the comparison *fair*. In other words, run the job with the same settings for both systems, i.e., in the same cluster on the same dataset and for the same topology (#tasks per stage). Most of the inefficiencies in RainStorm will be in accessing storage, so think of how it reads from and writes to DKVS. To measure

performance, use output tuples produced per time. Are you able to beat Spark Streaming?

Datasets: Pick or synthetically create datasets that ensure your **run times are at least several tens of seconds**, especially when comparing with Spark. You may need to use or create datasets that are 100s of MBs large.

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 the number of parking meters or the 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 use the CS VM Cluster machines. You will use 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 scheduled on the Monday right after the MP is due. The demos will be on the CS VM Cluster machines. You will use up to the max VMs for your demo. 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 (typed, not handwritten) of not more than three pages (12 pt font). Briefly describe the following. We recommend your report contain headings with bold keywords:

a) **Design**: Briefly describe your design (including architecture and programming framework) of RainStorm.

b) **MP3/DKVS Use**: Briefly, in 5-6 sentences, describe how task logging—both the storing and replay of log entries—uses DKVS.

c) **LLM use**:If you used LLMs, describe in <300 words your usage and experience---especially how beneficial or not it was. Full LLM transcripts must be submitted with the code. Place them in a folder named *llm_use/* in your repo. Instead of manually copying prompts/responses, you can use log-extraction tools. We provide one here.

d) **Measurements** (measure real numbers, do not calculate by hand or calculator!): see below. Please label each plot/data clearly so your report is easy to read.

Show plots comparing RainStorm's performance to Spark Streaming for two different datasets. You can pick from the aforementioned datasets or create/find your own. In any case, your report must include a brief description of the two datasets you picked. If generated, include in

your git repo the script/code used to generate it. If found, include the source and web-link in your report. For each dataset, describe it and the TWO streaming applications you implemented using that dataset (each with at least 2 stages); the grep+replace described above is one such example. Feel free to come up with your own applications. So, there should be a total of FOUR experiments. Your report should include a description of each of these applications.

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 implementation early!). Discuss your plots, don't just put them on paper, i.e., discuss trends, and whether they are what you expect or not, and why. Measurement numbers don't lie, but we need to make sense of them! Stay within the page limit.

Submission: More instructions on Piazza, but the checklist/workflow is the same as earlier MPs. Do not miss the deadlines for signing up for the demos (usually the Friday before submission Sunday), or to submit the form and the Gradescope report (both due on Sunday).

(1) **Report.** 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 must tag your partner.
(2) **Demo.** There will also be a **demo** of each group's MP4. Sign-up sheets for the demo & demo instructions will be posted on Piazza. Signup for demos on the Friday before the deadline.
(3) **Code.**

A. Submit your working **code** into a subfolder named mp4 in your **GitHub** course repo. In your repo, please include a README explaining how to compile, how to run your unit/demo tests, and how to run your MP4 code along with LLM transcripts.
B. Fill out the MP4 **code** submission **Google form**.

Further submission instructions will be posted on Piazza.

When should I start? You already know all the necessary class material to do this MP. This MP requires significant planning, design, implementation, debugging, and 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 plagiarism detection to check for copying within and outside this class. A 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.

Contributions by Group Members: We recommend you stick with the same group from one MP to the next (this helps keep the VM mapping sane on IT's end), except for exceptional circumstances. We expect all group members to contribute equally to the overall effort. If you believe your group members are not, please have "the talk" with them first, and give them a second chance. If that doesn't work, please email both instructors.

Happy Streaming!