Greed Is Not Enough: Adaptive load sharing in large heterogeneous systems

Abel Weinrib
Bell Communications Research
435 South Street
Morristown, NJ 07960

Scott Shenker
Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304

ABSTRACT

We consider the problem of job placement in load sharing algorithms for large heterogeneous distributed computing environments. We present simulation results on a simple model indicating that under heavy loads the usual policy of placing jobs where they will incur the shortest expected delay leads to inefficient system performance. Thus, purely greedy policies are not sufficient; we identify a simple threshold algorithm that does significantly better. We then introduce a novel adaptive algorithm whose performance is much closer to optimal.

1. Introduction

In contrast to the days when mainframes dominated computing, many users now rely on powerful personal workstations to serve their computational needs. These workstations are usually linked together through a local area network, and often there are some significantly faster compute servers on the network as well. The total computing power of these workstations and compute servers can be quite large, easily reaching hundreds of MIPS. Effective distributed computing environments should utilize this excess bulk processing power through load sharing mechanisms that enable overloaded computers to shift some of their workload to underutilized machines. These load sharing algorithms are currently a subject of much research interest (see, for example, [Eag86a,b, Her87] and references therein).

The problem is typically posed as a tradeoff between the decrease in execution delay realized by using a remote machine *versus* the transfer cost thereby incurred. Finding the optimal tradeoff is a hard problem, and its solution depends on the nature and speed of the underlying communication system (compare the radically different communication models in [Her87] and [Eag86a,b]). There is another critical issue in load sharing: that of where best to place a job even if transfer costs are negligible and exact knowledge of the global system state is assumed. In the case where all machines are identical, this choice is easy: use the machine with the fewest outstanding jobs. However, in heterogeneous distributed environments this simple rule does not suffice; choosing the optimal target machine is an unsolved problem.

The purpose of this paper is to investigate the issues involved in this choice of optimal target machine. Since we are not addressing the reduced execution delay *versus* transfer cost tradeoff that is the focus of most previous studies, we will consider the limit of free and instantaneous transfer of jobs, thereby removing the distinction between local and remote processing. We will first discuss a simple model of computation with N exponential servers, each having a (perhaps different) service rate μ_k with $k \in [1,N]$. Later, we will further simplify the model to have only two classes of servers; we then focus on the behavior of policies in the important crossover region where the system first starts to use the slow servers, and observe that the behavior in this region remains unchanged even if we let the number of slow servers become infinite. Each server has a local queue, and the number of jobs in each queue (including the job in service) is denoted by x_k. With our assumption of free and instantaneous job transfers, we can consolidate the system-wide arrival of jobs into a single Poisson stream of strength λ. Since the arrival and servicing of jobs are both memoryless processes, the instantaneous state of the system is completely characterized by the set of queue lengths $\{x_k\}$. The goal is to find a policy π that will tell us where to put newly arrived jobs. This policy will, in general, depend on the system parameters $\{\mu_k\}$ and λ in addition to the instantaneous state information $\{x_k\}$.

In this framework, the problem reduces to the more general problem of *joining the right queue* in a set of parallel queues with differing server rates (see the simplified schematic in Figure 1). As such, it has applications to many other problems of current

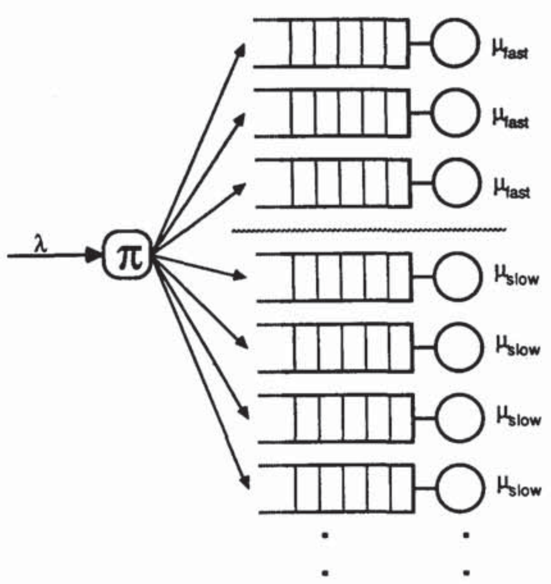

Figure 1. Model system with three fast servers of rate μ_{fast} and a large number of slow servers of rate μ_{slow}. The controller executes policy π to place jobs.

interest, such as routing in communication networks and scheduling in flexible manufacturing systems. The difference between applications typically lies in the amount of system and state information available when scheduling decisions are made. There is a vast literature on the general problem of *joining the right queue*, and we cannot attempt to do it justice here. The review articles [Sti85, 86] provide a good overview of the field and contain an extensive list of references.

In the next section we will discuss two optimization criteria, and their properties in the homogeneous and heterogeneous cases. In Section 3 we introduce the simple model that we believe captures the essence of the problem. In Section 4 we describe a number of dynamic policies for the model, observing that in the separable case (to be defined later) they reduce to threshold policies. These threshold policies fall between two limiting cases: never queue for a fast server, and queue until the expected delay for a fast server is the same as for a slow one (the greedy policy in which each job minimizes its own delay). Simulation data indicates, surprisingly, that never queueing is usually the better policy. Then, in Section 5, we introduce two adaptive policies that improve upon the performance of the *never queue* policy. Throughout the paper the graphs we present are based on simulation results for the performance of the various policies with the various choices of the model parameters. In Section 6 we briefly describe the simulation package. Finally, we provide a discussion of our results.

2. Individual and Social Optimization in Homogeneous and Heterogeneous Systems

It is important to clarify what we are trying to optimize. One possible goal is to minimize each individual job's expected delay (until completion). The policy that achieves this is quite straightforward. Since the quantity $(1+x_k)/\mu_k$ gives the expected delay the job will experience if sent to the k^{th} server, the individually optimal policy merely chooses the value of k that minimizes this quantity. We will call this policy the *shortest expected delay* (SED) policy. It is a *greedy* policy in that each job does what is in its own immediate best interest. However, for distributed load sharing, one is more likely to want a policy that minimizes the expected delay averaged over all jobs; such a policy is sometimes called the socially optimal policy.

The goals of individual optimization and social optimization do not always coincide [Nao69, Bel83]. This fact, which is somewhat counterintuitive, can be seen more clearly by noting that when a job chooses a server which minimizes the job's own delay, it does not consider the delay its presence will impose on future jobs that enter that queue. Socially optimal policies sometime require jobs to *sacrifice* themselves by going to a less than individually optimal server in order to allow the system to achieve better overall performance.

In the case where all servers are identical (equal μ_k's), the individually optimal SED policy, or the *join shortest queue* policy to which it reduces here, is also the socially optimal policy [Win77]. For nonexponential distributions of service times, [Whi86] has constructed examples where the *join shortest queue* policy is neither socially nor individually optimal. This may have ramifications for scheduling policies on real systems which have highly irregular service time distributions [Lel86].

While the load sharing literature usually treats the homogeneous case, reality often presents us with a heterogeneous set of servers. As distributed systems grow larger, the presence of different service rates will become increasingly common. Here the goals of individual and social optimization are often at odds. In fact, as we shall see later, the individually optimal policy of SED can, under moderately heavy loads, produce socially inefficient results. No general solution to the heterogeneous social optimization problem is known. While it is important to understand the structure of socially optimal algorithms [Lin84, Rub85], the exact solutions will depend in detail on all of the modeling assumptions, and hence may not easily generalize to more realistic systems. The goals of this paper are somewhat more modest: to find policies that produce good, but not necessarily optimal, results with minimal reliance on the exact knowledge of the system parameters. One can then hope that the insights gained from our simple model may also apply to real systems.

3. Model

Before constructing policies, we will simplify our model further; consider m_{fast} fast servers with service rate μ_{fast} and an infinite number of slow servers with service rate μ_{slow}. (We can, without loss of generality, always set $\mu_{slow}=1$ by redefining the time scale. From now on we shall assume this to be the case.) The rate of job arrival is given by λ. This set of assumptions produces the simplest nontrivial large heterogeneous system, and is not as unrealistic as it might first appear. One can think of the fast servers as a set of compute servers that are significantly faster than the typical workstation, of which there are very many. The question we are addressing is then: given that our goal is to achieve the social optimum of minimizing the mean delay averaged over all jobs, when is it better to use an idle workstation rather than to send the job to the more heavily loaded but faster compute server? (Note that, for this simplified model, there is never any queueing at the slow servers, so the model is equivalent to the control of arrivals to a set of identical parallel servers with penalties for rejecting jobs and for delays in the queues [Sti85].)

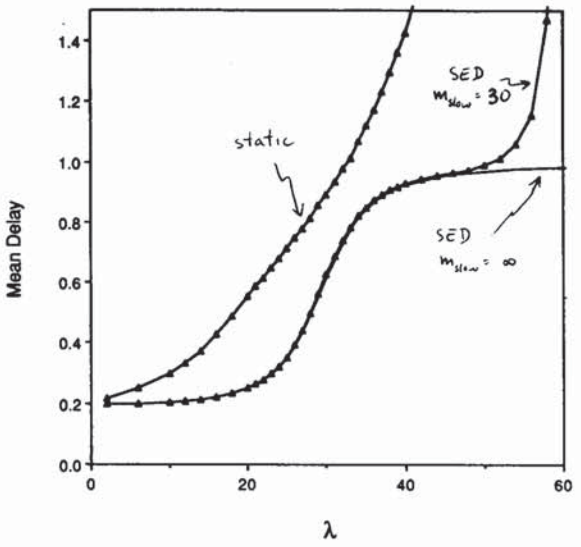

Figure 2. The shortest expected delay (SED) policy with $m_{slow}=30$ and $m_{slow}=\infty$ slow servers; $m_{fast}=6$. The server rates are $\mu_{fast}=5$ and $\mu_{slow}=1$, so that for the finite number of slow server case (with $m_{slow}=30$) the mean delay diverges at $\lambda=60$. We also display the mean delay for Bernouilli splitting, the optimal static random policy, for the finite case.

Figure 2 shows that the behavior of the system with a finite number of slow servers is very similar to that of a system with infinite slow servers. Figure 2 compares, for the SED policy, the average delay with $m_{slow}=30$ to that with $m_{slow}=\infty$ ($m_{fast}=6$ and $\mu_{fast}=5$). The curves are the same until $\lambda \approx m_{fast}\mu_{fast}+m_{slow}\mu_{slow}$, at which point the finite system's delay diverges. (The delay's insensitivity to the value of m_{slow} until the system is close to fully utilized is even more pronounced for larger systems, much as M/M/m systems increasingly resemble M/M/∞ systems as m gets large.) We are not addressing the extremely heavily loaded limit (see, e.g., [Fos78] for a study of this limit), where the delays diverge regardless of the load sharing policy, since this limit does not represent reasonable operating conditions. Instead, we will focus on the crossover region where one first needs to use the slow servers in order to provide enough power to serve the arriving jobs. This crossover occurs when $\lambda \approx m_{fast}\,\mu_{fast}$.

4. Policies

We will consider two main types of policies: *static* and *dynamic*, distinguished in part by the amount of information that is available. If no state information is available, then one can only use a static policy, where the decisions about where to send jobs are set in advance. The optimal randomized static policy is Bernoulli splitting [Eph80] where a job is assigned to a server with probability p_k. The result is to split the original source stream into N independent Poisson streams of strength $\lambda_k=p_k\lambda$. Bernoulli splitting chooses the p_k's so as to minimize the total system delay of these N M/M/1 queues subject to the constraint that $\sum\lambda_k=\lambda$. [Haj83] discusses static policies in which the splitting of the streams is done in a deterministic manner.

Dynamic policies make use of the state information $\{x_k\}$ in making decisions; in general, dynamic policies can always do better than static policies. See, for example, Figure 2 where the static Bernoulli split policy is compared to the *greedy* SED dynamic policy. Many load sharing situations lend themselves to dynamic policies because one does have available at least partial state information.

There are several classes of dynamic policies. *Separable* policies [Kri87] assume a model where each server computes a local cost or *toll* and the job is placed on the server with the minimal such cost. *Nonseparable* policies relax the independent cost constraint, allowing a decision to depend on the full set of state information. Nonseparable policies include separable ones as a special case, of course, and in general can be expected to perform better by using the extra information made available to them. In addition, in the next section we introduce two *adaptive* policies, one separable and one nonseparable. We use the term adaptive to indicate that the policy is explicitly history dependent, as opposed to merely using the instantaneous state information as do all dynamic policies.

Separable policies are especially relevant to distributed load sharing where polling strategies are commonly employed [Eag86a]; the workstation at which the job originates polls some of the servers (it might not be practical to poll all of them) and then sends the job to the processor with the lowest toll. (While in this paper we assume the availability of global state information, so that all servers are candidates for job placement, our policy analysis is also relevant to the case with incomplete polling.)

Given a choice between two servers of the same speed, jobs should always be assigned to the one with the shorter queue. Furthermore, given a choice of two idle servers with different speeds, jobs should always be assigned to the faster server. These two elementary observations imply that, for our simplified model of only two server speeds and an infinite number of slow servers, either we send a job to the fast server with the shortest queue or to an idle slow server. Thus, for our model, separable policies reduce merely to threshold policies: send to the fast server with the shortest queue if and only if its queue length is shorter than some threshold τ, and otherwise send the job to a slow server. Notice that only the integer part of τ matters, since the queue lengths x_k are integers.

Intuitively, one would never want to send a job to a fast server on which its expected time-to-service (x_k/μ_{fast}) is greater than its expected time-to-completion $1/\mu_{slow}$ on a slow server, so we expect that the optimal threshold always satisfies the inequality $\tau \leq r$, where we define $r \equiv \mu_{fast}/\mu_{slow}$. In fact, this inequality has previously been conjectured by [Lin84] for a similar model. The SED policy is equivalent to a threshold policy with $\tau = r$, so that it has the greatest threshold and the most queueing of all sensible policies. Conversely, the *never queue* policy where one never queues for a server is equivalent to a threshold policy with $\tau = 1$.

Figure 3 shows the behavior of various threshold policies with model parameters $m_{fast}=6$ and $\mu_{fast}=5$. For very small λ the threshold of $\tau = r$ (SED) is optimal. At small loads, queueing for the fast servers minimizes the individual job's delay; since the load is light, it is unlikely that the job will cause delays for future jobs.

For very large λ, a cutoff of $\tau=1$ (*never queue*) is optimal. Queueing for fast servers is inadvisable here since the slower servers must be used in order to provide enough total processing power. Soon after a fast server becomes idle a new job will arrive that can occupy the fast server (the time elapsed will be roughly $1/\lambda$); as long as all the fast servers are utilized, it is better to immediately serve a newly arrived job at an idle slow server than to wait for a fast server.

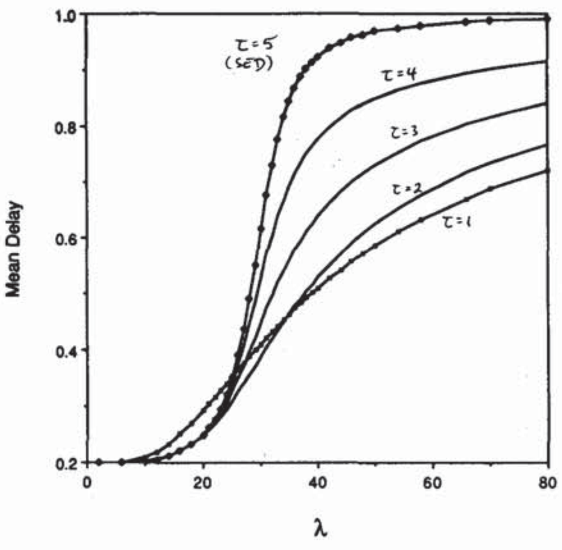

Figure 3. The mean delay for the threshold policies with $\tau=1,2,3,4,5$. $m_{fast}=6$, $\mu_{fast}=5$, and $\mu_{slow}=1$.

Queueing at fast servers is merely postponing the inevitable of having to use the slow servers, at a cost of adding delay to the jobs waiting at the fast servers.

The graph in figure 3 clearly illustrates that the SED policy performs poorly at high loads; in fact it is the worst threshold policy within the common sense constraint of $\tau \leq r$. Figure 3 also indicates that a threshold of $\tau=1$ performs reasonably well over the entire range of loads. It is somewhat surprising that this policy, which always sends jobs to the fastest open server and never has any queueing, seems to provide the best average performance of any single threshold policy. However, for loads in the range of $\lambda \approx m_{fast}\mu_{fast}$ the $\tau=1$ policy performs significantly worse than the best threshold policy. In Figure 3 at $\lambda=20$ the delay is 18% higher than optimal; also see Figure 8 where at $\lambda=40$ the delay is 63% higher than optimal. In general, the *never queue* policy will perform less well in these intermediate load ranges for larger values of r. In Section 5 we introduce some adaptive policies that do better in the crossover region. These policies will be most applicable in systems with large differences in processing speed (and hence large values of r) where the simpler never-queue-up policy is less attractive.

Between the extreme cases of small and large λ, where $\tau=r$ and $\tau=1$ are the best thresholds, we expect that there are parameter ranges $[\lambda_j^{opt}, \lambda_{j+1}^{opt}]$ where a threshold of $\tau=j$ is optimal. Finding the optimal separable policy in our model reduces to the problem of finding the optimal cutoffs λ_j^{opt} as a function of the system parameters. For a given set of servers one could, of course, use numerical simulation to build a table of the cutoffs. The performance of the optimal policy would then be given by the lower envelope of the curves in Figure 3. If one had a truly stable system, this might indeed work. However, this solution gives us no insight into the problem, and is inappropriate for evolving systems.

Alternatively, one can attempt to estimate the additional system delay caused by placing a job on a given server. The fast servers will have some cost function $C_{fast}(x)$ which is the system delay incurred by placing a job on a fast server when it already has x jobs in its queue. Since we are assuming an infinite number of slow servers, an idle slow server always exists, with cost function given by $C_{slow}=1/\mu_{slow}$. The policy is then to send to the server with the minimal cost. This approach produces a threshold τ, where τ is the solution to the equation $C_{fast}(\tau)=C_{slow}$. The SED policy is equivalent to using the expected delay for the individual job as the cost function: $C_k(x_k)=(x_k+1)/\mu_k$.

We can derive an estimate of the true system cost, which is better than the SED cost, by the following argument: assume that we know in advance the arrival times of all jobs and have already prescheduled them on the various servers according to some rule. Each server will then have some fraction of time idle, call it i_k. Now consider adding one more job. When we schedule the job, imagine that we will allow all previously scheduled jobs to preempt our present job, allowing it to be served only when the server would otherwise be idle. Then, on average, the time-to-completion on a server will be $(x_k+1)/(\mu_k i_k)$. By allowing, in this formulation, future jobs to preempt our additional one, the scheduling of this job will not cause future jobs any extra delay; everything is included in the expression for the additional job's delay. If the fractions of time idle i_k were indeed known, then the cost would be merely the above expression for the time to completion.

[Kri87] arrived at an equivalent expression for the cost using a different and more rigorous line of reasoning involving one step of policy iteration [How60]. Considering a static policy where each server had an independent Poisson arrival stream of strength λ_k, [Kri87] showed that the asymptotic difference in total delay of starting in state x_k *versus* starting in state x_k+1 is $C_k(x_k)=(x_k+1)/(\mu_k-\lambda_k)$. The fraction of time idle at each server is given by the M/M/1 result $i_k=1-\lambda_k/\mu_k$, so this cost function is the same as the one derived by the above heuristic argument.

Taking the λ_k's to be the static optimum Bernoulli split, [Kri87] then used the cost function to derive a separable policy that is guaranteed to do better than the static policy. Applied to our model with two classes of servers, this dynamic policy has the correct qualitative property of predicting a series of thresholds $\tau(\lambda)$ with $\tau=r$ for $\lambda\to 0$, and τ decreasing with increasing λ. However, for all $\lambda>\lambda_c$, with $\lambda_c=m_{fast}(\mu_{fast}-\sqrt{\mu_{fast}\,\mu_{slow}})$, the policy reduces to a threshold of $\tau=\sqrt{r}$, whereas we observed above that the correct limit is $\tau\to 1$ for large λ. In practice, for the parameter values we have studied, λ_c is small, and the simulation results for this policy are indistinguishable from those for a fixed threshold of $\tau=\sqrt{r}$. Thus, we observe that on our model the policy does well for small and intermediate λ, but not for large λ. See, e.g., the curve in Figure 3 with $\tau=2$. This curve is also the result for the policy with $\tau=\sqrt{5}$ since only the integer part of the threshold is relevant; also see Figures 7 and 8 for two other choices of the model parameters.

We now have three candidate separable policies. The first is the natural choice of SED (the $\tau=r$ threshold policy) that performs well at low loads but poorly at high loads. The second is the simple *never queue* policy (the $\tau=1$ threshold policy) that does well at high loads. Lastly, we have the policy of [Kri87], which is the only one of the three that explicitly uses the load information; however, it is indistinguishable from a threshold policy of $\tau=\sqrt{r}$ on our model. As discussed above, this choice of τ works well for small and intermediate loads, but does not exhibit desirable high load behavior. Other policies exist in the literature (see, for example, [Cho79] and [Yum81]), but these do not perform as well as the policies already discussed. Thus, we do not yet have a policy that performs well over the entire range of loads. Our attempt to find such a policy is the topic of the next section.

5. New Adaptive Policies

In real systems one does not *a priori* know the load. It is possible to measure arrival statistics and then use the resulting estimate of the load as an input to a separable policy. However, the load is not a quantity local to an individual workstation, and measuring it would require coordination and communication among the processors. Furthermore, once one considers the possibility of using statistical system measurements to determine the policy, certain locally measured statistics may be more useful than the load. For instance, one can easily measure the fraction of time idle i_k in each server. (In fact, many workstation operating systems already do this.)

This locally measured i_k can be used to define an adaptive policy using the cost function

$$C_k^{sep}(x_k)=\frac{x_k+1}{\mu_k i_k}$$

discussed in the previous section. Here we use the *measured* i_k for a given server to determine its cost function. The cost function affects the acceptance of jobs to the server, which in turn determines the future i_k. In this way the policy can adapt to the system and can easily accommodate to changes in arrival rate and system configuration. This approach is analogous to carrying out policy iteration [How60] restricted to separable cost functions of the form C_k^{sep}. As such, we can hope that it will perform well as long as the form of the cost function is adequate.

To realize the adaptive policy, we must measure the fraction of time idle i_k. We define

$$i_k(t) = \frac{<idle\ time>_k(t)}{<idle\ time>_k(t) + <service\ time>_k(t)}$$

where $<idle\ time>_k(t)$ denotes an estimate of the past idle time per job for server k at time t, and similarly for $<service\ time>_k(t)$. The brackets indicate an exponentially weighted average over past jobs for this estimate, requiring that a server maintain only a single number to keep the average, updating it as each job completes. This approach would be relatively easy to implement.

In general, define A to be a quantity related to each job such as the idle time or the service time, with A_n the value of A associated with the n'th job which finishes at time t_n. Then, when job $n+1$ completes, the exponentially weighted average is updated: $<A>_k(t_{n+1}) = (1-\alpha)<A>_k(t_n) + \alpha A_{n+1}$. The size of the exponential "window" is determined by α, the job α^{-1} in the past contributes e^{-1} compared to a recent job. (Note that $<service\ time>_k(t) \rightarrow 1/\mu_k$ for reasonably large values of α^{-1}, so that the system can measure its own μ_k if necessary.) In our simulation studies we found that the results were insensitive to choices of α^{-1} in the range 100 to 1000. In a real implementation the choice of α is a tradeoff between obtaining better averages and quickly responding to changes in loads and in the server population.

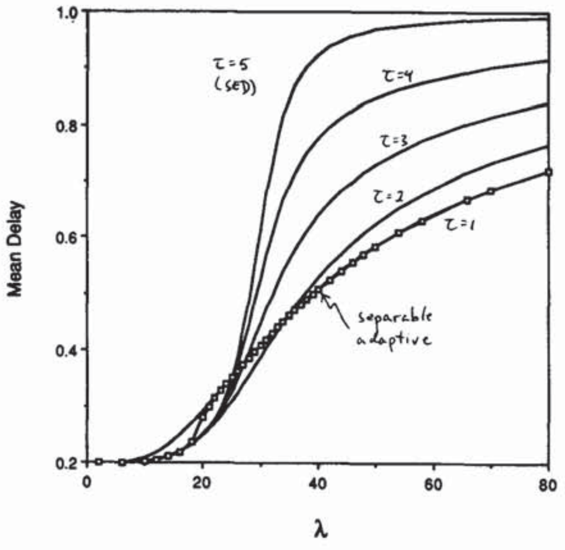

Figure 4. The mean delay for the separable adaptive policy. We also show the threshold policies for comparison. $m_{fast}=6$, $\mu_{fast}=5$, and $\mu_{slow}=1$.

Figure 4 shows the performance of this adaptive policy for the $m_{fast}=6$, $\mu_{fast}=5$ system. The results are compared to the various threshold policies. One can see that while the policy is not optimal, it does perform reasonably well. Furthermore, the results of simulations on systems with other model parameters show that the policy's performance improves as the size of the system grows.

To our knowledge, this kind of adaptive policy has not been previously used for the general *join the right queue* problem. It can be easily implemented in real systems, and does not assume knowledge of usually unknown system parameters. However, we can improve upon it. As we see in Figure 4, and have also observed for other choices of the model parameters not displayed here, the policy does not perform well in the crossover region $\lambda \approx m_{fast}\ \mu_{fast}$ where it first becomes necessary to utilize the slow servers (e.g., for $\lambda=22$ the delay for the adaptive policy is 17% more than the delay from the best separable policy). In this region it either settles down to a suboptimal threshold or oscillates between two thresholds, as it does for $\lambda \approx 20$ in Figure 4. (The oscillations occur when there is no threshold policy, with threshold τ, whose i_k's lead to cost functions C_k that are consistent with τ.)

For small λ the separable adaptive policy correctly mimics a high threshold $\tau \approx r$, but it switches over to a threshold of $\tau=1$ prematurely. The less than ideal behavior of this policy can be attributed, in part, to an overly simplified and restrictive cost function. We are presently working on generalizing the cost function to utilize more detailed local statistics such as state dependent arrival rates.

This adaptive policy also suffers from a granularity problem. Since only the integer part of the threshold function τ matters, the threshold jumps in unit increments. To loosen this restriction we introduce a nonseparable policy (although it still uses only local statistics): a policy where the job placement depends on more complicated state information than just the minimum of the cost functions. Nonseparable policies involve a single cost function C_{fast}^{nonsep} which is now a function of the entire instantaneous state of the fast servers $\{x_k\}$ with $k \in [1, m_{fast}]$; when this cost is less than C_{slow}, we send the job to the fast server with the shortest queue. We use the function

$$C_{fast}^{nonsep} = \frac{1 + \sum_k x_k}{\sum_k \mu_k i_k}$$

which, according to our heuristic derivation of the previous section, can be interpreted as the extra delay introduced into the system if the processing of the job could be divided among all of the fast servers. This cost function also leads to a policy of threshold form, but now the threshold is applied to the sum over the queue lengths instead of to the queues individually. The degree of granularity has been decreased roughly by a factor of m_{fast}.

Note that this policy still requires only statistics that can be measured locally: the i_k's. Nonseparable policies that use more detailed multi-server statistics are possible, and we plan to study them in the future. In the separable policy the server must provide only one number, C_k^{sep}, to the decision maker. This nonseparable policy requires that servers provide *two* numbers to the decision maker, λ_k and the product $\mu_k i_k$.

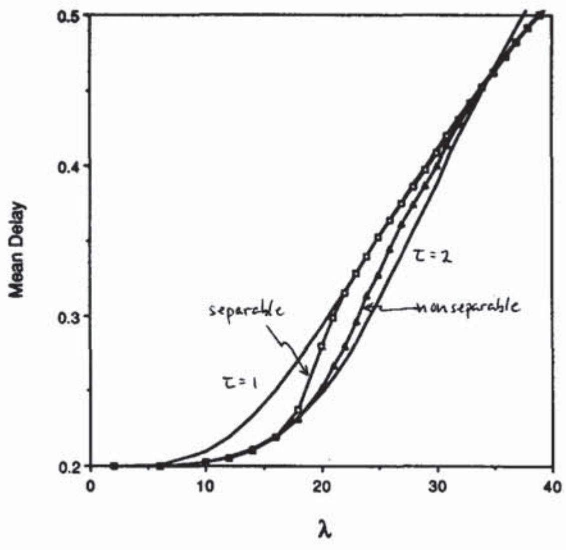

Figure 5. A comparison of the separable adaptive and non-separable adaptive policies. The threshold policies with $\tau = 1$ and $\tau = 2$ are also shown. $m_{fast} = 6$, $\mu_{fast} = 5$, and $\mu_{slow} = 1$.

Figure 5 shows a direct comparison of the two adaptive policies together with the bounding separable threshold policies. (Note that the scale is much enlarged compared to Figure 4 to better illustrate the differences.) We see that while neither policy attains the optimal separable threshold result, the nonseparable adaptive policy does substantially better.

To gain insight into the performance of our nonseparable adaptive policy, we have studied the set of nonseparable threshold policies where one sends a job to the slow servers if the sum of the fast server queue lengths is more than τ_{nonsep}. We find that the nonseparable threshold policies smoothly interpolate between the corresponding separable threshold policies. For the case of $m_{fast} = 6$ the nonseparable policy with $\tau_{nonsep} = 6$ is identical to the separable policy with $\tau_{sep} = 1$, and for $\tau_{nonsep} = 12$ and $\tau_{sep} = 2$ the results are indistinguishable. The nonseparable policies with $6 < \tau_{nonsep} < 12$ lie between the separable policies with $\tau_{sep} = 1$ and 2. In particular, the non-separable threshold policies do not significantly fall below the envelope defined by the best separable threshold policy. Thus, the improvement of the nonseparable over the separable adaptive policies can be attributed to the decrease in the granularity, rather than to the better use of the extra information available.

Based on the above observations, we conjecture that the performance of the best threshold policy is very close to that of the most general nonseparable optimal policy. To test this conjecture, we studied a lower bound obtained by considering a model with only one central queue. This model admits only threshold policies: only for queue lengths above a threshold value will the policy use the slow servers. By simulating all of the possible thresholds and taking the lower envelope of the curves, we determined the optimal performance of this single-queue model. This envelope is then a lower bound to the performance of any policy for our problem with separate queues. We found that this lower bound is typically within 10% of the optimal separable threshold policy, so that these policies look quite good. In addition, this bound cannot be realized with separate queues, since a central queue is unaffected by variations in job service time; thus, the best separable policy is sure to be considerably closer to optimal than this bound would indicate.

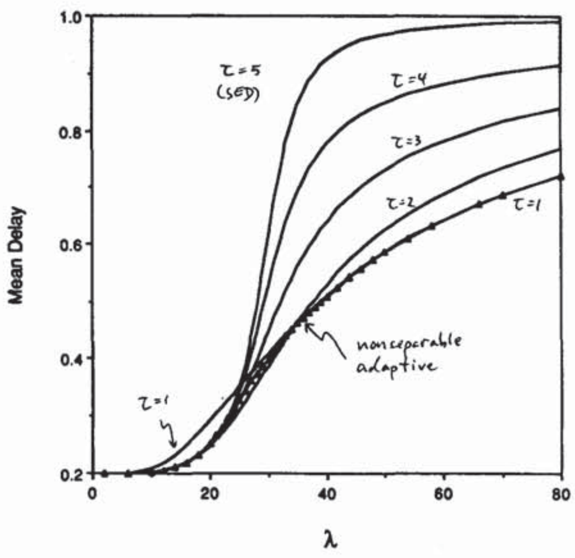

Figure 6. The non-separable adaptive policy. The threshold policies are also shown. $m_{fast} = 6$, $\mu_{fast} = 5$, and $\mu_{slow} = 1$. Note that $\tau = 5$ is equivalent to the SED policy.

In Figures 6-8 we show the performance of our best policy, the nonseparable adaptive policy, for a number of choices of model parameters. The policy closely matches the best separable threshold policy for all but a narrow range of λ near the crossover point $\lambda \approx m_{fast}\,\mu_{fast}$. Note that compared to the *greedy* policy with $\tau = r$ our policy does exceedingly well over the

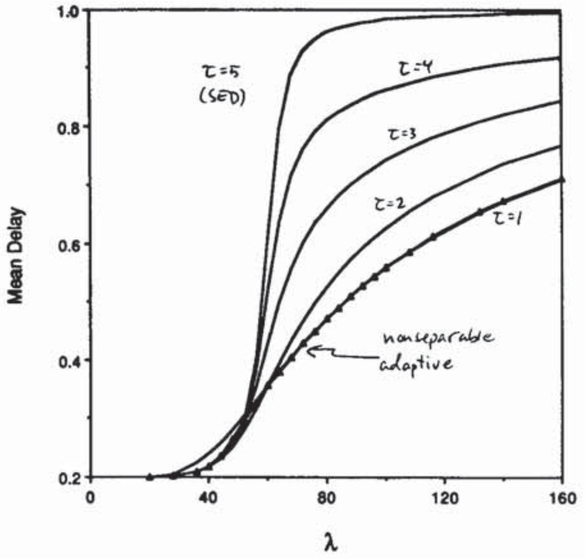

Figure 7. The non-separable adaptive policy. The threshold policies are also shown. $m_{fast}=12$, $\mu_{fast}=5$, and $\mu_{slow}=1$. Note that $\tau=5$ is equivalent to the SED policy.

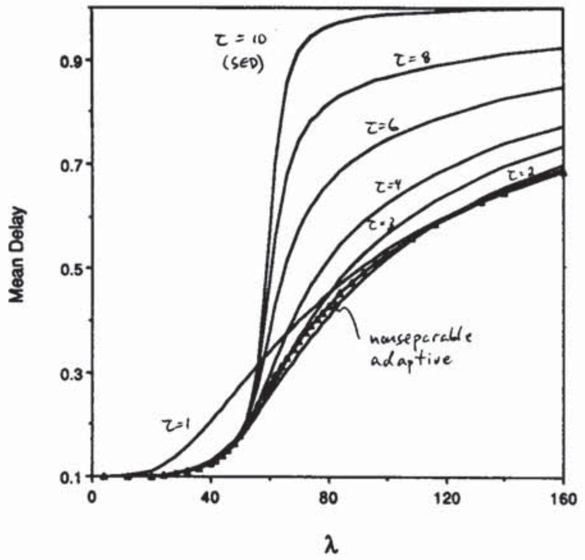

Figure 8. The non-separable adaptive policy. The threshold policies are also shown. $m_{fast}=6$, $\mu_{fast}=10$, and $\mu_{slow}=1$. Note that $\tau=10$ is equivalent to the SED policy.

whole range of arrival rates λ; and our policy does consistently better than the *never queue* policy with $\tau=1$.

The adaptive policies introduced in this section can be generalized to systems with more than two classes of servers. The separable adaptive policy immediately generalizes by simply assigning a job to the server with the minimum cost as measured by C^{sep}. A way to generalize the nonseparable policy is to use the cost function C^{nonsep} to determine whether to send the job to the fastest open server or to one of the faster servers with non-empty queues. If the decision is to send it to one of the faster servers, then C^{sep} can be used to decide between them. We plan to study the performance of these generalizations in future work.

6. Simulations

The results graphed in Figures 2-8 were obtained using a simulation package we constructed for the problem. The results were averages of the job delays over N_{jobs} after discarding the first $N_{discard}$ (typically 2×10^3) jobs to account for transient behavior as the system filled up. The data was taken in blocks so that confidence intervals could be determined. With typical values of N_{jobs} of 2.5×10^5 the confidence intervals were much less than 1%. Consequently, we have not displayed error bars on the graphs.

7. Discussion

We have introduced a simple model system that captures the essence of the problem of *joining the right queue*. For this model, all separable policies are equivalent to one of only a small number of possible simple threshold policies. With the use of computer simulation, we have studied these policies to show that the *greedy* SED policy is a very poor choice, illustrating that in queueing systems, as in life, greed is not enough. Perhaps surprisingly, we find that a simple policy of *never queue* displays quite good overall performance.

For cases where the *never queue* policy is inadequate (generally when there is a large disparity in server rates), we presented two adaptive algorithms; a separable and a nonseparable one. They require only that each server knows its queue length, its relative processor speed, and its average utilization.

The adaptive nature of the policies make them very attractive for situations where the system parameters change fairly often. In addition, because the adaptive policy in effect measures and reacts to how well it is doing (through i_k), we anticipate that it will perform well on more realistic system models in which the arrival and service time statistics are not of the simple memoryless type.

The results of this paper are based primarily on simulations; in a future publication [She87] we study similar models in the limit $m_{fast}\rightarrow\infty$. Here we can show that the *never queue* policy achieves optimal performance.

Acknowledgments

The authors thank K. R. Krishnan for several helpful discussions, and also G. Herman and G. Gopal for comments on a preliminary draft of this paper.

References

[Bel83] C. Bell and S. Stidham, "Individual versus Social Optimization in the Allocation of Customers to Alternative Servers," Management Science, Volume 29, pp 831-839, 1983.

[Cho79] Y. Chow and W. Kohler, "Models for Dynamic Load Balancing in a Heterogeneous Multiple Processor System," IEEE Transactions on Computers, Volume 28, pp354-361, 1979.

[Eag86a] D. Eager, E. Lazowska, and J. Zahorjan. "Dynamic Load Sharing in Homogeneous Distributed Systems," IEEE Transactions on Software Engineering, Volume SE-12, No. 5, pp 662-675, 1986.

[Eag86b] D. Eager, E. Lazowska, and J. Zahorjan. "A Comparison of Receiver-initiated and Sender-initiated Dynamic Load Sharing," Performance Evaluation, Volume 6, pp 53-68, 1986.

[Eph80] A. Ephremides, P. Varaiya, and J. Walrand, "A Simple Dynamic Routing Problem," IEEE Transactions on Automatic Control, Volume 25, pp 690-693, 1980.

[Fos78] G. Foschini and J. Salz, "A Basic Dynamic Routing Problem and Diffusion," IEEE Transactions on Communications, Volume 26, pp 320-327, 1978.

[Haj83] B. Hajek, "Extremal Splittings of Point Processes," Mathematics of Operations Research, Volume 10, pp 543-556, 1983.

[Her87] G. Herman and A. Holsinger, "Adaptive Resource Allocation in Homogeneous Processor Networks," to appear.

[How60] R. Howard, "Dynamic Programming and Markov Processes," The MIT Press, Cambridge, Massachusetts 1960.

[Kri87] K. R. Krishnan, "Joining the Right Queue and Routing in Data Networks," Bellcore Technical Memorandum; also see "Joining the Right Queue: A Markov Decision Rule," Proceedings of the IEEE Conference on Decision and Control, Dec. 1987 (to appear).

[Lel86] W. Leland and T. Ott, "Load-Balancing Heuristics and Process Behavior," Proceedings of the 1986 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems, 1986.

[Lin84] W. Lin and P. Kumar, "Optimal Control of a Queueing System with Two Heterogeneous Servers," IEEE Transactions on Automatic Control, Volume 29, pp696-703, 1984.

[Nao69] P. Naor, "On the Regulation of Queue Size by Levying Tolls," Econometrica, Volume 37, pp 15-24, 1969.

[Rub85] M. Rubinovitch, "The Slow Server Problem: A Queue with Stalling," Journal of Applied Probability, Volume 22, pp 879-892, 1985.

[She87] S. Shenker and A. Weinrib, "Asymptotic Analysis of Large Heterogeneous Queueing Systems," Submitted to the 1988 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems.

[Sti85] S. Stidham, "Optimal Control of Admission to a Queueing System," IEEE Transactions on Automatic Control, Volume 30, pp 705-713, 1985.

[Sti86] S. Stidham, "Scheduling, Routing, and Flow Control in Stochastic Networks," preprint.

[Whi86] W. Whitt, "Deciding Which Queue to Join: Some Counterexamples," Operations Research, Volume 34, No. 1, pp 55-62, 1986.

[Win77] W. Winston, "Optimality of the Shortest-Processing-Time Discipline," Journal of Applied Probability, Volume 14, pp 181-189, 1977.

[Yum81] T. Yum and M. Schwartz, "The Join-Biased-Queue Rule and Its Application to Routing in Computer Communication Networks," IEEE Transactions on Communications, Volume 29, pp 505-511, 1981.