Autonomous, Failure-resilient Orchestration of Distributed Discrete Event Simulations

Matthew Malensek[1], Zhiquan Sui[1], Neil Harvey[2], and Shrideep Pallickara[1]
[1]Department of Computer Science, Colorado State University, Fort Collins, CO, USA
[2]School of Computer Science, University of Guelph, Guelph, ON, Canada
{malensek, simonsui, shrideep}@cs.colostate.edu, neilharvey@gmail.com

ABSTRACT

Discrete event simulations model the behavior of complex, real-world systems. Simulating a wide range of relevant events and conditions naturally provides a more accurate model, but also increases the computational workload associated with the simulation. To manage these processing requirements in a scalable manner, a discrete event simulation can be distributed across a number of computing resources. However, individual tasks in the simulation are *stateful*, and therefore require inter-task communication and synchronization to produce an accurate model. This property not only complicates the orchestration of the discrete event simulation in a distributed setting, but also makes providing reliable, fault-tolerant execution a challenge, especially when compared to conventional distributed fault tolerance schemes.

In this paper, we propose an autonomous agent that provides fault tolerance functionality for discrete event simulations by predicting state changes in the simulation and adjusting its fault tolerance policy accordingly. This allows the system to avoid negatively impacting overall execution times while preserving reliability guarantees. To underscore the viability of our solution, we provide benchmarks of a production discrete event simulation that can sustain failures while running under the supervision of our fault tolerance framework.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Fault tolerance; I.6.8 [**Simulation and Modeling**]: Discrete Event

General Terms

Algorithms, Design, Performance, Reliability

Keywords

Fault tolerance, distributed discrete event simulation, checkpointing, neural networks, prediction

CAC'13, August 5–9, 2013, Miami, FL, USA.

1. INTRODUCTION

The longer a software process runs, the more likely it is to experience a hardware or software failure. This fact is only exacerbated by the trend towards distributed, multi-core architectures that take advantage of the vast processing power found in today's commodity hardware. In a cloud environment, each additional unit of processing involved in a task increases the overall probability of a failure occurring. Therefore, in the modern computing landscape, failures are an issue that must be expected and dealt with rather than simply avoided or ignored [25].

Discrete event simulations are one type of long-running computation that can benefit greatly from a distributed approach. Briefly, a discrete event simulation can be described as a model of a system that produces output based on how a series of events unfold. These simulations are generally computationally expensive, so dividing the workload among a number of computing resources is beneficial in multiple ways: outputs can be generated faster, more parameters can be explored, and additional iterations of the simulation can be run to verify output quality. However, these performance gains come at the expense of additional complexity through increased communication and synchronization required between distributed components. As more components are added to a system and executed across a diverse set of computing resources, the likelihood of a failure also becomes much higher.

In this work, we outline the design of an autonomous fault tolerance agent to oversee the execution of distributed discrete event simulations. Unlike conventional distributed fault tolerance approaches, our system must cope with constantly changing, stateful computations and ensure global consistency throughout the system in the event of a failure. This requires a number of system-level optimizations along with reliable prediction mechanisms that enable a dynamic, proactive approach to providing fault tolerant execution for our subject simulation.

To distribute the simulation across a cluster, we have expressed its units of computation as a set of iterative, streaming MapReduce phases. In each phase, Map tasks perform their individual stochastic operations and maintain local state information that can be collected at runtime in the form of *checkpoints*. We rely on an adaptive strategy that determines when and how checkpoints should be requested in order to reduce the amount of duplicate work and overhead incurred from state collection. This is made possible by forecasting state changes and having the system plan accordingly.

1.1 Research Questions

Creating a fault-tolerant, distributed implementation of a discrete event simulation that requires stateful, interdependent tasks led us to pose a number of research questions that we have addressed in this paper:

1. Can we incorporate support for failure resilience while minimizing overheads? Since failures are infrequent, our goal should be to achieve fault tolerance without introducing unacceptable overheads into the system.

2. Can these overheads be minimized while also retaining the ability to cope with multiple, concurrent failures, which can either be permanent or transient?

3. Can failures be detected efficiently? Simply executing duplicate tasks in parallel is not an adequate scheme for dealing with failures in our system, so active detection of failures is required.

4. Will simply collecting state information from individual tasks be efficient means to provide fault tolerance? How can state collection be optimized?

5. In the event of a failure, can we minimize the amount of work that will need to be duplicated?

1.2 Contributions

Our contributions in this work stem from: (1) the use of a learning-based, adaptive checkpointing strategy, (2) orchestration using a distributed stream processing engine, (3) the ability to handle multiple concurrent failures, both persistent and transient, (4) the verification of correctness of the recovery scheme, and (5) the amount of overhead introduced by our scheme.

1.3 Paper Organization

The remainder of the paper is organized as follows: in the next section, an overview of our parallel discrete event simulation and its components is provided. In Section 3, our fault tolerance strategy is detailed, including the components that are responsible for detecting and handling failures. Section 4 provides information about optimizations we developed for handling and transmitting state in the system, followed by Section 5, which explains how forecasting future simulation conditions allows our system to make efficient fault tolerance decisions. Section 6 includes performance benchmarks of recovery operations and overhead in the system. Section 7 surveys related work dealing with fault tolerance in discrete event simulations, and Section 8 outlines our conclusions and future work in this area.

2. SYSTEM ARCHITECTURE

Our system is designed to separate concerns between two components: the cloud runtime, which handles orchestration of the simulation, and the simulation itself. This separation allows different discrete event simulations to be run within the system, provided that they conform to our wire format specification. For this study, we utilized the Granules Cloud Runtime [24] for coordinating distributed activities within the system, and the North American Animal Disease Spread Model (NAADSM) [17] as our subject discrete event simulation.

2.1 Granules

Granules [24] is a distributed stream-processing framework that allows computations to be expressed using the MapReduce paradigm or as directed, cyclic graphs. The framework handles deploying, scheduling, and orchestrating computations on clusters of machines or in the cloud. Computations can be scheduled to run when data is available or at regular intervals, with a configurable number of execution iterations. Since computations can execute multiple times, it is possible to build state over the course of execution. Granules has been employed in several areas of study, including bioinformatics, brain-computer interfaces [12], clustering [11], and scientific data storage [22].

While Granules is implemented in Java and natively supports Java-based computations, the framework also provides bridging functionality that allows computations to be written in C, C++, Python, and R. NAADSM is written in C and uses this functionality to communicate directly with the Granules runtime. Granules is an open source effort, available at *http://granules.cs.colostate.edu*.

2.2 NAADSM

NAADSM [17] is an epidemiological model of disease outbreaks in livestock populations developed jointly by the US Department of Agriculture, the Canadian Food Inspection Agency, Colorado State University, the University of Guelph, and the Ontario Ministry of Agriculture, Food, and Rural Affairs. It has been applied to studies of several diseases including foot-and-mouth disease [26], avian influenza [15], and pseudorabies [28].

NAADSM is a Monte Carlo model: a simulation is run many times with each run representing one possible way that events could occur in a disease outbreak, which contributes to an overall picture of the *probability distributions* of the output variables. This means that simulations are generally run many times, and due to their processor-intensive nature can benefit greatly from a parallel, divide-and-conquer approach spread across multiple computing resources.

Discrete event simulations are used in a wide variety of problem domains, but generally have three main components in common: a set of state variables, a list of events, and a global clock [14]. The basic adaptation of NAADSM to run across multiple processes in a distributed environment managed by Granules required: (1) a field to indicate which farms in the simulation are managed by individual NAADSM instances, (2) addition of a bridge object that translates events between Granules and NAADSM, and (3) a simulation day barrier for synchronization in the NAADSM main event loop. Apart from these necessary modifications, we strived to make as few changes to the NAADSM codebase as possible.

In this study, we used a set of NAADSM directives, called a *scenario*, that simulated an outbreak of foot-and-mouth disease (FMD) set in Kansas, USA. The scenario consisted of 660,000 farms, which is the approximate number of FMD-susceptible farms in the 12 Midwest US states. Nine farms were infected initially and the disease was spread by airborne, direct, and indirect contact. The scenario's scale and extreme parameters were selected specifically to increase its computational footprint; on a single machine in our test cluster, the scenario takes about 18 hours to execute.

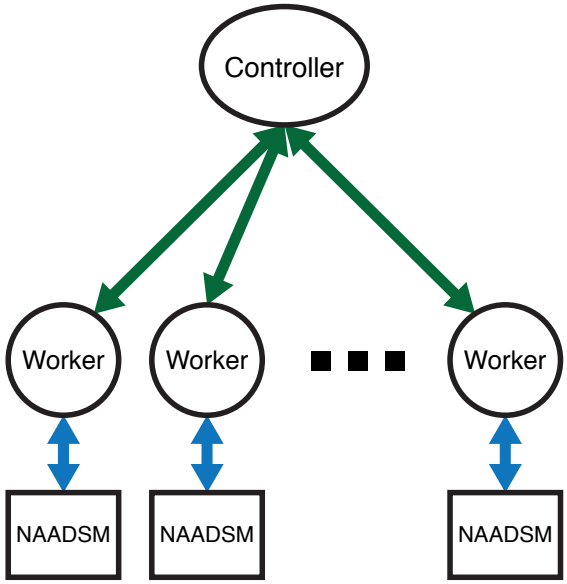

Figure 1: The network layout for our distributed discrete event simulation framework.

2.3 Test Environment

The benchmarks described in this study were carried out on a cluster of 16 HP DL160 servers, each equipped with a 2.4 Ghz quad-core Intel Xeon processor, 12 GB of RAM, and a 1000 Mb/s network interface. The Granules framework ran within OpenJDK 1.7. Each machine in the cluster managed four Worker instances for a total of 64 Workers executing our Midwest scenario in parallel, bringing the overall execution time of the simulation down to under an hour. Without debugging instrumentation used to produce the results in this work, the execution time is approximately 30 minutes.

2.4 Parallelization Framework

A distributed execution of NAADSM across a cluster of computing resources involves two primary components: a *Controller*, which coordinates the global state of the simulation, and multiple *Worker* instances, which are deployed to each node in the cluster and are responsible for managing individual instances of NAADSM. Figure 1 illustrates this configuration. These two components are the backbone of our discrete event simulation parallelization framework.

Internally, NAADSM is a set of code modules that produce and consume discrete events (e.g., movement of animals between two farms, or detection of a disease at a farm). When running within Granules, a simulation is divided by geography with each Worker managing a subset of farms in the scenario. Events that may have effects outside of a Worker's territory are packaged and transmitted to the Controller in a once-per-simulation-day step that acts as a synchronization barrier across the Workers. After receiving global state information from each Worker instance, the Controller broadcasts a single bundle of aggregate state out to the Workers to mark the start of a new simulation day. Compared to other cloud runtimes such as Hadoop [1] that deal with largely stateless processes, the stateful nature of this configuration is a challenge for both load balancing and fault tolerance operations.

To balance load across the available resources in a cluster, our system employs a dynamic *split* and *merge* strategy to divide or consolidate Workers depending on changing load characteristics. Due to the simulation day execution barrier, it is critical to place load as uniformly as possible across all the Workers in the system; the simulation can proceed only after *all* Workers have completed their tasks, so a run of a simulation day is only as fast as the slowest worker in the resource pool.

Our parallelization framework is best suited for simulations that have a spatial component that can be divided into individual subregions managed by a Worker instance. This includes atmospheric science [7], modeling object interactions in space [19], and cosmology [31]. If a particular feature of the framework is unnecessary for a given simulation, (such as the synchronization barrier) it can be disabled without affecting fault tolerance functionality.

These architectural decisions make the distributed orchestration layer highly adaptable to new types of simulations, but also make each component instance a single point of failure; the loss of either the Controller or even a single Worker prevents further progress of the simulation. Additionally, as the number of computing resources involved gets larger, the probability of a failure occurring becomes higher as well. To deal with these failure scenarios we have introduced another component, called the *Speculator*, which handles speculative execution, failure detection, and recovery operations from within the cloud runtime.

Unlike the speculative execution performed in implementations of MapReduce [8], our system requires transactional semantics to ensure the consistency of global state across the entire cluster during failures. Additionally, speculative tasks are launched in MapReduce based on the identification of *stragglers*, which are tasks that take an atypical amount of execution time. The stochastic nature of our problem leads to the development of execution *hot spots* that move across different processing elements during the simulation, precluding the use of execution time as a failure metric.

3. THE SPECULATOR

In our system, the Speculator is responsible for all activities related to fault tolerance. This includes detecting failures, launching new Workers, rolling the simulation back to a consistent state, and managing resources through the Granules API. The Speculator can also be used to suspend a simulation, save its state to disk, and then resume execution — even on completely different hardware.

Along with these fault tolerance features, the Speculator also plays a role as an autonomous manager of system events, deciding the frequency of system state collection and allocation of resources that can be used by the Controller. These decisions allow the Speculator to provide its services without imposing a significant performance penalty on the execution of the simulation.

One key aspect of our system architecture is that communication only occurs once per simulation time unit, which is a simulation day in the case of NAADSM. This means that querying a Worker and receiving a result is a two-day process. Because of this constraint, the Speculator must be highly proactive; simply reacting to events as they take place is not sufficient due to the ever-changing state of the simulation.

As a simulation progresses, global and individual state is

built during each simulated day. In the case of a failure at any node, the simulation cannot progress any further because all nodes must report their global state before moving on to the next day in the simulation. The challenges that arise in this situation are twofold: detection of failures, and failure recovery.

3.1 Failure Detection

Detecting failures requires a balanced approach; if the Speculator does not identify a failed machine in a timely manner, overall simulation throughput is reduced. Conversely, false positives needlessly delay simulation progress. These factors make failure detection a challenging task for the Speculator. Since Workers communicate with the Speculator and Controller on a regular basis, the absence of communication may indicate a failed Worker. However, it is also possible that the Worker is simply processing a particularly complex part of the simulation. Additionally, a sound failure detection system is not centralized; network segmentation may cause a single central node that is monitoring the system to mistakenly determine that an unreachable segment of the network has failed. In this scenario, the nodes may be reachable from another part of the network, so detection should be done from multiple locations in the network topology.

To mitigate these issues, the components in our system transmit small messages, called *Heartbeats*, to the Speculator. Heartbeats are generated periodically by a background process and contain system statistics, including load information, CPU utilization, available memory, and disk activity. This information provides the Speculator with data to monitor utilization on every node and make fault tolerance decisions based on overall cluster usage.

When a node has not sent a heartbeat message after a configurable failure timeout threshold, the Speculator assumes it has failed. This assumption is confirmed by instructing the Controller to also attempt to contact the node; if communications with the Controller have been severed, then the simulation will not be able to progress beyond the current simulation day and the Worker must be relaunched on another resource. In our tests, a heartbeat interval of 5 seconds resulted in a single failure being detected in about 7.82 seconds on average; Table 1 provides failure detection statistics for up to eight machine failures.

Table 1: Failure detection with a 5-second heartbeat interval

Failures	**Detection Time (s)**	
	Mean	SD
1	7.82	2.03
2	7.85	2.63
4	8.03	2.66
8	8.23	2.78

To detect the special case of a Controller failure, the Speculator ensures that: (1) the time of the last heartbeat message from the Controller is greater than the failure timeout threshold and (2) each Worker has submitted a state bundle, signaling the end of the current simulation day. If the Controller has not instructed the Workers to begin the execution of the next day, then it has failed. The Speculator starts a new Controller instance on a different node and transmits the current system state information to it, resuming execution.

Finally, it is possible that the Speculator itself experiences a failure. Due to the publish-subscribe design of the Granules framework, multiple instances of the Speculator can be started transparently and operate in parallel; if a failure is detected, the available Speculator processes elect a leader to handle it. This property also means that Speculator instances can transparently enter and leave the system at any time during execution; as long as one Speculator is available, then failures can be dealt with.

3.2 Failure Recovery

Workers in our system maintain both *local state* and *global state*. Global state information is exchanged during each new simulation day, but local state is only used at the individual Worker level. With no fault tolerance considerations, this leads to an unrecoverable simulation in the event of a Worker failure; there is no way of re-generating the local state necessary for a different node to take over for the failed Worker.

To cope with the possibility of a Worker failure, local state information must be also be shared; upon request, this information can be bundled by individual Workers and sent to the Speculator as a *checkpoint*. A checkpoint contains enough local state information for the Speculator to relaunch a Worker process. Unfortunately, simply relaunching a Worker is not enough to resume a simulation; all Workers must be executing the *same* simulation day, so the entire simulation must be rolled back to a consistent state. A rollback will place the simulation at the most recent point in time that the Speculator had a checkpoint for each worker in the system. This collection of all distributed state information in the system at a given point in time is called a *snapshot*. Rollback operations are applied in terms of snapshots available to the Speculator.

Further complicating failure scenarios, there is no guarantee that the number of active Workers will stay constant during the execution of the simulation; in fact, it is highly likely that there will be a different number of active Workers at any point in time due to the system's dynamic splitting and merging functionality for load balancing. This means that the Speculator must also start up or suspend the appropriate number of Worker instances during a rollback operation.

Since the Speculator monitors the nodes' system status information and the simulation state as well, it can reason about what information it will need to provide fault tolerance without impacting the overall performance of the system. For example, given a particular livestock disease, the simulation may progress rapidly and then slow down as the disease spreads and becomes more computationally expensive. Figure 2 illustrates this situation, showing execution times in a 64-Worker simulation run of our Midwest scenario. In the early stages of a simulation, the Speculator does not need to request checkpoints frequently because a restart of the entire simulation would have a minimal impact on overall execution time. This property makes setting a hard "checkpoint interval" inefficient, and lead us to develop a fault tolerance component designed around mak-

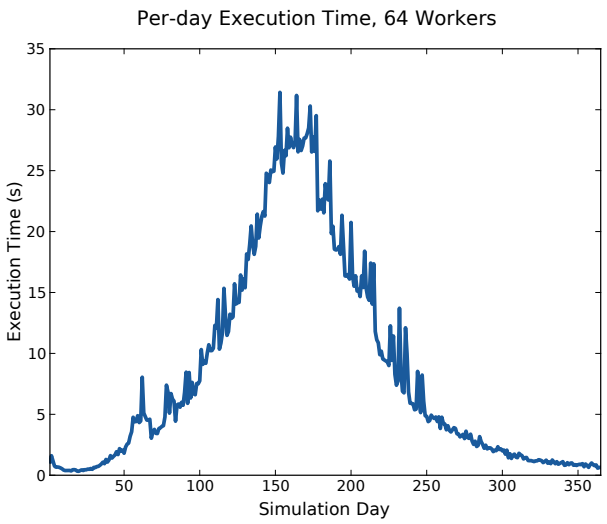

Figure 2: Per-day execution time of a 64-Worker Simulation of foot-and-mouth disease in the Midwest USA.

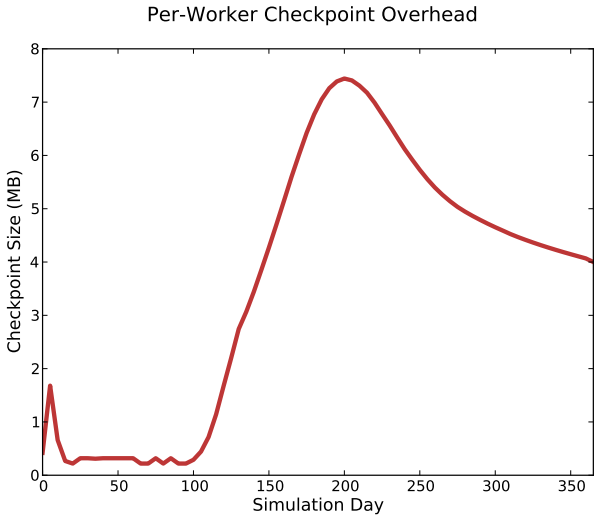

Figure 3: Checkpoint size (in MB) as a simulation progresses.

ing intelligent, autonomous decisions about when and how checkpoints should be collected.

Giving the Speculator the ability to request checkpoints at specific times during the simulation allows for novel fault tolerance approaches that balance recovery speed and system load. The Speculator can also accept user-defined parameters that provide additional constraints on how fault tolerance operations are carried out; a user may need results within a given time frame, or want to guarantee that lost execution time due to failures does not exceed a specific threshold.

4. CHECKPOINT OPTIMIZATIONS

Checkpoints are used for two primary purposes in our system: load balancing and fault tolerance. As one might expect, the amount of state information in a simulation tends to grow over the course of execution, with the most growth seen during highly-complex events. We have observed that the most CPU-intensive portions of a simulation are also accompanied by a growth in checkpoint sizes; Figure 3 illustrates this trend in our Midwest scenario.

Given a 64-Worker run of the simulation, individual checkpoints of about 8 MB will result in 512 MB of data being sent to the Speculator at a time, easily saturating its network interface. This IO-bound operation makes building a complete snapshot of the simulation costly, and causes delays while the Workers transmit their state information. Decreasing the size of these checkpoints allows the system to collect state information more frequently, which reduces the amount of simulation progress lost in the event of a failure. To better utilize resources at the Worker level, we moved state transfer operations to a separate thread to help interleave computation and IO operations. We also implemented two approaches to reduce the overall size of messages being sent in the system: *transparent compression*, and *dynamic patching*. Both of these optimizations do not require modification of the discrete event simulation.

4.1 Transparent Compression

Checkpoints consist of several data structures serialized to a binary format. In most cases, a simple translation of in-memory data to serialized form does not produce the smallest checkpoints possible. Therefore, we added a new feature to the Granules cloud runtime, called *Transparent Compression*, that compresses and decompresses information as it is sent and received. This framework allows client applications to take advantage of a number of compression algorithms without needing to write any code; events are automatically compressed before they are sent across the network, and the decompressed when they reach their destination. For event payloads that would not benefit from compression, the functionality can be toggled at runtime.

We utilized the Java JDK implementation of the DEFLATE algorithm [10] to compress our checkpoints, which provides a balance of speed and output file sizes. Since our simulation is highly CPU-bound, we favored a fast algorithm to help reduce the amount of time spent on activities orthogonal to simulation progress. Table 2 provides a summary of the compression ratios achieved and their computational cost using the default compression settings. While the 1 MB checkpoints were not highly compressible, the 4 and 7 MB checkpoints achieved a compression ratio of 0.35 and took about 65 ms per megabyte to create.

Table 2: Checkpoint compression evaluation, averaged over 1000 runs.

Checkpoint Size (MB)	Compression Ratio		Compression Time (ms)	
	Mean	SD	Mean	SD
1	0.90	0.04	8.38	2.19
4	0.35	0.002	262.88	32.87
7	0.35	0.002	458.95	7.21

4.2 Delta Checkpoints

At a given point in time, the current state in a discrete event simulation simply represents how a sequence of events has unfolded. The progression of these events causes state changes in an iterative fashion, meaning each checkpoint in our system has some common attributes with previous checkpoints. To exploit this property, we created *delta checkpoints*: checkpoints that contain the binary differences from the last checkpoint. This enables our system to "patch" an old checkpoint to produce an up-to-date representation of state in the simulation.

A number of algorithms have been developed for generating binary patch files, such as VCDIFF [20] and bsdiff [27], which are frequently used in the distribution of software: rather than transmitting an entirely new version of a software binary, a small patch can be used to update an older version of the software. For this work, we created a native Java implementation of bsdiff due to the relatively small patch files it produces. bsdiff has been employed in a number of software products, and, notably, was used for deploying quick, frequent updates to users of Google's Chrome browser.

One key step in the process of creating a bsdiff patch is applying a suffix sorting algorithm on the original source file; the reference implementation of bsdiff uses Larsson and Sadakane's qsufsort [21]. Our implementation, however, provides a simple interface for changing the sorting algorithm at runtime. Different suffix sorting implementations tend to perform better on different types of data, and since execution time is very important in our system we benchmarked a number of suffix sorting algorithms from the jSuffixArrays library [4] on our checkpoint files. The results of this benchmark can be seen in Table 3, which lead us to choose the deep-shallow sorting algorithm [23] for our particular delta checkpoints.

Table 3: Suffix soft comparison, executed 100 times on various 7 MB checkpoint files.

Sorting Algorithm	Sort Time (ms)	
	Mean	SD
qsufsort	2175.24	37.03
divsufsort	1223.67	19.57
SAIS	1989.67	9.84
deep-shallow	738.12	8.00

To further speed up the delta checkpoint creation process, we replaced the bzip2 compression used in bsdiff with the same Java implementation of DEFLATE used to compress our standard checkpoints. While bzip2 offers better compression ratios, it also tends to consume more processing time. Table 4 summarizes the delta checkpoint sizes and their creation times.

Ultimately, there are tradeoffs associated with each of our checkpointing methods. Small checkpoints generally do not need further compression or patching. Larger checkpoints that are created during fast-executing parts of the simulation benefit most from compression; the compressed files save network bandwidth and can be generated quickly. During the longest-running portions of the simulation, delta checkpoints enable the system to collect state information frequently to mitigate the large amount of lost processing time a failure would cause. Figure 4 illustrates the size differences between uncompressed, compressed, and delta checkpoints. These options give the Speculator flexibility in its state collection operations to choose an approach that is most appropriate for the current simulation environment.

Table 4: Delta checkpoint generation, averaged over 1000 runs.

Checkpoint Size (MB)	Delta Size (KB)		Creation Time (ms)	
	Mean	SD	Mean	SD
1	38.66	3.53	517.26	79.07
4	116.01	4.16	1037.43	36.57
7	126.40	9.72	1579.68	56.48

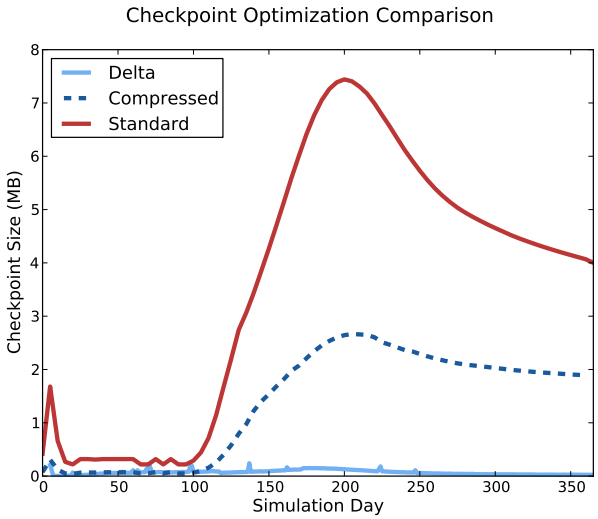

Figure 4: Per-Worker checkpoint sizes for each checkpointing strategy.

5. FAULT TOLERANCE POLICY

The Speculator has a number of options for providing fault tolerance at its disposal, each with their own tradeoff space. ***How*** and ***when*** the Speculator requests checkpoints ultimately determines the performance impact of our fault tolerance functionality and its scalability as more Workers are added. While thresholds can be set to guide the Speculator's choices, hard rules tend to be brittle and ineffective when faced with different simulation and disease types. To provide a flexible model for determining the appropriate checkpointing policy, we predict the per-day execution time of the simulation, as well as the cost of generating a checkpoint. These future costs can then be evaluated against the likelihood of failures to produce a fault tolerance strategy.

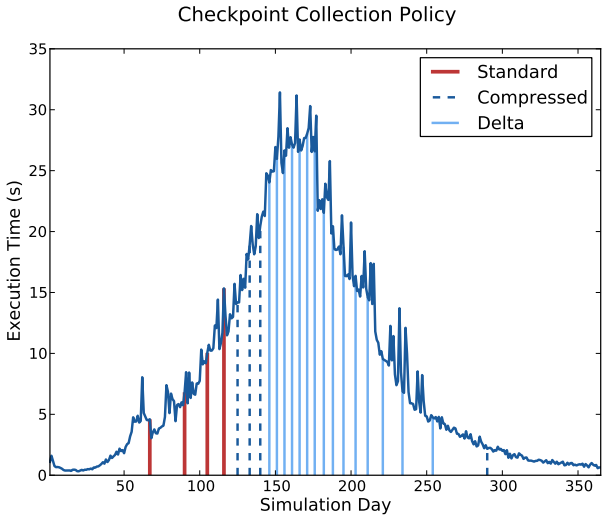

Figure 5: Prediction-based checkpoint policy changes over time. Days that checkpoints were requested on are indicated by a vertical line.

5.1 Predictions: Artificial Neural Networks

Since execution times in our simulation tend to exhibit non-linear patterns, we utilized an Artificial Neural Network (ANN) from the Encog Machine Learning Framework [18] for predicting execution time and checkpointing costs. Collecting a complete Snapshot of the system state requires at least two simulation days; one to request the checkpoints, and one to receive them. Due to this constraint, we trained the neural network to predict three simulation days into the future. This gives the Speculator time to choose an appropriate checkpointing strategy for upcoming events and inform Workers of when the checkpoints should be generated. Figure 5 illustrates how and when the Speculator requested checkpoints in an iteration of our Midwest scenario. The policy ensured that no more than two minutes of execution time could be lost at any point due to a failure. Parts of the simulation that execute quickly required fewer checkpoints, and if a failure occurs before day 67 the entire simulation will simply be restarted.

The neural network can be trained with information from two sources: system data in heartbeats collected by the Speculator, and simulation parameters published by the discrete event simulation. To provide the latter data points, we created an additional message type in our communication wire format that contains an arbitrary array of information about the current state of the simulation. In our Midwest scenario, parameters include variables such as the number of herds managed by a Worker, the number of herds exposed to the disease, vaccinations, and herds that have died. This data does not have to be inspected or understood by the Speculator, but can simply be fed into the neural network for training and predictions.

While additional state information from the simulation is not required to predict future execution times, the availability of such information helps model how particular scenarios will behave. Additionally, the trained neural network can be serialized to disk and used in other installations of the system to provide predictions without requiring a training phase. The implementation of this functionality is entirely optional; if publishing additional state variables is not possible, then our system can still provide its prediction-based fault tolerance functionality.

6. PERFORMANCE EVALUATION

To evaluate the recovery performance and overhead incurred by our fault tolerance functionality, we designed and executed a number of benchmarks. Due to the stochastic nature of the simulation, we also implemented functionality to support repeatable random runs so that the outcome of the scenario could be held constant while other system variables were changed. Each benchmark was also verified for correctness with the single-threaded, non-distributed version of NAADSM.

6.1 Repeatable Random Runs

As a Monte Carlo simulation, NAADSM makes heavy use of a global random number generator object. This complicates performance evaluations because each iteration of the simulation will produce different output. Further complicating matters, simply choosing the same random number seed for each iteration will still produce different output in a distributed setting. We solved this problem by attaching an independent random number stream to each farm, initialized by combining a global seed and the farm's unique ID. All stochastic decisions pertaining to a farm are made using the farm's own random number stream. Because farms are not divided across Workers, this scheme will guarantee the same sequence of events will occur, regardless of how the simulation is split across computational resources. This modification is not a requirement for running a discrete event simulation in Granules, but allows us to hold simulation variables constant while we benchmark fault tolerance features.

6.2 Correctness

A distributed implementation of a discrete event simulation, no matter how fast, is only useful if it produces correct output. To ensure our optimizations did not compromise the validity of simulation runs, we verified that the output of each iteration matched that of an unmodified NAADSM, farm-for-farm. Our regression test suite was composed of several scenarios, run across a wide range of simulation sizes, random number seeds, and load balancing patterns.

6.3 Recovery

After a failure has been detected, the Speculator must ensure that an adequate number of Workers is available and then roll the simulation state back to the last snapshot. This involves:

1. Stopping active Workers.
2. Starting and stopping resources to ensure an adequate number of Workers is available.
3. Transmitting state to the Workers and Controller.
4. Waiting for Workers to decompress and apply new state information.
5. Resuming execution of the simulation.

Table 5 outlines the amount of time it takes to recover from a failure at different stages of execution in the simulation. Recovery tends to require more time as checkpoint sizes increase, although a considerable portion of the recovery time can be attributed to latencies involved with global coordination in the recovery process. The results were gathered from 100 simulation runs that experienced a failure on each day noted.

Table 5: Failure recovery times at different stages of a scenario's execution.

Simulation Day	Recovery Time (s)	
	Mean	SD
50	5.502	1.36
100	7.320	1.43
150	7.684	2.02
200	10.403	2.78
250	9.621	2.79
300	7.981	2.13
350	7.783	2.08

6.4 Adaptive Checkpointing

To illustrate the advantage of using our adaptive checkpointing strategy, we compared its fault tolerance overhead with three simulation runs that had a static checkpoint interval of 3 days and a fixed checkpoint type. Table 6 shows the stark difference in overhead between a static and adaptive strategy.

Table 6: Comparison of adaptive and fixed checkpointing strategies.

Strategy	Overhead
Fixed, Standard	7.32%
Fixed, Compressed	5.8%
Fixed, Delta	5.98%
Adaptive	***1.1%***

This table also demonstrates that simply using a delta strategy for the smallest possible checkpoint size does not result in the lowest overhead of the fixed strategies. The slight increase in overhead compared to compression is due to the additional processing time required to compute delta checkpoints.

6.5 Fault Tolerance Overhead

In order to evaluate the worst-case performance overhead incurred from our fault tolerance system, we ran two identical iterations of our Midwest scenario: one with no fault tolerance functionality enabled, and one with checkpoint requests occurring every simulation day. Figure 6 shows the per-day cost of collecting checkpoints in our system; the overhead accounted for a five-minute increase in execution

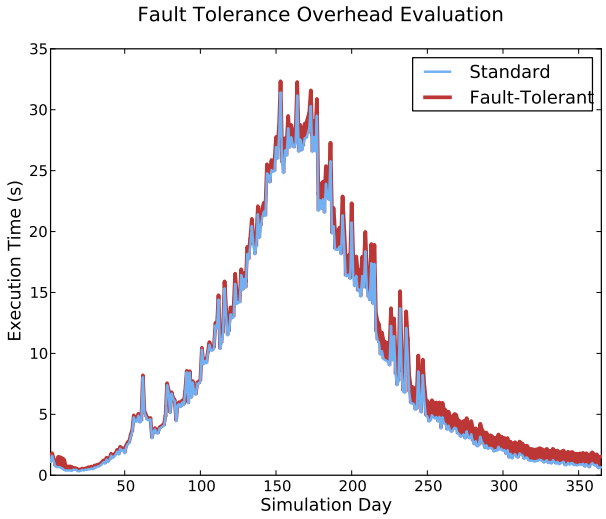

Figure 6: Shaded areas above the standard execution times represent overhead due to checkpointing operations.

time, but the majority was during non-critical execution at the end of the simulation. These costs can be easily avoided by the Speculator through forecasting, as our benchmarks have shown.

7. RELATED WORK

While our system primarily targets cloud or cluster deployments, checkpointing also plays an important role in creating fault-tolerant grid environments. Recognizing the importance of checkpointing schemes that dynamically adapt to their environment, Chtepen et al. propose a number of heuristics for determining a checkpoint interval [6]. Unlike our system, their design does not require global state synchronization, but it does account for estimated execution time and checkpoint overhead to determine whether a job should be allowed to checkpoint its state or not.

Failure detection research for distributed discrete event simulations has produced several innovative solutions [9,29]. Gupta, Chandra, and Goldszmidt [16] postulate that the key components of an effective failure detection mechanism are completeness, speed, and accuracy. In the case of our framework, completeness and speed are significantly relaxed in favor of ensuring accuracy to avoid needless rollback operations. Only weak completeness is required for our purposes, meaning all the components in the system do not need to know when a failure has occurred, and the heartbeat interval sets an approximate upper bound for detection times. These factors and examples from the literature guided our decision to keep the failure detection component of our framework as simple as possible.

DACE introduces a failure model that tolerates crash failures and partitioning, while not relying on consistent views being shared by the members through a self-stabilizing exchange of views [13]. This however may prove to be very expensive if the number and rate at which the members change their membership is high. In our case, the number of workers for a given simulation day can vary because of

the split-and-merge operations being performed to balance computational loads.

Rollback operations and speculative execution in discrete event simulations have also been researched by Ortiz and Jiménez [30]. In their work, a *coordinator* handles snapshotting the simulation, but requires execution to completely halt while the process is carried out; our solution interleaves snapshot operations with other processing, and only requires the simulation to stop if a failure has occurred.

The Recovery-Oriented Computing project, a joint research effort of Stanford University and UC Berkeley, takes the point of view that hardware and software failures and user errors are inevitable. Therefore, making systems recover quickly and automatically is at least as important an endeavor as fixing problems [25]. The architecture and software produced by the project hinges on 2 building blocks: the micro-reboot, a selective restart of only the failed component(s), and system-wide undo capability.

The practicality of micro-rebooting depends on three factors: an ability to locate the component at fault, a loosely-coupled modular architecture, and an ability to store state externally to the application. Our system shares some concepts with this approach. It can locate failed components, via heartbeats, although this is a simpler approach than the Pinpoint software developed by the ROC project, which watches all client requests in the system, and uses data clustering to identify combinations of components likely associated with the failure to fulfill a particular request [5]. The system described in this paper has the kind of loosely-coupled modular architecture needed to allow just one component to restart, and it does store state externally to the simulator (in checkpoints held by the Speculator). However, the ROC project explicitly distinguishes micro-reboots from failover to a new node, which they describe as a more time-consuming and less desirable operation [3].

System-wide undo capability as envisioned by the ROC pro-ject depends on what they call the "three R's": rewind, repair, and replay [2]. Rewind is seen in many software systems (for example, rollbacks in databases) but building an architecture for repair and replay can be more challenging. Any operations done after the fault occurred are "replayed" in the context of the repaired system, which may differ from the original context in which the operations were taken (for example, end-users may have already seen and interacted with faulty results) [2]. The system described in this paper performs a rewind step, but opts for simply running the simulation from that point rather than attempting to "replay" logged operations. We sidestep the problem of potentially delivering faulty results to the end-user due to the built-in barrier of a completed simulation day: a day's simulation results will not be delivered to an end-user until all nodes have reported results.

8. CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

We have devised an adaptive checkpointing strategy and fault tolerance framework for discrete event simulations which dynamically selects both the timing and mechanism to perform checkpoints. The mechanism to perform checkpoints could either be full-fledged or deltas computed based on differences between successive full-fledged checkpoints. The decision is based on the time it takes to compute the checkpoint (and the overhead it adds to the overall simulation), the transmission overhead, and the amount of work that needs to be duplicated should the simulation be rolled back.

To our knowledge, this is the first attempt to incorporate fault tolerance into a discrete event simulation orchestrated using a stream processing engine. All interactions and control messages are orchestrated as streams. Distributed workers that orchestrate this simulation run inside computations that are activated when data is available on their input streams.

Our framework can handle an arbitrary number of failures. The system can sustain permanent failures of all workers in the system. The system allows incorporation of redundancy for the checkpoint orchestrator (Speculator). With multiple Speculators, when the primary fails, one of the surviving speculators is promoted to be the primary. Where there is just one instance of the Speculator, we can sustain transient failures (such as restart of a machine) because Speculators manage their state by writing to, and reconstructing from, persistent local storage.

We have verified the correctness of our recovery strategy for the stochastic discrete event simulation. We have done this in the presence of multiple, concurrent failures and an adaptive checkpointing strategy.

The overheads introduced by our strategy are acceptable in situations where failures do not take place. In situations where a failure occurs, we reduce the recovery costs by minimizing the amount of work that needs to be duplicated.

8.2 Future Work

While the simple neural network we used to predict changes in system state performed well for our purposes, a Recurrent Neural Network (RNN) may provide better performance for making predictions due to their internal memory. In the future, we will evaluate different prediction techniques, including utilizing Elman networks.

Our delta checkpoint scheme provides different options to improve performance by allowing both the compression and suffix sorting algorithms to be changed at runtime. Selection of the appropriate algorithms could be performed automatically during the training process, or could be changed dynamically by the Speculator as conditions in the system change over time.

We also plan to adapt our parallelization and fault tolerance frameworks to oversee the execution of a different discrete event simulation to help identify common traits and execution patterns that affect fault tolerance functionality. This could incorporate new system health metrics and learning techniques to make our framework even more generalizable.

9. ACKNOWLEDGMENTS

This research has been supported by funding (HSHQDC-10-C-130) from the US Department of Homeland Security's Long Range program.

10. REFERENCES

[1] A. Bialecki, M. Cafarella, D. Cutting, and O. O'Malley. Hadoop: a framework for running applications on large clusters built of commodity hardware. *http://hadoop.apache.org/*.

[2] A. B. Brown and D. A. Patterson. Rewind, repair, replay: three R's to dependability. In *Proceedings of*

the 10th workshop on ACM SIGOPS European workshop, pages 70–77. ACM, 2002.

[3] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot–a technique for cheap recovery. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, volume 6, 2004.

[4] Carrot Search Labs. jSuffixArrays: Suffix arrays for java. *Web page at http://labs.carrotsearch.com/jsuffixarrays.html.*

[5] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 595–604. IEEE, 2002.

[6] M. Chtepen, F. H. Claeys, B. Dhoedt, F. De Turck, P. Demeester, and P. A. Vanrolleghem. Adaptive task checkpointing and replication: Toward efficient fault-tolerant grids. *Parallel and Distributed Systems, IEEE Transactions on*, 20(2):180–190, 2009.

[7] W. R. Cotton, R. Pielke Sr, R. Walko, G. Liston, C. Tremback, H. Jiang, R. McAnelly, J. Harrington, M. Nicholls, G. Carrio, et al. Rams 2001: Current status and future directions. *Meteorology and Atmospheric Physics*, 82(1-4):5–29, 2003.

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

[9] R. Debouk, S. Lafortune, and D. Teneketzis. Coordinated decentralized protocols for failure diagnosis of discrete event systems. *Discrete Event Dynamic Systems*, 10(1-2):33–86, 2000.

[10] L. Deutsch. DEFLATE compressed data format specification version 1.3. 1996.

[11] K. Ericson and S. Pallickara. On the performance of high dimensional data clustering and classification algorithms. *Future Generation Computer Systems*, 2012.

[12] K. Ericson, S. Pallickara, and C. Anderson. Analyzing electroencephalograms using cloud computing techniques. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 185–192. IEEE, 2010.

[13] P. T. Eugster, R. Boichat, R. Guerraoui, and J. Sventek. Effective multicast programming in large scale distributed systems. *Concurrency and Computation: Practice and Experience*, 13(6):421–447, 2001.

[14] R. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, 1990.

[15] C. Green et al. Simulation modeling of alternative control strategies for an HPAI outbreak using NAADSM. In *Canadian Association of Veterinary Epidemiology Preventive Medicine (CAVEPM) Meeting, May 29 - 30 2010, Guelph, Ontario, Canada*, 2010.

[16] I. Gupta, T. D. Chandra, and G. S. Goldszmidt. On scalable and efficient distributed failure detectors. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, PODC '01, pages 170–179, New York, NY, USA, 2001. ACM.

[17] N. Harvey, A. Reeves, M. Schoenbaum, F. Zagmutt-Vergara, C. Dubé, A. Hill, B. Corso, W. McNab, C. Cartwright, and M. Salman. The north american animal disease spread model: A simulation model to assist decision making in evaluating animal disease incursions. *Preventive Veterinary Medicine*, 82(3):176–197, 2007.

[18] Heaton Research, Inc. Encog machine learning framework. *http://www.heatonresearch.com/encog.*

[19] D. Jefferson and J. Leek. Application of parallel discrete event simulation to the space surveillance network. In *Proceedings of the Advanced Maui Optical and Space Surveillance Technologies Conference, S. Ryan, ed.(Maui Economic Development Board, 2010) E*, volume 34, 2010.

[20] D. Korn and K. Vo. The VCDIFF generic differencing and compression data format. 2002.

[21] N. J. Larsson and K. Sadakane. *Faster suffix sorting.* Univ., 1999.

[22] M. Malensek, S. Pallickara, and S. Pallickara. Exploiting geospatial and chronological characteristics in data streams to enable efficient storage and retrievals. *Future Generation Computer Systems*, 2012.

[23] G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1):33–50, 2004.

[24] S. Pallickara, J. Ekanayake, and G. Fox. Granules: A lightweight, streaming runtime for cloud computing with support, for map-reduce. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–10. IEEE, 2009.

[25] D. Patterson, A. Brown, P. Broadwell, et al. Recovery-oriented computing (ROC): Motivation, definition, techniques, and case studies. Technical report, Technical Report UCB//CSD-02-1175, UC Berkeley Computer Science, 2002.

[26] D. Pendell, J. Leatherman, T. Schroeder, and G. Alward. The economic impacts of a foot-and-mouth disease outbreak: a regional analysis. *Journal of Agricultural and Applied Economics*, 39(0):19–33, 2007.

[27] C. Percival. *Matching with mismatches and assorted applications.* PhD thesis, University of Oxford, 2006.

[28] K. Portacci, A. Reeves, B. Corso, and M. Salman. Evaluation of vaccination strategies for an outbreak of pseudorabies virus in US commercial swine using the NAADSM. In *ISVEE 12: Proceedings of the 12th Symposium of the International Society for Veterinary Epidemiology and Economics, Durban, South Africa*, page 78, 2009.

[29] W. Qiu and R. Kumar. Decentralized failure diagnosis of discrete event systems. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 36(2):384–395, 2006.

[30] J. L. Ramírez Ortiz and R. Marcelín Jiménez. Fault-tolerant distributed discrete event simulator based on a P2P architecture. In *SIMUL 2011, The Third International Conference on Advances in System Simulation*, pages 21–26, 2011.

[31] V. Springel. The cosmological simulation code gadget-2. *Monthly Notices of the Royal Astronomical Society*, 364(4):1105–1134, 2005.