Fault Tolerant Runtime System For Distributed Graphical Simulations with a Centralized Controller in the Cloud

Distributed Systems Course Project
Omid Mashayekhi (omidm@stanford.edu)

1 Introduction

Graphical simulation is a cornerstone of modern digital media. Currently, these simulations are run using expensive super computers, or a small cluster of highly customized machines with high bandwidth and low latency communications. On the other hand, recent developments in the cloud computing industry makes cloud services, like Amazon EC2, an interesting and cost effective domain for graphics simulations.

Nimbus is a runtime system for distributed graphics simulations in the cloud. Simulation state is distributed among a number of workers that accept commands from a centralized controller and execute operations to change the state of the simulation. Nimbus is actively under development.

If a worker is down, its covered state is gone and the entire simulation progress is halted. This paper explains the design and implementation of a fault tolerance mechanisms to make Nimbus robust against worker failures. Specifically, checkpoints are created by the runtime system, periodically. At a checkpoint, each worker is responsible for saving a portion of the state that is necessary to rebuild the lost state in case of failure.

2 Nimbus In A Nutshell

This section introduces Nimbus architecture and application abstraction concisely. The intention is not to justify the design and goals of Nimbus; it requires its own paper and is out of the scope of this report. However, this section provides enough information and explains Nimbus key abstraction features, such that an unfamiliar person will be able to follow the rest of this paper about adding fault tolerance to the system, with no problem.

2.1 Nimbus Architecture

Nimbus has a centralized controller that drives a number of worker in the cloud. Controller distributes the simulation states among the workers and sends commands to the workers to mutate, duplicate, or exchange data among themselves. Controller makes runtime decision for communication and load distribution based on dynamic changes in cloud resources.

In a way, this architecture resembles MapReduce's[4]. Currently, there are distributed runtime systems for scientific computation in the super computing domain (e.g. Legion [3]). However, we believe that in the cloud environment with dynamic and unreliable resources a centralized decision maker could benefit a lot from global knowledge. Also, any distributed runtime solution requires duplicating and updating the state over all the nodes, which results in excessive latency in the cloud domain which has way worse communication resources compared to what is assumed in the super computing domain.

2.2 Application Abstraction

Nimbus application is serries of `job` units that operate over well defined `data` objects. Each job has a **read** set and a **write** set of the data objects it accesses. In addition, each job is associated with another set of jobs called **before** set, which is a set of jobs that has to finish before job can start execution

safely. This information produces a Directed Acyclic Graph(DAG) called **job graph** with jobs as nodes and each before-set relation as directed edge. Figure 1 shows one example job graph. Nimbus enforces that all writer jobs over a specific data form a well defined order (**lineage**) in the form of the before set relationship in the job graph.

The meta data per job (read/write/before set) is enough for the runtime system to exploit possible parallelism and handle required data exchanges or duplications. In oder words, **data flow** is deciphered from the job graph and read/write set of jobs. Specifically, the version of a data that a job, say A, reads is the version that a writer job of the data, say B, produces where B is the latests job in the data lineage that also appears directly or indirectly in the before set path of job A; note that before set relationship is transitive. For example, in figure 1, job A sees version of the data that job B produces, because it is the latest writer job in A's before set (Not C).

Each nimbus data is declared over a well defined **geometric region**. This is very beneficial as it helps runtime system for data placement strategies and also lets the application writer to specify job read/write set only by declaring a data type and a geometric region to read and write.

2.2.1 Parent Job vs. Sterile Job

Application stars by executing a special job called `main` and from there on jobs can spawn other jobs in addition to operating over the data. Jobs that spawn other jobs are called **parent** jobs and implicitly appear in the before set of their children. To simplify the abstraction Nimbus does not impose any scope restriction, and the fact that jobs could spawn other jobs with any access pattern in the future makes data management strategies at runtime way less efficient compared to static compilation of the entire computation in advance. However, only a few jobs in the system spawn other jobs. For example, for each iteration of implicit solver, only one job spawns the rest of the jobs for the iteration depending on the convergence requirements. So, as an optimization, application writers mark jobs that will not spawn other jobs as **sterile**. This helps the centralized runtime controller to have faster and more efficient algorithms for enforcing correctness. As we will see, this optimization also helps in checkpoint creation and recoveries in the event of failures.

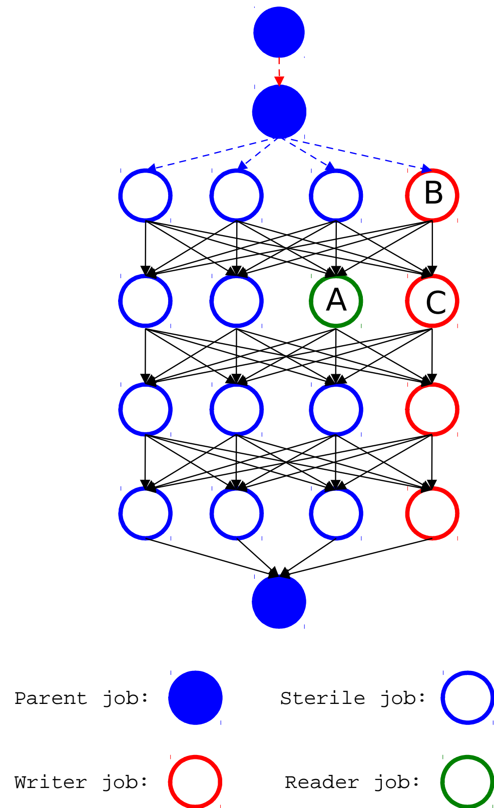

Figure 1: Nimbus sample job graph.

3 Design And Model

This section explains the failure model and assumptions in designing a fault tolerant mechanism for Nimbus. Also, I will describe the expected behaviour in case of failure. In addition, we will see why the well known checkpoint creation and rewinding mechanism is a well suit for Nimbus and will compare it with other possible options.

3.1 Failure Model And Expected Behaviour

We assume that controller never fails, but any number of workers could fail at any time. Since the failure rate of controller is way less than the worker's,

this is a reasonable assumption. Note that currently in case of controller failure human interaction is required. Features could be added to the system to launch a back up controller automatically, however it is out of the scope of this paper.

Worker failure could happen due to many reasons, including, but not limited to, bugs in the code, network disconnection, disk failures that causes application to fail (currently this causes worker to stop), and random errors that we face in the application due to missed memory corruptions that cause nondeterministic segmentation faults.

Let's assume that in case of no failure, the application would ultimately produce version V of a specific data, say D. Then, the goal for the fault tolerance algorithm is to create version V of D despite any worker failure and without restarting the application from scratch.

3.2 Design Space: Checkpoints vs. Lineage

In order to recover from a failure, one well known solution in literature is creating a snapshot of the system state periodically and rewind back from a safe checkpoint and reproduce the entire sate from there. This solution is usually used in super computing solution.

Another solution is keeping the lineage meta data for each data separately and in the event of worker failure only reproduced the lost data using their lineage. Systems like Spark [5] use this solution.

Here, I use the checkpointing mechanism. There are a couple of reasons why this solution fits the Nimbus application space better. First the job graph and related data lineage in graphics simulations is way more complicated compared to the big data applications that Spark[5] supports. If there are N unique data objects and S steps from the last reliable version of the data, then the meta data required to keep the lineage for all the data objects separately would have $O(NS)$ complexity. Note that the applications that we are aiming have around 1.04 million unique data objects. Also there are stages in the simulation for the implicit solver that the lineage grows to 100 hops within few seconds. Note that Nimbus's centralized controller could easily become a bottleneck and saving this meta data is a source of considerable latency.

Moreover, jobs usually have bigger read set compared the write set in graphical simulations. It is due to the well-known ghost values concept in scientific computation (the value of a cell in next step depends on the neighbor values as well). This means that to recompute a lost data, we need to recompute a prior version of a neighbor data as well (and for that, neighbors of neighbor, etc.). In addition, note that jobs in graphical simulation have read/write set of multiple variables. This means that even for lineages with moderate length, almost the entire lineage for all data objects in the system has to be revisited. In other words, this practically results in reproducing the entire state after the reliable version for each data.

All in all, checkpointing mechanism sounds more promising for Nimbus, and I have implemented this solution. In the next section, I will explain the states that are required for reliable rewinding from a checkpoint for Nimbus applications.

3.3 Checkpointing The System State

Note that in case of Nimbus, state is not just the data distributed among the workers. The parent jobs that produce the job graph and spawn the next generation of jobs also need to be saved to properly reproduce the jobs that operate over the saved data. Note that as discussed in the previous section, it is not practical to keep the jobs in the lineage for each data separately, and so re-executing the parent jobs would reproduce the job graph and lineage for the data.

Next question is what are required and sufficient data versions that has to be saved to re-execute jobs spawned form the parent jobs after rewinding? The answer is all the data versions that a parent job in the checkpoint sees. Note that this is required, since the parent job could spawn jobs with any meta data. For sufficiency, not that any job that parent job spawns will need a data version that is either saved or can be re-created from other children of the parent job. This is true, since a set of parent jobs in the job graph snap shot create a cut in the job graph.

4 Implementation

To implement the checkpointing mechanism for Nimbus I have written around 2500 lines of extra c++ code for the system, during the past quarter. In the following sections, I will discuss algorithms required to create safe checkpoints in the system and also rewinding safely from a checkpoints in case of a failure. In addition, to save the data I implemented a distributed data base on top of leveldb [1], that I will explain further in section 4.3.

4.1 Checkpoint Creation

For each checkpoint, controller assigns a unique checkpoint id and saves data objects needed for the checkpoint over the workers that hold the required data version in volatile memory. Checkpoint ids are monotonically increasing and controller announces the latest successful created checkpoint id periodically. Workers could safely remove all the data associated with old checkpoints. Note that this is important as the meta data for each checkpoint is big enough that workers could run out of disk after a few checkpoints.

Depending on the checkpoint creation rate algorithm, centralized controller attempts to create checkpoints periodically. When checkpoint creation is triggered, followings are the steps that controller takes for safe checkpoint creation:

1. Give a unique identifier to this checkpoint; these identifiers are monotonically increasing. Controller stops assigning further jobs for execution.

2. If there are any running parent jobs in the system, wait until they finish completely. We need this so that any other parent job that could be produced by running jobs could be detected properly.

3. Detect all spawned parent jobs in the system. They from a cut in the job graph. Save them with their meta data as a list of jobs for re-execution upon rewinding.

4. Find the required data versions that needs to be saved for the checkpoint as explained in Section 3.3. Save these data versions at the worker that produces them, by assigning a specific save job to the target worker.

5. If all the data required for checkpoint is saved successfully, then we have a stable checkpoint of the system. Announce the new safe checkpoint id to the workers with a `CheckpointCreated` command.

4.2 Recovery From Failure

After controller detects worker failure, it attempt to re-execute the simulation from the latest safe checkpoint. Note that, controller needs to avoid race conditions at the worker, since live workers could be still running jobs assigned previously. Followings are the steps that controller takes for rewinding:

1. Ignore the current job graph of the system. Also send a `PrepareRewind` command to the workers to cancel all pending jobs and terminate all running jobs. This command has a unique rewind id.

2. Wait until all the healthy workers respond with a `RewindAck` command to acknowledge the rewind id.

3. Start assigning the parent jobs from the latest safe checkpoint. Notice that the meta data of those jobs are saved at the controller.

Note that as discussed in Section 3.3 all the required data versions for all the jobs spawned by these parent jobs could be either reproduced or loaded form the saved state in non-volatile memory.

4.3 Distributed Data Base

In order to save the data versions needed for each checkpoint on non-volatile memory, I implemented a simple distributed data base module on top of leveldb [1]. Data required for each checkpoint is split among the workers depending on which worker has the data

in its local volatile memory. Each worker saves the data on a local leveldb module and announces a handle that could be used to retrieve the data from any other node in the system. This handle has enough information on how to find the disk (e.g. ip address of the machine, path, etc.) that has the target leveldb instance holding the data. In our failure model, even if the worker (process) is down, the disk (machine) is still reachable.

This is just a temporary solution and we are planning to use better data bases with higher availability in our system. Amazon S3 would be a better solution, however for current stage of the system that we uses local machines for development, the distributed data base on top of leveldb is better suited.

5 Evaluation

In this section, the implemented fault tolerance mechanism is evaluated. Specifically, we will look in to two main factors: correctness and performance. First, we make sure that correctness is not violated and system can survive properly from diverse worker failure models. Second, we consider the overhead of adding fault tolerance feature to the system. I have run two experiments to show this.

First, I used simple stencil application distributed among 4 workers and took down workers randomly. This application applies an stencil over the data for 150 iterations and saves the final results. Experiment has been run a couple times and the output of the system was always correct, regardless of the worker failures. Controller could detect the failed worker properly and redistribute the work load over other workers after rewinding. Figure 2 shows sample traces of simulation progress in case of worker failures. Here, the checkpointing happens at a fixed rate for every 30 iterations and as you can see the periodic jumps show the checkpoint creation overhead. As the workers fail system rewinds back to a previous checkpoint, and so some iterations are run multiple times.

Second, I ran water simulation from PhysBAM [2] library that has been ported in to Nimbus. This is a very complicated simulation that incorporates both level set and particle methods and produces complex data dependencies.

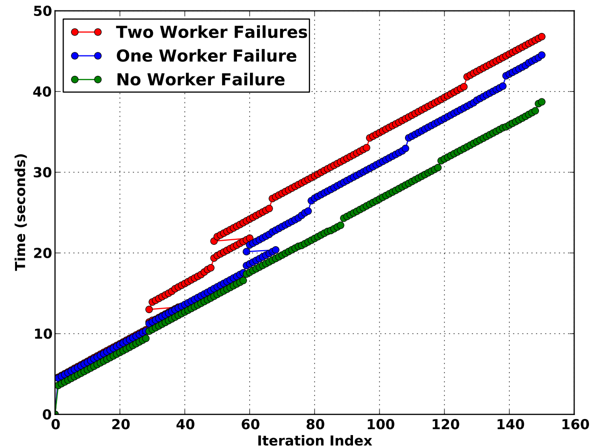

Figure 2: Stencil application traces. It shows the progress based on iteration number and how the system rewinds back to checkpoints upon worker failures.

Simulation had 64 partitions and was distributed over 8 workers each with 8 cores (total of 64 nodes). I ran the simulation on EC2 and used c3.2xlarge instances for the workers. Initially the load was evenly distributed over all nodes. Then, workers were brought down randomly one by one. The simulation frames were computed properly despite random multiple worker failures, for a couple of trials. Also the checkpoint creation overhead was measured to be 8.2 seconds in average. Considering that each iteration of computation takes 32 seconds and there are 15 iterations in each frame, if the checkpointing is triggered every frame the overall overhead would be 8%. Not that the checkpoint rate could be set by the application writer to fit the requirements of the application.

6 Conclusion and Future Work

In this paper, I explained the design and implementation of a fault tolerance mechanism to make Nimbus robust against worker failures. The key point, is regular checkpointing of the simulation state, and rewinding back to the latest safe checkpoint upon failure. Experiments show that the algorithm keeps correctness and has an acceptable overhead of 8% for

a reasonably large and complex simulation.

Currently application writer decides on a fixed checkpoint creation rate to fit the application requirements. However, one interesting future work would be to explore dynamic checkpoint creation rate adaptation to minimize the execution time of the simulation. Note that each checkpoint creation adds over head to the system and there should be as few of them as possible. On the other hand, if the checkpoints are sparse, the lost state upon creation is huge and it takes longer to get to the most recent lost state from an old checkpoint. This trade off has an optimum point based on workers expected failure rate.

References

[1] Leveldb. `https://github.com/google/leveldb`.

[2] PhysBAM. `http://physbam.stanford.edu/`.

[3] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11. IEEE, 2012.

[4] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[5] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.