USING DATA ANALYTICS FOR RELIABILITY AND AUTONOMIC MANAGEMENT OF LARGE-SCALE SYSTEMS

BY ALEJANDRO PELÁEZ

A thesis submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Electrical and Computer Engineering

Written under the direction of

Manish Parashar

and approved by

_____

_____

_____

_____

New Brunswick, New Jersey

October, 2015

ABSTRACT OF THE THESIS

Using Data Analytics for Reliability and Autonomic Management of Large-Scale Systems

by Alejandro Peláez

Thesis Director: Manish Parashar

Large-scale clusters are growing at a rapid pace, and the resulting amount of monitoring data produced in these systems is also increasing. The goal of this research is to investigate tools that improve the reliability and help manage such systems using this wealth of data. This is a challenging problem as the scale of these machines increases the complexity, the amount of monitored data, and amount of interactions between different nodes, making the system much harder to manage and also resulting in high failure frequency. In this thesis we focus on online failure prediction and policy based management as mechanisms that can help address these issues. First, in case of failure prediction we focus on achieving an acceptable accuracy that is comparable other algorithms, but with the objective of being able to scale to thousands of nodes (given that typical centralized solutions suffer from high transmission and processing overheads at very large scales). Our solution to this problem is based on a decentralized online clustering algorithm (DOC) to detect anomalies in resource usage logs. We show that we can in fact achieve a similar accuracy as other algorithms while scaling to thousands of nodes with less than 2% overhead. Second, high level policies are an attractive option for managing complex systems and ensuring that they run within certain restrictions,

as policies can be specified in terms of business goals and do not require low level knowledge of the machines. In order to enable this, we need a way of dynamically mapping the state of the system to the high level policies. We consequently propose a machine learning solution based on monitoring data, wherein we make predictions of the high-level indicators of the state of a system in order to determine what actions have to be taken to satisfy a given policy. We evaluate our approach using a sample system, and demonstrate that neural networks do an excellent job at predicting the required state, only incurring an error of at most 8.78%, 98% of the time.

Acknowledgements

I want to acknowledge my family and my girlfriend, their constant support and encouragement made all this possible. I also want to thank everybody at RDI^2, they made the lab a more interesting place, specially Mehmet and Gia who became great friends. Many thanks to Prof. Manish Parashar, his support and guidance were a great help during my studies and research. I also want to thank Dr. Andres Quiroz for his support and input in my research. Finally thanks to Prof. Ivan Rodero, Prof. Ivan Marsic and for being part of my committee.

Part of the research presented in this thesis was presented and publish in the HiPC conference, and can be found here [1]. The research presented in this work is supported in part by the US National Science Foundation (NSF) via grants numbers ACI 1339036, ACI 1310283, DMS 1228203 and IIP 0758566, by the Director, Office of Advanced Scientific Computing Research, Office of Science, of the U.S. Department of Energy through the Scientific Discovery through Advanced Computing (SciDAC) Institute of Scalable Data Management, Analysis and Visualization (SDAV) under award number DE-SC0007455, the RSVP grant via subcontract number 4000126989 from UT Battelle, and by an IBM Faculty Award. The research was conducted as part of the NSF Cloud and Autonomic Computing (CAC) Center at Rutgers University and the Rutgers Discovery Informatics Institute (RDI2). The research has used NSF grants 1203560 for the SUPReMM project which developed the rationalized logs, 0622780 for Stampede and XSEDE resources as part of the grant TG-CCR130025.

Dedication

My parents, my brother and my beautiful girlfriend Laura.

Table of Contents

List of Tables

List of Figures

Chapter 1

Online Failure prediction using decentralized clustering

1.1 Introduction

1.1.1 Motivation

The first chapter in this thesis focus on reliability of large-scale clusters, specifically on the aspect of failure prediction using monitoring data in a scalable manner. This is important as large-scale clusters often fail due to either the failure of an individual component or an error arising from a fault in the complex interactions among components, and as these clusters get bigger and bigger, the frequency of failures is becoming a real and very important concern [3] [4]. As a result, monitoring these complex systems and interactions, and detecting and predicting failures and other conditions based on the large amount of monitoring data have become critical for ensuring reliability. There have been approaches in the form of diagnostics [5] [6] [7], which help in understanding and fixing the errors after they happen, and in some cases may prevent some future failures. While detection and diagnosis is essential once the error has occurred, the reliability constraints and size of these systems make it impossible for system administrators to address all failures in a timely manner and thus avoid undesired downtime to the system. For this reason, prediction is becoming increasingly important, as it allows preventive measures to be taken by administrators, or even automatically, that guarantee the reliability of the system. Moreover, the ideal solution would run in real-time which forces it to scale linearly with the number of nodes in the system, and thus decentralized solutions could provide an advantage.

1.1.2 Real world example

At the Texas Advanced Computing Center at the University of Texas at Austin, failures in the form of compute node soft lockups are some of the most frequent source of problems for the systems administrators of the Ranger supercomputer. Faults recorded in system event logs have been shown to provide good indicators of failures [8], often minutes or even hours before the system failure actually occurs. While flagging these faults at runtime is already a valuable tool in the prediction of failures, the list of faults is static and can only be derived through an exhaustive offline analysis of multiple sets of logs. The same study, however, found that resource anomalies in resource usage data contained in these logs are present most of the time in faulty nodes. Thus, detecting anomalies in node resource usage at runtime could also be used as a means to predict future soft lockups, potentially with the same or greater lead time than that provided by the fault events, and with the additional advantage of having a greater capacity to adapt to changing system conditions.

1.1.3 Overview and approach

In order to predict failures and be able to react fast enough, system log data must be monitored and analyzed continually online (i.e. in soft real-time). Although multiple monitoring and anomaly detection approaches are available, centralized solutions can become cost-prohibitive as system scales grow due to the sheer amount of data movement as well as the computation (and energy) required to transport and process the entirety of the data. Distributed solutions, on the other hand, have the potential of scaling much better but present the challenge of attaining a global solution comparable to that of centralized approaches by using only local interactions, and restricting communication to a minimum in order to ensure high node scalability.

We present a solution to the problem of predicting compute node soft-lockups[1] in a scalable and real-time way, via online anomaly detection in system event logs using a Decentralized Online Clustering (or DOC) algorithm [9] [10] [2], which has been

[1] A compute node soft lockup is a hang-up or a crash of a compute node.

developed for distributed system monitoring and resource provisioning, in addition to addressing the issue of scalability being able to scale to thousands of nodes while maintaining low bandwidth and memory usage with less than a 2% time overhead for running applications and thus contributing to the feasibility of online failure prediction. The application of DOC is also meant to provide support to the hypothesis that resource usage anomalies are good predictors of failures in large supercomputer systems. We will show that this approach, including enhancements that will also be described, can achieve good prediction accuracy when compared to similar algorithms and to the state of the art in machine learning or data mining based failure prediction. Thus, our approach can be generalized as a tool for monitoring and prediction in these systems.

1.2 Problem description and related work

Compute Node Soft Lockups are some of the most frequent sources of problems for the systems administrators of the Ranger supercomputer at the Texas Advanced Computing Center (TACC) at the University of Texas at Austin. These soft-lockups have led to premature termination of several running jobs which had to be manually restarted by the systems administrators. A soft-lockup event is identified by a log event in the rationalized logs that contains the following keywords: BUG: softlockup stuck for. Predicting these events is crucial for ensuring system reliability, but is getting more complicated as the size and complexity of large-scale systems continue to increase. Our process focuses on recognizing the potential occurrence of compute node soft-lockups because many faults, both hardware and software, eventually lead to the node where the fault originates locking up and thus blocking execution of a job.

On these systems, we have at our disposal a special type of logs, called the rationalized logs [11]. They provide a standardized format which contains information about resource usage, job id, node id, and time. It has been shown [8] that resource anomalies contained in these logs are present most of the time in faulty nodes, and moreover a strong correlation has been found between certain types of events and the occurrence of future compute soft lockups. These two observations lead us to the hypothesis that by considering anomalies in only a subset of features from the rationalized logs (which has

	1-Mar	**6-Mar**	**9-Mar**	**11-Mar**
Count of nodes	16	1	2	1
% of nodes	0.4%	0.025%	0.05%	0.025%
Count of soft lockup events	177	951	801	1393

Table 1.1: Summary of the compute nodes that locked up in March 2012. The Ranger supercomputer is configured with 3936 nodes

less dimensionality than the whole set of features), namely the resource usage features that are contained in the events correlated with compute node soft-lockups, we can predict future soft lockups accurately.

In order to test this we used the Ranger supercomputer at the Texas Advanced Computing Center at the University of Texas at Austin and collected the rationalized logs for the month of March 2012 and extracted the soft lockup events. A distribution of soft lockup events is shown in Fig. 1.1, and the counts of nodes that locked-up is given in Table 1.1. In Fig. 1.1, we observed that these soft lockups occurred more than 300 times during March 2012. We also observed, in Table 1.1, that these soft lockups occurred on a very small count of nodes. In March 1st, we identified 0.4% of compute nodes that locked-up and 171 soft lockup events that were generated. In March 6th, we identified 0.025% of compute nodes that locked-up and 951 soft lockup events that were generated. In March 9th, we identified 0.05% of compute nodes that locked-up and 801 soft lockup events that were generated. In March 11th, we identified 0.025% of compute nodes that locked-up and 1,393 soft lockup events that were generated. We observed that compute node soft lockups occurred on 4 out of 31 days in March 2012, and that a very small percentage of compute nodes that locked-up and the large quantity of soft lockup events that were generated by these nodes showed that compute node soft lockups are clustered on a very small number of nodes, which suggest that they are indeed related to some kind of anomalous behavior.

Assuming that this correlation is a good indicator for failures and since these resource usage features can be represented as points in an n-dimensional space, any

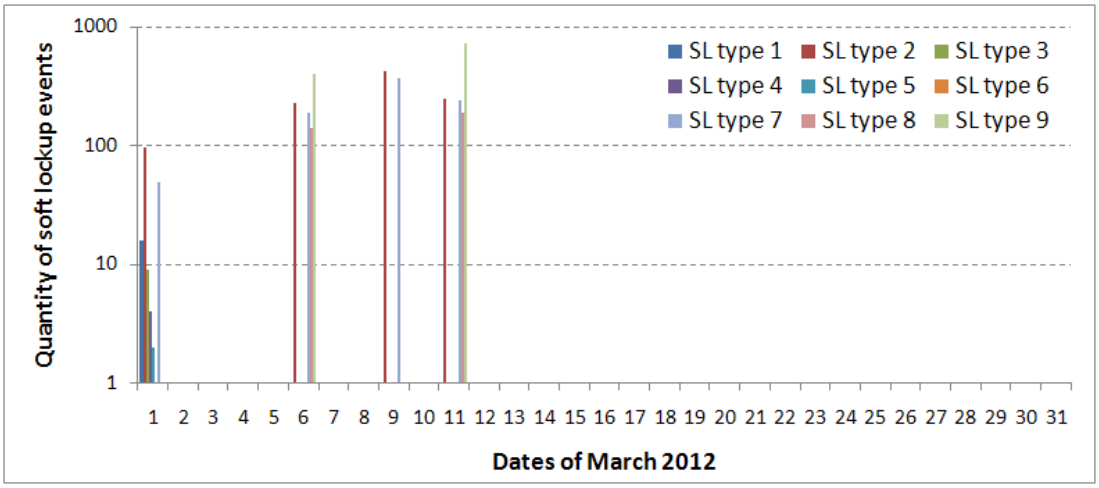

Figure 1.1: Distribution of compute node soft lockup in March 2012

anomaly detection or classification algorithm could help us predict errors. For this purpose, we have at our disposal several techniques that allow us to tackle the problem. For example there has been research on data mining and machine learning techniques [12] [13] [14] where data points extracted from message logs are either categorized as normal or anomalous. They have been found to provide very good results in terms of false negatives, but they tend to suffer in the false positive rate due to the fact that failures are very rare events, making it hard for classifiers to pinpoint them without dragging some more events along the way. Another alternative are probabilistic and statistical models, based on techniques such as Bayesian networks [15] or Semi-Markov processes [16], which usually model the probability of failure of the system given current characterization of the systems state. These approaches, while achieving very good results, are very computationally expensive, and thus are limited in their practical use on extreme large-scale systems.

Despite the advantage of the classification techniques that do not need a very big history or training set to be able to work, their scalability when applied to very large clusters is still a real concern. Any failure prediction system meant to be used in a production environment must be lightweight, meaning that it should require the minimum amount of resources as possible, as it is a supporting system that is going to run periodically on every node. It should also be able to deal online with large amounts

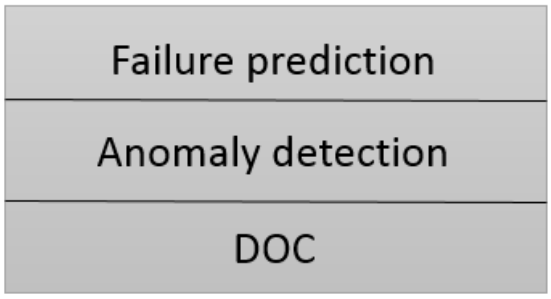

Figure 1.2: Software stack. The bottom layer is in charge for data distribution and clustering. The middle layer is in charge of getting anomalies from the cluster results. The top layer is responsible for predicting failures based on the anomalies.

of data, as is not unusual for current large-scale supercomputers to exceed 5 thousand nodes, each one producing detailed monitoring data every second that must be checked for anomalous behavior. This is indeed a hard problem, and is where current solutions fall short, since anomalous behavior should be detected as soon as it happens in order to anticipate failure events, and, at these scales, even data collection starts to become a bottleneck.

A totally distributed solution can avoid the data bottleneck problem and also leverage the power of all the nodes present in the system and scale with it, as long as it is able to optimize data movement by doing part of the computation in the nodes that produce the data. With this in mind, our proposed approach is in the vein of the data mining and machine learning methods, using a distributed clustering algorithm called DOC. We show that we can achieve similar or better results (with a combination of methods) as other data mining techniques, specifically other related clustering algorithms, with the added advantage of being lightweight, distributed, and optimizing data movement. We also show that we can achieve low bandwidth usage, and less than 2% overhead while still being able to run in under 3 minutes on real-time system-wide data, to be able to predict failures online.

1.3 Outlier detection approach

Our solution has 3 main components as seen on Fig. 1.2 The bottom layer is called DOC, and it will be described later in section 1.3.1, but it will the one responsible for handling how the points get distributed to the nodes, and implements the clustering algorithm.

Built on top of it, we have the anomaly detection layer, which directly uses the clusters generated by DOC and extract the anomalies from them, it has the responsibility of keeping track of the logs and extract the relevant features from them in order to be able to invoke DOC with the correct data.

The top layer is in charge of making predictions based on the anomalies found in the middle layer. At first, it may seem that it does not have any purpose, as the anomalies will be quite easily converted to predictions (i.e just extract the node id). But as we will see in section 1.3.3, we can apply several techniques to improve the accuracy of the solution, which requires us to have this layer as it simplifies the actual implementation.

Having said this, we focus on the problem of anomaly detection, for this we can find plenty of ways to detect outliers, but not all techniques are created equal: some are more accurate than others, and some require more resources. We specifically need a solution that can quickly detect anomalies in a distributed manner, which can scale without problem to thousands of nodes, and that runs with a small overhead for existing applications. For this purpose we have developed a Decentralized Online Clustering algorithm or DOC [9] [10] [2].

1.3.1 Decentralized online clustering (DOC)

DOC is a clustering algorithm designed to run in a distributed setting (i.e. on a set of networked nodes, which we call processing nodes) on data that is likewise distributed, usually generated by or input via the processing nodes. In the log monitoring use case, nodes generate data points, each one of which is an entry in the rationalized logs. Instead of collecting data from the nodes at a centralized location, DOC redistributes data among the nodes without centralized control, so that each node receives data that

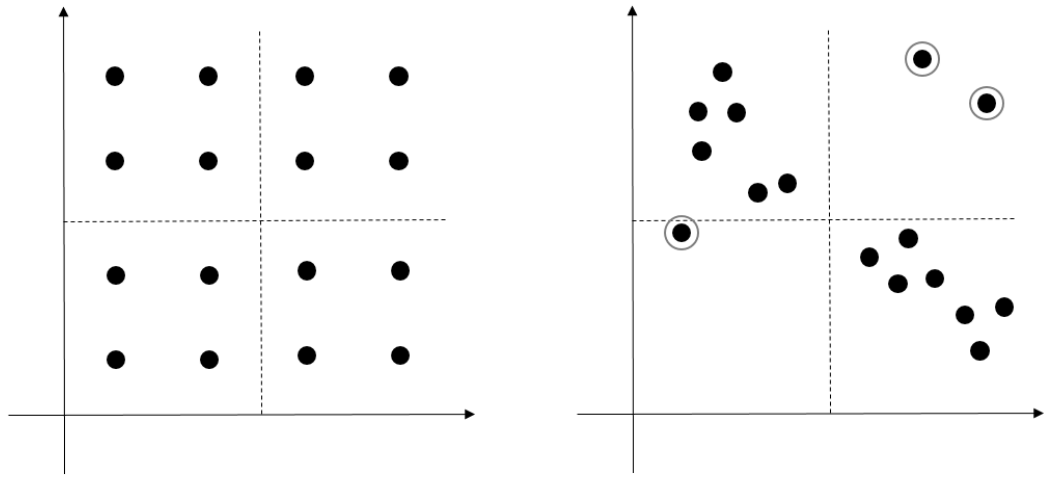

Figure 1.3: (a) Uniform distribution of data points in the information space. (b) Point clustering and anomalies among regions; the circled points are potential anomalies. Taken from [2]

is close together in the n-dimensional data space (referred to as the information space). Thus, if a cluster is present in the data, most of the clusters points will be distributed to one or a reduced set of nodes that will be able to detect it.

DOC applies a density-based clustering approach. In other words, cluster detection is based on evaluating the relative density of points within the information space. If the total number of points in the information space is known or can be reasonably estimated, then a baseline density for a uniform distribution of points can be determined and used to calculate an expected number of points in a given sub-region of the space. Clusters are recognized within a region if the region has a relatively larger point count than this expected value. Conversely, if the point count is smaller than expected, then these points may potentially be outliers or isolated points. These concepts are illustrated in Figure 1.3.

In DOC, the information space is subdivided dynamically into regions, and each region is assigned to a particular processing node. Processing nodes can then analyze the points within their region independently to detect clustered points and outliers, performing a very lightweight computation (basically comparing point counts to the

expected count), and only communicating with nodes responsible for adjoining regions to deal with boundary conditions (for clusters that span multiple regions) and for aggregating cluster data if necessary.

The subdivision of the information space into regions and the decentralized distribution of points to the different nodes are supported by a content-based Distributed Hash Table (DHT) that uses a Peano-Hilbert space-filling curve [17] as a locality preserving hash function. As shown in Figure 1.4, the space-filling curve determines a dynamic mapping of regions of the information space to processing nodes that are organized into a one-dimensional address space in a ring topology. The DHT implementation is responsible for getting the data points within each region to the distributed processing nodes in a scalable fashion.

It should be noted that the process which has been described using the rationalized logs [11] from the now retired Ranger supercomputer can be applied to any Linux based system which uses standard syslog event logging. The only data in the rationalized logs that is used in the DOC based process is the job ids. The job ids can also be obtained from the logs generated by most job schedulers including Slurm [18], Torque [19], etc. The Lariat [20] system processes logs from these schedulers to correlate job ids with system resources such as the nodes upon which a job is running. Therefore, our process is readily adapted to most Linux based cluster systems.

1.3.2 Caching for execution time reduction

DOC was designed for a more general type of distributed system, and as such it is very flexible regarding topology changes. Because we are running in tightly coupled clusters, which provide a more stable environment, a caching layer can be added to DOC to speed up point distribution and cluster data aggregation. The caching strategy works as follows. Every time a lookup operation of a point in the DHT gets resolved to the node responsible for its containing region, we also return the address space range that node is responsible for. That range, along with the node identifier, we will then be inserted in the local cache. In that way if another point is going to be inserted and it belongs to that same range, no lookup is performed (each lookup in the DHT carries

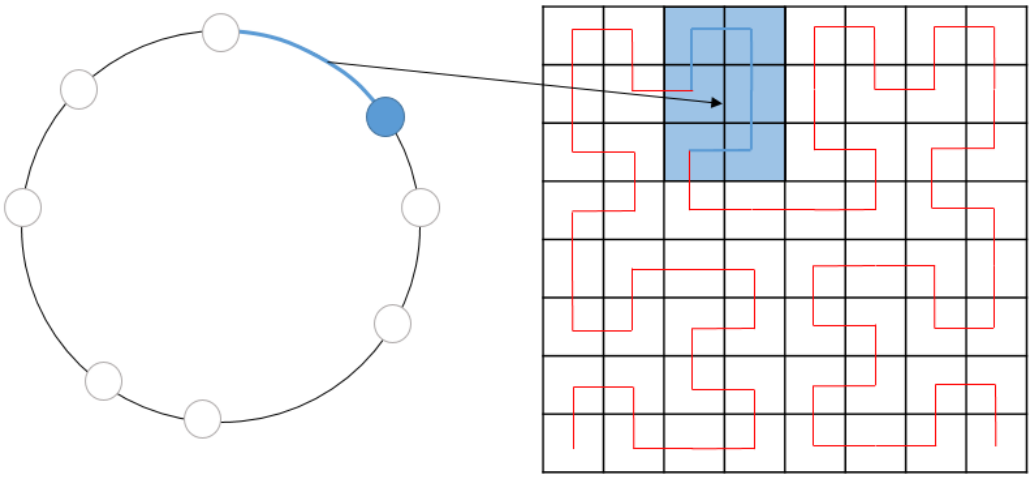

Figure 1.4: The highlighted node is responsible for the highlighted region, which is mapped onto a multidimensional space using an SFC. Taken from [2]

involves node communication overhead), and the request is sent directly to the node found in the cache. In order to avoid high memory overheads, we bound the number of such range-node pairs that are stored and use a LRU replacement policy. The reason for this policy is that points are not evenly distributed, so a reduced number of nodes have more probability of containing more points. Cache coherence is guaranteed because each node that receives a point over the network first checks to see if it is indeed contained in its own information space region. If not, it will perform a normal DOC insertion (ignoring the cache) and then will send a cache invalidation request to the original sender. Notice that in the case that a cache entry of the original sender was outdated, the average number of hops needed to insert the data will increase by at most 1. We consider this to be a very good trade-off, since cache entries will be valid most of the time (because the cluster is stable). In this way we can achieve $O(1)$ insertion rate most of the time.

1.3.3 Reducing false positives

As with other machine learning and data mining failure prediction strategies, our clustering approach with DOC suffered from a large false positive rate. In order to mitigate

this problem and thus improve the precision of our approach, we developed two complementary strategies, which can be used easily alongside DOC.

Multiple time bins

This strategy follows an observation by Chua et al. [8] , which says that outliers identified as such in a time bin[2] T1 which are also identified by a separate run of the algorithm in a larger time bin T2 that contains T1 (meaning that the start and end times o T1 are within the start and end times from T2) are more likely to be correlated to soft lockups. In other words, outliers corresponding to the intersection of the results of the algorithm run on the events over T1 and T2 are better predictors of soft lockups.

We empirically found that using a pair of time bins of size 5 and 10 minutes can increase the precision of the system. To apply the strategy, the usual outlier detection is run after two consecutive time bins of 5 minutes, saving but not reporting the results. Then, the algorithm is run with all of the points of the last two time bins (i.e. a 10 minute time bin), and only outliers found in the final result that were present in at least one of the two 5 minute time bins are reported (notice that we have two 5 minutes time bins, and one 10 minute time bin, as shown in Figure 1.5).

Multiple clustering

The dimensionality of the data, which forces us to only pick subsets of features to use for clustering, causes us to potentially leave important information behind even if the features used were picked based on past correlation results. Thus we need a way to be able to retain such information without increasing the dimensionality of the data. Following a similar approach as before and reporting results that are reinforced by more than one run of the algorithm, we found that by using a slightly different set of features and clustering according to those, as well as using the original set, the outliers present in both of runs are more likely to be correlated to the soft lockups. So, we can just run parallel clustering using the different sets of features, and then report only the outliers

[2]A time bin is an interval of time where data points are collected

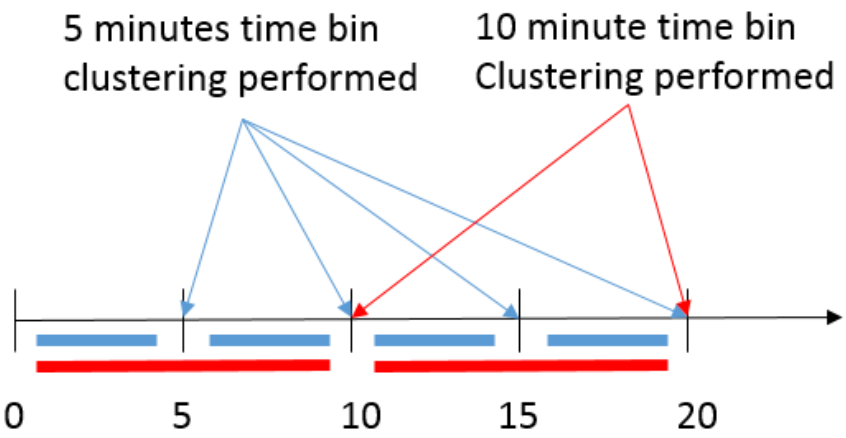

Figure 1.5: 5 minute time bins require data in the blue lines, and 10 minute time bins require data in the red line. So by just keeping the logs from the last 10 minutes we can perform both algorithms and avoid any extra data movement

found in both runs. The amount of false positives and false negatives achieved using this strategy will vary depending on the features that differ between the two sets. We found that a variation of only 20% of the features gives the best results. The drawback of this approach is that each data point needs to be sent to two different nodes, and thus the distribution overhead is higher, which is why we only use two different sets of features.

1.4 Experiments and results

1.4.1 Accuracy

There are two main dimensions to test in order to show the usefulness of the system. The first one is the accuracy of the prediction; and in this respect we want to show that our solution can achieve similar results as other related approaches such as DBSCAN [17] or PCA [21]. For this purpose we used the Ranger supercomputer at the Texas Advanced Computing Center at the University of Texas at Austin and collected the rationalized logs for the first week of March 2012, extracting the soft lockup events.

We have identified a total of 1128 soft lockups distributed over 17 nodes, which will be the ground truth to test our system against. We partitioned the rationalized logs in time bins of 5 minutes each; that is, we considered only those log entries that happened inside that specific 5 minute interval and nothing else. The reason for this specific time comes from an observation made by Chua et al. [8] which says that reducing the time bin size can yield better results up to a point. We performed several tests and arrived empirically at the value of 5 minutes, which gives good results and also is long enough to guarantee our system will run in time. This is meant to mimic the behavior of the real world, online scenario, where logs will be collected for a short time window and then analyzed. Our algorithm was then run for each time bin, and each outlier detected constituted a prediction of a soft lockup event (note that, in previous work [8], soft lockups were found to be highly correlated to previous anomalous resource utilization within a 100 to 318 minute interval).

To check the accuracy of each prediction, the corresponding outlier was matched against the ground truth by checking for soft lockups that happened at most 318 minutes into the future (i.e. 318 minutes after the timestamp of the outlier) on the node at which the outlier was detected. An outlier is considered a true positive if such a matching lockup is found and a false positive otherwise. All unmatched soft lockup events are counted as false negatives. Note that soft lockups found beyond of the 318 minute range are not counted as true positives, even though that would improve the measured prediction accuracy, since our previous study does not support a correlation outside of this time window (i.e. the match would probably be more a coincidence than an actual prediction).In order to avoid any bias in our tests, we limit ourselves to the time windows mentioned before.

All tests were run on Stampede at the Texas Advanced Supercomputer Center, using 32 processing nodes. We only used 32 nodes for this test because it deals with accuracy, which does not depend on the number of nodes. Accuracy was measured using the usual precision and recall metrics, which are defined as:

$$\text{Precision} = \frac{tp}{tp + fp}, \quad \text{Recall} = \frac{tp}{tp + fn}$$

Algorithm	Job outliers	Node outliers	Job precision	Job recall	Node precision	Node recall
PCA	2130	377	0.53	1.00	0.05	1.00
DBSCAN	1552	65	0.64	0.88	0.26	1.00
DOC	1559	102	0.56	0.80	0.15	0.88

Table 1.2: Base algorithms. Ground truth is 17 faulty nodes and 1128 soft lockups

Algorithm	Job outliers	Node outliers	Job precision	Job recall	Node precision	Node recall
PCA	2130	377	0.53	1.00	0.05	1.00
DBSCAN	1552	65	0.64	0.88	0.26	1.00
DOC	1559	102	0.56	0.80	0.15	0.88
DOC + MC	1431	37	0.68	0.86	0.41	0.88
DOC + MC + TB	1385	31	0.73	0.89	0.52	0.94

Table 1.3: DOC with several enhancements. SL = Soft lockups. MC = Multiple clustering. TB = Multiple time bins

Where tp, fp and fn means true positives, false positives and false negatives respectively. The results are presented in table 1.2.

As intended, all three algorithms achieve similar accuracy results, with PCA being most inclusive and thus minimizing false negatives and DBSCAN being least accurate overall. DOC trades off some accuracy in the false negative rate, as seen on the recall rate, to achieve better accuracy than the others in the false positive rate, shown by the highest precision rate. However, these results show what we mentioned before, which is that these algorithms have bad precision in general, but very good recall rates. Next we tested our different accuracy enhancement modifications from Section 1.3.3 using the same number of nodes and the same dataset. The results are presented in table 1.3.

We can see that the precision increased considerably when using the two algorithms together, with only a small decrease in the recall rate. This comes from the multiple

clustering approach, as the consolidation can discard a lot of false positives which commonly appear in only one of the outlier results, but on the other hand there will be a few true positives that are lost in the same way, yielding the observed results.

Other state of the art techniques achieve higher precision rates [12] [13], between 0.7 and 0.8. But considerably slower recall rates, ranging from 0.2 to 0.7. Overall our approach achieve comparable results showing that these outliers can indeed be used for fault prediction.

1.4.2 Scalability and performance

The second dimension is scalability and performance, and in this regard we have several parameters to test in order to show the behavior of the system at high scales and how it impacts existing running applications.

Scalability in terms of number of events

Sometimes the rationalized logs may be updated more or less frequently for various reasons. Because of this we have to test how the system behaves given different numbers of events. We therefore fixed the number of processing nodes at 100, increased the total number of points input for clustering, and measured execution time. As before, Stampede was used for all the runs. We split the time measurement in two parts: a) the insertion part, which measures the time taken to redistribute all the data to DOC processing nodes, and b) the clustering part, which measures the time taken to find all outliers, and report them back to the node responsible for gathering the results. We can see the results in figure 1.6

We note that the clustering time is not a significant part of the total time, only about 9.5%. This was expected because inter-node communication at clustering time is bounded and does not depend too much on the amount of data. The insertion time, on the other hand, is the most significant component, increasing a bit more in percentage, but still maintaining a sub-linear growth. This is expected, since insertion times in DOC have a theoretical bound of $O(\log n)$. Overall the results are good as for our test data from Ranger we never exceed 15k points on a single time bin.

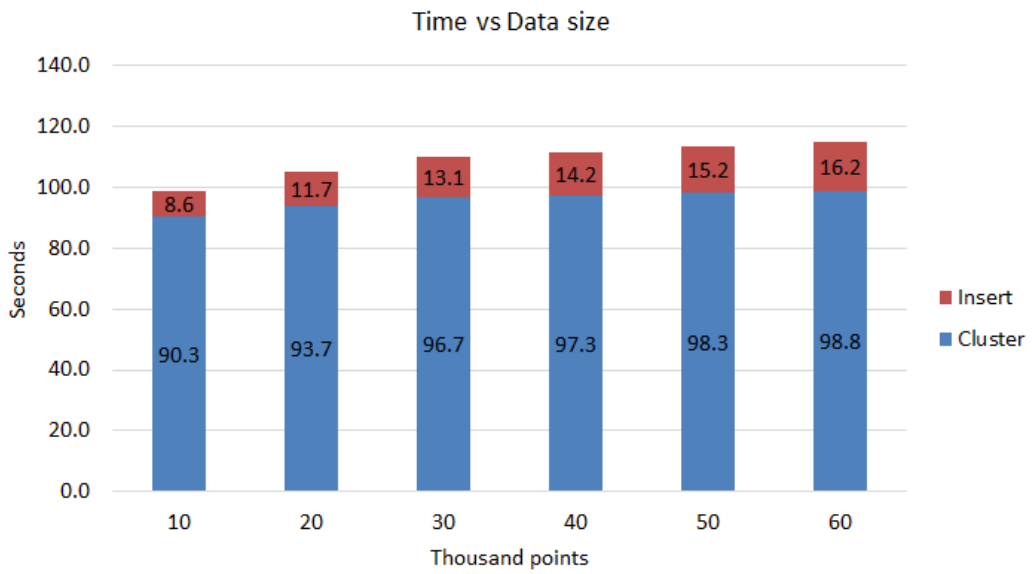

Figure 1.6: Time required to perform a whole run (insertion and clustering), for different number of points.

Scalability in terms of number of nodes

One of the most important metrics to test in order to ensure high scalability is how the system execution time scales with the number of nodes being monitored, which can grow to the order of thousands of nodes in large supercomputer clusters. For this purpose we fixed the number of points to be clustered at 30 thousand (much more than was generated at Ranger during a single time bin), and then we performed different runs increasing the number of monitored nodes (which, in our system setup, are the same as the processing nodes). For this test, we used the Stampede supercomputer at Texas Advanced Computing Center, which enabled us to do tests on smaller scale, results are presented in 1.7. For larger tests we used the Titan supercomputer, located at Oak Ridge National Laboratory; Titan enabled us to test our system using up to 6 thousand nodes. Results are presented in figure 1.8.

We can see that clustering times don't increase much as the number of nodes rises and this is expected, as inter node communication at the clustering phase is bounded, and does not depend on the number of nodes (only neighbors have to be considered).

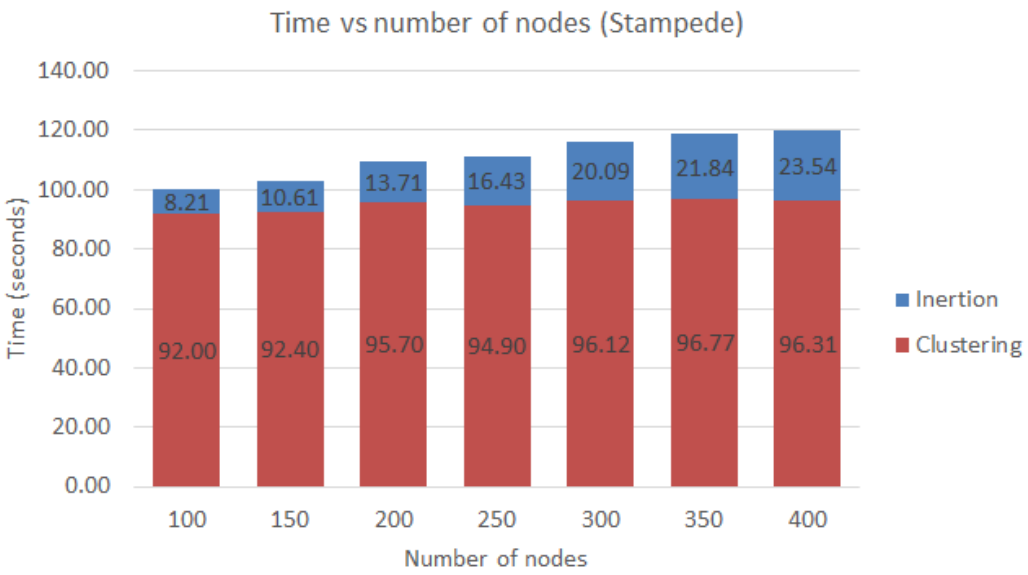

Figure 1.7: Time required for a whole run on Stampede varying the number of nodes. We only use one core per node

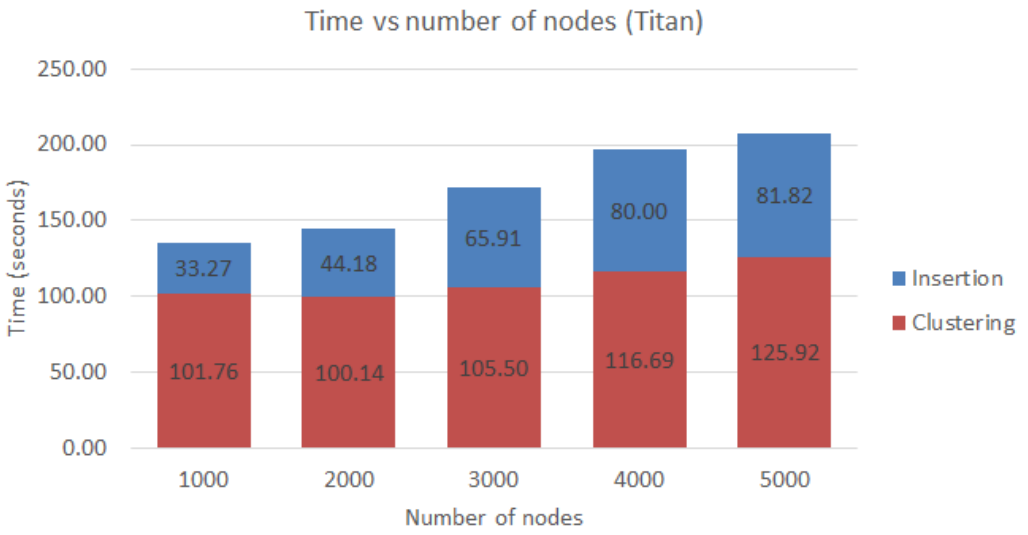

Figure 1.8: Time required for a whole run on Titan varying the number of nodes. We only use one core per node

Insertion times on the other hand is what constrain the overall scalability of the system, as they do depend on the number of nodes. We can observe that for scales less than

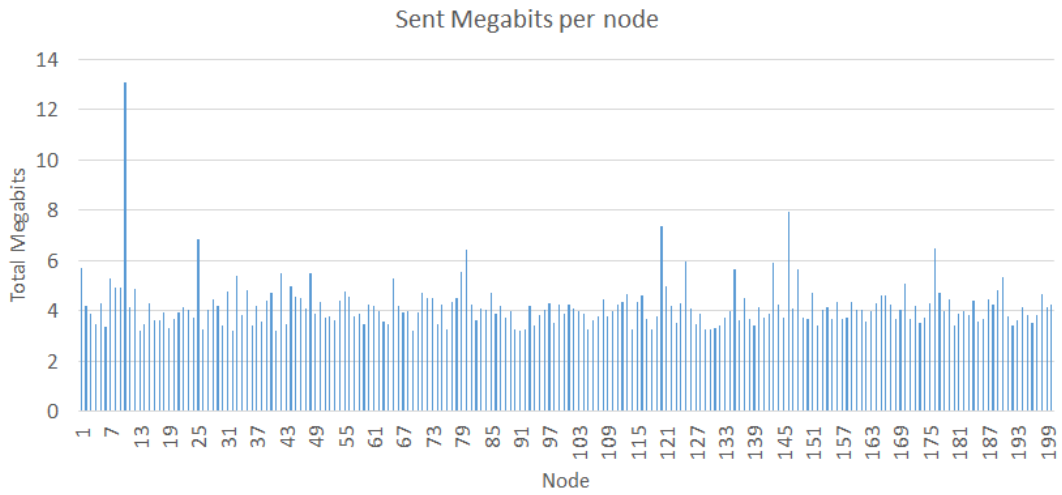

Figure 1.9: Overview of how many megabits where sent by each node during a whole run.

1000 they are almost insignificant in comparison, but once we get to scales of thousands, we see a loss in performance. Still the system is able to perform well and takes just a bit more than 3 minutes for 5k nodes, which is an acceptable number, as the time contain is 5 minutes (which is the time bin size).

Resource usage

In order to show that our system can be used effectively as a monitoring tool, it must use as few resources as possible (i.e. incur low overhead resource utilization). We performed tests to measure the amount of RAM and bandwidth used, given 30 thousand points and 200 nodes. We noticed that RAM usage never exceeded 10Mb; on the bandwidth side, figures 1.9 and 1.10 show both the total sent bandwidth and received bandwidth at each node during one whole run of the algorithm. These metrics were taken directly from the Linux logs, in order to be able to consider every possible message sent. The reason they were measured separately is to ensure that there is a low bandwidth usage under both scenarios, and not just in the average. Figure 1.11 shows that the total bandwidth used by the system is relatively low considering a usual run last between 2 and three minutes.

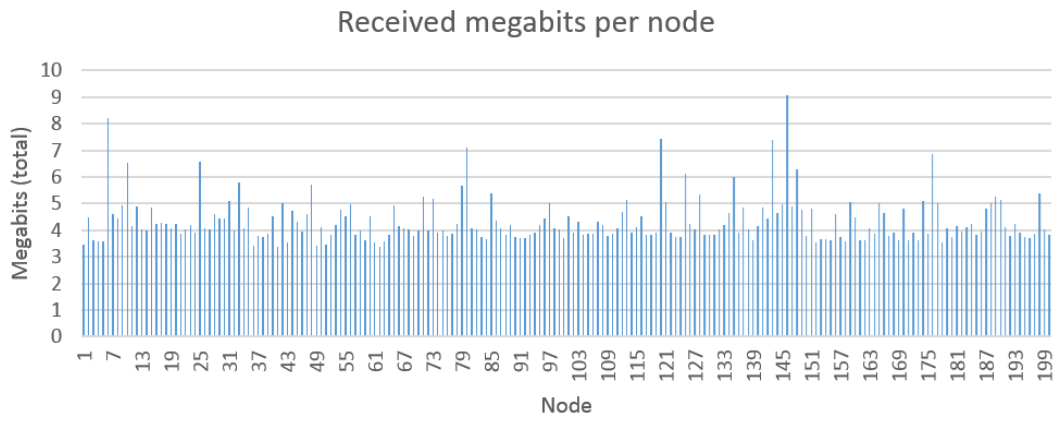

Figure 1.10: Overview of how many megabits where received by each node during a whole run.

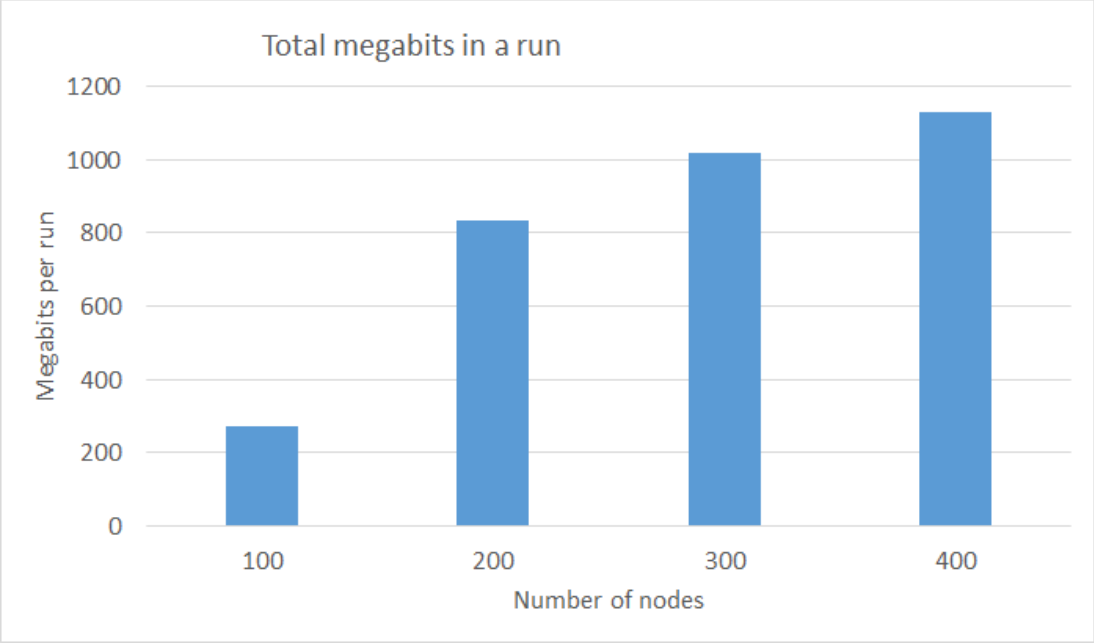

Figure 1.11: Total bandwidth used by the system varying the number of nodes.

Overhead for existing applications

To be able to test how much performance overhead DOC imposes over existing running applications, we used two common benchmarks, namely LINPACK and NPB (NAS Parallel Benchmarks). We timed how long they took to run for different number of nodes, and then we measured how long they took when DOC was running alongside

Nodes	Alone	With DOC	Overhead
100	309.24s	314.3s	1.6%
300	259.36s	263.72s	1.7%
500	253.58s	257.63s	1.6%
700	249.81s	254.27s	1.8%
1000	240.46s	244.41s	1.6%

Table 1.4: LINPACK. Times are in seconds. everything is averaged over 5 runs.

Nodes	Alone	With DOC	Overhead
64	48.1s	49.12s	2.1%
128	32.79s	33.15s	1.1%
256	26.32s	27.11s	3.0%
512	20.23s	20.58s	1.7%
1024	15.69s	16.02s	2.1%

Table 1.5: NPB FFT class D. Times are in seconds. everything is averaged over 5 runs.

them. For LINPACK, the objective was to test mainly the CPU overhead, so we used a problem size of 150k in order to ensure it takes a bit more than DOC so it had time to be influenced by the CPU intensive phase. The results are presented in table 1.4.

We can see that the overhead stayed under 2% and that it was pretty stable, meaning that there are not unexpected peaks in CPU usage and that applications should expect a very stable performance. We then used NPB in order to test the overhead incurred by the bandwidth usage of DOC. For this purpose, we used the FFT test (Fast Fourier Transform), which is highly demanding in terms of overall bandwidth as it uses a many to many communication pattern. For this experiment we chose the class D problem size of the FFT problem that comes bundled with the benchmark, which will make it last more than the main bandwidth intensive phase of DOC. Table 1.5 contains the results.

We can observe that the overhead is on average less than 2%, and the maximum overhead is just 3%. There is one peak, and it may happen when there is a burst on bandwidth usage by DOC, but these don't happen very often, and when they do, they are not big nor last long.

1.5 Conclusions

As supercomputers get bigger, failure prediction starts to become a fundamental tool in order to ensure proper reliability. In this chapter, we proposed DOC as a solution to enable decentralized online monitoring and prediction by means of resource anomaly detection using system logs. We showed that this approach yielded good results, having similar accuracy as other related methods in terms of precision and recall, and even surpassing them in some cases, proving that anomalies found by just using a subset of features of the rationalized logs can indeed yield a good failure predictor. Since the success of any failure prediction system depends on it being lightweight and able to scale up to thousands of nodes, in this regard we performed several experiments and showed that DOC has very little impact on CPU, memory, and bandwidth usage (with around 2% performance overhead, using less than 10MB of RAM, and consuming on average 5MB of bandwidth per node in total). On the scalability side, we demonstrated that DOC is capable of handling at least 60k events using up to 5k nodes, a scale at which typical centralized techniques fail due to the huge communication and performance overheads.

There are still some improvements that need to be addressed, in that we can further exploit the more static nature of these systems (when compared to other more loosely coupled distributed systems) to improve communication speed between nodes and reduce clustering times.

Our long term goal is to use these predictions to trigger automatic action and proactive measures to prevent the imminent failures. This requires further validation of the real predictive power of resource usage anomalies for node failures, and possibly the combined use of additional features in the system logs, leading to improvements in

prediction accuracy. We are also looking at several other machine learning and data mining techniques to help us improve the accuracy of the results by applying some precomputation to the data before analyzing it. This work was only a first step in that direction, but nonetheless can already be considered a valuable tool for monitoring and prediction in large supercomputer systems.

Chapter 2

Dynamic policy adaptation

2.1 Introduction

The second chapter of this thesis focus on high level management. Dynamic policy adaptation provides an efficient and scalable way for controlling large and complex infrastructure (think of supercomputers or large commodity clusters). It is also more convenient, because these dynamic policies could be defined over a higher set of variables where the business goals are easier to identify, and this is clearer for system administrators to define the correct actions. Dynamic policies help in this case because even if the system could not directly perform an action to change higher level variables, it can indirectly affect them by modifying lower level ones, and the dynamic nature of the policy is able to capture the complex relationships between these two set of variables. Also, it helps with the problem of observability, which happens because by the time we could measure the higher level variables, it may already be too late. It will be ideal to apply the policy right before this, and thus we require that dynamic policies predict the behavior of the system in the near future, and adapt accordingly.

2.1.1 Problem definition

Intuitively, a policy specifies some desired state or property of a system (or, conversely, an undesired state or property). Examples of policies are: *The system must respond in x milliseconds*, *The system must guarantee 99% availability*, or *the system must not have more than 25% idle capacity over a one hour period*, etc. Actions, meanwhile, are pieces of code that change the state of the system, and they can be used to make sure that its state satisfies a set of policies as closely as possible. Given a way to infer the right actions for a system to continually meet that goal, it is simple to see how policies

can provide a concise and high-level means for administrators to manage their systems and make sure they run within the desired parameters.

Definitions

The following definitions, which will be used through the rest of the chapter, attempt to expand on the previous ideas and make them more formal in order for us to clearly define our problem and approach for policy-based management.

An *attribute* is any parameter of the system that can be measured. In practice, each attribute only takes values in a bounded subset of the real numbers.

The *attribute space* (which we also refer to as A) is a vector space where each attribute corresponds to one dimension. Note that in practice, because each attribute is bounded, we are only interested in a bounded subset of the attribute space.

We need to distinguish between two sets of attributes:

- **Monitoring attributes:** These are the attributes that can be measured constantly, and more importantly, that are directly controllable. These attributes generate the Monitoring subspace that we denote by M.
- **Business attributes:** These are attributes that clearly relate to the business goals, and whose values are what really matters to system administrators. These attributes generate the Business subspace that we denote by B.

Notice that we don't not necessarily have that M and B contain all of the attributes that are meaningful in the description of a system's state, because there may be others that are not known or that are hard to measure. For example, we typically have a situations where we have some auxiliary applications running on the same machine, they of course influence the performance of the whole system, so we should measure them in order to get a complete system description, but this is not usually the case.

A *system state* is a vector in the attribute space. In other words, it is a particular value assignment of each attribute. If a system is in state S, then $S_{|B}$ denotes the projection onto the business subspace, and $S_{|M}$ the projection onto the monitoring subspace.

A *policy* is a tuple (l, r), where l is a function (called the label function) from B to a set of labels L, and r is another function (called the ranking function) from L to $\mathbb{R}^+$. The purpose of l is to partition the business space into regions. For example if $L = \{\text{good,underused,overused}\}$, the we will partition B into a region considered to be *good*, another considered to be *underused* and another considered to be *overused*. The r function is intended to provide a quantitative value of the desirability of a state. This is because, given the projection of state S onto the business subspace B (i.e $S_{|B}$), we have that $r \circ l(S_{|B})$, represents a numeric value that can be used for comparing against other states. Intuitively, we want desirable states to rank higher.

An *action* is a piece of code that modifies the state of the system. If the system is in state S we denote by S_a the new state of the system after applying the action a. We say that an action a *satisfies* a policy $p = (l, r)$ at state S if we have that

$$r \circ l(S_{a|B}) > r \circ l(S_{|B}),$$

This means that after applying a we obtain a state that ranks strictly higher for the policy. This definition captures the notion of making the actions keep business attributes in more desirable regions of the business subspace, as specified by the policies.

2.1.2 Dynamic policy adaptation problem

The problem under consideration then reduces to being able to compute $S_{a|B}$ for a given state S and action a. This is nontrivial in general, as we usually only know (or can estimate) how a affects $S_{|M}$ (the monitoring attributes), but we do not know how it affects $S_{|B}$ (the business attributes), and can only measure it after the fact. Furthermore, a system may be dynamic within a particular attribute space A, which we have when, for a given action a and state S, $S_{a|B}$ varies over time, which happens when

$$S_{a|B}(t_1) \neq S_{a|B}(t_2), \quad t_1 \neq t_2.$$

This can happen in practice because all meaningful attributes are not included in the attribute space. The problem of dynamic policy adaptation is in fact determining when an action a satisfies a policy p in a dynamic system.

As a starting point to approach this problem, we make two assumptions. The first is that $S_{a|M}$ is static or can be learned apriori; in other words, that we know how actions affect the monitoring attributes. The second is that the change over time in the relationship between the monitoring attributes and the business attributes exhibits a knowable pattern, which can also be learned. For example, it is reasonable to think that, even if the system response time changes over time for a particular measurement of memory, CPU usage, and bandwidth, it still depends on these attributes in a definite way. More formally, this means that there is a family of possibly non-linear functions $G_t : M \to B$, such that if the system is in state S at time t, and we apply an action a, the system transition to state $S_a(t) = \big(S_{a|M}, G_t(S_{a|M})\big)$, and thus $S_{a|B}(t) = G_t(S_{a|M})$. So, the problem of obtaining $S_{a|B}$ is equivalent to the problem of finding or estimating the family of functions G_t.

In practice we cannot find G_t exactly because we usually don't consider (or can measure) all the attributes which determine the family of functions. Also, G_t may be too complex to be able to find or compute effectively. For these two reasons we focus on estimating G_t rather than computing it exactly, and thus we rely on machine learning techniques which are explained in section 2.3.

2.1.3 Real world example

We want to see that the problem of dynamic policy adaptation is a non-trivial one, and that we do have systems in practice which exhibit this variability over time. Remember from before that the main issue was to be able to determine when an action satisfies a policy, and the only thing we did not know how to compute was the resulting state $S_{a|B}$. For this, we made the assumption that we could find it by determining a hidden relationship G_t, which varies over time, between the monitoring and business attributes. Here we will show a system for which this G_t is indeed dynamic, meaning that for the same monitoring state $S_{|M}$, we have two different business states $S_{|B1}$, $S_{|B2}$, and thus G_t changes over time. This is of course because we are not considering the whole possible monitoring space, but only a subspace of it, which is the case in practice.

For this purpose we used RabbitMQ as our system to be analyzed. We chose this

one as it is a basis for a lot of applications, and is one of the most widely used queuing systems. In this scenario, we used RabbitMQ to implement a master/worker workflow that solves simple linear algebra problems.

Our *business attributes* would be the cost (calculated based on the number and types of machines running), and the *message send/delivery time* of the system.

The monitoring attributes include *send rate, deliver rate, memory usage, messages acknowledged, messages pending, messages delivered*, etc. For this we focused mainly on the memory usage and message delivered attributes.

We could have the following possible set of labels $L = \{\text{normal, overused, underused}\}$ (in a more real scenario, we would have different type of overused and underused labels, in order to deal with unexpected peaks in the system behavior). We aim to keep the system running under the normal label. The overused state is meant to capture scenarios where the system is not being able to handle all the traffic and response times are getting affected. The underused state is meant to capture those situations where the system is overprovisioned, and we could probably take down resource without compromising the response times. The normal state is everything else. With this in mind we could define a ranking function that gives the highest ranking to the normal state, and actions will trigger when transitioning outside of this state. So for example if we go from normal to overused, we could have an action that increases the number of machines in the RabbitMQ cluster.

Our RabbitMQ cluster was formed by 4 amazon EC2 instances. Our simulation consisted of 4 master instances generating tasks, and 8 worker instances consuming them. The cost remained constant as we did not added any new node to the cluster.

We run the same exact simulation 4 times without making any configuration changes to the cluster, all variations will be due to external influences such as network delays, virtualization overhead, etc. Our hypothesis is that these outside and unmeasurable influences are the main causes of the dynamism in the system.

Figure 2.1 and Figure 2.2 show the behavior of our business attributes, the cost is not shown as it was constant during the simulation (i.e we used the exact same machine configuration).

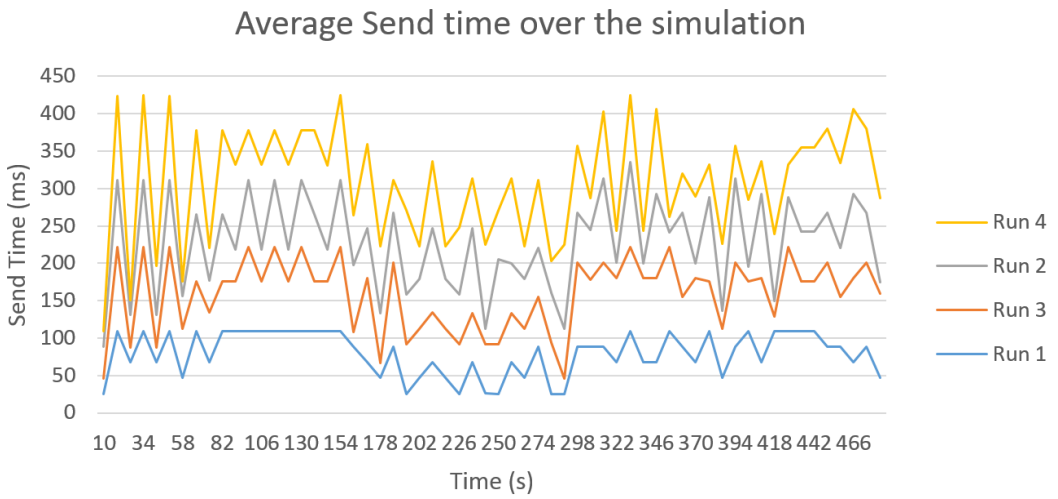

Figure 2.1: Average time spent by the masters to send work over the whole simulation.

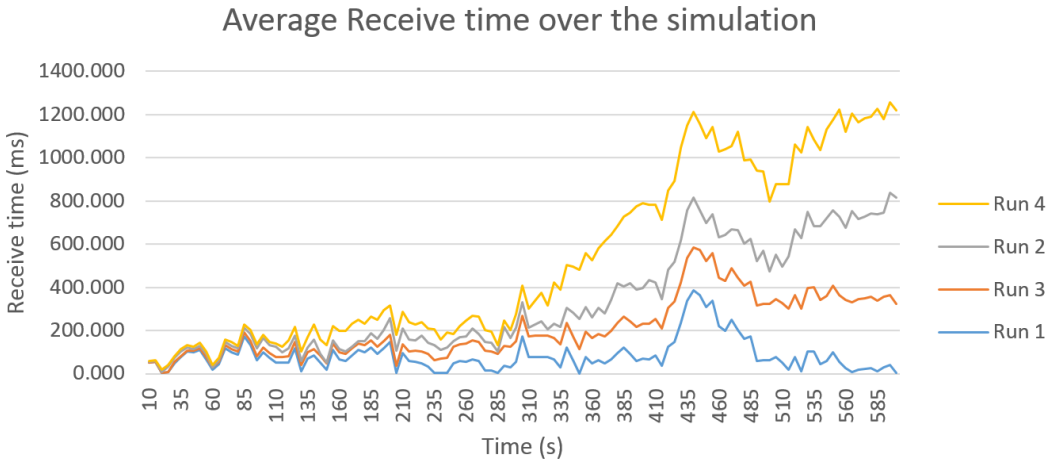

Figure 2.2: Average time a job remained in the queue until it was delivered to a worker over the course of the simulation.

Figure 2.3 shows a plot of the average memory usage of the 4 nodes in the cluster, we see that for the most part there seems to be a linear dependency between receive times (Figure 2.2) and memory (i.e if one increase the other one does also), even the spikes seem to align, but it all breaks at the end, where we see the opposite trend.

Figures 2.1, 2.2 and 2.3 show the problem of trying to compute the dependency G_t directly. Focus on Run 3 at times 230s and 370s, clearly we have the same memory (Figure 2.3) at those two places (with the value around 250mb), but going back to

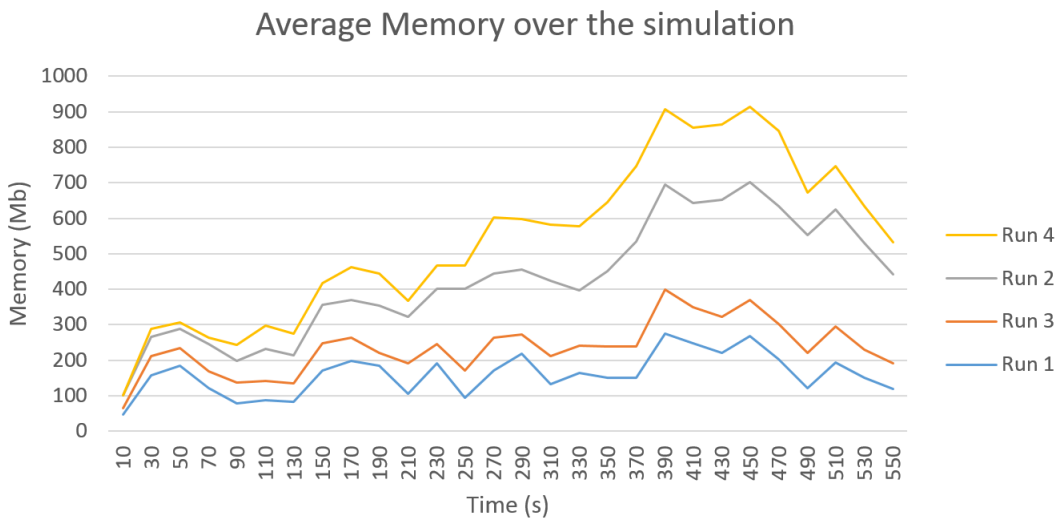

Figure 2.3: Average memory used by the RabbitMQ nodes over the simulation.

Figures 2.1 and 2.2, we observe that at time 230s we have an average receive time of 100ms and an average send time of also 100ms, but at time 370s we measured an average receive time of 220ms and an average send time of 160ms. Drastically different.

Let us consider another monitoring attribute, maybe with the hopes of being able to figure the relationships ourselves. Let us look at the average deliver rate, we see that at times 230s and 370s, we measured similar deliver rates, and thus even considering the pairs (memory,deliver rate) we still have a dynamic G_t. Also, notice that for example in some simulations (Run 2 and 4), we have that the average receive time had a rising tendency all the time, but clearly the memory and deliver rate went down at the end. This suggest that G_t has to be nonlinear in order to capture these type of dependencies, and thus is not easy find by a human.

These empirical observations, the dynamic behavior of G_t and its apparent complexity, shows that even for simple master worker workflows that implement basic algorithms exhibit dynamic behavior, and thus they present the problem of dynamic policy adaptation. Notice, however, that the runs in each plot do follow each other in a similar pattern, they rise and fall together, and tend to form peaks at similar points in time. This suggests the validity of the assumption that the dynamic behavior can be learned.

Finally it is important to note that there may be *business attributes* which are easy

to predict, and for which we know how actions affect them, an example of this is the *cost*. But more often than not, we have the complex relationships shown above.

2.2 Related Work

The idea of separating the policies that govern the behavior of a system, from the actual functionality of the system or the mechanisms that enforces them, was proposed by Sloman et al. [22]. This of course requires the ability to be able to adapt policies to a system at runtime, and without a dynamic approach the behaviors that can be managed are too restricted. Some of the research on policy based management have been geared on how to specify them using description languages, some focus on simple definitions [23], and others on automatically detecting conflicts between different policies [24]. Although very helpful in allowing people to easily express policies, they still suffer from not being able to capture the dynamic system behavior and thus have restricted applicability.

In a research agenda for policy management, [25] Dam et al. identified the importance of being able to recompute low level policies due to changes in the system state. And in fact, this has been dealt with before by Moffett and Sloman using policy hierarchies [26]. However, their approach failed to provide automatic and dynamic mechanisms for building the low level policies from top levels.

A different approach to this problem, which employs utility functions, has been presented in [27] and [28]. The main idea is to define the desired goals using some cost functions, and the optimal state is then found by solving an optimization problem whose output specifies the actions that need to be taken. This approach is radically different from ours as it does not apply incremental actions in order to achieve the desired state; rather, it determines the optimal scenario and applies all relevant actions. The problem with these approaches is that they may lead to non-smooth changes of the state of the system over time, because two different solutions to the utility functions may actually differ drastically. Depending on the cost of applying some policies, this may be undesired. A reactive approach, like ours, in which actions are taken due to

observed changes in the system, does not have this problem. This is because only one change is made at a time, resulting in a smooth evolution of the state of the system.

Another different perspective is given by Aly and Lutfiyya, who use a control theoretic approach [29]. They proposed to use feedback in the system in order to tweak the threshold values, but again, this requires low level policy specifications, and the thresholds are necessarily linear, which may not cover all cases.

Using machine learning for dynamic policy adaptation is not new. An early approach was proposed by Kephart and Walsh using intelligent agents [30], wherein they identify 3 type of policies: Action policies which are basically static, goal policies where a desired state is specified and intelligent agents decide the actions to lead the system to that state, and utility function policies where the agents try to maximize the specified function. Their first approach wont work on a dynamic system, and their third approach is similar to the utility function approaches described before. Their goal policies are closer to what we are trying to achieve, but in our case the *goals* are implicitly coded in the high level policy specification, giving us a little bit more flexibility regarding what states our system can be in, which may be desired and cheaper.

Finally, a similar approach to the one presented in this thesis is given by Quiroz et al. in [31]. In fact we use a similar formulation of representing the system attributes as a vector space, and arguing there is a relationship between some dimensions called monitoring attributes to the other dimensions called business attributes. In their approach, they try to find this relationship using only the recent monitoring data of the system, and thus must rely on unsupervised algorithms, in their case clustering, to be able to capture the dynamic behavior of the system. That is, they cannot have a fixed pre-trained model. We improve upon this by using supervised models that actually look at the short-term evolution of the system, and capture the dynamic behavior from there. The advantage of our approach is that we would be able to use faster and more powerful algorithms such as support vector machines and neural networks.

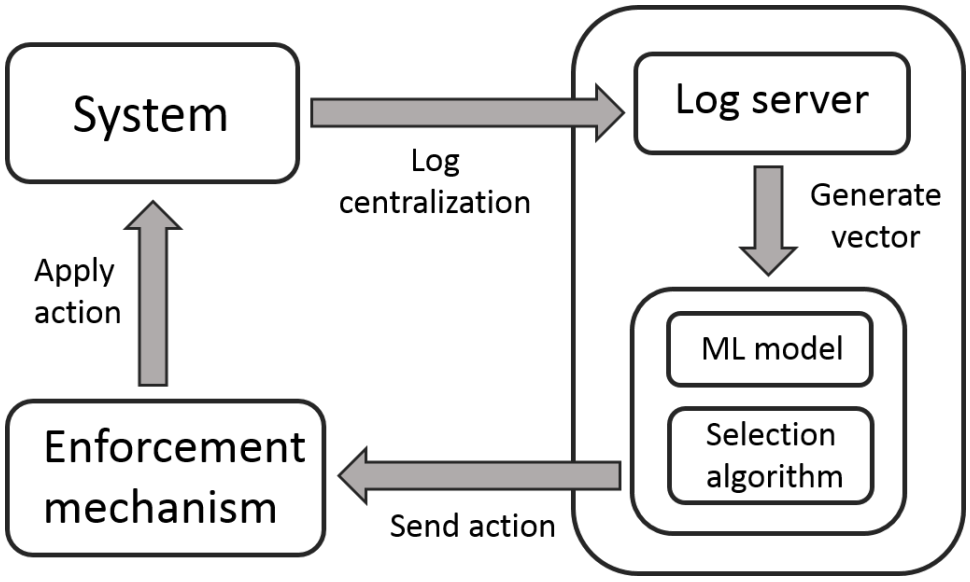

Figure 2.4: High level overview of the policy adaptation and enforcement system.

2.3 Estimation via machine learning

2.3.1 System Architecture

First, we will work with the assumption that the systems to be monitored have a log centralizer mechanism, and that we can deploy our code in that central system. We also require that the mechanisms that enforce policies are completely decoupled from the ones that decide which policies to apply, in order to be able to run both of them on different machines.

Figure 2.4 shows the architecture we are looking into for the case in which the monitoring logs are being centralized in one server (which is the case for most large-scale clusters). In it, a log server aggregates the logs from the whole system, and the system transforms 30 seconds worth of logs into a vector that is fed into our model in order to predict the current state of the system, as well as the possible states our system will transition to for each of the actions. Given that information, the selection algorithm chooses the best action to take in order to satisfy the policies, which is then sent to the enforcement mechanism (possibly outside our system) in order to modify the system.

In the next sections we explain the regression (or ML) model, how to generate

vectors to feed this model,, and how the selection algorithm works.

2.3.2 Regression model

As we mentioned in section 2.1.2, our problem can be solved if we are able to estimate the family of functions G_t which allow us to obtain $S_{a|B}$ for a state s and an action a. Clearly we cannot base our algorithm only on a specific point int time, that is we cannot use only $S_{a|M}$, the state of the monitoring attributes after applying the action, or even S itself, the state of the system before performing the action. The reason for this is that if we find an algorithm that makes a prediction based on these two states at that specific point in time, then it will make the same prediction every time it encounters the same configuration, and thus we would not be effectively estimating the whole family G_t, but only one of its members.

In order to solve this issue, we will use a well-known idea called sliding windows [32] [33]. Let's assume we are at state $S(t)$ and we want to predict $G_t\left(S_{a|M}(t)\right)$, which gives the value of the business attributes. Instead of just using $S(t)$ and $S_{a|M}(t)$, we consider the vector

$$V_t = \left(S(t-k), S(t-k+1), \ldots, S(t), S_{a|M}(t)\right),$$

Where $S(t)$ stands for the state of the system at time t and k is a constant called the window size. Now we estimate another function f, which is independent of time, with the property that $G_t\left(S_{a|M}(t)\right) = f(V_t)$. The intuitive idea that justifies this equivalence is that the time window provides enough context to differentiate the cases in which $S(t)$ is equal at different times, even if we don't know the additional attributes to obtain that differentiation. If that were not the case, then all of the functions in G_t would be equivalent, and thus our assumption that there is not a single function G would be incorrect.

In order to do this, we will try several methods which have been shown to give good results for different types of sequential and time series data [34], [35] and [33]. The models we will try are Support Vector regression, Ridge regression, Decision trees regression, nearest neighbor regression and neural networks.

2.3.3 Transforming data

Assume we have monitored the system for enough time that we have been able to record multiple measurements of both the *monitoring attributes* and the *business attributes*. These measurements give us a sequence of states $S(0), S(1), \ldots, S(N)$. In total we can build $N - k$ vectors V_t, as described above by the sliding windows technique (we can start at any of the first $N - k$ positions). Also, for each of these vectors V_i, we know exactly the value of $f(V_i)$, which is just the projection of the business attributes of the last state (i.e $S(i+k)_{|B}$). If we monitor the system some more, we can obtain a different sequence of states, and produce more vectors with their respective values of f. In this way, we can extract training data, with their respective target values, in order to feed the learning algorithms.

2.3.4 Selection Algorithm

The algorithm for choosing which action to apply works like this. Given the current state of the monitoring attributes $S_{|M}$, we estimate the current state of the business attributes $S_{|B}$ using the regression model. By our assumption, we know how actions affect monitoring attributes, and thus for each action a we can compute the resulting monitoring state $S_{a|M}$, having this we can estimate $S_{a|B}$ again using the regression model. Now, consider the set of policies as tuples (l_p, r_p), we can compute the set of all actions A that satisfy at least one policy (l_p, r_p), then we pick some $a \in A$, and that will be the action we need to apply. The pseudocode for this is presented in Figure 2.5.

Notice that at the end we just return a random action from all the possible ones. This often gives good results, but there could be scenarios where you have to policies that conflict each other. We did not address this issue in this research.

2.4 Experiments

2.4.1 Setup

We will use the same RabbitMQ setup and simulation described in section 2.1.3. Our business attributes remain the same, that is, *average message receive time, average*

```
function selectAction(State S, List of actions A, list of policies P, model m)
    candidates = [ ]
    for a ∈ A do
        for (l_p, r_p) ∈ P do
            oldRank = r_p ∘ l_p (m.estimate(S_|M))
            newMonitoringState = a.futureState(S)
            newRank = r_p ∘ l_p (m.estimate(newMonitoringState))
            if newRank > oldRank then
                candidates.insert(a)
            end if
        end for
    end for
    chosenAction = candidates.pickRandom()
    return chosenAction
end function
```

Figure 2.5: Policy selection algorithm using the machine learning regression model.

message send time and *cost*. Our monitoring attributes consist of the whole output of the monitoring plugin for RabbitMQ, which includes parameters like *memory, pending messages, delivered messages, arrival rate, deliver rate*, etc. We have 32 attributes in total.

In order to capture enough of the trends that allow us to predict the dynamic behavior of the system, we need to run the simulation changing some parameters. These include the number of nodes in the RabbitMQ cluster, which we changed to 2, 3 and 4. The number of workers, which we varied from 4, 8, 12 and 16. The number of masters, which were varied from 2, 4, 6 and 8. Also, in order to be able to capture unexpected behavior, we performed some runs where we started with some workers, and suddenly killed several during the simulation. The same for masters and nodes in the cluster.

We collected data every second, and a typical simulation lasted for about 6 minutes,

giving around 360 states, and after aggregating all the runs together, we ended up with about 35000 points.

From here we created the actual training data using the sliding window method. Notice that the total number of points available to train depends on the window size k. One final thing we need to address, these models have parameters that we need to choose, and the windows size also needs to be given a near optimal value. In order to do this we are going to use a standard 10-fold cross validation technique.

2.4.2 Evaluation

In order for us to compare different algorithms, parameter configurations, and to be able to have a notion of how well they are performing, we will use the square root of the mean square error, defined for a particular output variable as

$$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{n} \|\hat{y}_i - y_i\|}{n}}$$

Where $\hat{y}_i$ is our estimate of the output variable, or in other words it is $f(V_i)$, and y_i is the true value. For testing purposes we will use the RMSE by looking only at the message receive time and message send time. This is because they have the same units (which are seconds), and thus it will make the interpretation of the RMSE easier, as we can give units to it (i.e seconds).

Also, a particularly good property of the RMSE is that by Chebyshev we know that for any random variable X with finite mean and nonzero variance, and any positive number k, we have that

$$\mathbb{P}(|X - \mu| > k\sigma) \leq \frac{1}{k^2}$$

And because our RMSE is equal to the standard deviation in the case of unbiased estimators, we approximately get that for an RMSE of x seconds, we would have an error of kx seconds with probability at least $1 - 1/k^2$. For example for an RMSE of 1, would produce an error of 4 seconds with probability 0.9375. We can use this not only to compare algorithms against each other, but also to determine if a given result is good or bad.

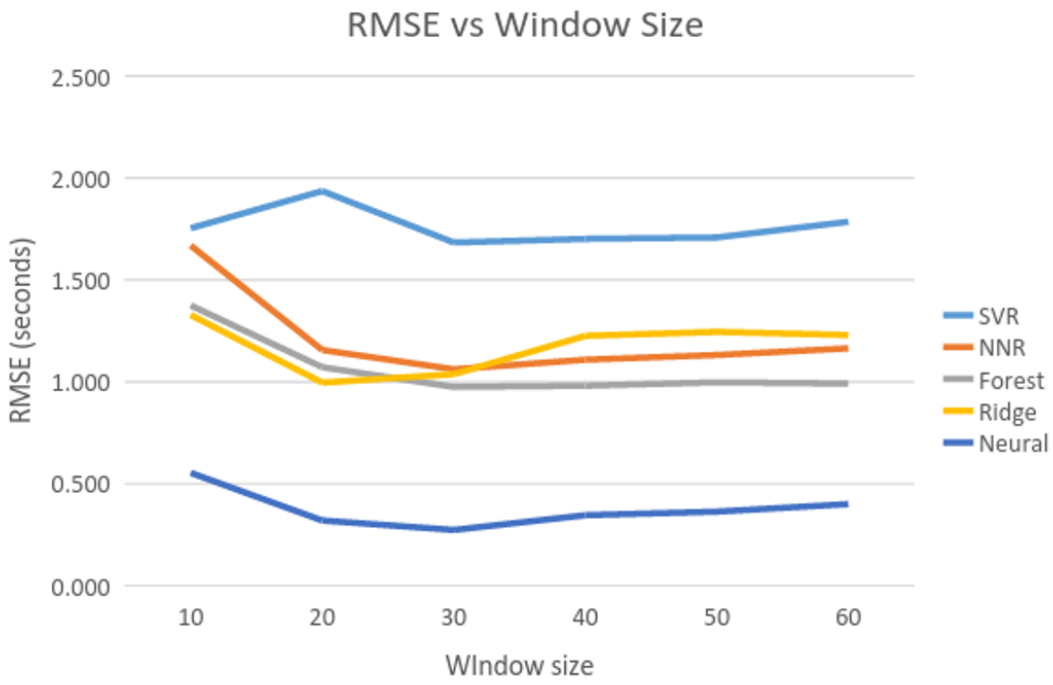

Figure 2.6: Result of performing cross validation for different window sizes and different algorithms.

2.4.3 Results

Window Size

Figure 2.6 shows the result of performing cross validation for different window sizes for each of the algorithms. We can see in most cases, they performed better around $K = 30$, and that neural networks outperformed the rest. Also by looking at the nearest neighbors, random forests and ridge regression, we can see that the choice of K can make one algorithm better than the other.

This tells us that we need to look around 30 seconds into the past in order to fully capture the current trend of the system. We would expect that considering values further into the past should make the algorithms behave even better, but the problem is that the dimensionality of the problem increases linearly with the time window K, and thus in these scenarios, the models are much more complex, so they need much more data in order to perform well.

Even without fine tuning parameters we already see very good results for our simple

Algorithm	**RMSE**
SVR	1.18
NNR	1.00
RFR	0.97
Ridge	0.97
Neural	0.21

Table 2.1: RMSE after fine tuning the parameters for the different algorithms.

neural network, getting a RMSE as low as 0.273s, which considering that the average receive time was 11.96 seconds, it means that we produce an average error (using Chebyshev with $k = 5$) of at most 11.41%, 96% of the time, which looks promising.

2.4.4 Fine Tuning

Table 2.1 contains the results for the different algorithms after performing parameter selection using cross validation. The best results were obtain again by the nueral network, and it got an RMSE of 0.21 seconds which gives an error of at most 8.78% at least 96% of the time

The following are the optimal parameters that we found using cross validation.

- **Support Vector regression (SVR):** A Gaussian kernel with a radius of influence (gamma parameter) of 0.01 and a degree of 3.
- **Nearest neighbor regression (NNR):** The optimal was achieved by looking at the 25 nearest neighbors.
- **Random forest regression (RFR):** We found the best results when using 100 trees and requiring at least 5 training samples per leaf.
- **Ridge Regression (Ridge):** A regularization (or alpha) parameter of 2 gave the best results.

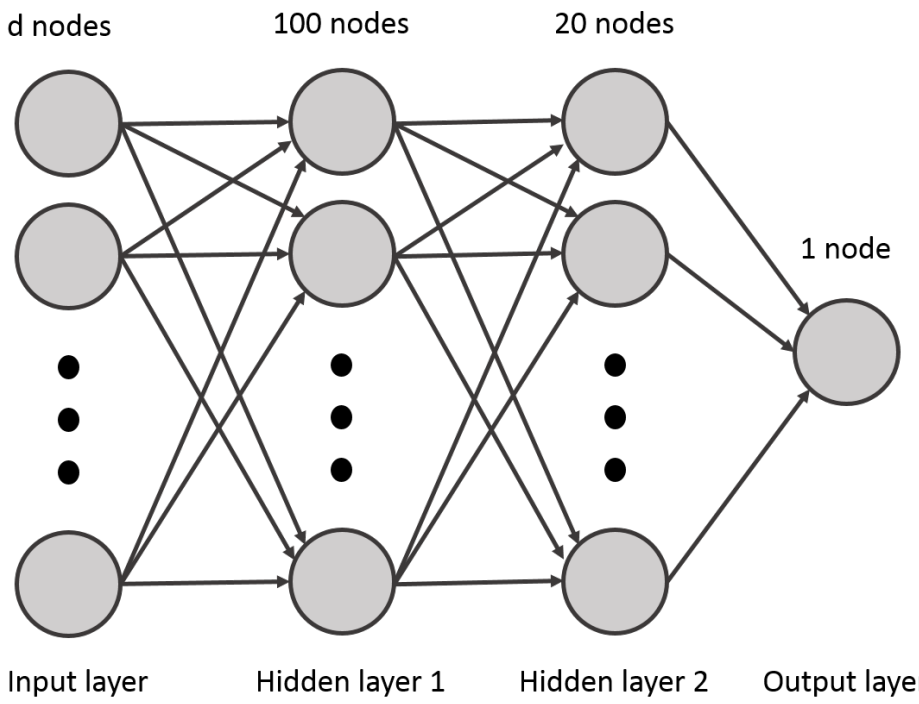

Figure 2.7: Best Neural network architecture. d is the number of dimensions.

- **Neural Network (Neural):** The neural networks shown in Figure 2.7 is the one that gave the best results for us.

2.4.5 Generalization

Because it is unfeasible to train the models with data that contains every single possible system configuration, we awant to know how our model performs with unseen behavior. For this purpose we gathered data from our system using a completely different set of parameters. Table 2.2 shows the results of our algorithms on this dataset, compared to the old results.

Although the RMSE is higher when presented with unseen configurations, it is still a very good result, producing an error of at most 12.79% at least 96% of the time.

Dataset	RMSE
Old	0.210
Unseen	0.306

Table 2.2: RMSE under unseen circumstances.

2.5 Future work

As Sloman suggested in his seminal paper on policy driven management [22], adaptation and enforcement mechanisms of policies are two different areas that should be logically separated in order to simplify the design and implementation of the management software. In this thesis we focused on the former, that is policy adaptation. We proposed a machine learning regression model that is able to estimate the near future state of the system, and thus the action to be applied can be chosen by considering the predicted state as the real one.

However in order to fully build a dynamic policy management software, we require the enforcement mechanisms to be present, and our current work lies in this area. There are several technical challenges that need to be addressed in order to achieve this. First, we need to fully implement and deploy our proposed architecture for the system, also because actions make take some time before they have an impact on the state of the system, we need to determine how often to apply them in order to avoid flooding the system with configuration changes, which may actually harm the performance. Finally we need to extend our algorithm for choosing the correct action to apply based on our regression model, the main issue arises when we have conflicting policies, in which case we need to build some resolution algorithm to deal with this.

Regarding how often to apply actions, we have two possible solutions. First we may require the system administrators to specify timeouts along with the actions, this will work by making sure no other action is applied during the timeout period, and thus giving the system enough time to react to the change. We are also looking at a more dynamic approach that leverages our abstract policy definition, in which we require a

label function l. Remember that the idea of this labeling is to partition the system state into a set L of regions, such as $L = \{\text{good,underused,overused}\}$, and now, an action will only be applied whenever we observe a transition between regions. This will effectively avoid the flooding problem as once an action is applied we won't act anymore until we observe another change in regions, which will happen after the action has made an impact on the system. Both approaches have their benefits and drawbacks, and we are currently looking at which one gives the best results, or even have a combination of both.

With respect to the conflict resolution algorithm, recall the algorithm shown at 2.5. In the last step, we may have several candidates to choose our finally action from, but what if some of them are conflicting or incompatible. To solve this, we need to develop a conflict resolution algorithm, and we have several choices on how to proceed here. First we can do it randomly, and in this case we have to prove that we can achieve a near optimal solution with high probability. We are also looking at some approaches made by others such as Lupu and Sloman [36], where they propose to add a precedence relationship between actions. In [37] Davy et al. shows an analytic tool that is able to detect the conflicts at the moment of specifying the policies, and we are looking into adapting that technique to our specification method. Finally in [38] Rusello et al. present an automatic conflict resolution approach based on a on the ponder policy specification language [23], we want to see if we can modify their ideas so they work with our approach.

Finally, there are some optimization that could help with the speed and accuracy that we want to look into. First, we may be able to somehow discard most of the actions based on a heuristic, as most of them are clearly not applicable to some states; this will significantly reduce the number of predictions we need to do, and thus improving the performance. Also, the system may have permanent changes over time, and thus the machine learning algorithm must be retrained in order to fully capture this scenario. We are looking both at online version of the algorithms we used, or retrain mechanisms in order to solve this issue.

2.6 Conclusions

Dynamic policies offer an easy way of managing large scale systems. We have presented an approach for dynamic policy adaptation that, unlike static solutions, allows administrators to define their policies in terms of high level goals and lets the system determine which actions to apply (and when to apply them) in order to guarantee that those goals are met.

We achieve this by making the assumption that there is some state that can be measured constantly, called the monitoring space, and that there is a learnable relationship that varies in time between this space and the high level goals, also called the business space. With this we model the problem as a regression problem and solve it using machine learning techniques by learning to determine the unknown state using the known short term evolution of the system.

We showed that this relationship can indeed be learned by different algorithms, and also observed that tracking the evolution of the systems for the last 30 seconds is optimal. We concluded that neural networks achieve the best performance with an RMSE of 0.21s, which represents an error of at most 8.78% at least 96% of the time. We also show that the model was robust enough to deal with completely unseen behavior and was able to generalize well, getting a RMSE of 0.306s with new data, which represents an error of 12.79% at least 96% of the time.

Finally, this work is more focused on the theoretical part of how to dynamically adapt policies, and it was out of scope to implement and evaluate the actual mechanisms that enforce the actions. Nevertheless we outline one possible software architecture that could use this regression model in order to enforce the policies and make sure a system stays within the parameters specified in the high level policies. We are currently working on building such software, and looking at possible mechanisms to automatically retrain the regression model in order to better deal with unexpected harsh changes in the behavior of the system and looking on how to solve the situations where conflicts arise between policies.

References

[1] A. Pelaez, A. Quiroz, J. Browne, E. Chuah, and M. Parashar, "Online failure prediction for hpc resources using decentralized clustering," in *High Performance Computing (HiPC), 2014 21st International Conference on*, Dec 2014, pp. 1–9.

[2] A. Quiroz, M. Parashar, N. Gnanasambandam, and N. Sharma, "Clustering analysis for the management of self-monitoring device networks," in *Autonomic Computing, 2008. ICAC '08. International Conference on*, June 2008, pp. 55–64.

[3] D. A. Reed, C.-d. Lu, and C. L. Mendes, "Reliability challenges in large systems," *Future Gener. Comput. Syst.*, vol. 22, no. 3, pp. 293–302, Feb. 2006.

[4] B. Schroeder and G. Gibson, "A large-scale study of failures in high-performance computing systems," *Dependable and Secure Computing, IEEE Transactions on*, vol. 7, no. 4, pp. 337–350, Oct 2010.

[5] U. Lerner, R. Parr, D. Koller, and G. Biswas, "Bayesian fault detection and diagnosis in dynamic systems," in *In Proc. AAAI*, 2000, pp. 531–537.

[6] J. Gertler, *Fault Detection and Diagnosis in Engineering Systems*, 1998.

[7] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "Sober: Statistical model-based bug localization," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 286–295, Sep. 2005.

[8] E. Chuah, A. Jhumka, S. Narasimhamurthy, J. Hammond, J. C. Browne, and B. Barth, "Linking resource usage anomalies with system failures from cluster log data," in *Proceedings of the 2013 IEEE 32Nd International Symposium on Reliable Distributed Systems*, ser. SRDS '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 111–120.

[9] A. Quiroz, M. Parashar, N. Gnanasambandam, and N. Sharma, "Design and evaluation of decentralized online clustering," *ACM Trans. Auton. Adapt. Syst.*, vol. 7, no. 3, pp. 34:1–34:31, Oct. 2012. [Online]. Available: http://doi.acm.org/10.1145/2348832.2348837

[10] A. Quiroz, M. Parashar, and I. Rodero, "Autonomic management of distributed systems using online clustering," in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, April 2010, pp. 1–4.

[11] J. L. Hammond, T. Minyard, and J. Browne, "End-to-end framework for fault management for open source clusters: Ranger," in *Proceedings of the 2010 TeraGrid Conference*, ser. TG '10. New York, NY, USA: ACM, 2010, pp. 9:1–9:6.

[12] X. Fu, R. Ren, J. Zhan, W. Zhou, Z. Jia, and G. Lu, "Logmaster: Mining event correlations in logs of large-scale cluster systems," in *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on*, Oct 2012, pp. 71–80.

[13] Z. Lan, J. Gu, Z. Zheng, R. Thakur, and S. Coghlan, "A study of dynamic meta-learning for failure prediction in large-scale systems," *Journal of Parallel and Distributed Computing*, vol. 70, no. 6, pp. 630 – 643, 2010.

[14] Y. Zhang and A. Sivasubramaniam, "Failure prediction in ibm bluegene/l event logs," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, April 2008, pp. 1–5.

[15] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam, "Critical event prediction for proactive management in large-scale computer clusters," in *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '03. New York, NY, USA: ACM, 2003, pp. 426–435. [Online]. Available: http://doi.acm.org/10.1145/956750.956799

[16] F. Salfner, M. Schieschke, and M. Malek, "Predicting failures of computer systems: a case study for a telecommunication system," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, April 2006, pp. 8 pp.–.

[17] M. Ester, H. peter Kriegel, J. S, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise." AAAI Press, 1996, pp. 226–231.

[18] Slurm: A highly scalable resource manager. Lawrence Livermore National Laboratory. [Online]. Available: https://computing.llnl.gov/linux/slurm/

[19] Torque resource manager. Adaptive Computing. [Online]. Available: http://www.adaptivecomputing.com/products/open-source/torque/

[20] R. McLay. Lariat. [Online]. Available: https://github.com/TACC/Lariat

[21] C. Callegari, L. Gazzarrini, S. Giordano, M. Pagano, and T. Pepe, "A novel pca-based network anomaly detection," in *Communications (ICC), 2011 IEEE International Conference on*, June 2011, pp. 1–5.

[22] M. Sloman, "Policy driven management for distributed systems," *Journal of Network and Systems Management*, vol. 2, pp. 333–360, 1994.

[23] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The ponder policy specification language," in *LECTURE NOTES IN COMPUTER SCIENCE*. Springer-Verlag, 2001, pp. 18–38.

[24] A. Bandara, E. Lupu, and A. Russo, "Using event calculus to formalise policy specification and analysis," in *Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on*, June 2003, pp. 26–39.

[25] M. Dam, G. Karlsson, B. S. Firozabadi, and R. Stadler, "A research agenda for distributed policy-based management," in *nd SEAS DTC Technical Conference - Edinburgh 2007 C1*.

[26] J. Moffett and M. Sloman, "Policy hierarchies for distributed systems management," *Selected Areas in Communications, IEEE Journal on*, vol. 11, no. 9, pp. 1404–1414, Dec 1993.

[27] D. Chess, A. Segal, I. Whalley, and S. White, "Unity: experiences with a prototype autonomic computing system," in *Autonomic Computing, 2004. Proceedings. International Conference on*, May 2004, pp. 140–147.

[28] J. Kephart and R. Das, "Achieving self-management via utility functions," *Internet Computing, IEEE*, vol. 11, no. 1, pp. 40–48, Jan 2007.

[29] W. Aly and H. Lutfiyya, "Dynamic adaptation of policies in data center management," in *Policies for Distributed Systems and Networks, 2007. POLICY '07. Eighth IEEE International Workshop on*, June 2007, pp. 266–272.

[30] J. Kephart and W. Walsh, "An artificial intelligence perspective on autonomic computing policies," in *Policies for Distributed Systems and Networks, 2004. POLICY 2004. Proceedings. Fifth IEEE International Workshop on*, June 2004, pp. 3–12.

[31] A. Quiroz, M. Parashar, N. Gnanasambandam, and N. Sharma, "Autonomic policy adaptation using decentralized online clustering," in *Proceedings of the 7th International Conference on Autonomic Computing*, ser. ICAC '10. New York, NY, USA: ACM, 2010, pp. 151–160. [Online]. Available: http://doi.acm.org/10.1145/1809049.1809074

[32] S. Yoshida, K. Hatano, E. Takimoto, and M. Takeda, "Adaptive online prediction using weighted windows," *IEICE Transactions*, vol. 94-D, no. 10, pp. 1917–1923, 2011.

[33] T. Dietterich, "Machine learning for sequential data: A review," in *Structural, Syntactic, and Statistical Pattern Recognition*, ser. Lecture Notes in Computer Science, T. Caelli, A. Amin, R. Duin, D. de Ridder, and M. Kamel, Eds. Springer Berlin Heidelberg, 2002, vol. 2396, pp. 15–30.

[34] N. K. Ahmed, A. F. Atiya, N. E. Gayar, and H. El-shishiny, "An empirical comparison of machine learning models for time series forecasting."

[35] G. Bontempi, S. Ben Taieb, and Y.-A. Le Borgne, "Machine learning strategies for time series forecasting," in *Business Intelligence*, ser. Lecture Notes in Business Information Processing, M.-A. Aufaure and E. Zimnyi, Eds. Springer Berlin Heidelberg, 2013, vol. 138, pp. 62–77.

[36] E. Lupu and M. Sloman, "Conflicts in policy-based distributed systems management," *Software Engineering, IEEE Transactions on*, vol. 25, no. 6, pp. 852–869, Nov 1999.

[37] S. Davy, B. Jennings, and J. Strassner, "Efficient policy conflict analysis for autonomic network management," in *Engineering of Autonomic and Autonomous Systems, 2008. EASE 2008. Fifth IEEE Workshop on*, March 2008, pp. 16–24.

[38] G. Russello, C. Dong, and N. Dulay, "Authorisation and conflict resolution for hierarchical domains," in *Policies for Distributed Systems and Networks, 2007. POLICY '07. Eighth IEEE International Workshop on*, June 2007, pp. 201–210.