Resource Constrained Failure Management in Networked Computing Systems

Praveen Bommannavar
Management Science and Engineering
Stanford University
Stanford, CA 94305 USA
bommanna@stanford.edu

Nicholas Bambos
Electrical Engineering and
Management Science and Engineering
Stanford University
Stanford, CA 94305 USA
bambos@stanford.edu

***Abstract*—We examine the problem of fault detection in networked computing systems and highlight the tradeoff between diagnosing/reacting to potentially harmful real-time events and minimizing the number of times the system is reset or scanned for malicious activity. The various health states of a system are modeled as states in a Markov chain, and we use a model fitting approach to estimate the transitions between these states. We proceed by considering a scenario in which a system is to be deployed over a fixed horizon but with a limit on the number of times that the health state can be scanned and the system can be reset. Each health state is assigned a cost according to the performance of the system while in that state. Dynamic Programming is then used to find an optimal *admissible* policy (one that obeys the usage limitation constraints) which achieves the lowest expected aggregate cost. Finally, we examine some properties of the solution.**

***Index Terms*—fault detection; dynamic programming; budgeted estimation; failure management**

I. Introduction

As the landscape of distributed storage and computing becomes increasing complex, it has become clear that new levels of automation are required to manage the operation of such services. Indeed, modern cloud computing systems can have as many as thousands of cloud servers which are interconnected by multi-layer networks [1]. Proactive failure management has proven to be an effective way of improving the reliability of systems [2]. The need for reliability, availability and serviceability (RAS) of systems is the impetus behind recent efforts to find automated failure management systems [3].

There are a myriad of approaches used to this end. Logged histories of past failures and current runtime execution states have been shown to provide a foundation on which resource allocation, computation reconfiguration and system maintenance can be performed [4]. The models used today use a combination of current system states along with past observations to make predictions, in contrast to more conventional methods such as classical reliability theory, which focus on long term averages of historical data.

Statistical learning has provided a foundation for many existing failure detection techniques [5]. Supervised methods such as those proposed in [6],[7],[8], and [9] utilize log data from failed and and normal execution paths to train a predictor on various performance features. In [10], an ensemble of Bayesian methods are used to perform anomaly detection by estimating the probability distribution of several features from normal execution paths.

Other mathematical techniques have also been used for event prediction, such as belief networks, time series analysis and wavelet analysis [11],[12]. Dynamic belief networks have been used for root cause isolation of problems in Microsoft Windows operating systems [13],[14]. Time series approaches have mostly been adapted from previous works aiming to predict telecommunication related issues [15] [16]. Unfortunately, the heterogeneity of system health data makes these techniques less effective. Perhaps it is due to this heterogeneity that many heuristic methods are used in practice today, such as the Dispersion Frame Technique (DFT), among others [17].

These methods often do not account for the fact that there are usually limitations in the accessibility of information that can be used for failure prediction and management. They also offer no guidelines for efficient scheduling of maintenance to react to failures. This paper aims to address these points with a Markov chain approach to this ubiquitous problem. Specifically, a key tradeoff between budget limitations and performance optimization is highlighted. We propose a model for the time evolution of the health of a system and offer a policy for scheduling observations and maintenance in an optimal way.

The approach taken here considers a limit on the number of times some actions can be taken over a problem horizon of finite length. This type of modeling has been used in similar situations where there is a budget on resources and a decision maker must allocate them most efficiently, such as point to point communications and networked control systems [18].

The structure of the paper is as follows. Section II addresses a motivating scenario for the modeling and optimization framework that will follow. In Section III, we introduce our Markov model for the evolution of the health state of a distributed system. Section IV discusses estimation of the parameters governing the evolution of the health state based on observations of the system. Then, Section V builds on this discussion by using the estimated parameters together with a dynamic programming approach to optimally schedule health state scans and system resets. Numerical results are presented

in Section VI, followed by a consideration of practical issues in Section VII along with concluding remarks and a brief discussion of future directions in Section VIII.

II. Motivating Scenario

In this paper we will be concerned with a scenario in which there are many nodes in a distributed computing infrastructure, each of which continuously performs tasks. Such systems are widely in place in industries which warehouse and process massive amounts of data, such as Google [19]. In an environment with many nodes sharing the computational load of the system, it is possible for nodes to fail and for restarts of tasks to take place. It is also possible that certain routine maintenance, such as patching against malware, would allow nodes to perform better. We can imagine such a system to look like Fig. 1.

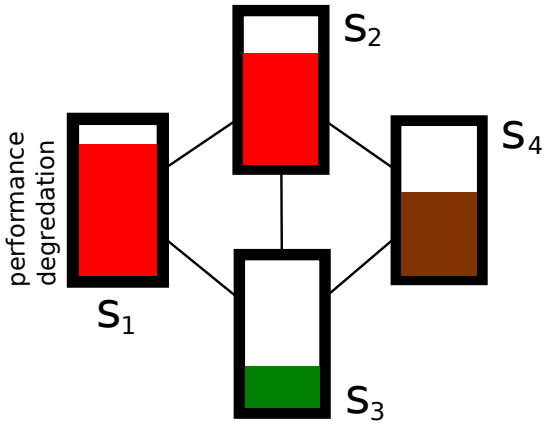

Fig. 1. Nodes in a computing cluster, each with a certain level of performance degredation.

The scenarios we wish to consider contain elements of randomness. That is, the health state of the nodes in the system may randomly fail or begin processing jobs with suboptimal performance at a random time, signaling the need for maintenance. Also, there could be correlations in failures, such as the tendency for similar issues to appear on machines that are on the same rack. Finally, we suppose that the number of nodes in the system is very large, and so scanning the system for problems or resetting the system to a workable state (flushing out no longer needed processes) is disruptive and expensive. This paper focuses on situations in which there is a limit on the number of service discontinuities that are experienced due to scanning or resetting the system.

III. Model

We now proceed by considering a general mathematical abstraction for modeling failures in a large-scale computing system. Indeed, although every network is different in physical arrangement, specifications and the jobs executed by each node, all clusters will have some things in common. For instance, the types of failures that occur can be broadly classified as hardware failures, the need to patch software vulnerabilities or other issues which are common to all systems. Hence, the health of a system can be summarized by the health of these components. Also, event logs are typically available for systems as well, which can provide a way to use data to diagnose this health state. In short, the problem of failure management in distributed computing lends itself nicely to probabilistic modeling.

A. Markov Chain

We can view the health state of a system as a summary of the health of several interconnected components, where the failure of a particular unit can cascade and result in failures of other parts of the system. As time passes, the chance occurrence of problems in some parts of the system can result in failures in other parts of the system. Conversely, if the health of a particular component is good, perhaps due to a recent update or patch being applied, we can expect that other units will as a result also be less prone to failure. In light of this, we model the evolution of the health state to be governed by a Markov chain whose parameters depend on the specific scenario at hand.

More formally, let us consider a finite state space of health states, $\mathcal{S}$. The choice of states is context dependent and can be adjusted according to the level of granularity which the modeler wishes to capture. For instance, one may take $\mathcal{S}$ to be the cross product of the health states for each component in the system. Specifically, we can consider the system to be comprised of several subunits $1, 2, .., K$ each of which has a health state $s_k \in \mathcal{S}_k$. Each $\mathcal{S}_k$ could be any set of descriptions for the state such as $\mathcal{S}_k = \{Bad, Medium, Good\}$. The overall state space could then be set as $\mathcal{S} = \mathcal{S}_1 \times \cdots \times \mathcal{S}_K$ to capture each component's health. This, however, introduces an unnecessary level of precision, since we are not interested in each individual unit's health but rather the health of the overall system, and it causes the state space to grow at an unmanageable rate.

In reality, we group health states in such a way that similar states get combined into one. This process of agglomeration can take place until an acceptable balance is achieved between the size of the state space, $|\mathcal{S}|$ and the level of granularity with which we describe the system. In addition to the health states, we also include a "reset" state, which corresponds to a situation in which the system has been reset and the health states of all components is perfect. Fig. 2 shows how the health profiles of two nodes can be agglomerated into three system health states which summarize the overall system health profile. Fig. 3 depicts a larger state space with the probabilistic transition drawn between them.

B. State Transitions

After arriving at an appropriate health state space $\mathcal{S}$, we must also model the transitions that occur from one state to the next. Depending on the implementation of the system under consideration, a particular state $s \in \mathcal{S}$ is taken to transition to another state $\hat{s} \in \mathcal{S}$ with probability $P(s, \hat{s})$. These transition probabilities together make up a transition matrix $\mathbf{P} \in \mathbf{R}^{|\mathcal{S}| \times |\mathcal{S}|}$.

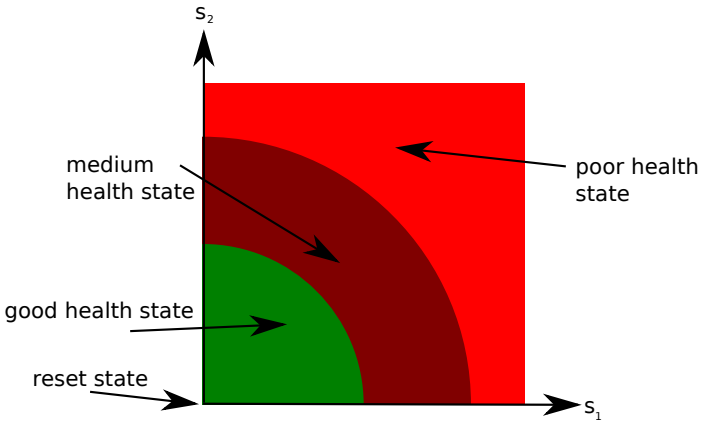

Fig. 2. Risk profiles of two machines are combined into three overall health states.

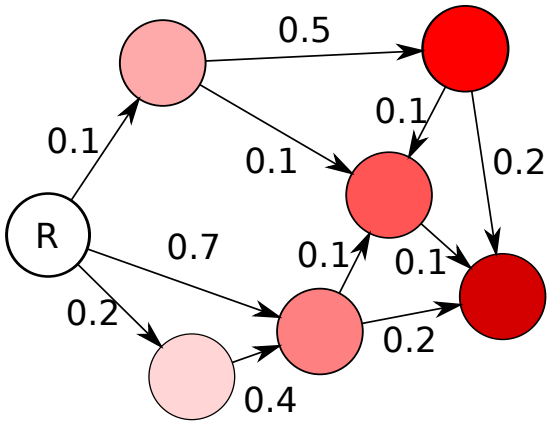

Fig. 3. Example health states Markov model. Darker red corresponds to higher cost (worse system performance) and residual probability mass at any node is taken as a self loop (transition to the same state).

C. Cost Structure

Each state in the Markov chain corresponds to a different health state - therefore it is natural to assign a cost to each state in $\mathcal{S}$. Higher costs correspond to a worse overall system health. Let us denote the costs by a vector $\mathbf{c}$ where each component corresponds to the cost of being in a particular health state.

D. Observations and Control

Due to the fact that scanning the system to determine the health state at any particular time is expensive, we consider a problem over N time steps in which a decision maker has a limited number, M_1, opportunities to scan the system for assessment of the health state. Similarly, resetting the health state of the system is also something that can be done a limited number of times because of the service disruption such an action causes. Therefore, we model the decision maker to have M_2 chances to reset the health state of the system. We use a shorthand for the action u_k a decision maker takes at time k as follows:

- $u_k = 0$: no action, cost $\hat{c}_0$.
- $u_k = 1$: system scan, cost $\hat{c}_1$.
- $u_k = 2$: system reset, cost $\hat{c}_2$.

where the quantity $\hat{c}_i$ corresponds to the cost of taking action i. Typically $\hat{c}_0 = 0$ but it can be set to any non-negative quantity to reflect a 'cost of not knowing'. In this paper we shall assume that $\hat{c}_0 \leq \hat{c}_1$, and $\hat{c}_0 \leq \hat{c}_2$.

E. Problem Statement

Let us now formally state the problem being solved in this paper. Consider the class of policies consisting of a sequence of functions $\Pi = \{\mu_1, ...\mu_{N-1}\}$, where each function μ_k maps the information available to the decision maker at time k to an action u from the set $A = \{0,1,2\}$ with the additional constraint that one may take action $u = 1$ at most M_1 times and action $u = 2$ at most M_2 times.

We want to find a policy $\pi^* \in \Pi$ to minimize the cost

$$J^{\pi}_{(M_1,M_2,N)} = E\left\{\sum_{k=1}^{N-1} c(x_k) + \hat{c}_{\mu_k(x_k)}\right\} \tag{1}$$

The objective function is a sum of the cost due to being in a particular state x_k and the cost from taking a particular action, but the difficulty of the problem lies in abiding by the given constraints.

IV. Parameter Estimation

Before solving the problem developed in the previous section, we must first address a critical issue: how are the parameters of the Markov chain in Fig. 3 determined? Once the quantities governing the evolution of the state space are determined, we can turn our attention to the problem at hand, but in a real system, these values would not be readily available from the modeling phase. Also, the costs for applying control (the costs $\hat{c}_0$, $\hat{c}_1$, $\hat{c}_2$) are domain specific and depend on the amount of operational disruption caused by scanning and resetting. Finally, we need to determine the appropriate cost vector $\mathbf{c}$ due to performance loss from being in a particular state. All of these can be determined by making observations on the system during a 'calibration' phase. Specifically, we allow the process to run for a set amount of time in a test environment so that observations can be made about its behavior.

A. Transition Probabilities

By observing the evolution of the Markov chain in a test environment we can keep track of each transition from state s to $\tilde{s}$ for all $s, \tilde{s} \in \mathcal{S}$. Then, when testing is over we obtain the estimated transition probabilities by computing an estimate for $P(s, \tilde{s})$ as

$$\hat{P}(s, \tilde{s}) = \frac{\text{\# transitions from } s \text{ to } \tilde{s}}{\text{\# transitions from } s \text{ to any state}}$$

so long as the number of observations in state s is nonzero. One can introduce LaPlace smoothing in the event that there are too few observations for a reliable estimate.

B. State and control costs

Just as in the case of estimating transition probabilities, we can average the performance loss in each state to come up with an associated cost. The cost assigned for a health state should be monotonically increasing as the performance loss increases. The cost of control can be measured by the performance degradation due to scanning or resetting.

V. Dynamic Programming

We now return to the problem defined in Section II - we would like to find an optimal admissible policy with respect to the objective function (1). This is done with a dynamic programming approach.

A. Backwards Induction

Define an extended state space comprised of five variables:

- r: the number of time steps since the last scan or reset action.
- s_1: the number of scanning opportunities remaining in the problem.
- s_2: the number of resetting opportunities remaining in the problem.
- t: the number of time steps remaining in the problem.
- x: the last known health state of the system.

These variables capture the state for a decision maker - all information necessary to make an optimal decision with respect to the problem statement is contained in this set. We are interested in using this information to find the optimal policy. To this end, let us denote a function as follows: $J_{(r,s_1,s_2,t)}(x)$ is the optimal cost-to-go given the variables (r, s_1, s_2, t, x). That is, given that the optimal decisions are made, this function returns the aggregated cost from the time $N - t$ onwards.

We can express this function's value recursively with the well-known Bellman's equation [20]:

$$
\begin{aligned}
&J_{(r,s_1,s_2,t)}(x) = \\
&\min_{u\in\{0,1,2\}} \Big\{ \mathbf{1}(u=0)\left(\hat{c}_0 + \mathbf{e}_x^T\mathbf{P}^r\mathbf{c} + J_{(r+1,s_1,s_2,t-1)}(x)\right) + \\
&\mathbf{1}(u=1)\big(\hat{c}_1 + \mathbf{e}_x^T\mathbf{P}^r\mathbf{c} + \sum_{y\in\mathcal{S}} \mathbf{P}^r(x,y) J_{(r+1,s_1-1,s_2,t-1)}(y)\big) \\
&+\mathbf{1}(u=2)\left(\hat{c}_2 + \mathbf{e}_x^T\mathbf{P}^r\mathbf{c} + J_{(r+1,s_1,s_2-1,t-1)}(R)\right) \Big\}
\end{aligned}
$$

where $r, t \geq 1$, $0 < s_1 \leq M_1$, $0 < s_2 \leq M_2$ and $\mathbf{1}(\cdot)$ is the indicator function. The vector $\mathbf{e}_x$ is a vector of zeros except in the x component. We need boundary conditions for the cases where $s_1 = 0$ or $s_2 = 0$ (corresponding to the cases where there are no more opportunities to scan or reset the system). These require separate computations. For the $s_1 = 0$ case we can write a similar equation to obtain a recursive cost-to-go expression:

$$
\begin{aligned}
&J_{(r,0,s_2,t)}(x) = \\
&\min_{u\in\{0,2\}} \Big\{ \mathbf{1}(u=0)\left(\hat{c}_0 + \mathbf{e}_x^T\mathbf{P}^r\mathbf{c} + J_{(r+1,0,s_2,t-1)}(x)\right) \\
&\qquad +\mathbf{1}(u=2)\left(\hat{c}_2 + \mathbf{e}_x^T\mathbf{P}^r\mathbf{c} + J_{(r+1,0,s_2-1,t-1)}(R)\right) \Big\}
\end{aligned}
$$

where $r, t \geq 1$ and $s_2 > 0$. And finally, for the $s_2 = 0$ case, it is worth noting that the future cost due to damage will be the same whether or not scans are done, due to the fact that there is no remaining budget to reset the system. In light of our assumption that $\hat{c}_0 \leq \hat{c}_1$, we can take sum of the expected action costs $\hat{c}_0$ and future performance costs to get the boundary condition:

$$
J_{(r,s_1,0,t)}(x) = \sum_{k=0}^{t-1} \left(\hat{c}_0 + \mathbf{e}_x^T\mathbf{P}^{k+r}\mathbf{c}\right)
$$

where $r, t \geq 1$, $s_2 \geq 0$ and $t \geq 1$.

We require one final boundary condition. In the event that the recursion requires a value for $t = 0$, let us assign $J_{r,s_1,s_2,0}(x) = 0$ for all $x \in \mathcal{S}$. This corresponds to a stage of the problem which is out of scope, and hence no future cost should be accrued. If we want to drive the system to a particular terminal state, we can assign costs to $t = 0$ stages to influence this. For example, if we want to strongly insist that the health state be returned to a reset state at the end, we can put a large terminal cost on all other states except this one. The optimal policy will then strongly favor one in which a reset action is used in the very last step.

B. Offline computation

We now use these equations to obtain a method of determining an optimal policy. Once the approximate dynamics of the system have been learned after a training phase (as described in Section IV), we can do all further computation offline, before deployment of a system.

One would proceed by using the backwards induction equations to obtain cost-to-go values for all reachable stages of our problem. Our computations could be done by using any off-the-shelf dynamic programming method, such as value iteration or policy iteration. The order in which the recursive equations above would be used is:

1) Set $J_{(r,s_1,s_2,0)}(x) = 0$
2) Compute $J_{(r,s_1,0,t)}(x)$ for all $t \leq N$ and $x \in \mathcal{S}$
3) Compute $J_{(r,0,s_2,t)}(x)$ for all $t \leq N$, $s_2 \leq N_2$ and $x \in \mathcal{S}$
4) Compute $J_{(r,s_1,s_2,t)}(x)$ for all $t \leq N$, $s_2 \leq N_2$ and $x \in \mathcal{S}$

Once these cost-to-go values are obtained, they can be stored in memory and used when the system is actually deployed. The arguments minimizing the expressions which are used to obtain items 3 and 4 should also be stored in memory, since they are the actions u associated with getting the minimized cost-to-go. These saved arguments are looked up at the time of deployment.

VI. Numerical Results

Let us now examine some numerical results to illustrate the performance of our approach to scheduling scanning and resetting actions. We use a small system as an example, with only $|\mathcal{S}| = 5$ health states. In what follows, it is assumed that the estimation phase of this problem has been completed and the necessary problem parameters are available to us, such as the transition matrix $\mathbf{P}$ and the cost vector $\mathbf{c}$. Using these parameters, we consider a scenario in which we fix the horizon of the problem, N, and the number of opportunities to scan and reset, M_1 and M_2, respectively. We then examine several plots which illuminate the structure of the solution and discuss this structure. We also compare the performance of the given algorithm to a baseline heuristic.

A. Problem

The problem we use for numerical insight into our model is one in which we have six health states (labeled $R, s_1, \ldots, s_5$), including one reset state, with associated transitions and cost vector:

$$\mathbf{P} = \begin{bmatrix} 0.6 & 0.2 & 0.1 & 0.1 & 0 & 0 \\ 0 & 0.5 & 0 & 0.3 & 0.2 & 0 \\ 0 & 0 & 0.6 & 0 & 0.4 & 0 \\ 0 & 0 & 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0 & 0 & 0.4 & 0.6 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{c} = \begin{pmatrix} 0 \\ 2 \\ 1 \\ 3 \\ 5 \\ 7 \end{pmatrix}$$

Additionally, we use costs $\hat{c}_0 = 0$, $\hat{c}_1 = 2$, $\hat{c}_2 = 5$. The transitions can be visualized in Fig. 4.

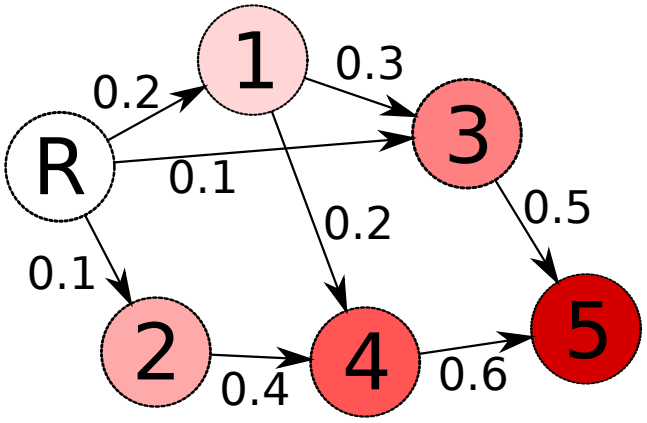

Fig. 4. Model for numerical experiments.

We consider a problem with a duration of $N = 20$ time steps and examine some properties of the solution for various values of M_1 and M_2.

B. Error Curves

Let us examine structural properties of the solution while fixing the values of M_1 and M_2. That is, we allow each of these to vary independently while fixing the other and plot the expected error. The resulting curve when we fix $M_2 = 5$ and allow M_1 to vary is plotted in Fig. 5. The plot also includes the cost of the baseline strategy of evenly spacing $M_2 = 5$ system resets.

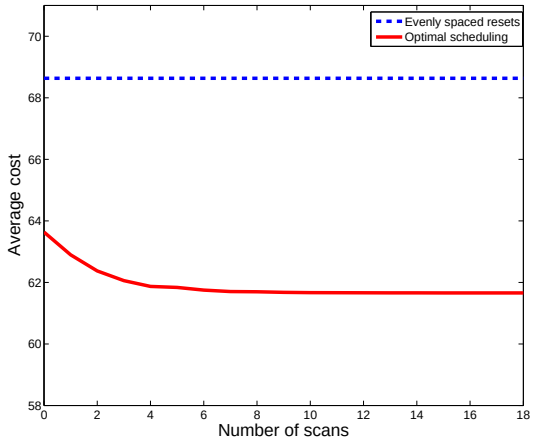

Fig. 5. Error curve for fixed $M_2 = 5$.

We see that using only a small number of system scans, we are able to obtain near optimal performance. Also, it appears that system scans are not nearly as important to schedule as system resets. Indeed, the greatest benefit of scanning the system comes from the first one, and the rest provide relatively little improvement to performance. Similarly, we can look at a curve for the scenario in which we fix $M_1 = 2$ and allow M_2 to vary. This is shown in Fig. 6.

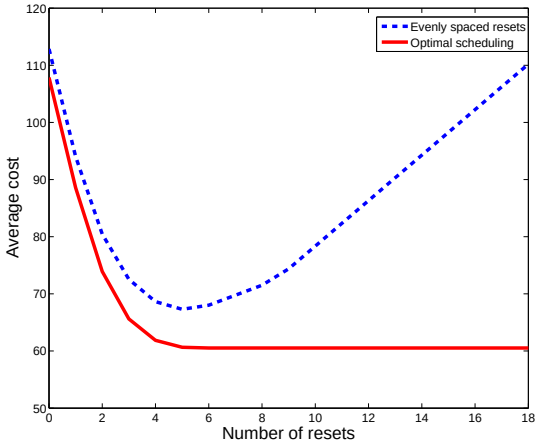

Fig. 6. Error curve for fixed $M_1 = 2$. When the number of resets is large, the cost for even spacing is dominated by reset actions.

As in the fixed M_2 case, we see that once again, the number of resets required to get near optimal performance is a small fraction of N. Indeed, after a point, the cost of the baseline algorithm of equally spacing rises rapidly. One consequence of this is that we can limit the number of scans and resets apriori. That is, before beginning the problem, guarantees can be made about the number of times system performance is

disrupted, since we know that the performance gain of using many scans and resets is small.

Finally, we point out some properties of these curves. Although they can be proved rigorously, we simply state them here due to space constraints.

1) Monotonicity: Holding one of M_1 or M_2 constant and varying the other shows that the error monotonically decreases in each of these parameters. In terms of the problem, it demonstrates that having more opportunities to scan or reset the system never hurts, which is intuitively true.

2) Convexity: Once again holding each of the two resources constant, we can see that the curve generated by varying each of the other parameters is convex. This is interpreted to mean that resources in this framework exhibit diminishing returns: the first resources that are received account for more cost reduction than later resources. This is important decision making information, since knowing this can allow us to achieve near optimal performance with far fewer resources. It should be noted that the degree of convexity relies greatly on the parameters of the problem at hand.

VII. Practical Issues

A number of modeling considerations and tradeoffs would come into play if such a dynamic failure management policy were implemented on a live system. Indeed, one would need a good way to determine the number of states in the Markov model and how to agglomerate health states of the individual nodes. The issue of how well the learned model represents the true functioning of the system must also be evaluated. With more states (more granularity in describing the risk state), we expect that the model matches reality more closely. However, this representation must be traded off with the cost of keeping track of more states. Finally, the possible actions one may take to service a poorly performing system could be more varied than simply deciding whether or not to reset. However, these details are specific to the instance of the problem which is encountered, and a similar approach to the one given here can be used to handle them. This paper aims to demonstrate a mathematical approach to scheduling diagnostics and maintenance with limits on the number of times such actions may be performed.

VIII. Conclusion and Future Work

In this paper we have presented a model for keeping track of the health state in a complex networked system and for making decisions on when the system should be reset to restore a clean working condition. Markov chains are used to probabilistically model the health state as it evolves in time. The transitions between states are learned during a "tuning" phase, when observations can be made on the health state at each time and on the associated performance of the system. Once the transition probabilities and health state costs have been assessed, a dynamic programming approach is used. This backwards induction process gives a method of optimally scheduling scan and reset actions.

Simulations reveal that in some cases, it is possible to get near optimal performance from the system with only a few opportunities to scan or reset. Additionally, several structural properties of the solution are explored.

Research into fault management in networked systems is still in its nascent stages and there are several opportunities for further research. Future work can consider distributed approaches for collecting log information in the even that communication between nodes in a cluster is expensive. It can also be a worthwhile effort to consider situations in which the state space for health states is unmanageably large for dynamic programming and approximate solutions must be considered.

References

[1] N. Adiga and B. G. Team. An overview of the bluegene/l supercomputer. In *Supercomputing (SC2002) Technical Papers*, November 2002.

[2] R. K. Sahoo, A. J. Oliner, I. Rish, and et al. Critical event prediction for proactive management in large-scale computer clusters. In *Proceedings of ACM International Conference on Knowledge Discovery and Data Dining (SIGKDD)*, 2003

[3] P. Horn. Automatic computing: IBM's perspective on the state of information technology. *http://www.research.ibm.com/autonomic, IBM Corporation*, 2001

[4] A. J. Oliner, R. K. Sahoo, J. E. Moreira, and et al. Fault-aware job scheduling for BlueGene/L systems. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[5] F. Salfner, M. Lenk, and M. Malek. A survey of online failure prediction methods. *ACM Computing Surveys*, 42:10:110:42, 2010

[6] J. W. Mickens and B. D. Noble. Exploiting availability prediction in distributed systems. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2006

[7] R. K. Sahoo, A. J. Oliner, I. Rish, and et al. Critical event prediction for proactive management in large-scale computer clusters. In *Proceedings of ACM International Conference on Knowledge Discovery and Data Dining (SIGKDD)*, 2003.

[8] S. Fu and C. Xu. Quantifying event correlations for proactive failure management in networked computing systems. Journal of Parallel and Distributed Computing, 70(11):11001109, 2010

[9] J. Gu, Z. Zheng, Z. Lan, J. White, E. Hocks, and B.-H. Park. Dynamic meta-learning for failure prediction in large-scale systems: A case study. In *Proceedings of IEEE International Conference on Parallel Processing (ICPP)*, 2008

[10] Q. Guan, Z. Zhang and S. Fu. Ensemble of Bayesian Predictors for Autonomic Failure Management in Cloud Computing. In *International Conference on Computer Communication Networks (ICCCN)*, 2011.

[11] F. Jensen. *An introduction to Bayesian Networks*. UCL Press, London, 1996.

[12] K. P. Murphy. Active learning of causal bayes net structure. In *http://citeseer.nj.nec.com/451402.html*.

[13] J. Berger. *Statistical Decision Theory and Bayes Analysis*. Springer-Verlag, New York, 1985.

[14] D. Heckerman. A tutorial on learning with Bayesian networks. *Tech. Rep. MSR-TR-95-06, Microsoft Research*, 1996.

[15] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. Wiley-Interscience, 2001

[16] C. Lim, N. Singh, and S. Yajnik. A log mining approach to failure analysis of enterprise telephony systems. In *Proc. of IEEE Conference on Dependable Systems & Networks (DSN)*, 2008

[17] M. F. Buckley and D.P. Siewiorek. Vax/vms event monitoring and analysis. In *FTCS-25, Computing Digest of Papers*, pages 414-423, June 1995.

[18] O.C. Imer. Optimal Estimation and Control under Communication Network Constraints. Ph.D. Dissertation, UIUC, 2005.

[19] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In OSDI04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation, 2004.

[20] D. P. Bertsekas, *Dynamic Programming and Optimal Control*. Belmont, MA: Athena Scientic, 1995.