Towards Autonomic Computing: Adaptive Network Routing and Scheduling

Shimon Whiteson and Peter Stone
Department of Computer Sciences
The University of Texas at Austin
1 University Station, C0500
Austin, TX 78712-0233
{*shimon,pstone*}@*cs.utexas.edu*
http://www.cs.utexas.edu/~{shimon,pstone}

Track: Emerging Application
Application Domain: Network Routing and Scheduling
AI Techniques: Q-routing
Application Status: Research Prototype

Abstract

Computer systems are rapidly becoming so complex that maintaining them with human support staffs will be prohibitively expensive and inefficient. In response, visionaries have begun proposing that computer systems be imbued with the ability to configure themselves, diagnose failures, and ultimately repair themselves in response to these failures. However, despite convincing arguments that such a shift would be desirable, as of yet there has been little concrete progress made towards this goal. We view these problems as fundamentally *machine learning* challenges. Hence, this article presents a new network simulator designed to study the application of machine learning methods from a system-wide perspective. We also introduce learning-based methods for addressing the problems of packet routing and CPU scheduling in the networks we simulate. Our experimental results verify that methods using machine learning outperform heuristic and hand-coded approaches on an example network designed to capture many of the complexities that exist in real systems.

Introduction

Computer systems are rapidly becoming—indeed some would say have already become—so complex that maintaining them with human support staffs will be prohibitively expensive and inefficient. Large enterprise systems, such as those found in medium-sized to large companies, are prime examples of this phenomenon. Nonetheless, most computer systems today are still built to rely on static configurations and can only be installed, configured, and re-configured by human experts.

In response, visionaries have begun proposing that computer systems be imbued with the ability to configure themselves, diagnose failures, and ultimately repair themselves in response to these failures. The resulting shift in computational paradigm has been called by different names, including *cognitive systems* (Brachman 2002) and *autonomic computing* (Kephart & Chess 2003), but the underlying motivation and goal is remarkably similar.

However, despite convincing arguments that such a shift would be desirable, as of yet there has been little concrete progress made towards this goal. There has been preliminary progress on adaptive system *components* such as network routing (Boyan & Littman 1994; Caro & Dorigo 1998; Clark *et al.* 2003; Itao, Suda, & Aoyama 2001). But to our knowledge, there has not been any previous work on improving system performance from a system-wide perspective.

Our long-term goal is to enable large-scale integrated computer systems, consisting of tens to hundreds of machines with varying functionality, to be delivered in a default configuration and then incrementally tune themselves to the needs of a particular enterprise based on observed usage patterns. In addition, the systems should be able to adapt to changes in connectivity due to system failures and/or component upgrades.

We view these goals as fundamentally *machine learning* challenges. For entire systems to be able to self-configure, self-diagnose failures, and repair themselves, there will need to be machine learning components at multiple levels, including the operating systems, databases, and networking modules. Furthermore, individual computers and local systems will need to adapt their interactions with remote systems whose connectivity and computing capabilities may vary unpredictably.

This article introduces a simulator designed to facilitate the study of machine learning in enterprise systems and reports on our initial experiments in this domain. These experiments underscore key complexities that arise when optimizing routing and scheduling in computer networks. We present machine learning approaches to address these challenges.

The remainder of this article is organized as follows. The next section provides background on our network simulator and the particular network we use in all our experiments. After that, we detail our methods for optimizing routing and scheduling and present the results of our experiments evaluating these methods. Finally, we discuss the implications of these results and highlight some opportunities for future work.

Copyright © 2004, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

Background

This section introduces the simulator that we use as the substrate system for our research as well as a detailed example network that is the setting for the experiments presented in this article.

The Simulator

To pursue our research goals, we need a high-level simulator that is capable of modeling the relevant types of interactions among the many different components of a computer system. While detailed simulators exist for individual system components, such as networks, databases, etc., we were unable to locate any simulator that models system-wide interactions. Therefore, we have designed and implemented a system that simulates the way a computer network processes user requests from a high-level perspective.

The simulator represents a computer network as a graph: nodes represent machines or users and links represent the communication channels between them. Users create jobs that travel from machine to machine along links until all of their steps are completed. When a job is completed, it is assigned a score according to a given *utility function*, which depends on how much time the job took to complete. The utility function may vary depending on the type of job or the user who created it. The job's score is added to the simulation's global, cumulative score, which the agents controlling the network are trying to maximize.

The simulator includes the following components:

Nodes: A node is a component of the network that is connected to other nodes via links. There are two primary types of nodes: users and machines.

Users: Users are special nodes who create jobs and send them to machines for processing. Once a job is completed, it is returned to the user, who computes its score.

Machines: A machine is a node that can complete portions of a job. Each type of machine is defined by the set of steps it knows how to complete.

Links: A link connects two nodes in the network. It is used to transfer jobs and other packets between nodes.

Packets: A packet is a unit of information that travels between nodes along links. The most common type of packet is a job, described below, but nodes can create other types of packets in order to communicate with other nodes about their status.

Jobs: A job is a series of steps that need to be completed in a specified order. Completing these steps could require the job to travel among several machines. A system usually has several types of jobs which differ in the list of steps they require for completion.

Steps: A step is one component of a job. Each step can only be carried out by a subset of machines in the network. For example, the retrieval of information in response to a database query must happen at a database server.

A simulation proceeds for a specified number of discrete timesteps. At each timestep, machines can allocate their CPU cycles towards the completion of steps on the jobs in its possession, packets can be sent along links, and packets can arrive at new nodes.

We believe that this simulator provides a valuable testbed for new approaches to autonomic computing. Because its design is very general, it can be used to represent a wide variety of computer systems with arbitrary topology. Furthermore, it is highly modular which makes it easy to insert intelligent agents to control any aspect of the system's behavior. Most importantly, the simulator captures many of the real world problems associated with complex computer systems while retaining the simplicity that makes experimental research feasible.

An Example Network

All of the experiments presented in this article were performed on the network depicted in Figure 1. In this network, the users (the CEO and Intern) can check their mail or perform database queries via a web interface. Two load balancers do not complete any steps but make important decisions about how to route jobs. The speed associated with each machine represents the number of CPU cycles it can execute in one turn. Note that the web servers and mail servers are not fully connected.

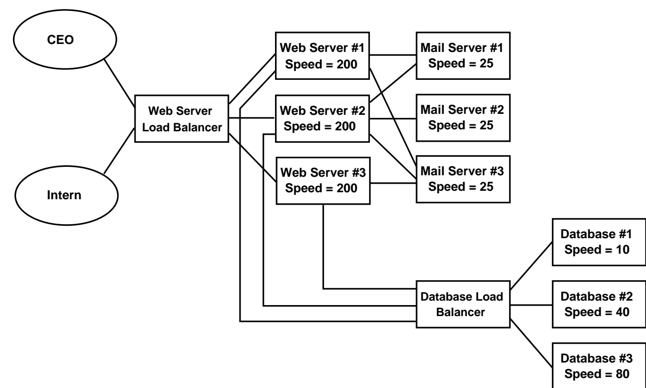

Figure 1: The network used in our experiments; ovals represent users and rectangles represent machines; the lines between them represent links that allow communication of jobs or other packets. The speed associated with each machine represents the number of CPU cycles it can execute in one turn. Note that the web servers and mail servers are not fully connected.

There are two types of jobs that the users create: Mail Jobs and Database Jobs. Each Mail Job consists of the following three steps: 1) Web Step (work = 50), 2) Mail Step (work = 100), and 3) Web Step (work = 50). The work associated with each step is simply the number of CPU cycles required to complete the step. As one might expect, only Web Servers can complete Web Steps and only Mail Servers can complete Mail Steps. A Database Job is identical except that its second step is a Database Step (work = 200).

In this article, we explore ways of applying machine learning techniques to the problems of routing and scheduling jobs efficiently. The network just described, although abstract, nonetheless captures the essential difficulties posed

by such optimization problems. For example, each Database has a different speed, which means that if the Database Load Balancer routes packets randomly, it will overload the slower Databases and underload the faster ones. Even if we take machine speed into account, routing on this network is far from trivial. Consider a Web Server that wishes to forward a Mail Job to a Mail Server. Since there is no load balancer governing the Mail Servers, the Web Server must pick directly from among the Mail Servers to which it is connected. Doing so optimally requires considering not just the speed of each Mail Server but also how busy that server is completing jobs sent by other Web Servers. Since the connections between Web and Mail Servers are not just incomplete but also uneven (i.e. some Web Servers are connected to more Mail Servers than others), determining this is not easy. Similarly, the Web Server Load Balancer cannot consider only the relative speed of a Web Server to which it is considering routing a Mail Job. It must also consider how much access that Web Server has to Mail Servers and how busy those Mail Servers are.

Load Updates Each machine periodically (every five timesteps) sends a special packet called a Load Update to each of its neighbors. A Load Update indicates how many jobs that machine already has in its queue. The contents of such an update can help an intelligent router make better decisions. Load updates incur network traffic overhead but are quite useful for making routing decisions. As long as they are not too frequent, including them as a routine occurrence is not unrealistic.

Job Creation At each timestep, each user chooses randomly between creating one or two new jobs. For each job, it chooses randomly between a Mail Job and a Database Job. The creation of new jobs by each user is subject to an important restriction: each user must remain below a maximum number of incomplete jobs (set to fifty in our experiments). When a user is at this maximum, it does not create any new jobs until older jobs are completed. This simple method of generating jobs models features of real user behavior: users tend to reduce their use of networks that are overloaded and the creation of new jobs depends on the completion of older ones. For example, a user typing a document on a slow terminal is likely to stop typing momentarily when the number of keystrokes not reflected on the screen becomes too great. In addition, this demand model allows us to easily test our methods on a network that is busy but not overloaded. Any demand model that is not tied to the system's capacity is likely to either under or over utilize network resources. In the former case, weak methods may still get good performance since there is spare capacity (i.e. a ceiling effect). In the latter case, even good methods will perform badly because the available resources, regardless of how they are allocated, are insufficient to meet demand. Our demand model, by striking a balance between these alternatives, allows us to more effectively compare methods of optimizing the network's performance.

Utility Functions The ultimate goal of our efforts is to improve the network's utility to its users. In the real world, that utility is not necessarily straightforward. While it is safe to assume that users always want their jobs completed as quickly as possible, the value of reducing a job's completion time is not always the same. Furthermore, each user may have a different importance to the system. In our network, the utility of quickly completing the CEO's jobs should be higher than the utility of doing so for the intern.

In order to capture these complexities, we assign different, non-linear utility functions, shown in Figure 2, to our two users. Jobs created by the intern are scored according to the following function: $U(t) = -20\ln(t)$ where t is the job's completion time. By contrast, jobs created by the CEO are scored by the function $U(t) = -(t^{1.2})$. Though our experiments study only this particular pair of metrics, our algorithms are designed to work with arbitrary functions of completion time, so long as they are monotonically decreasing. These metrics were selected because they are significantly different from each other and because they are non-linear, features which, as explained below, create the complications that make intelligent scheduling non-trivial. The specific coefficients were tuned for this scenario only in the sense that the region of interest, where one curve crosses the other, lies in the neighborhood of the average completion time for jobs in this network.

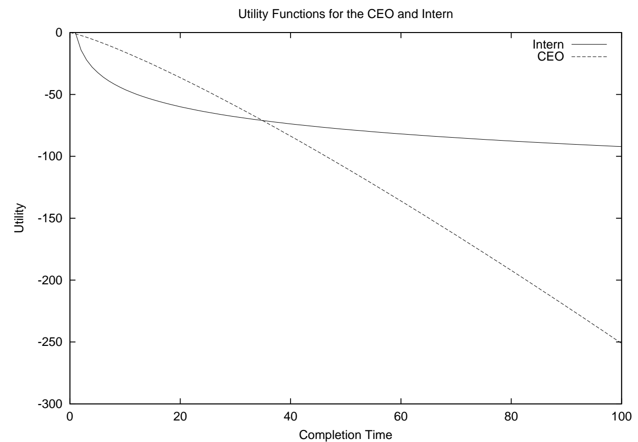

Figure 2: Utility Functions for the CEO and Intern.

Method

In this section, we present our approach to developing intelligent routers and schedulers for networks like the one detailed above. Due to the complications described above, achieving good performance in such a network using fixed algorithms and hand-coded heuristics is very difficult and prone to inflexibility. Instead, we use reinforcement learning to develop routers and schedulers that are efficient, robust, and adaptable. The rest of this section explains the details of our approach to these two problems.

Routing

As traditionally posed, the packet routing problem requires a node in a network to decide to which neighboring node to

forward a given packet such that it will reach its destination most quickly. In the network simulation described above, each machine faces a similar but not identical problem each time it finishes processing a job. When it is unable to complete the next step required by the job, it must search among its neighbors for machines that can complete that step (or that can forward the job to machines that can complete it). If more than one neighbor qualifies, the machine should make the choice that allows the job to be completed as quickly as possible.

In both our task and the traditional routing problem, the router tries to minimize the travel time of a packet given only local information about the network. However, in the traditional problem the goal is only to get the packet from its source to a specified destination. In our domain, this goal is not relevant. In fact, since a job returns to its creator when it is complete, the source and destination are the same. Instead, we want the job to travel along a path that allows the appropriate machines to complete its various steps in sequence and return to its creator in minimal time.

In this section, we present three ways of addressing this modified routing problem: a random method, a heuristic method, and a method, called Q-routing, based on reinforcement learning.

Random Router As its name implies, the random router forwards jobs to a machine selected randomly from the set of contenders C. A neighboring machine is a contender if it is capable of completing the job's next step. If no such machines exist, then C is the set of all neighbors who can forward the job to a machine that can complete its next step. In the random router, the probability that a given job will be forwarded to a specific contender $c \in C$ is:

$$P_c = \frac{1}{|C|}$$

where $|C|$ is the size of C. Despite its simplicity, the random router is not without merit. For example, if all the neighbors have the same speed and do not receive load from anywhere else, the random router will keep the load on those neighbors evenly balanced. Of course, it does not address any of the complications that make routing a non-trivial problem and hence we expect it to perform poorly in real world scenarios.

Heuristic Router Without the aid of learning techniques or global information about the network, a heuristic router cannot be expected to perform optimally. However, it can do much better than the random router by exploiting the available local information, like the speed of its neighbors, to make routing decisions. Hence, our experiments also test a heuristic in which the likelihood of routing to a given neighbor is in direct proportion to that neighbor's speed. That is, for each $c \in C$,

$$P_c = \frac{speed(c)}{\sum_{c' \in C} speed(c'))}$$

Hence, if there are two qualifying neighbors and one is twice as fast as the other, a given packet will have a 2/3 probability of going to the fast machine and a 1/3 probability of going to the slower one. This algorithm ignores both the load these neighbors might be receiving from other machines and the status of any machines the packet might be sent to later. Hence, it acts as a myopic load balancer.

Q-Router Despite the distinctive features of our version of the routing problem, techniques developed to solve the traditional version can, with modification, be applied to the task faced by machines in our simulation. In this article, we adapt one such technique, called Q-routing (Boyan & Littman 1994), to improve the performance of our network. Q-routing is an on-line learning technique in which a reinforcement learning module is inserted into each node of the network.

In Q-routing, each node x maintains a table of estimates about the *time-to-go* of packets if they are routed in various ways. Each entry $Q_x(d, y)$ is an estimate of how much additional time a packet will take to travel from x to its ultimate destination d if it is forwarded to y, a neighbor of x. If x sends a packet to y, it will immediately get back an estimate t for x's time-to-go, which is based on the values in y's Q-table:

$$t = min_{z \in Z} Q_y(d, z)$$

where Z is the set of y's neighbors. With this information, x can update its estimate of the time-to-go for packets bound for d that are sent to y. If q is the time the packet spent in x's queue and s is the time the packet spent traveling between x and y, then the following update rule applies:

$$Q_x(d, y) = (1 - \alpha)Q_x(d, y) + \alpha(q + s + t)$$

where α is a learning rate parameter (0.7 in our experiments). In the standard terms of reinforcement learning (Sutton & Barto 1998), $q+s$ represents the instantaneous reward (cost) and t is the estimated value of the next state, y.

By bootstrapping off the values in its neighbors' Q-tables, this update rule allows each node to improve its estimate of a packet's time-to-go without waiting for that packet to reach its final destination. This approach is based directly on the Q-learning method (Watkins 1989). Once reasonable Q-values have been learned, packets can be routed efficiently by simply consulting the appropriate entries in the Q-table and routing to the neighbor with the lowest estimated time-to-go for packets with the given destination.

State Representation. To make Q-routing more suitable for our unique version of the packet routing problem, we must change the state features on which learning is based. Instead of simply using the job's destination, we use three features that indicate in what general direction the job is headed, what machine resources it will likely tax if routed in a particular way, and what priority it will be given by subsequent schedulers:

- the type of the job,
- the type of the next step the job needs completed, and
- the user who created the job

In addition, we want a fourth state feature that allows the router to consider how much load is already on the neighbors to which it is considering forwarding a job. We could

add a state feature for every neighbor that represents the current load on that machine. However, this would dramatically increase the size of the resulting Q-table, especially for large, highly-connected networks, and could make table-based learning infeasible. Fortunately, almost all of those state features are irrelevant and can be discarded. Since we are trying to estimate a job's time-to-go if it is routed to a given machine, the only information that is relevant is the load on *that particular machine*. Hence, we use an action-dependent feature (Stone & Veloso 1999). As the name implies, action-dependent features cause an agent's state to change as different actions are considered. In this case, our action-dependent feature always contains the current load on whatever neighbor we are considering routing to. The load on all other neighbors is not included and hence the Q-table remains very small.

Update Frequency. The original formulation of Q-routing specifies that each time a node receives a packet it should reply with a time-to-go estimate for that packet. However, it is not necessarily optimal to do so every time. In fact, the frequency at which such updates are sent represents an important trade-off. The more often a reply is sent, the more reliable the router's feedback will be and the more rapidly it will train. However, if replies are sent less often, then more network bandwidth is reserved for actual packets, instead of being clogged with administrative updates. In our implementation, replies are sent with a 0.5 probability, which we determined through informal experimentation to be optimal.

Action Selection. Like other techniques based on reinforcement learning, Q-routing needs an exploration mechanism to ensure that optimal policies are discovered. If the router always selects the neighbor with the lowest time-to-go, it may end up with a sub-optimal policy because only the best neighbor's estimate will ever get updated. An exploration mechanism ensures that the router will occasionally select neighbors other than the current best and hence eventually correct sub-optimalities in its policy. In our implementation, we use ϵ-greedy exploration (Sutton & Barto 1998), with ϵ set to 0.05. In ϵ-greedy exploration, the router will, with probability ϵ, select a neighbor randomly; with probability $1 - \epsilon$ it will select the currently estimated best neighbor.

Scheduling

The routing techniques discussed above all attempt to minimize the time that passes between the creation and completion of a job. However, this completion time is only indirectly related to the score, which it is our goal to maximize. The score assigned to any job is determined by the utility function, which can be different for different types of jobs or users. The only requirement is that the function decrease monotonically with respect to completion time (i.e. users never prefer their jobs to take longer).

At first, this monotonicity constraint may seem to make the routing approach sufficient on its own: if we are minimizing the completion time, we must be maximizing the score. However, this is true only in the very limited case where all jobs have the same importance. There are two important ways that jobs can vary in importance.

Firstly, the jobs may be governed by different utility functions. Suppose jobs created by the intern were scored according to the function $U(t) = -t$ while jobs created by the CEO were scored according to the function $U(t) = -100t$. In this case, the CEO's jobs are vastly more important. Clearly, a network that devotes as much of its capacity towards the intern's jobs as the CEO's jobs will be very suboptimal.

Secondly, utility functions may be non-linear. Even if all jobs are controlled by the same function, if that function is non-linear then some jobs will matter more than others. Imagine a utility function that slopes down sharply until $t = 50$ and then completely flattens out. Now consider two jobs working their way through the network, one that was created 25 timesteps ago and one that was created 100 timesteps ago. In this scenario, the former job is much more important than the latter. The job that has been running for 100 timesteps is a "lost cause": it is already past the region in which there is hope of improving its score so spending network resources to speed up its completion would be fruitless. By contrast, the job that has only run for 25 timesteps is very important: if it is possible to complete the job in less than 50 timesteps, then every step that can be shaved off its completion time will result in an improved score.

Hence, when jobs do not all have equal importance, minimizing the completion time of less important jobs can be dramatically suboptimal because it uses network resources that would be better reserved for more important jobs. In this sense, the Q-routing technique explained above has a greedy approach: it attempts to maximize the score of a given job (by minimizing its completion time) but does not consider how doing so may affect the score of other jobs.

In principle, this shortcoming could be addressed by revising the values that the Q-router learns and bases its decisions on. For example, if the Q-values represented global utility instead of time-to-go, the router would have no incentive to favor the current job and could eventually learn to route in a way that maximizes global utility, even at the expense of a particular job's time-to-go. However, such a system would have the serious disadvantage of requiring each node to have system-wide information about the consequences of its actions, whereas the current system is able to learn given only feedback from immediate neighbors.

Another alternative would be to change the router's action space. Currently, an action consists of routing a particular job to some neighbor. Instead, each action could represent a decision about how to route *all* the jobs currently in the machine's queue. While such a system would reduce the router's myopia, it would create a prohibitively large action space. Given a queue of length n and a set of m neighbors, there would be m^n possible actions!

Given these difficulties, we believe the challenges posed by complicated utility functions are best addressed, not by further elaborating our routers, but by coupling them with intelligent schedulers. Schedulers determine in what order the jobs sitting at a machine will be processed. They decide

how the machine's CPU time will be scheduled. By determining which jobs are in most pressing need of completion and processing them first, intelligent schedulers can maximize the network's score even when the utility functions are asymmetric and non-linear.

FIFO Scheduler The default scheduling algorithm used in our simulator is the *first-in first-out* (FIFO) technique. In this approach, jobs that have been waiting in the machine's queue the longest are always processed first. More precisely, the scheduler chooses the next job to process by selecting randomly from the set J_l of jobs that have been waiting the longest. If $time(j)$ is the time that job j arrived at the machine and J is the set of waiting jobs, J_l is determined as follows:

$$J_l = \{j \in J \mid time(j) \leq time(j'), \forall j' \in J\}$$

Clearly, the FIFO algorithm does nothing to address the complications that arise when jobs have different importance.

Priority Scheduler An alternative heuristic that does address these concerns is a *priority scheduler*. This algorithm works just like the FIFO approach except that each job is assigned a priority. When allocating CPU time, the priority scheduler examines only those jobs with the highest priority and selects randomly from among the ones that have been waiting the longest. In other words, the priority scheduler selects jobs randomly from the following set:

$$J_l = \{j \in J \mid \quad time(j) \leq time(j') \wedge priority(j) \geq priority(j'), \quad \forall j' \in J\}$$

If all the utility functions are simply multiples of each other, the priority scheduler can achieve optimal performance by assigning jobs priorities that correspond to the weight of their utility function. However, when the utility functions are truly different or non-linear, the problem of deciding which jobs deserve higher priority becomes much more complicated and the simplistic approach of the priority scheduler breaks down.

Insertion Scheduler To develop a more sophisticated approach, we need to formulate the problem more carefully. Every time a new job arrives at a machine, the scheduler must choose an ordering of all the n jobs in the queue and select for processing the job that appears at the head of that ordering. Of the $n!$ possible orderings, we want the scheduler to select the ordering with the highest utility, where utility is the sum of the estimated scores of all the jobs in the queue. Hence, to develop an intelligent scheduler, we need to decide 1) how to estimate the utility of an ordering and 2) how to efficiently select the best ordering from among the $n!$ contenders.

The utility of an ordering is the sum of the constituent jobs' scores and a given job's score is a known function of completion time. Thus, the problem of estimating an ordering's utility reduces to estimating the completion time of all the jobs in that ordering. A job's completion time depends on three factors:

1. How old the job was when it arrived at the current machine,
2. How long the job will wait in this machine's queue given the considered ordering, and
3. How much additional time the job will take to complete after it leaves this machine.

The first factor is known and the second factor is easily computed given the speed of the machine and a list of the jobs preceding this one in the ordering. The third factor is not known but can be estimated using machine learning. In fact, the values we want to know are exactly the same as those Q-routing learns. Hence, if the scheduler we place in each machine is coupled with a Q-router, no additional learning is necessary. We can look up the entry in the Q-table that corresponds to a job of the given type. Note that this estimate improves over time as the Q-router learns.

Once we can estimate the completion time of any job, we can compute the utility of any ordering. The only challenge that remains is how to efficiently select a good ordering from among the $n!$ possibilities. Clearly, enumerating each possibility is not computationally feasible. If we treat this task as a search problem, we could use any of a number of optimization techniques (e.g. hill climbing, simulated annealing, or genetic algorithms). However, these techniques also require significant computational resources and the performance gains offered by the orderings they discover are unlikely to justify the CPU time they consume, since the search needs to be performed each time a new job arrives. Given these constraints, we propose a simple, fast heuristic called the *insertion scheduler*. When a new job arrives, the insertion scheduler does not consider any orderings that are radically different from the current ordering. Instead, it decides at what position to insert the new job into the current ordering such that utility is maximized. Hence, it needs to consider only n orderings. While this restriction may prevent the insertion scheduler from discovering the optimal ordering, it nonetheless allows for intelligent scheduling of jobs, with only linear computational complexity, that exploits learned estimates of completion time.

Results

To evaluate our routing and scheduling methods, we ran a series of experiments on the example network described above. Figure 3 shows the result of the first set of experiments, in which our three routing methods — random routing, heuristic routing, and Q-routing — are each employed in conjunction with a FIFO scheduler in simulations that run for 20,000 timesteps. In each case, the simulation runs for another 20,000 "warmup" steps before timestep #0 to get the network load up to full capacity before tallying scores. In the case of Q-routing, a random router is used during the warmup steps; Q-routing is turned on and begins training only at timestep #0. At timestep #10,000, a network catastrophe is simulated in which Web Server #1, Mail Server #1, and Database #3 simultaneously go down and remain down for the duration of the run. At any point in the simulation, the score for each method represents a uniform moving aver-

age over the scores received for the last 100 completed jobs. The scores are averaged over 25 runs.

The graph clearly indicates that routing randomly is dramatically suboptimal. The heuristic router, which routes in proportion to the speed of its neighbors, performs much better. Using a t-test, we verified that the heuristic router's advantage is significant with 95% confidence after timestep #1,000. The Q-routing method, whose initial policy lacks even the primitive load balancing of the random router, starts off as the worst performer but improves rapidly and plateaus with a higher score than the heuristic. This difference is significant with 95% confidence after timestep #4,000. Both the Q-router and the heuristic router are able to respond gracefully to the loss of machines in the middle of the simulation (the reason for the heuristic router's robustness is discussed in the following section); the Q-router maintains its superiority afterwards. The random router, however, suffers a major loss of performance from the sudden change in network configuration.

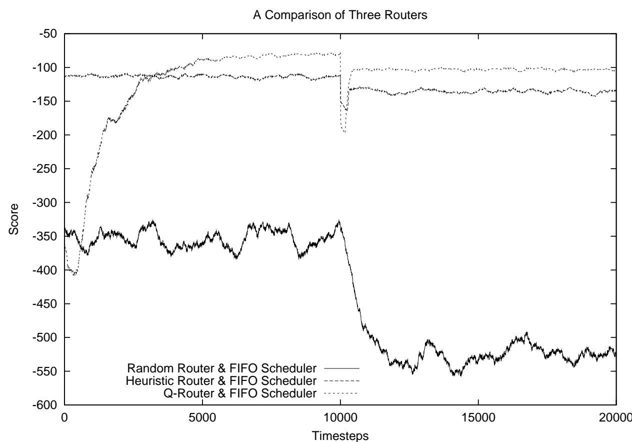

Figure 3: A comparison of three routing methods. A network catastrophe occurs at timestep #10,000.

Figure 4 shows the results of a second set of experiments. These experiments are identical to the first except that three different schedulers—the FIFO scheduler, the priority scheduler, and the insertion scheduler—are tested, all in conjunction with our best router, the Q-router.

The FIFO scheduler, which ignores complications that arise from the utility functions, performs relatively poorly, while the priority and insertion schedulers fare much better. Though the lines appear close in the graph, the insertion scheduler, by capitalizing on the information learned in each machine's Q-table, obtains scores that are approximately 20% higher than those of the priority scheduler. A t-test verified that these differences are significant with 95% confidence after timestep #4,000. Since all of these runs use Q-routing, they all recover from the network catastrophe at timestep #10,000.

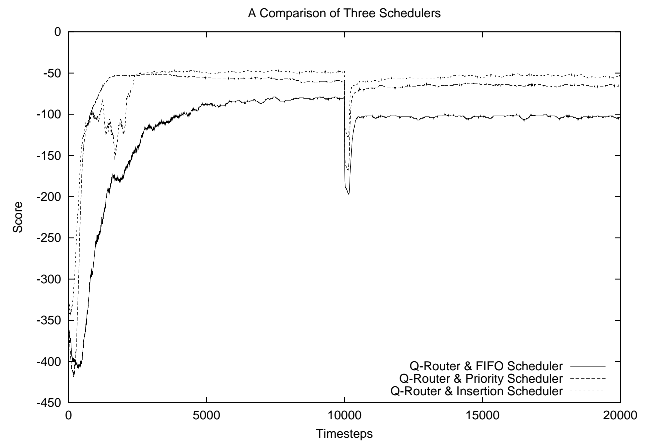

Figure 4: A comparison of three scheduling methods. A network catastrophe occurs at timestep #10,000.

Discussion

Our experimental results indicate clearly that machine learning methods offer a significant advantage in optimizing the performance of complicated networks. Both the router and scheduler placed in each machine benefit substantially from the time-to-go estimates discovered through reinforcement learning. Furthermore, the best performance is achieved only by placing intelligent, adaptive agents at more than one level of the system: the Q-router and the insertion scheduler perform better together than either could apart. Hence, they benefit from a sensible division of optimization tasks; the router focuses on routing jobs efficiently and balancing load throughout the network while the scheduler focuses on prioritizing those jobs whose effect on the score will be most decisive.

One of the chief advantages of the reinforcement learning techniques we employ is that they learn on-line, while the system is running. Unlike off-line methods, our system is never locked in to a specific policy but can continually adapt to changes in network topology or shifts in user demand. The superior recovery of Q-routing after machines have been ablated, when compared to random routing, verifies the worth of this feature. However, Q-routing, while continuing to get the best performance, does not recover more gracefully than the heuristic router, which uses no learning at all. Though initially surprising, this result makes sense after a careful consideration of the heuristic policy. Because the heuristic router does not learn, nothing in its policy is tuned to the current status of the network. Hence, adapting to changes is easier. So long as the heuristic router gets updated information about the state of its neighbors (which can be provided via Load Updates without learning), its policy remains viable. Since Q-routing's policy depends highly on the current configuration, it faces a greater task of adaptation when that configuration changes. In this light, it is not surprising that the heuristic router suffers little degradation in performance after a sudden system change. Instead, it is remarkable that our system, despite a greater burden to adapt, is able to recover just as robustly.

Future Work

In ongoing research, we plan to investigate new ways of applying machine learning methods to further automate and optimize networks like the one studied in this article. In particular, we hope to automate the decision of how frequently machines should send updates to their neighbors. Both load updates and Q-updates are more useful if they are sent more often; however, both kinds of updates also tax precious network bandwidth. Rather than finding the balance between these two factors through manual experimentation, we would like to devise a network intelligent enough to determine optimal update frequencies without human assistance.

In addition, we would like to use machine learning to determine what network topology gives optimal performance. In the research presented here, network performance is optimized given a certain network configuration; the structure of the network is not within the learner's control. We hope to develop a system in which machine learning helps determine the most efficient structure of the network when it is initially designed, when it needs to be upgraded, or when it is repaired.

Our on-going research goal is to discover, implement, and test machine learning methods in support of autonomic computing at all levels of a computer system. Though this initial work is all in simulation, the true measure of our methods is whether they can impact performance on real systems. Whenever possible, our design decisions are made with this fact in mind. Ultimately we plan to implement and test our autonomic computing methods, such as Q-router and insertion scheduler, on real computer systems.

Conclusion

The three main contributions of this article are:

1. A concrete formulation of the autonomic computing problem in terms on the representative task of enterprise system optimization.
2. A new vertical simulator designed to abstractly represent all aspects of a computer system. This simulator is fully implemented and tested. It is used for all of the experiments presented in this paper.
3. Adaptive approaches to the network routing and scheduling problems in this simulator that out-perform reasonable benchmark policies.

The research presented in this article indicates that machine learning methods can offer a significant advantage for routing and scheduling jobs on complex networks. It also provides evidence of the value of combining intelligent, adaptive agents at more than one level of the system. Together these results provide hope that machine learning methods, when applied repeatedly and in concert, can produce the robust, self-configuring, and self-repairing systems that tomorrow's computing needs will demand.

Acknowledgments

We would like to thank IBM for a generous faculty award to help jump-start this research. In particular, thanks to Russ Blaisdell for valuable technical discussions and to Ann Marie Maynard for serving as a liaison. This research was supported in part by NSF CAREER award IIS-0237699. Finally, we would like to thank Gerry Tesauro for his insightful suggestions about implementing Q-routing.

References

Boyan, J. A., and Littman, M. L. 1994. Packet routing in dynamically changing networks: A reinforcement learning approach. In Cowan, J. D.; Tesauro, G.; and Alspector, J., eds., *Advances in Neural Information Processing Systems*, volume 6, 671–678. Morgan Kaufmann Publishers, Inc.

Brachman, R. J. 2002. Systems that know what they're doing. *IEEE Intelligent Systems* 17(6):67–71.

Caro, G. D., and Dorigo, M. 1998. AntNet: Distributed stigmergetic control for communications networks. *Journal of Artificial Intelligence Research* 9:317–365.

Clark, D. D.; Partridge, C.; Ramming, J. C.; and Wroclawski, J. 2003. A knowledge plane for the internet. In *Proceedings of ACM SIGCOMM*.

Itao, T.; Suda, T.; and Aoyama, T. 2001. Jack-in-the-net: Adaptive networking architecture for service emergence. In *Proc. of the Asian-Pacific Conference on Communications*.

Kephart, J. O., and Chess, D. M. 2003. The vision of autonomic computing. *Computer* 41–50.

Stone, P., and Veloso, M. 1999. Team-partitioned, opaque-transition reinforcement learning. In Asada, M., and Kitano, H., eds., *RoboCup-98: Robot Soccer World Cup II*. Berlin: Springer Verlag. Also in *Proceedings of the Third International Conference on Autonomous Agents*, 1999.

Sutton, R. S., and Barto, A. G. 1998. *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press.

Watkins, C. J. C. H. 1989. *Learning from Delayed Rewards*. Ph.D. Dissertation, King's College, Cambridge, UK.