Mandrake:Implementing Durability for Edge Clouds

Kyle Carson, John Thomason, Rich Wolski, and Chandra Krintz
Computer Science Dept.
Univ. of California, Santa Barbara

Markus Mock
Computer Science Dept.
Univ. of Applied Sciences, Landshut Germany

***Abstract*—In this paper we present Mandrake, a software infrastructure for edge clouds (private clouds located at the network edge), designed to provide reliable, "lights out" unattended operation and application hosting in IoT deployments. Mandrake both implements reliable private cloud operation in restricted resource environments and data durability features that hosted applications can leverage. We describe leveraging Mandrake for hosting Hadoop applications at the edge. Our empirical evaluation shows that Mandrake is able to ensure Hadoop's data durability guarantees efficiently in the presence of relatively frequent failures even when resources are scarce.**

***Index Terms*—edge cloud, cloud computing, fault tolerance, high availability, resource allocation, data durability**

I. Introduction

Cloud computing systems have been continuously evolving to meet ever-changing workload demands. The volume of data produced by IoT systems [1] has recently created a need for processing this data "at the edge", i.e. where it is produced rather than in traditional data centers. The co-location of processing infrastructure and IoT devices is crucial both to reduce the latency between data acquisition and device actuation, and to alleviate network congestion to cloud sites.

However, given limited hardware resources and the lack of a controlled data center environment for the equipment to operate within, edge computing infrastructure faces additional challenges. In particular, failures ranging from hardware breakdowns to power and networking outages can occur at any time, and unlike traditional cloud environments, these failures might take significant portions of the cloud's entire resource pool offline. Once in this impaired state, it can take an extended amount of time before any manual human intervention restores the cloud to its original healthy state.

Mandrake addresses this robustness challenge by defining an architecture for deploying a "lights out" edge cloud – one that automatically reconfigures itself in response to failures until a human operator can respond with a repair. The goal of Mandrake is

- to provide a general purpose hosting environment for applications running at the edge,
- to automatically reconfigure the hosting environment, and any applications hosted within it, in response to resource failures such that application functionality (but not necessarily capacity) is preserved, and
- once a repair to the infrastructure is completed, to restore full capacity.

The edge cloud hardware consists of a small cluster of robust computing and storage elements configured as a private cloud [2]. Mandrake implements resource management and automated reconfiguration. It detects when a physical host within the cloud has experienced a failure, or conversely recovered from a failure, and in response automatically updates the cloud topology. Mandrake also operates at the virtualization layer, where user applications run, and manages the distributed applications running within each virtual machine (VM) allocated by the cloud's virtualization hypervisor. It exports an API so that applications that are hosted on a Mandrake edge cloud can make informed decisions regarding service failover, co-tenancy of replicated data in VMs, etc. That is, Mandrake "breaks" the cloud abstractions and exposes tenancy information (through an API) when a resource failure occurs.

To demonstrate and evaluate the capabilities of Mandrake, we have assembled an edge cloud and used it to test our system. Using a set of benchmarking workloads for big data processing, we outline the different phases of handling physical machine failure in the cloud. We show that on the physical hosts, Mandrake detects and handles failures quickly. We also show that the reallocation of resources to replace lost computational power proceeds quickly as well. To study the effect of failures on a big data processing workload, we use the popular Hadoop [3] parallel processing platform and exhibit how Mandrake enhances its availability characteristics and data placement strategies.

Additionally, we restrict our study to Hadoop as it is currently available (as an image) in Amazon EC2 [4]. Our goal is to understand edge cloud auto-reconfiguration in a context where the edge cloud and a public cloud are part of a tiered deployment architecture for IoT. The edge cloud we use is interface compatible with Amazon EC2 making it possible to automatically download and use EC2 images without modification, at the edge.

In this way, Mandrake facilitates both moving the code (the EC2 image) to the data or the data (stored in the edge cloud) to the code based on whichever is more efficient for a given deployment. Thus, we present a system in which the Hadoop installation is unmodified with respect to the publicly available version (although we do customize its configuration for Mandrake's execution environment).

II. Mandrake

Mandrake consists of two logical sets of processes: a cloud management system that is implemented on the physical hosts logically as part of the private cloud software infrastructure,

and an application management system that is implemented inside the virtual machines (VMs) which are hosted on the edge cloud. User workloads run in VMs i.e., alongside Mandrake's application management processes in VMs.

The cloud management system (termed the Mandrake Coordinator, or MC) is responsible for allocating and deallocating the appropriate VMs in response to changes in physical resource availability. To do this, it maintains a global state map, with a list of hosts, a mapping of VMs to hosts, and the distributed application(s) running on each VM. When the MC detects a host failure, it marks the failed host in this state map and dynamically reconfigures it out of the "view" presented to hosted applications. The updated view of the cloud's topology is propagated to the application management system, which will adapt to the reduced size of the cloud. Conversely, when a host has recovered from a failure, the application management system is informed of the change and can integrate the newly provisioned resources relatively quickly.

The MC's actions are guided by a user-defined policy configuration that includes information such as the number of VMs to run on a host, the VM configuration, etc. In the experimental results presented in this paper, we use a configuration in which Mandrake hosts a single Hadoop cluster. However, the extensible design allows for the multi-tenancy of different hosted applications, while enforcing the modularity by using the MC as the same underlying cloud management system which facilitates resource allocation.

The application management system (termed the Mandrake Application Orchestrator, or MAO) provides services that can manipulate an application's configuration and operation dynamically to meet reliability or performance objectives. It coordinates its activities with the MC via the system view that the MC produces and serves to it via an API. For example, in this paper, the MAO implements a mechanism for placing data blocks for the Hadoop File System (HDFS) to ensure replica failure independence when VM-to-host mappings change dynamically – a feature not available to Hadoop at present. Thus, Mandrake enables host-aware data replication for Hadoop when Hadoop is using VMs in an edge cloud. This mechanism is not limited to Hadoop, however. Rather, it is a general mechanism for sharing (in a controlled way) the VM-to-host mapping information so that any system implementing replica independence could make informed placement decisions.

A. Mandrake Coordinator

The Mandrake Coordinator (MC) is the primary host-level service responsible for controlling the other host services and making decisions about how to manage the system's resources. A process belonging to the MC runs on every host and maintains the global mapping of the edge cloud's state. To do this in a consistent way, a distributed consensus mechanism is necessary. Mandrake maintains global state information pertaining to the cluster using Zookeeper [5] – a well-known open source system for implementing distributed consensus. In particular, it uses Zookeeper to implement group membership for identifying the set of hosts currently available, to replicate a consistent view of the resources to all hosts, and to denote a leader responsible for initiating reconfiguration actions.

Mandrake uses a heartbeat mechanism to detect host failure. When a "follower" machine fails, the MC "leader" notices it missing from the host list due to the lack of a heartbeat, and proceeds to perform the dynamic reconfiguration of the cloud using the other software services that make up the cloud management system. Conversely, when a machine comes back online, the MC instance running on that machine starts up automatically and rejoins the Zookeeper group of active hosts. To enable this, we leverage a new feature in a beta version of Zookeeper (r3.5.4-beta) for dynamic group reconfiguration [6] by the MC.

Mandrake is designed to use a private cloud for resource provisioning (i.e. VM allocation and deallocation, network subnet provisioning, local DNS naming, etc.). The current implementation uses Eucalyptus [2], [7] which is an open source private cloud that implements the Amazon AWS APIs. Eucalyptus is, itself, designed to tolerate failures, so Mandrake treats it as a fault-tolerant, distributed provisioning service.

In addition to the AWS user APIs, Eucalyptus exports an administrator API that the MC uses to determine virtual to physical mappings. To avoid tying Mandrake to a specific VM provisioning service, the MC code uses the Eucalyptus API through a thin, generic wrapper layer. Thereby the MC can be extended to use any private cloud or distributed provisioning service that supports similar functionality as Eucalyptus (e.g. OpenStack [8], Open Nebula [9], etc.).

The MC ensures consistency across separate calls to the provisioning services it uses. For example, if it detects a failure while it is in the process of performing a reconfiguration requiring multiple calls to the provisioning service, it will bring the system to a stable configuration before handling the newly detected failures.

Mandrake also implements two separate services that it uses internally alongside the MC. The State Service exposes an endpoint for the MC's consistent view of the system's state and a Watchdog Service tracks the activity of the main MC process and attempts to restart it if it becomes inactive. The State Service runs as a single instance on the MC leader node, fielding requests from both the MC and MAO. The system mapping is sent from the MC to the State Service via a single RESTful POST request. Similarly, the MAO periodically sends a GET request to obtain the current version of the mapping, and if the mapping is newer than its copy, it sends another request to obtain the current mapping. Since edge clouds are by definition resource constrained, we expect the size of the clouds to be on the order of tens of machines and the size of the state description to be on the order of bytes to kilobytes.

B. Mandrake Application Orchestrator

The Mandrake Application Orchestrator (MAO) runs within the VMs spawned at the request of the MC and acts as an automated application management system, configuring and running applications in place of a human operator. While the MC provisions and tracks the underlying resources, the MAO

ensures that the resources are correctly configured for use by a specific application and that there is minimal degradation when reconfiguring an application after a failure.

The MAO comprises a number of internal services. The MAO also uses a Watchdog Service to ensure its liveness. Together with the application it manages, it instantiates its own instance of Zookeeper and a Key Exchange Service (for SSH key exchange during bootstrapping) in each VM. All other VM services and the MAO communicate via SSH. There is no single master (thus no single point of failure) in the MAO – instances use the distributed consensus service, Zookeeper, to communicate, elect leaders for individual tasks, and store critical information. Finally, MAO instances poll the State Service hosted on the leader MC node for the system mapping via HTTP. Each instance stores this information locally and, when the polled version of the system mapping differs from the local version, the MAO requests the new mapping and initiates a reconfiguration.

Similar to the MC, the MAO uses Zookeeper's watcher, ephemeral znode, and dynamic reconfiguration [6] mechanisms. The Zookeeper clusters running in the MC and MAO are completely separate pools and do not communicate.

The application we use for Mandrake evaluation is Hadoop (version 3.0.3), a mature, widely used, and highly configurable data processing framework that implements the MapReduce [10] parallel computation paradigm, which has proven useful in solving many kinds of problems [11]. Hadoop is interesting for our study since it includes its own internal data replication mechanism that assumes it can assign replicas to separate resources to ensure failure independence. In a virtualized, resource constrained, and multi-tenant setting (e.g. in an edge cloud) VMs can be mapped to the same physical hosts with no way for Hadoop's mechanisms to properly determine this co-location.

Therefore, we attempt to determine in our experiments whether Mandrake can force Hadoop to use a replica placement pattern that ensures failure independence without modifying Hadoop internally so that it works "out of the box" in an edge cloud setting. Not only does this save us the potentially difficult work of modifying Hadoop's internals, it allows us to work with publicly available and vetted images pre-configured with Hadoop (such as those on the AWS Marketplace [12]), accruing the additional benefit that applications that run on a Mandrake-enabled edge cloud could also run unmodified in a traditional cloud environment.

C. *Mandrake's Replica Mover*

Hadoop makes its decisions about data placement to improve availability and reliability statically, i.e., using the host and network topology (e.g. rack placement) in place when its processes start up. While this is not a problem in a "bare metal" setting, it can result in sub-optimal decisions when executing or reacting to Hadoop node failures in a virtualized environment *if the probability of co-locating VMs is high* as it might be on an edge cloud.

To deal with such scenarios, we extend the MAO with a "Replica Mover" that ensures replica independence for Hadoop externally. Thus, using only configuration options that disable the internal replica placement mechanisms of Hadoop, the MAO can induce Hadoop to preserve replica independence without modifying its internal mechanisms. Thus, "standard" Hadoop images available in public clouds like AWS are compatible with Mandrake. [1] We overview how this is done in Mandrake in this subsection.

Data in a Hadoop cluster is stored using the Hadoop Distributed File System (HDFS), a distributed file system that uses data replication as the mechanism for achieving availability and reliability. Files are partitioned into blocks, which are replicated across the cluster using a specified replication factor (the default is three) and stored on multiple nodes called "DataNodes". When fewer copies of data blocks are available than required by the replication factor, HDFS creates a new copy on a different node.

The current implementation of HDFS does not provide an interface for manually moving data under user control.[2] As such, to transfer a replica from node A to node B, the Replica Mover first copies its metadata and data files from A to B. It then deletes the metadata file from node A and restarts it. The restart is required to force Hadoop to detect that the replica is no longer on node A.

The Replica Mover then restarts node B, causing HDFS to recognize the new replica. Finally, the data file from node A is deleted. This method of moving replicas requires that we disable Hadoop's own replica repair features, which we achieve by increasing its repair interval, giving the Replica Mover time to finish before Hadoop's mechanisms kick in.

The Replica Mover is started periodically (every minute) by MAO, and additionally using Zookeeper's watch callback mechanism [5] whenever the MAO learns about a system change from the MC. For simplicity and to ensure consistency, the Replica Mover always runs from a single elected leader.

To determine a replica movement plan, the Replica Mover uses Hadoop's `fsck` command to determine the location and status of replicas. Based on that information, it computes where replicas should be placed to achieve failure independence. Next it determines how replicas should be moved from co-located VMs and also where replicas for under-replicated blocks should be created. Finally, it attempts to place replicas so that each physical host ends up with a roughly equal number of replicas, i.e., it balances the replicas across physical rather than virtual hosts (Hadoop's balancing mechanism only considers virtual hosts). For every replica identified in a

[1]Note that while currently, we have implemented the Replica Mover for Hadoop only, the mechanism is general and can be re-targeted to other distributed file systems or data stores potentially used in edge clouds.

[2]The current release does support a "balancer" [13] that attempts to distribute data evenly among DataNodes, but it does not have a way to incorporate cloud tenancy information in its placement decisions. It may be possible to leverage BlockPlacementPolicyRackFaultTolerant [3] (treating physical hosts as racks) or a more advanced custom policy (as described in [14] and [15]) to prevent co-tenancy but hosts with multiple VMs would hold significantly more data than others, hurting failure resiliency.

movement plan the Replica Mover records the source and destination of the replica, and flags whether the source should be deleted after the move.

This plan is executed as follows: the Replica Mover decides on an order of source DataNodes (rather than moves). After all moves from a source are completed, the node is restarted so that the HDFS learns about any deleted replicas. This ensures that the node does not classify the blocks as over replicated. All DataNodes that now have no more moves planned to them (from any source) are restarted to make them report that the replicas are available at their respective locations. The Replica Mover then repeats this for the next source in the order.

One exception to this strategy must be made in the case where restarting the DataNode would cause the last reported copy of a data block to disappear from HDFS's accounting. In this case, the Replica Mover restarts a different DataNode with another copy first. This ensures that the block replication does not reach 0. This case may result in over-replication of other blocks but we attempt to avoid his using the plan's source order.

If the Replica Mover is activated while it is in the process of performing an earlier execution, e.g., when the MAO learns about an additional change to the cloud before a plan is fully executed, the Replica Mover cancels the current plan and restarts the current source DataNode as well as all DataNodes with unreported replicas. It then creates a new plan based on the latest system state mapping and proceeds as described. As long as mapping changes happen less frequently than the time it takes the Replica Mover to carry out its work, the system will eventually stabilize. In section III we evaluate the performance of the Replica Mover and its timeline of operation.

Since Hadoop and Mandrake are operating independently of each other and are not synchronized, race conditions between Hadoop and Mandrake can occur. Two scenarios are possible:

1) If Hadoop writes to a replica on a DataNode when the Replica Mover restarts it, the write will fail, and Hadoop will simply create a replica on some other DataNode. This results either in an over replication of the data block, or the data block being placed on a Hadoop-chosen location (potentially not where the Replica Mover wants it to be).
2) If Hadoop writes to a block while some replica has already been moved by the Replica Mover but not yet reported to Hadoop (via restarts), Hadoop will classify this replica as outdated (as it performed a later write to one of them) and will delete them.

Both may result in sub-optimal placement or over/under replication. The Replica Mover fixes both cases during its subsequent run.

III. Results

A. Experimental Setup

We test and evaluate Mandrake using a Eucalyptus [2] edge cloud and Hadoop v3. For all software the Linux distribution is CentOS 7. Each VM is allocated 2 CPU cores, 4 GB of memory, and 100 GB of disk space. The cloud consists of 9 Intel Next Unit of Computing (NUC) machines connected via a gigabit switch.

TABLE I
EXPERIMENT CATEGORIES

Category	Replica Repair	Spawn VM	Use New VM
RM	Replica Mover	Yes	Yes
R	Hadoop	Yes	Yes
NR	Hadoop	Yes	No
NR2	Hadoop	No	No

Our evaluation of Mandrake studies the effects of its use on the Hadoop service while attempting to ensure that data replicas are always hosted on separate physical nodes (to preserve failure independence). We consider three Hadoop benchmarks from the HiBench [11] benchmarking suite: Indexing (Nutch), TeraSort, and Naive Bayes.

The Mandrake Coordinator (MC) attempts to maintain the invariant that Hadoop will be allocated 8 VMs regardless of physical node count. At the start of each experiment, it assigns one VM per physical node (the 9th node in the cloud is a control node). The Mandrake Application Orchestrator (MAO) configures these VMs into a working, empty Hadoop cluster. The Hadoop specific configuration settings we use are available at https://github.com/MAYHEM-Lab/Mandrake-App-Hadoop-Configs. In particular, we use Hadoop's default replication count of three replicas in all experiments.

The MAO installs a HiBench benchmark and any necessary data in HDFS and executes the benchmark. We use the same data for every experiment that uses the same benchmark. Once the Map phase completes approximately 10% of execution time, we simulate a failure by rebooting a physical host. We chose this point in the execution because doing so (without recovery) causes both work and data loss (i.e. all workers are busy) in Hadoop. Since each host has sufficient resources to host two VMs at a time, Mandrake has sufficient capacity to maintain the invariant of 8 VMs through co-location.

Hadoop's own internal repair mechanisms require a significant amount of time (more than 10 minutes, by default) to detect and reschedule work from a failed VM. To aid experimentation, we configured this repair response interval to be 1 minute instead of 10.

Importantly, we are not yet accounting for any "NameNode" (Hadoop's master process) failures since Hadoop has options for its high availability (HA) [16]. Unfortunately, our use of this option in our resource constrained setting destabilizes Hadoop (e.g. causes it to crash, hang, etc.), when we deploy it with the other Hadoop processes (without Mandrake). So, our results do not include the use of this Hadoop option. We hope to explore this option more in future work.

B. Experiment Categories

Table I shows the different types of experiments (in terms of failure occurrences and Mandrake responses) to which we subjected our system in this study. RM (Reconfigure + Replica Mover) implements Mandrake's full functionality (Hadoop's

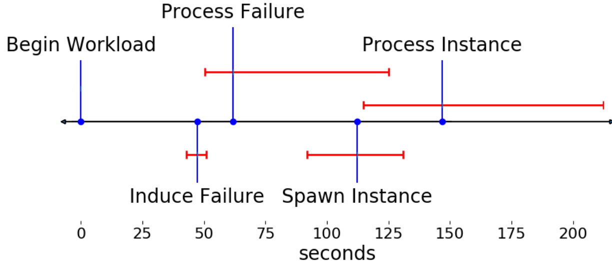

Fig. 1. Time line of phase duration during host fail over, with average latencies (in blue) and error bars (in red) depicting the empirical 0.95 confidence intervals.

internal replica repair system is disabled for this category). In all other experimental categories, we disable the Replica Mover and allow Hadoop to implement replica repair using its own internal mechanisms. In these cases, however, Hadoop may assign replicas to VMs that share physical nodes (Mandrake will not), thus introducing the possibility for correlated failures. As such, the R (Reconfigure) experiments implement VM replacement without data movement latencies; R depicts base cloud reconfiguration times.

For the NR (No Reconfigure) and NR2 (No Reconfigure v2) categories, we do not configure a replacement VM into Hadoop. With NR, we spawn a VM but do not use it; with NR2, we do not spawn a VM. We compare RM to R to gauge Replica Mover interference (vs Hadoop's replica repair). We compare R and RM to NR and NR2 to gauge the performance impact of replacing a VM. We also compare NR to NR2 to measure any interference caused by the MC's repair operations.

We remove outliers in our experimental results that are more than three standard deviations away from the median. This process did not remove more than one point (out of 50+ runs) from any category. We did consider relative significance with and without the outliers and found the results to be the same. Our results include 50 runs from every category.

C. Mandrake Coordinator Time Line

Figure 1 shows average execution time in seconds for the phases most relevant to the MC when we enable the Replica Mover (i.e. perform an RM experiment) for a typical, single-failure, repair operation. We compute the average phase durations using 50 runs of the Naive Bayes Hadoop benchmark. We induce a failure approximately 50 seconds after startup. The error bars show the empirical 0.95 confidence interval for each average which exposes the variability of this real system.

On average, the MC detects and processes the failure approximately 20 seconds after the failure is induced (marked *Process Failure* in the figure). Next, the MC spawns an instance about a minute after detection (*Spawn Instance*). At this point, the MC informs the MAO that a new instance is available. The MAO requires 30 seconds on average (marked *Process Instance*) to deploy Hadoop and all other MAO services in the newly spawned VM.

Note that the 0.95 empirical confidence intervals (shown as red horizontal error bars in the figure) of the later phases are quite large relative to the average. This is primarily due to the variability of when each phase occurs, and how this variability propagates such that it magnifies the "range" of the confidence intervals with each succeeding phase.

TABLE II
TERASORT RUNTIME

Category	Average Runtime (seconds)	Standard Deviation
RM	460	9.84
R	447	12.0
NR	444	9.21
NR2	446	8.87

TABLE III
TERASORT SIGNIFICANCE

Comparison	Significance		
	P-Value	*T-Value*	*Mean Perc. Diff.*
RM v R	1.264e-07	5.716	2.788%
RM v NR	8.158e-13	8.236	3.507%
RM v NR2	5.340e-11	7.384	3.085%
NR v R	0.1407	-1.486	0.7191%
NR2 v R	0.5347	-0.6231	0.2968%
NR v NR2	0.3061	-1.029	0.42223%

This time line also provides some insight into the frequency of failures Mandrake can tolerate without halting the application to wait for stability. For this benchmark, the MC "hands off" the mapping of the reconfigured VMs to the MAO for "ingestion" by the Replica Mover (and, subsequently, Hadoop) approximately 80 seconds after a failure. Again, Hadoop does not pause its execution so this time period represents the average time from a failure to a full repair of the virtualized infrastructure. Thus Mandrake can completely restore the infrastructure needed by Hadoop approximately every 80 seconds.

Internally, the critical time period during which Mandrake may experience additional failures is short. Indeed, the period of stability required to synchronize the current Eucalyptus state with MAO using Zookeeper is approximately 6.5 seconds. The rest of the delay observed in the time line is the amalgamation of latencies associated with the eventual consistency of the Eucalyptus and Zookeeper platforms. Thus it is possible that Mandrake can tolerate a much higher failure frequency than one failure every 80 seconds. We have not, at this juncture, tested the upper bound on failure rate and, instead, report only the rate at which Mandrake can make a full restoration of the virtualized infrastructure.

D. Mandrake's Impact on Application Performance

Mandrake's infrastructure repair times are remarkably similar across benchmarks. Thus, for brevity, rather than providing a time line similar to that shown in Figure 1 for each benchmark, we illustrate the impact of Mandrake on the "end-to-end" benchmark performance in this subsection.

Table II shows the average and standard deviation across the 50 runs of TeraSort. We use a 16GB data set (48GB with replication) for this test. TeraSort is CPU bound in the Map phase and I/O bound in the Reduce phase. The average run times for TeraSort shown in Table II are quite similar, and the standard deviations relatively low. Comparing RM (full Mandrake reconfiguration with durability guarantees)

TABLE IV
NAIVE BAYES RUNTIME

Category	Average Runtime (seconds)	Standard Deviation
RM	1571	43.98
R	1519	39.58
NR	1543	47.69
NR2	1540	38.47

to R (no durability guarantees) shows that the "cost" of preserving replica failure independence for data durability is approximately 13 seconds out of a 450 second execution time.

Indeed, the data in Table II warrants a test for statistical significance for the differences of the means. In Table III, we show Student t-test statistics and p-values between means for each pair of experiment categories for TeraSort. Gray rows of the table indicate a rejection of the NULL hypothesis that the means are the same at the $\alpha = 0.05$ significance level. For illustration purposes, we also show the percentage difference between the means.

From the table, it is clear that the effect of the Replica Mover on the mean execution time is small, but significant. However the effect of the MC (as shown by NR versus R and NR2 versus R) is not detectable at the $\alpha = 0.05$ significance level. Thus, it is the overhead of moving replicas to ensure failure independence and not the overhead of virtual infrastructure repair that accounts for the difference in the mean benchmark times end-to-end.

The results for Naive Bayes are slightly different than for TeraSort. Table IV shows a comparison of the average run times and their standard deviations for the 50 runs of Naive Bayes. For this benchmark RM, NR, and NR2 are similar, but R is dramatically lower. In this case, allowing Hadoop to reconfigure its replicas internally (the R experiment) improves average execution time over the case where we use the external Mandrake Replica Mover to move the replicas. Unlike TeraSort, this workload consists of multiple "chained" jobs, each requiring replica reconfiguration. Hadoop's internal replica management mechanisms are, thus, more efficient but *they do not ensure 3-replica failure independence* in this setting. That is, R is faster, but replicas are often co-located on the same physical node. Thus the difference between R and RM in this case shows the cost of maintaining data durability guarantees when many short jobs are chained together in a Hadoop workload. Again we performed a t-test statistic on these results, which showed that the difference between all means (except NR versus NR2) is statistically significant at the $\alpha = 0.05$ significance level (significance table omitted due to space constraints).

The Nutch Indexing benchmark results are qualitatively between those for TeraSort and for Naive Bayes. We summarize them due to space constraints. Like TeraSort, there is a small but significant difference in mean execution time introduced by Mandrake end-to-end (the overall execution time for the Indexing benchmark is approximately 580 seconds). Like Naive Bayes, Hadoop's replica management is faster than the Replica Mover (although only slightly in the case of Indexing) but it again does not preserve failure independence.

TABLE V
REPLICA MOVER ACTIVITY

Rep. Mover Activity	Naive Bayes		TeraSort	
	Avg. Secs	***Avg. Percent***	***Avg. Secs***	***Avg. Percent***
Prepare	17.0s	6.97%	14.2s	3.30%
Execute	88.0s	35.3%	272s	61.3%
Restart	77.4s	28.1%	86.1s	20.4%
Cancel	55.3s	22.1%	54.5s	12.6%
Clean	11.5s	4.14%	9.95s	2.20%
Avg. Total	248 Seconds		435 Seconds	

E. Replica Mover Performance

Clearly, from the data shown in the previous subsection, Mandrake's ability to preserve failure independence imposes a performance overhead with respect to Hadoop's execution performance. To understand these effects more completely, we instrument the Replica Mover's internal mechanisms and describe their timings.

The Replica Mover interacts with Hadoop by interrogating HDFS to determine the current replica status. We denote this activity as *Prepare* in our analysis. The current implementation of Hadoop requires the DataNode to restart in order to "sense" a change in the population of replicas that are co-located with it. The Replica Mover carefully coordinates these restart events (as described in section II-C and denoted *Restart*) when it moves replicas from one VM to another. The DataNode on the source VM must be restarted to "forget" the replica and the DataNode on the target VM must be restarted to record the arrival of a replica. Moreover, at no time can more than 3 active DataNodes record a replica as being local (or the NameNode will mark a replica as "EXCESS" and begin to ignore it, eventually deleting it). The Replica Mover ensures this invariant. If a failure happens while the Replica Mover is engaged in this coordination or if a new VM is added to the resource pool for Hadoop, the Replica Mover performs a global replica state synchronization by restarting all DataNodes with unreported replicas (denoted as *Cancel*). The data movement phase of the Replica Mover is denoted *Execute* in our analysis and the time when the Replica Mover deletes a replica that has been moved from a source to a target from its source is denoted as *Clean*.

Table V shows average times for these phases after activations of the Replica Mover due to a failure. For these experiments, we allowed the Replica Mover to complete after each failure and Mandrake to complete a repair (hence the timings for *Cancel*).

Table V shows how factors change as the amount of data to manage increases. The Naive Bayes experiments used 2 GB of data (6 GB after replication), and the TeraSort experiments used 16 GB (48 GB). As expected, the percentage of time spent executing relative to other phases increased. The total time of *Restart* increased as well with the larger data amount, but the relative percentage decreased. This is because the low amount of data for Naive Bayes meant not every DataNode got involved in every Replica Mover run. This data scaling suggests that improving the way the Replica Mover executes

data transfers to work in a more distributed manner would be beneficial. Currently, only the Replica Mover leader (as determined by Zookeeper) actually executes the transfers. However, such a system would still incur the costs of *Restart* and *Cancel*.

We also investigated whether network and/or disk I/O contention could be the cause of the Replica Mover overhead and found, instead, that the primary source of overhead was, in fact, *Restart* and *Cancel* activities. Apparently, a combination of setting Hadoop's replica repair interval to much higher than usual and aggressively restarting DataNodes causes the slowdown. We hypothesize that part of the effect may be due to a loss of locality, consistent with results reported in [17], but we have yet to identify the root cause for this overhead.

IV. Related Work

Various groups have investigated running cloud applications on cheap and resource-limited devices [18] [19] [20]. The authors of [18] provide guidance for running Hadoop in resource constrained settings.

Similarly, fault tolerance and automated recovery has been widely studied for cloud systems. Li et al. [21] describe a cloud auto-healing framework consisting of a multi-part system in which the user defines a policy to guide the health and recovery services in handling failures. Javed et al. [22] explore fault tolerance and automated recovery in a small edge cluster via Containers, Kubernetes, and Apache Kafka. Su et al. [23] provide automated recovery of failed components in a M2M sensor network by invisibly replicating them. Similar to our system, theirs aims for the "deploy once, run forever" model of enabling applications to run without human intervention.

V. Conclusion

Mandrake is a system for automatically recovering from failures in small edge clouds without human intervention. When a failure occurs, the Mandrake Coordinator reallocates resources to fit user-defined goals, allowing for continued operation at degraded performance (but not degraded functionality). The Mandrake Application Orchestrator, running on the provisioned VMs, manages the application configuration changes needed to continue operation in the face of system configuration changes (in place of a human administrator). With Hadoop as a case study, we have demonstrated that our collection of services introduces little overhead compared to "native" Hadoop execution. Furthermore, Mandrake can resiliently place and balance HDFS data with only minor slowdowns and without modifying Hadoop. We hope to explore the effects of this in future work, as well as expand on the fault tolerance mechanisms and multi-application-aware reconfiguration capabilities of the system.

This work is funded in part by NSF (CNS-1703560, OAC-1541215, CCF-1539586, ACI-1541215), and ONR NEEC (N00174-16-C-0020).

References

[1] C. Systems, "The internet of things," 2019. [Online]. Available: https://www.cisco.com/c/dam/en/us/products/collateral/se/internet-of-things/at-a-glance-c45-731471.pdf

[2] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The eucalyptus open-source cloud-computing system," in *IEEE Cluster Computing and the Grid*, 2009.

[3] A. S. Foundation, "hdfs-default.xml," 2018. [Online]. Available: https://hadoop.apache.org/docs/r3.0.3/hadoop-project-dist/hadoop-hdfs/hdfs-default.xml

[4] Amazon.com, "Amazon ec2," 2019. [Online]. Available: https://aws.amazon.com/ec2/

[5] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," *USENIX Annual Technology Conference*, 2010.

[6] A. Shraer, B. Reed, D. Malkhi, and F. P. Junqueira, "Dynamic reconfiguration of primary/backup clusters," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX, 2012, pp. 425–437. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/shraer

[7] Eucalyptus Open Source Project, "http://www.eucalyptus.cloud," Feb. 2019.

[8] O. O. S. Project, "https://www.openstack.org," Feb. 2019.

[9] O. N. O. S. Project, "https://opennebula.org," Feb. 2019.

[10] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251264

[11] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, March 2010, pp. 41–51.

[12] Amazon, "Aws marketplace," 2019. [Online]. Available: https://aws.amazon.com/marketplace

[13] A. S. Foundation, "Hdfs user guide," 2018. [Online]. Available: https://hadoop.apache.org/docs/r3.0.3/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html

[14] D. Borthakur, "Hdfs block replica placement in your hands now!" 2009. [Online]. Available: http://hadoopblog.blogspot.com/2009/09/hdfs-block-replica-placement-in-your.html

[15] ——, "Design a pluggable interface to place replicas of blocks in hdfs," 2009. [Online]. Available: https://issues.apache.org/jira/browse/HDFS-385

[16] A. S. Foundation, "Hdfs high availability using the quorum journal manager," 2018. [Online]. Available: https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithQJM.html

[17] Z. Guo, G. Fox, and M. Zhou, "Investigation of data locality in mapreduce," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, ser. CCGRID '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 419–426. [Online]. Available: https://doi.org/10.1109/CCGrid.2012.42

[18] N. J. Schot, "Feasibility of raspberry pi 2 based micro data centers in big data applications," 2015.

[19] P. Abrahamsson, S. Helmer, N. Phaphoom, L. Nicolodi, N. Preda, L. Miori, M. Angriman, J. Rikkil, X. Wang, K. Hamily, and S. Bugoloni, "Affordable and energy-efficient cloud computing clusters: The bolzano raspberry pi cloud cluster experiment," in *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, vol. 2, Dec 2013, pp. 170–175.

[20] F. P. Tso, D. R. White, S. Jouet, J. Singer, and D. P. Pezaros, "The glasgow raspberry pi cloud: A scale model for cloud computing infrastructures," in *2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops*, July 2013, pp. 108–112.

[21] X. Li, K. Li, X. Pang, and Y. Wang, "An orchestration based cloud auto-healing service framework," in *2017 IEEE International Conference on Edge Computing (EDGE)*, June 2017, pp. 190–193.

[22] A. Javed, K. Heljanko, A. Buda, and K. Frmling, "Cefiot: A fault-tolerant iot architecture for edge and cloud," in *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*, Feb 2018, pp. 813–818.

[23] P. H. Su, C. Shih, J. Y. Hsu, K. Lin, and Y. Wang, "Decentralized fault tolerance mechanism for intelligent iot/m2m middleware," in *2014 IEEE World Forum on Internet of Things (WF-IoT)*, March 2014, pp. 45–50.