In *Engineering Applications of Artificial Intelligence*,
17(7):855-869, 2004.

CORRECTED VERSION: See Errata Section

Adaptive Job 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}

June 23, 2005

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 job routing and CPU scheduling in the networks we simulate. Our experimental results verify that methods using machine learning outperform reasonable heuristic and hand-coded approaches on example networks designed to capture many of the complexities that exist in real systems.

Keywords: Autonomic Computing, Reinforcement Learning, Q-Learning, Routing, Scheduling.

1 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 to large companies, are prime examples of this phenomenon. Nonetheless, most computer systems today are still built to rely on static configurations and can be installed, configured, and re-configured only 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.

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 computer systems to optimize their own performance without human assistance, they will need to learn from experience. To continually adjust in response to a dynamic environment, they will need the adaptability that only machine learning can offer.

Fully automating the maintenance and optimization of a large computer system via machine learning methods is a staggering challenge. If it is not achievable today, what short-term goals should we set to maximize the likelihood that it will be achievable tomorrow? One approach is bottom-up: instead of optimizing an entire system, we can optimize its individual components. Though the study of autonomic computing is still in its early stages, there has already been a lot of preliminary progress with the bottom-up approach (see Section 7 for a thorough survey of related work). For example, many researchers have used machine learning to optimize network routing (Boyan & Littman, 1994; Di Caro & Dorigo, 1998; Clark et al., 2003; Itao et al., 2001). In addition, Gomez et al. (2001) used neuroevolution to optimize dynamic resource allocation on a chip multiprocessor. Chen et al. (2004) used decision trees to diagnose system failures and Mesnier et al. (2004) used decision trees to classify different file types and therefore improve disk performance. Furthermore, Brauer and Weiss (1998) used reinforcement learning methods to optimizing scheduling of tasks on multiple machines.

The bottom-up approach is appealing because optimizing individual components is much more feasible than optimizing entire systems. Eventually, these components can be assembled into an autonomic system that should perform better than manually configured ones. However, such an approach fails to address the effects of interactions between various components and does not capitalize on opportunities to optimize at a system-wide level. Therefore, we propose a top-down approach to developing autonomic systems. Our emphasis is on optimizing the entire system by developing autonomic components that work well, not just independently, but in concert with other such components.

Since doing so in a real, full-fledged enterprise system is not currently feasible, we introduce in this article a high-level simulator designed to facilitate the study of machine learning in enterprise systems. Our simulator captures some of the key complexities that make system-wide autonomic computing a challenge, while abstracting away the low-level details that currently make it impractical to create fully autonomic systems on real hardware.

In addition to introducing this new tool, we present machine learning approaches to optimizing routing and scheduling in the networks represented in the simulator. Our results, that adaptive routers and schedulers can do better together than either can individually, validate the notion of a top-down approach to autonomic computing.

The remainder of this article is organized as follows. Section 2 provides background on our network simulator and Section 3 details our methods for optimizing routing and scheduling. Section 4 explains our experimental framework and Section 5 presents the results of those experiments. Section 6 discusses the implications of these results, Section 7 reviews related work, and Section 8 highlights some opportunities for future work.

2 Simulation

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., to our knowledge there is no 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 is very general purpose and can be used to represent many different kinds of networks. For example, Figure 1 depicts a commercial enterprise system in which a set of users use a web interface to check their mail or query a database.

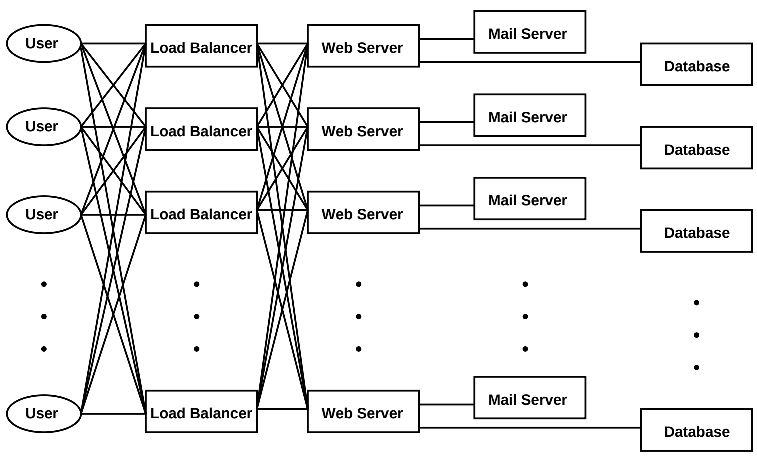


Figure 1: An example of a network implemented in our simulator; ovals represent users and rectangles represent machines; the lines between them represent links that allow communication of jobs or other packets.

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. In Figure 1, users forward their requests to a Load Balancer which selects a Web Server to handle the job. The Web Server forwards the job to a Mail Server or Database as appropriate. When completed, the job is sent back to the user who created it.

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. Completing steps uses the machine's CPU time so each machine must have an algorithm for allocating CPU time among the jobs currently in its possession. If a machine cannot complete the next step of a given job, it must look for a neighboring node that can. If several neighbors qualify, the machine must make a routing decision, i.e. it must try to determine which of the contending neighbors to forward the job to so as to optimize performance of the whole system. An intelligent agent can be used to control a machine and make decisions about how to allocate CPU time and route jobs.

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. For example, a user who wishes to buy something off a web site might create a "purchase job." This job might include steps such as accessing the customer database, confirming credit card information, and generating an order confirmation. 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.

In the example shown in Figure 1, Mail Jobs must travel to a Web Server, a Mail Server, and back to a Web Server before returning to the user. Database Jobs must travel to a Web Server, a Database, and back to a Web Server before returning to the user. The goal of the agents controlling the system is to process these user requests as efficiently as possible.

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 their 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.

2.1 Load Updates

Each machine periodically (in our case, 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. However, the very presence of the update is also important information: if a machine does not receive any updates from a given neighbor for a certain period of time (ten timesteps), it concludes that that neighbor has gone down and will no longer route any jobs to it until it receives another update. The system has a strong incentive to quickly detect when machines go down, since any jobs routed to a down machine receive a stiff penalty. For the purposes of scoring, such jobs are given a completion time of 500 timesteps, though in fact they are never actually completed. Since Load Updates are communicated via packets, they incur real network traffic overhead in the simulator. As long as they are not too frequent, including them as a routine occurrence is not unrealistic.

2.2 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 order to capture these complexities, the simulator allows different utility functions for each user or job type (Walsh et al., 2004). Each utility function can be any monotonically decreasing function that maps a job's completion time to its utility. Hence, the goal of the agents controlling the network is not to minimize average completion time, but to maximize the cumulative utility over all the jobs it is asked to process.

3 Method

In this section, we present our approach to developing intelligent routers and schedulers for networks like the ones shown in Figure 1. 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.

3.1 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 completed, 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 four ways of addressing this modified routing problem: a random method, two reasonable heuristic methods, and Q-routing, a method based on reinforcement learning.

3.1.1 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:

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

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.

3.1.2 Speed-Based Router

Without the aid of learning techniques or global information about the network, a 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 $c \in C$,

$$P_c = \frac{speed(c)}{\sum_{c' \in C} speed(c'))} \tag{2}$$

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.

3.1.3 Load-Based Router

Another heuristic approach to routing is to utilize information about the load on neighboring nodes, received in Load Updates. In this case, the router always routes to the qualifying neighbor with the lowest currently estimated load. Hence, for each $c \in C$,

$$P_c = \begin{cases} 1, & if\ load(c)\ \leq\ load(c')\ for\ all\ c' \in C \\ 0, & otherwise \end{cases} \tag{3}$$

If the capacity of the neighboring machines is a critical factor in the system's performance, then load information is likely to be highly useful and the Load-Based Router will perform very well. However, if the system's bottleneck is not adjacent to the machine making a routing decision, then its neighbors will often have no load and this heuristic will perform identically to the Random Router.

There are many other feasible routing heuristics besides those presented here (e.g. considering both load and speed when routing). However, all such heuristics must make their decisions based only on information about immediate neighbors, which may or may not be a useful guide to efficient routing. By contrast, a machine using Q-routing, presented below, can learn to route well even when the critical parts of the networks are not adjacent to it.

3.1.4 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 reinforcement learning modules are inserted into each node of the network. Reinforcement learning (Sutton & Barto, 1998) agents attempt to learn effective control policies by observing the positive and negative rewards they receive from behaving in different ways in different situations. From this feedback, reinforcement learning agents learn a *value function*, which estimates the long-term value of taking a certain action in a certain state. Once the value function is known, deriving an effective policy is trivial: in each state the agent simply takes the action that the value function estimates will reap the greatest long-term reward.

In Q-routing, the agents do not attempt to maximize a reward function but instead to minimize a *time-to-go function*, which estimates how long a given packet will take to complete if it is routed to a particular neighbor. Each node x maintains a table of estimates about the time-to-go of different types of packets. 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:

$$t = min_{z \in Z} Q_y(d, z) \tag{4}$$

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:

$$Q_x(d, y) = (1 - \alpha)Q_x(d, y) + \alpha(q + s + t) \tag{5}$$

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 routing problem, we must change the state features on which learning is based. Instead of containing simply the job's destination, the Q-tables contain three features that indicate in what general direction the job is headed (and therefore what machine resources it will likely tax if routed in a particular way):

- 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 to be reasonable through informal experimentation.

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.

3.2 Scheduling

The routing techniques discussed above all attempt to distribute load on the system evenly so as to minimize the time that passes between the creation and completion of a job. Doing so correctly plays an important role in overall system performance, but it is not the only factor. Our goal in this article, and the point of introducing a high-level vertical simulator, is to investigate the possibility of employing autonomic elements at more than one level of the system. It is with this goal in mind that we attempt to couple the routing mechanisms already described with schedulers, which must determine how to most efficiently allocate a given machine's CPU cycles.

Because our goal is to maximize overall utility, according to the utility functions given for each user, optimizing routing alone would not be optimal. The routing methods presented in this paper attempt to minimize the completion time of a given packet. However, 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 a 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 one user are scored according to the function $U(t) = -t$ while jobs created by another user scored according to the function $U(t) = -100t$. In this case, the latter user's jobs are vastly more important. Clearly, a network that devotes as much of its capacity towards the former user's jobs will be very sub-optimal.

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 while $t \leq 50$ 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. Since current reinforcement learning methods scale poorly to large action spaces, such a representation would render our approach intractable.

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. In the following subsections, we present two simple scheduling heuristics and introduce a new technique called the *insertion scheduler*, which utilizes the time-to-go estimates contained in the router's Q-table to assess a job's priority.

3.2.1 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:

$$J_l = \{j \in J \mid time(j) \leq time(j'), \forall j' \in J\} \tag{6}$$

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

3.2.2 Priority Scheduler

An alternative heuristic that does address these concerns is a *priority scheduler*, which is similar to multilevel feedback queues (Tanenbaum, 2001). 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:

$$J_l = \{j \in J \mid time(j) \leq time(j') \wedge priority(j) \geq priority(j'), \forall j' \in J\} \tag{7}$$

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 slope of their utility functions. 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.

3.2.3 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.

3.2.4 Sample Scheduler

The insertion scheduler uses a heuristic to select which n queue orderings to examine. In order to test the value of this heuristic, we developed another, similar scheduler that randomly selects which orderings to test. The sample scheduler estimates the utility of each ordering it examines in exactly the same manner as the insertion scheduler. It also selects the ordering that produces the highest estimated utility, just like the insertion scheduler. The only difference is that it selects n orderings from the $n!$ possibilities randomly instead of using the heuristic described above. If the insertion scheduler's heuristic is worthwhile, we should expect it to outperform the sample scheduler.

4 Experimental Framework

Our experiments test all of the above methods on three different networks, each of which simulates a commercial enterprise system serving two companies whose employees generate requests to access mail and database programs. In this section, we detail the features that all three networks have in common: the job types, the mechanism for job creation, and the utility functions that are used by each user.

4.1 Job Types

In all three networks, 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
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. Hence, in order to be completed, each Mail Job must travel along a path that includes the following stops: 1) visit a Web Server, 2) visit a Mail Server, 3) return to a Web Server, 4) return to the user who created it.

Each Database Job consists of the following three steps:

1. Web Step, work = 50
2. Database Step, work = 200
3. Web Step, work = 50

Since only a Database can complete a Database Step, the path of a Database Job must include: 1) visit a Web Server, 2) visit a Database, 3) return to a Web Server, 4) return to the user who created it.

4.2 Job Creation

All three networks use the following mechanism for determining when users create new jobs. At each timestep, the users create enough jobs to bring the total number of incomplete jobs up to a threshold, set to 100 in our experiments. Each newly created job has an equal probability of being created by each user in the system. 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.[1]

4.3 Utility Functions

In order to capture the complexities raised by users of differing importance, we assign different utility functions, shown in Figure 2, to our two users. The utility functions are used in all three networks. Jobs created by Company #1 are scored according to the following function:

$$U(t) = \begin{cases} -10t, & if\ t<50 \\ -t/10-495, & otherwise \end{cases} \tag{8}$$

where t is the job's completion time. By contrast, jobs created by Company #2 are scored by the function

$$U(t) = \begin{cases} -t/10, & if\ t<50 \\ -10t+495, & otherwise \end{cases} \tag{9}$$

The crucial feature of these metrics is that they do not have constant slope. Hence, the change in utility that the system reaps for reducing the response time of a job is not constant. As explained above, this feature gives rise to the complications that make intelligent scheduling non-trivial. The point at which each function changes slope was chosen so as to lie in the region of the x-axis that corresponds to typical completion times for jobs in our networks. If the threshold were nowhere near this region, then the utility functions would de facto have constant slope, yielding a much easier scheduling problem (i.e. one that the priority scheduler could handle optimally). The utility functions were not tuned to this problem in any other way. 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.

[1] This section is corrected from the orginal paper. See the Errata Section for more details.

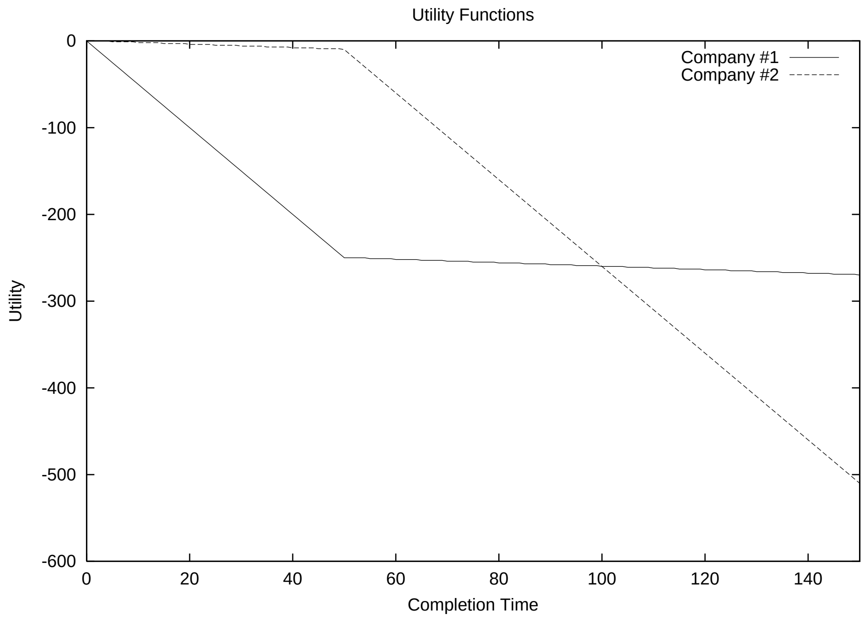


Figure 2: Utility Functions for the two companies.

5 Results

In this section we describe experiments conducted on three different networks comparing the various routing and scheduling methods described above. The first two networks we use are designed to establish proof-of-concept for our methods. Hence, they are the simplest networks we could construct that exhibit the complications our methods attempt to address. The third network is larger and designed to provide some confidence that these methods will scale up.

Each experiment runs for 20,000 timesteps. Each simulation is preceded by an additional 5,000 "warmup" steps 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. The purpose of the warmup is to ensure that the network is at full capacity before learning begins. Doing so helps distinguish changes in performance due to discovering superior policies from those due to load building up in an initially empty network. At any point in the simulation, the score for each method represents a uniform moving average over the scores received for the last 100 completed jobs. The scores are averaged over 20 runs.

For each network, we present the results of pairing each routing method with a FIFO scheduler and pairing each scheduling method with a Q-router. For the sake of clarity, we do not present the other possible pairs though our experiments confirm that those combinations perform worse than the best methods.

At timestep #10,000 in each experiment, a system catastrophe is simulated in which the speed of a few critical machines is cut in half. Since our learning methods are designed to work on-line, we expect them to adapt rapidly to changes in their environment, a feature tested by these simulated catastrophes. The details of which machines are effected in each experiment are explained below.

In all of the results presented below, assertions of statistical significance are based on a student's t-test (at 95% confidence) comparing the scores of the two given methods immediately before the catastrophe (at timestep #10,000) and at the end of the experiment (at timestep #20,000).

5.1 Network #1

Figure 3 depicts the network used in our first experiment. In this network, the Web Servers are relatively slow. Hence, they act as a bottleneck to system performance. Note that the machines that must make important routing decisions (the Load Balancers), are neighbors of the machines that are critical to system performance. Hence, the speed-based and load-based routers, which rely on information about their neighbors, can perform very well.

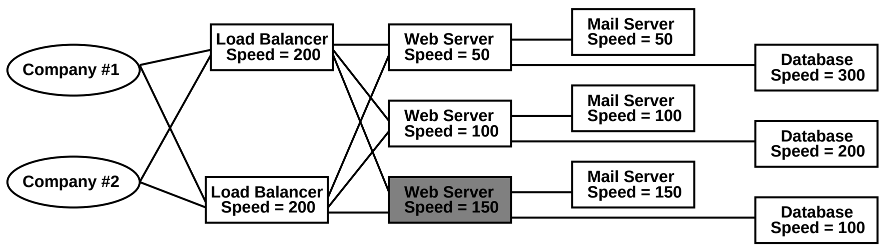


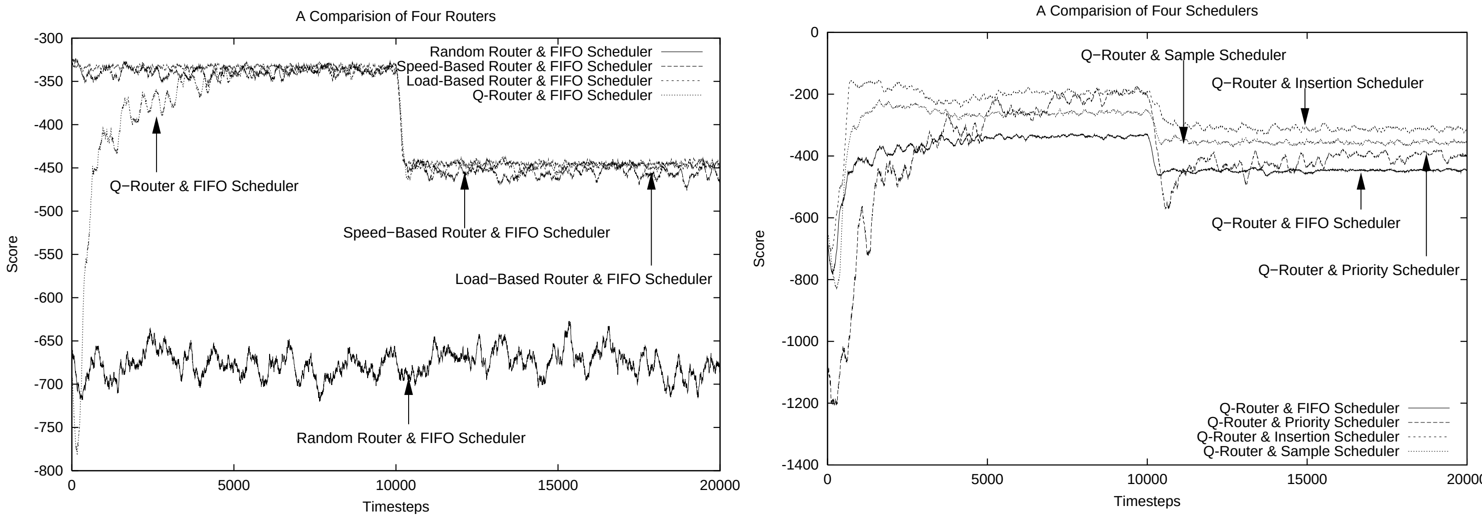
Figure 3: Network #1, in which the Web Servers act as a bottleneck to system performance. Ovals represent users and rectangles represent machines; the lines between them represent links that allow communication of jobs or other packets. The speed associated with each machine represents the number of CPU cycles it can execute in one turn. The gray box indicates a machine whose speed is reduced by half during a system catastrophe.

Figure 4a compares the performance of all four routing methods when paired with a FIFO scheduler and applied to Network #1.

The graph clearly demonstrates that routing randomly is dramatically suboptimal. The gap in performance between the random router and its competitors is statistically significant. The Q-routing method initially performs as poorly as the random router but improves rapidly. It performs erratically while exploring different policies but quickly plateaus at the same level as the speed-based and load-based routers. In this network, it is not surprising that Q-routing does not outperform these heuristics. Since the system's bottleneck occurs in machines that neighbor the load balancers, the heuristic methods are able to route efficiently based on the speed and load information they receive.

At timestep #10,000, a system catastrophe is simulated in which the speed of the fastest Web Server (indicated by a gray box in Figure 3) is cut in half. Since the Web Servers act as bottlenecks in this network, the catastrophe causes a drop in performance for all methods except the random router, which already routes so inefficiently that the catastrophe does not further narrow its bottleneck.

Figure 4b compares the performance of all four schedulers when paired with the Q-router. In this case, using the insertion scheduler yields a statistically significant performance boost



(a) Routers (b) Schedulers

Figure 4: Results from Network #1. In (a), all four routing methods are paired with a FIFO scheduler. In (b), all four scheduling methods are paired with Q-routing.

over all the other schedulers. This result suggests that even when Q-routing does not itself improve performance, it is worth doing because the values it learns can be successfully exploited in scheduling. The sample scheduler, which evaluates as many orderings as the insertion scheduler, performs significantly worse, which supports our claim that the insertion scheduler is a useful heuristic.

5.2 Network #2

Figure 5 depicts the network used in our second experiment. It is identical to Network #1 except that the speed of the Web Servers has been significantly increased. As a consequence, the system's bottleneck moves from the Web Servers to the Mail Servers and Databases. Because of this change, the local information that the heuristic routers rely on is no longer useful. The Load Balancers can see only the Web Servers and their speeds and loads are no longer critical to system performance.

Figure 6a compares all four routing methods on Network #2. Since performance now depends on machines that are not directly visible to the Load Balancers, it is not surprising that Q-routing outperforms the other methods in this network. Due to their increased speed, the Web Servers never have load accumulated in their queues, which causes the load-based router to perform just like a random router. The speed-based router actually performs worse than random because it is misled by the irrelevant speeds of the Web Servers and attempts a counterproductive load balancing.

In this network, the catastrophe at timestep #10,000 involves cutting in half the speed of the fastest Mail Server and Database (indicated by gray boxes in Figure 5). Since the heuristic routers were underloading the faster machines before the catastrophe, the reduction

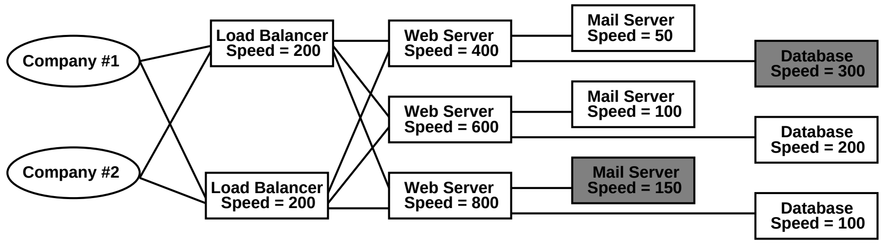


Figure 5: Network #2, in which the Mail Servers and Databases act as a bottleneck to system performance. Ovals represent users and rectangles represent machines; the lines between them represent links that allow communication of jobs or other packets. The speed associated with each machine represents the number of CPU cycles it can execute in one turn. Gray boxes indicate machines whose speed is reduced by half during a system catastrophe.

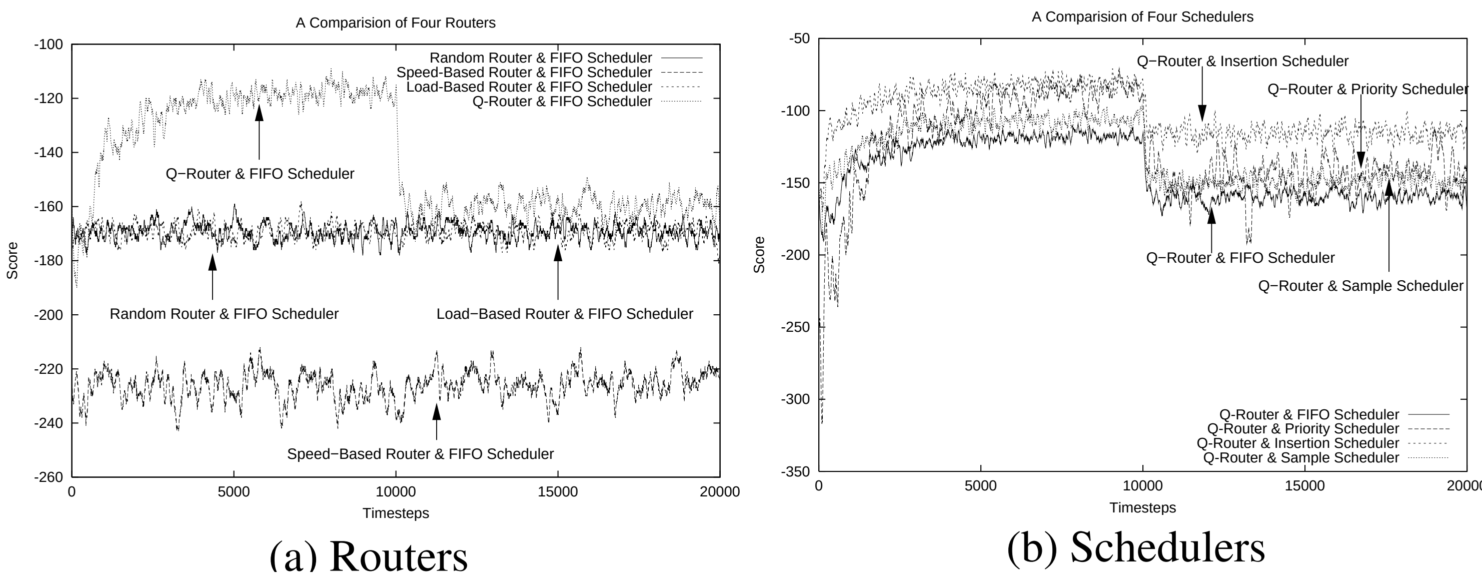


Figure 6: Results from Network #2. In (a), all four routing methods are paired with a FIFO scheduler. In (b), all four scheduling methods are paired with Q-routing.

in speed does not effect them. With fewer resources available, the performance of Q-routing inevitably degrades, though it is able, through on-line adaptation of its policy, to retain a small advantage over the other methods.

Figure 6b pairs all four schedulers with the Q-router to evaluated their performance on Network #2. As above, the insertion scheduler provides a statistically significant boost in performance over the other methods. The relatively weak scores of the sample scheduler further confirm the usefulness of the insertion scheduler's heuristic for selecting orderings to evaluate.

5.3 Network #3

Networks #1 and #2 are intended to provide proof-of-concept for the advantages of learning-based routing and scheduling. To demonstrate that these advantages scale up, we also tested our various methods on the larger network depicted in Figure 7. In this network, the same two Load Balancers must choose between nine Web Servers, each of which is connected to its own Mail Server and Database. As in Network #2, the Web Servers have enough CPU cycles that the bottleneck to system performance lies in the Mail Servers and Databases. In order to keep this larger network busy, the users are allowed to have 500 incomplete jobs at any time (as opposed to the 100 allows in Networks #1 and #2).

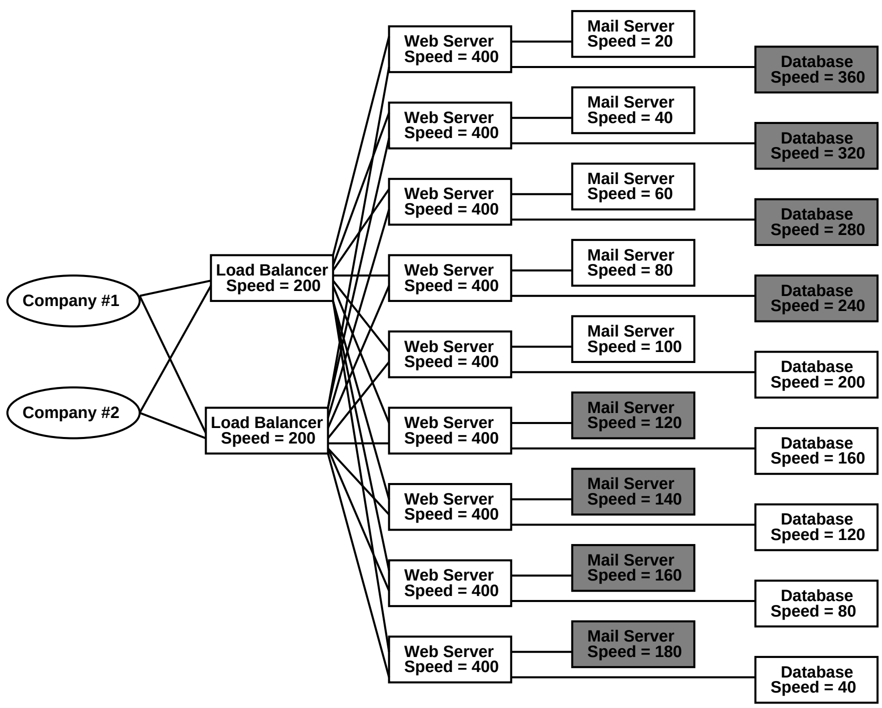
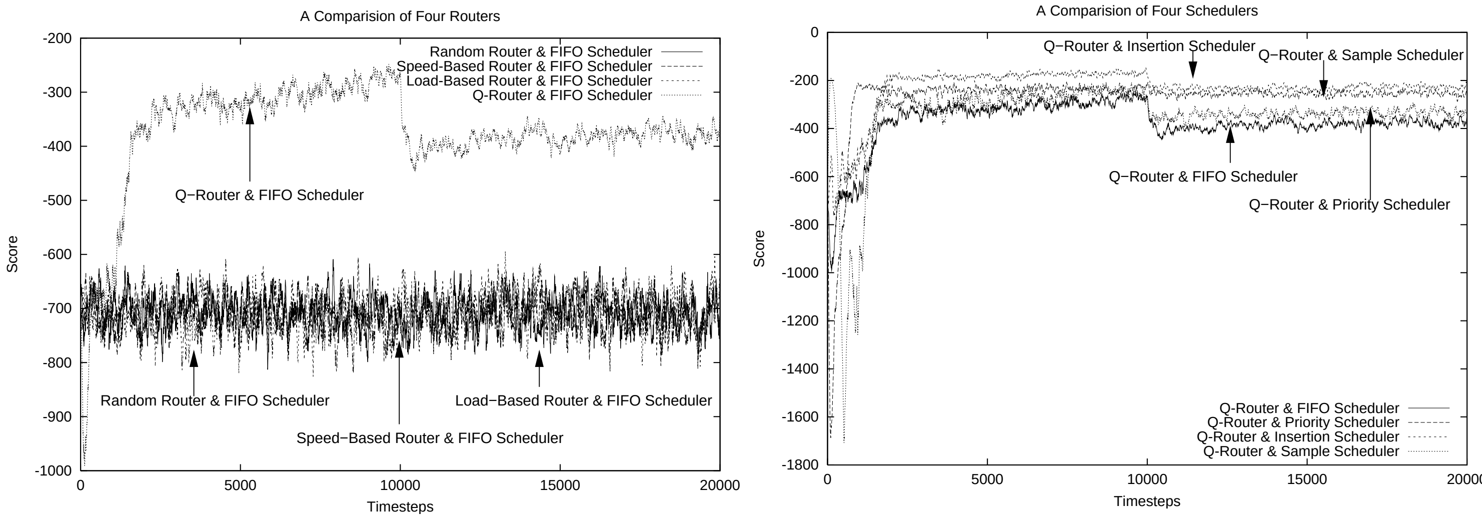


Figure 7: Network #3, in which the Load Balancers must choose between nine Web Servers instead of three. Ovals represent users and rectangles represent machines; the lines between them represent links that allow communication of jobs or other packets. The speed associated with each machine represents the number of CPU cycles it can execute in one turn. Gray boxes indicate machines whose speed is reduced by half during a system catastrophe.

Figure 8a compares the performance on this larger network of all four routing methods when paired with a FIFO scheduler. Since the Web Servers all have the same speed in this network, the speed-based router performs similarly to the random and load-based routers. As in Network #2, Q-routing achieves by far the best performance, obtaining a statistically significant improvement over the other methods.

The catastrophe that occurs at timestep #10,000 consists of cutting in half the speed of the four fastest Mail Servers and Databases (indicated by gray boxes in Figure 7). Though its performance inevitably degrades, Q-routing recovers gracefully by adjusting its policy



(a) Routers (b) Schedulers

Figure 8: Results from Network #3. In (a), all four routing methods are paired with a FIFO scheduler. In (b), all four scheduling methods are paired with Q-routing.

on-line in response to environmental changes.

Figure 8b compares all four schedulers on Network #3 by pairing them with Q-routing. As before, the insertion scheduler scores the highest, yielding a statistically significant improvement over FIFO scheduling, the fixed heuristic of the priority scheduler, and the random evaluations of the sample scheduler.

6 Discussion

Our experimental results indicate clearly that machine learning methods offer a substantial advantage in optimizing the performance of computer 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.

This advantage is especially compelling in systems like Networks #2 and #3 in which the machines that make important routing decisions (the Load Balancers) are not direct neighbors of the machines that are most critical to system performance (the Mail Servers and Databases). In these scenarios, the information necessary to route efficiently is not directly available and a good policy can be discovered only by learning from experience. However, even when this is not the case, as in Network #1, and heuristic routing methods perform well, learning-based systems can still reap an advantage by exploiting Q-routing's time-to-go estimates to improve CPU scheduling. Furthermore, the success of the Q-router and the insertion scheduler on a larger system (Network #3) is cause for optimism that these methods will continue to be successful on networks of realistic complexity.

Note that while our simulator models the network overhead that our learning algorithms generate (in the form of Load Updates and Q-Updates), it does not model the computational cost of running them. This simplification is probably not significant for routing, since Q-routing's update rule requires only a handful of arithmetic operations for each job. However, scheduling algorithms can be significantly more expensive, which is why we are committed to finding fast heuristics for addressing this problem. For example, the insertion scheduler examines only n of the $n!$ possible queue orderings each time it must rearrange its jobs.

Since all of the results we present were obtained in simulation, it remains to be seen how these methods will perform in real systems. In particular, while our simulator strives to capture many of the intricacies that make adaptive routing and scheduling a challenge, it also glosses over many complicating aspects of real systems. For example, in our simulator the time required to complete a step is a deterministic function of the machine's speed, whereas in real systems it depends on the state of the machine's memory, disks, caches, etc. In addition, our simulator assumes that jobs never need resources on more than one machine simultaneously, thus avoiding the need to reason about locks, pin data in caches, etc. These, and many other, issues will need to be addressed before our methods will be deployable in real systems. However, we believe that our top-down, system-wide, perspective will play an essential role in the emerging field of autonomic computing.

7 Related Work

The study of autonomic computing is still in its early stages. Nonetheless, there already exists a broad body of work that relates, in terms of both methods and goals, to the efforts described here.

The method with the closest relationship to our approach is of course the original Q-routing technique (Boyan & Littman, 1994), of which our method is a close adaptation. A chief limitation of canonical Q-routing is that distinguishes between packets based only on their ultimate destination, which is insufficient in enterprise systems where jobs always return to their creator upon completion. In this article, we show that by expanding the learner's state representation, the principle behind Q-routing can be successfully applied to resource management tasks that do not exactly match the routing problem as traditionally posed.

In addition, there are many other approaches to network routing with the aid of machine learning (Di Caro & Dorigo, 1998; Clark et al., 2003; Itao et al., 2001). In particular, AntNet (Di Caro & Dorigo, 1998) uses an ant colony metaphor to create agents that concurrently explore the network and exchange information about it. Though it could be easily adapted to manage workloads on enterprise systems, it does not explicitly learn time-to-go estimates for each type of packet and hence its routing tables could not easily be exploited by CPU schedulers, as we have done with Q-routing.

The operating systems literature discusses many techniques for CPU scheduling, including multilevel feedback queues (Tanenbaum, 2001), which are similar to the priority scheduler discussed here. However, we know of no other work that exploits reinforcement learning methods to optimize CPU scheduling.

Beyond the scope of routing and scheduling, there is also considerable research devoted to using machine learning to optimize resource management in complex systems. For example, Brauer and Weiss (1998) use reinforcement learning methods to optimize scheduling of tasks on multiple machines. Gomez et al. (2001) use neuroevolution to optimize dynamic resource allocation on a chip multiprocessor. Abdezaher et al. (2002) present a technique for managing web servers that considers, as we do, jobs of differing utility, though their method uses only classical feedback control. Also, Yellin (2003) presents an algorithm for dynamically selecting among component implementations, though each individual implementation must still be manually configured. These methods are important contributions to the goal of optimizing individual parts of complex systems but they do not address our specific focus: the problems and possibilities of simultaneously optimizing multiple components from a system-wide perspective.

Whereas our research employs reinforcement learning techniques, other researchers have used supervised learning methods to address classification problems of interest to autonomic computing. Chen et al. (2004) use decision trees to diagnose system failures, an important problem. However, their solution does not address the question of how to deal with a failure once it is diagnosed, which would require either manual intervention or a reinforcement learning agent. Mesnier et al. (2004) use decision trees to classify different file types and therefore improve disk performance. Again, such a system is useful only if we know (or can get the computer to learn) what kind of disk behavior is best suited to each file type.

Finally, there is also research in autonomic computing that does not address machine learning at all, but instead redesigns system architectures to create new possibilities for adaptation. For example, Jann et al. (2003) present an architecture that allows dynamic movement of hardware resources across logical partitions without rebooting. They do not address how an intelligent agent might determine when resource reallocation should occur. Hence their work, while striving for different goals, fits cohesively with our approach: as they try to create opportunities for adaptability, we try to create agents that can exploit those opportunities.

8 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 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 the Q-router and insertion scheduler, on real computer systems.

9 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 simulator that we introduce facilitates the study of autonomic computing methods from a top-down perspective. Rather than simply optimizing individual components, we focus on optimizing the interactions between components at a system-wide level. The results presented here, in addition to demonstrating that machine learning methods can offer a significant advantage for job routing and scheduling, also validate this top-down approach. They provide evidence of the value of combining intelligent, adaptive agents at more than one level of the system. Together these results offer hope that machine learning methods, when applied repeatedly and in concert, can produce the robust, self-configuring, and self-repairing systems necessary to meet tomorrow's computing needs.

Errata

Section 4.2 originally stated that each user creates either 1 or 2 jobs per turn until a maximum threshold is reached. However, this description was in error. Due to an implementation bug, the experiments reported in this paper used a slightly different job creation model. On each turn, the users create however many jobs is necessary to bring the total number of incomplete jobs in the network up to the given maximum, which is 100 for Networks #1 and #2 and 500 for Network #3. Each newly created job has an equal probability of being created by Company #1 or Company #2. For Network #3, this fact is important, as the job creation model reported in Section 4.2 will not create enough jobs to cause significant network congestion.

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. Also, we would like to thank Gerry Tesauro for his insightful suggestions about implementing Q-routing and Emmett Witchel for his constructive comments on the feasibility of our model. Finally, we wish to thank Eugénio Oliveira and Luis Nunes for bringing to our attention the error reported above.

References

Abdelzaher, T., Shin, K. G., & Bhatti, N. (2002). Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, *13*.

Boyan, J. A., & Littman, M. L. (1994). Packet routing in dynamically changing networks: A reinforcement learning approach. *Advances in Neural Information Processing Systems* (pp. 671–678). Morgan Kaufmann Publishers, Inc.

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

Brauer, W., & Weiss, G. (1998). Multi-machine scheduling - a multi-agent learning approach. *Proceedings of the International Conference on Multi-Agent Systems* (pp. 42–48).

Chen, M., Zheng, A., Lloyd, J., Jordan, M., & Brewer, E. (2004). Failure diagnosis using decision trees. *Proceedings of The International Conference on Autonomic Computing (ICAC-04)*. To appear.

Clark, D. D., Partridge, C., Ramming, J. C., & Wroclawski, J. (2003). A knowledge plane for the internet. *Proceedings of ACM SIGCOMM*.

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

Gomez, F., Burger, D., & Miikkulainen, R. (2001). A neuroevolution method for dynamic resource allocation on a chip multiprocessor. *Proceedings of the INNS-IEEE International Joint Conference on Neural Networks* (pp. 2355–2361). IEEE.

Itao, T., Suda, T., & Aoyama, T. (2001). Jack-in-the-net: Adaptive networking architecture for service emergence. *Proceedings of the Asian-Pacific Conference on Communications.*

Jann, J., Browning, L. M., & Burugula, R. S. (2003). Dynamic reconfiguration: Basic building blocks for autonomic computing on IBM pSeries servers. *IBM Systems Journal*, *42*, 29–37.

Kephart, J. O., & Chess, D. M. (2003). The vision of autonomic computing. *Computer*, *36*, 41–50.

Mesnier, M., Thereska, E., Ellard, D., Ganger, G. R., & Seltzer, M. (2004). File classification in self-* storage systems. *Proceedings of The International Conference on Autonomic Computing (ICAC-04).* To appear.

Stone, P., & Veloso, M. (1999). Team-partitioned, opaque-transition reinforcement learning. In M. Asada and H. Kitano (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., & Barto, A. G. (1998). *Reinforcement learning: An introduction.* Cambridge, MA: MIT Press.

Tanenbaum, A. (2001). *Modern operating systems.* Englewood Cliffs, NJ: Prentice Hall.

Walsh, W. E., Tesauro, G., Kephart, J. O., & Das, R. (2004). Utility functions in autonomic systems. *Proceedings of the International Conference on Autonomic Computing.* To appear.

Watkins, C. J. C. H. (1989). *Learning from delayed rewards.* Doctoral dissertation, King's College, Cambridge, UK.

Yellin, D. M. (2003). Competitive algorithms for the dynamic selection of component implementations. *IBM Systems Journal*, *42*, 85–97.