Finding Idle Machines in a Workstation-based Distributed System

Marvin M. Theimer
IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120

Keith A. Lantz
Olivetti Research Center
2882 Sand Hill Road
Menlo Park, CA 94025

Abstract

Remote execution facilities allow the user of a workstation-based distributed system to offload programs onto idle workstations, thereby providing the user with access to computational resources far beyond that provided by his personal workstation. In order to make good use of such facilities, one must be able to find suitable candidate hosts for remote execution, typically the least-loaded host with the necessary number of resources to run a given program. In this paper we describe the design and performance of scheduling facilities for finding idle hosts in a workstation-based distributed system. We focus on the tradeoffs between centralized and decentralized architectures with respect to scalability, fault tolerance, and simplicity of design, as well as several implementation issues of interest when using multicast communication. We conclude that the principal tradeoff between the two approaches is that a centralized architecture can be scaled to a significantly greater degree and can more easily monitor global system statistics, while a decentralized architecture is simpler to implement.

1 Introduction

A distributed computer system consisting of a cluster of workstations and server machines represents a large amount of computational power, much of which is frequently idle. For example, the research system used as the testbed for the work reported here consisted of about 70 Sun-class workstations and server machines, providing a total of about 150 MIPS processing power. Yet one third of the workstations were completely unused, even at the busiest times of the day. Use of such workstations as "computation servers" increases the processing power available to (active) users and improves the utilization of the hardware base.

In a previous paper [11] we described the design and performance of remote program execution and migration facilities for such a system, leaving open the question of what kinds of global scheduling facilities would be best to provide. In this paper we provide half of the answer to that question—a comparative study of *architectures* (hence *mechanisms*) for global scheduling, as well as several implementation issues of interest when using multicast communication. Policy issues, including fairness, are discussed in [10].[1]

From an architectural point of view, the principal goals for a global scheduling facility are:

- **performance:** The scheduling facilities should impose minimal overhead on the system; they should neither steal cycles from applications that are not using the facilities nor significantly delay the execution of applications being scheduled. Specifically, we would like the scheduling facilities to, on average, consume less than 1% of any client workstation's CPU cycles, consume less than 1% of the network bandwidth, and require less than 100 ms to select a host.
- **scalability:** The architecture should support systems containing hundreds of machines.
- **fault tolerance:** The remote execution facilities of the system should not be disabled for more than a few seconds by the crash of one or even a few machines.

The principal thrust of this paper is to demonstrate that facilities for finding suitable hosts for remote execution (or migration) can be constructed that perform well, are highly available, and can be scaled to systems containing hundreds or even thousands of machines. Assuming the availability of efficient broadcast/multicast facilities, we further show that both centralized and decentralized designs can be constructed which are similar in their performance and fault tolerance characteristics; however, centralized designs can be scaled to significantly larger system sizes.

There are two principal features of workstation-based distributed systems that distinguish them from the environments typically assumed in the literature on global scheduling (see [13] for an overview). First, most previous work has assumed a homogeneous workload. However, an environment consisting principally of personal workstations tends to produce a load distribution that can vary greatly in magnitude over time and is *not* homogeneous in nature. Hosts may be idle or running interactive applications or compute-intensive batch jobs or both. "Clusters" of hosts may be idle when events such as group meetings occur. Consequently, techniques based on interactions with "nearest neighbors" of a host or scheduling

[1] Our "architectures" are roughly equivalent to the "placement policies" of Zhou and Ferrari [14], whereas our "policy issues" include their "information" and "transfer" policies.

CH2541-1/88/0000/0112$01.00 © 1988 IEEE

designs that rely on statistical averages to determine their selections will have difficulties with this environment.

Second, most previous work has assumed very simple communications models, frequently with zero communication cost. Little work has been done on determining how realistic such models are; that is, on *how* to actually gather global state information about a system. What has been done, with a few exceptions, assumes point-to-point communications, ignoring the availability of efficient broadcast or multicast support in many local area network technologies. However, in a workstation-based environment, in which the set of machines available as "remote execution servers" is constantly changing, a communications model based on some form of broadcast can provide a much more efficient means of disseminating information than can be provided by point-to-point communications. Naturally, indiscriminate use of broadcast can inflict a significant overhead on the system, particularly on hosts that are not interested in participating in the activity using broadcast (e.g., global scheduling). The provision of multicast communication facilities can alleviate much of this overhead by directing communications only to interested parties in the system.

Unfortunately, broadcast/multicast, as implemented at the data link level, is typically unreliable. If reliable broadcast/multicast is required, it must be implemented in software, with the attendant overhead. Furthermore, the loss rate of replies to query messages can be very significant if the number of replies generated to a query is larger than the buffering capacity of the sender's hardware and/or software. With multicast, one must also worry about how to manage the membership of machines in multicast groups.

This paper presents our scheduling designs with particular focus on how we have addressed both the general scheduling issues mentioned as well as the specific issues of using unreliable multicast facilities to implement our designs. The next two sections describe the target environment and provide an overview of the major architectural issues in designing a global scheduling facility. Sections 4 and 5 describe our centralized and decentralized scheduling architectures, respectively. Section 6 discusses related work, and Section 7 concludes the paper.

2 Target Environment

The scheduling architectures described in this paper should apply to most contemporary workstation-based distributed systems. By "workstation-based distributed system" we mean a system in which most machines are autonomous personal workstations, each dedicated primarily to serving its local user, interconnected by high-speed local area networks. We assume that the operating systems on each workstation are able to communicate with each other so as to provide transparent remote execution and scheduling facilities. By "transparent" we mean that use of workstations as computation servers should not require programs to be written with special provisions for executing remotely. Examples of systems that conform to this model include the V-System [2], Mach [8], LOCUS [12], and NEST [1]. The performance figures quoted in this paper are based on implementations we have done under the V-System.

Many of the assumptions made in this section with respect to parameter values can be characterized as "rule-of-thumb"—for example, assuming that at most 1% of the system network bandwidth be consumed for scheduling activities. The values assumed are intended to be representative rather than authoritative and we argue that reasonable changes to those values will not significantly affect the conclusions we arrive at.

2.1 Hardware

We assume workstations with at least 1 MIPS processing power. We assume a network with speed and failure characteristics like those of Ethernet[2] or token rings and that can provide efficient broadcast or multicast communications between machines. Of particular concern is the overhead the use of broadcast/multicast imposes on machines that are not members of the intended set of recipients. Although hardware support in the network interface can make the overhead close to zero,[3] many network interfaces only support broadcast, requiring that multicast filtering be done in software. Fortunately, evaluation and measurements we have done of the V-System's communication facilities indicate that a 1 MIPS machine (such as a Sun-2) can reasonably be expected to need about 0.4 milliseconds to discard a multicast packet not destined for it in software. Thus, it could discard up to 25 packets per second using less than 1% of its CPU cycles.

We also must make some assumptions about the handling of multiple nearly simultaneous replies to a single broadcast (or multicast) query. We will assume that network interfaces are capable of receiving at least every second network packet of a stream of "back-to-back" packets and that machines and their operating systems are capable of buffering at least 20 to 40 packets before reaching the limits of their capacity. This corresponds to our empirical observations on the performance of a Sun-2 workstation with a 3COM interface board, connected to a 10 Mb Ethernet, and running the V-System. Even with these assumptions, the broadcast/multicast reply loss rate can easily exceed 50% when more than a few replies are generated to a query [4].

The final major parameter that must be taken into account is network bandwidth. If we assume a 10 Mb/sec network and require that, on average, at most 1% of its bandwidth be consumed by our scheduling facilities, then we have 12.5 Kbytes/sec to play with. This corresponds to about 120 messages of length 100 bytes—sufficient space, in particular, for the 64 bytes of packet header required on Ethernet and the 32 bytes of data comprising a V-System message.

2.2 Workload Assumptions

Our primary interest is an environment in which most machines are autonomous personal workstations that are dedicated principally to serving their local users. However, nothing in our design precludes the use of dedicated "compute server" machines. We assume a workload that consists primarily of interactive and "fast-turnaround" applications that exist for relatively short periods of time—such as text editing, electronic publishing, program

[2] Ethernet is a trademark of Xerox Corporation.

[3] An example is the Intel 85286-based interface used in the Sun-3 workstation.

development, and CAD—rather than compute-intensive, long-running applications. The total number of such applications running in the system varies considerably, depending on how many users are present and active at any given time. Typically, many machines are "idle", where idle is defined here to mean a level at which low-cost activities such as text editing are still handled. Periodically, however, a few users may grab the entire system's free resources to run large, distributed computations. The heterogeneity of this workload contrasts with an environment in which compute-intensive, long-running applications take up all available resources in the system most of the time, thereby keeping the workload uniformly high.

Informal observations indicate that our users spend *at least* one minute thinking, debugging, or text editing, between submitting tasks such as compilation or text formatting for execution. Therefore, in order to obtain an upper bound on the frequency of remote execution requests, we assume that *all* users generate requests randomly, with an average of one per minute—that is, according to a Poisson distribution with a mean, k, of one per minute. We claim that this remote execution frequency is ***significantly*** higher than will occur in reality since, as observed, many workstations are completely idle and many of the rest are occupied by users engaged in non-compute-intensive activities such as editing of files.

We assume, in turn, that each user request corresponds to an application with some number of subprograms, each of which may be executed on a separate machine. Based on experience with our own system, we argue that in contemporary workstation-based distributed systems highly concurrent applications are rare and that the average number of subprograms per application, d, is small—between one and five, say. Moreover, we claim that the remote execution requests associated with each application will almost always be batched into one "multiple" selection action or will have program loads interspersed between them. Since program loads typically take several seconds to perform and different programs will take different amounts of time to load, we argue that in the latter case there will effectively be no correlation between remote execution requests. Therefore, we assume that all remote execution requests generated by a machine will be distributed randomly—according to a Poisson distribution with mean $k \cdot d$.

Finally, we require that selection of an idle host takes no more than 100 ms.

2.3 Load Metrics

It is well known that schedulers make better decisions if they can take advantage of information about the nature of the jobs they are scheduling. Unfortunately, in most systems—especially timesharing systems, of which our environment is a generalization—exact knowledge is unavailable and estimates are either unreliable or time-consuming to compute. Therefore, as in most commercial timesharing systems, we take the simple but surprisingly effective approach of scheduling programs on the basis of a few readily-available metrics—memory requirement, processor type, and processor utilization. However, the discussion that follows is independent of chosen load metrics, since they can be computed in a completely decentralized manner; in particular, each host periodically computes its own processor utilization independent of all other hosts. It is the manner in which the load information is disseminated that is of interest here.

3 Dimensions of Global Scheduling

There are three principal architectural issues to be addressed in designing a global scheduling facility for a workstation-based distributed system, namely:

1. How is load information disseminated—by *advertisement* or in response to *queries*?
2. Who initiates a request for remote execution—the *source* (or *sender*) or the *server* (or *receiver*)?
3. Who "makes the match" between program and host—the initiator of the request or a central server (in which case the initial request was sent to that server)?

Server-initiated strategies are best suited to environments where it is desired to maintain a balanced load across all machines at all times. In our target environment this is not only unnecessary but undesirable. Since most machines are under-utilized, balancing the load will typically cause processes to be migrated from one lightly loaded machine to another, to no advantage when overhead is taken into account. Therefore, we concern ourselves only with source-initiated strategies.

The remainder of the paper addresses the remaining two issues, using the distinction between centralized scheduling and decentralized scheduling as the means of subdividing the discussion.

4 Centralized Scheduling

We define centralized scheduling to mean that state information is collected at a single physical location at which all scheduling decisions are made. We will show that replication for performance reasons will be unnecessary, but may be desirable for fault tolerance.

4.1 The Basic Model

The basic model of a centralized scheduling service consists of machines periodically sending status update messages and clients sending remote execution requests—to a central server.[4] Thus, machines advertise their load information. One might consider having the server query the system for state, but the same query could just as well be performed directly by clients, resulting in the decentralized scheduling architecture described in Section 5. Having a central server perform the state queries would reduce message traffic if state information were used to answer several host selection requests, but, since machines can change their load at any time due to local activities, this would introduce problems of stale state information.

[4] This corresponds to the CENTRAL algorithm of Zhou and Ferrari [14].

4.1.1 Model Parameters

The most important quantities that determine a scheduling server's performance are the frequency of selection requests, f_r, the frequency of state updates, f_u, and the average service times needed to process a selection or update request, s_r and s_u, respectively. If N is the number of hosts in the system, then:

$$f_r = N \cdot k \cdot d \tag{1}$$

For idle machines, we claim that state updates will occur only infrequently. For active machines, we need to filter out transient load changes by using time-averaged load values. We argue that once every 10 seconds is the highest frequency one should consider, given our assumptions of remote program durations of a minute or more and our desire to filter out insignificant local events. Examples of these include a brief flurry of editor page scrolling events, checkpoints of edit files (which can take several seconds), and the loading and running of small programs—such as directory listing and system status programs.

If we assume that the server maintains an ordered list of host candidates of size N_g, then s_r is essentially a constant. If we assume that update messages arrive randomly then s_u will be proportional to N_g. We shall take N_g to be equal to the average number of host selection requests arriving during one time interval since this is sufficiently large to satisfy most bursts of requests. We shall show later how multicast communication can be used to quickly refill the server's candidate list in the event of too many requests.

4.1.2 Model Evaluation

Given these assumptions and definitions, the load placed on the server is:

$$l = l_r + l_u \tag{2}$$

where l_r is the selection request load:

$$l_r = f_r \cdot s_r = N \cdot k \cdot d \cdot s_r \tag{3}$$

and l_u is the update message load:

$$l_u = f_u \cdot [\frac{N_g}{N} \cdot s_{u1} + \frac{(N - N_g)}{N} \cdot s_{u2}] \tag{4}$$

Here s_{u1} and s_{u2} are the service times needed to process host update messages that respectively belong or don't belong to the host selection candidate set. Assuming an ordered candidate list with pointers to the first and last element, s_{u1} is linearly proportional to N_g and s_{u2} is a constant. For our implementation we found that $s_{u1} = s_0 \cdot (1 + 0.0025 \cdot N_g)$ and $s_{u2} = s_0$, where s_0 is basically the time needed to satisfy a "null" request—that is, the time required to receive and reply to a network message with all server "maintenance" activities averaged in. Indeed, since it is difficult to delineate the time spent on each machine for a inter-machine message, we overestimate s_0 by including the entire time required to perform an inter-machine send/receive/reply transaction. That is, s_0 will be approximated as the sum of the network IPC time and the time needed to satisfy a "null" request. With this overestimation in mind, we can also approximate s_r as s_0 since the time required for the scheduler to pick the first element off an ordered list of candidates is negligible.

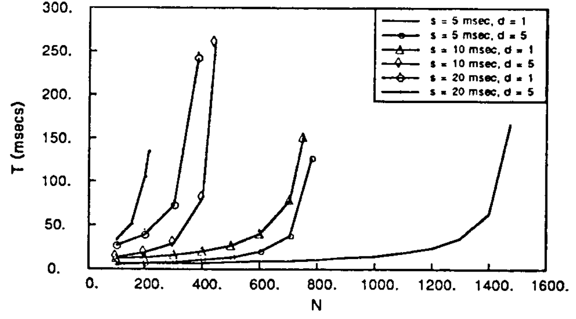

Figure 1: Central server performance with all hosts updating it. In all cases hosts generate one host selection request per minute and one status update message every 10 seconds. s in the figure stands for s_0.

We estimate the average response time of the server by assuming that it can be approximately modeled as an M/M/1 queue [7]:

$$T = \frac{s}{1 - \rho} \tag{5}$$

where s is the average service time of the server and ρ is the utilization factor of the server. For the constant service time required for an M/M/1 queue, we simply take the maximum service time our server might need to process an incoming message, namely, s_{u1}. ρ is the normalization of l to the interval $0-1$. The details of this analysis may be found in [10].

Finally, with respect to network bandwidth, remember that our scheduling facilities should not generate more than 120 100-byte messages per second. Assuming that our selection requests, replies to those requests, and load updates do indeed average 100 bytes in length, adhering to this limit implies that:

$$2 \cdot f_r + f_u \leq 120 \tag{6}$$

Figure 1 shows how a central scheduler can be expected to perform under various perturbations of the quantities just discussed. As a convenience to the reader, we first summarize the various parameters that are relevant to this figure before drawing conclusions from it:

- k : the average number of applications to execute remotely per user per unit time (we assume one per minute per workstation in all cases shown)
- d : the average number of (remotely executed) subprograms per application
- N : the number of hosts in the system
- f_u : the frequency of update messages generated by the system (we assume each workstation generate one every 10 seconds in all cases shown)
- N_g : the number of host selection candidates recorded at the server (in an ordered list)
- s_0 : the "base" service time for a message by the scheduler.

The server is quite capable of handling systems of 100 to 200 machines, even when a base service time of 20 milliseconds is assumed for s_0 and all applications are generating 5 remotely executing subprograms. As will be shown in Section 4.3, a base service time of 6 msecs is possible with an operating system with optimized IPC (such as the V-System), running on a 1 MIPS machine (such as a Sun-2). Examining the behavior for a base time of 20 msecs illustrates how a system with less efficient IPC facilities might perform, such as Berkeley UNIX.

If the server has a base service time of 10 milliseconds, then slightly over 700 machines can be handled, each of which is generating one remotely executing *non-distributed* application per minute. If we can reduce the base service time by a factor of 2—e.g. by buying a single more powerful server machine—then our design would support a system of up to about 1500 "standard" workstations. If, on the other hand, we make the average "degree of distribution" of applications, d, equal to 5, then a server with a 10 msec base service time should still be able to accommodate systems containing up to about 400 machines.

Unfortunately, taking our constraints on network bandwidth consumed into account significantly reduces the number of hosts that can be handled in some scenarios. In particular, for $d = 1$, we can handle at most 900 machines, and for $d = 5$, at most 450 machines. Whereas this limit is not a serious constraint for systems where the base service time is 10 – 20 msec, it will be for a system with a 5 msec base service time.

4.2 Restricting the Number of Updaters

We can substantially lower the load on the network (as well as the server) by realizing that we are only interested in update information from those hosts that are likely candidates for remote execution, namely, the hosts that comprise the host candidate set described previously. An "update group" can be created whose members send updates to the server, and all other hosts in the system do not. Membership in this group can be determined in a decentralized manner by using broadcast to send out a load cutoff value (to all hosts in the system). Hosts with a load below this value join the update group if they are not already a member of it. Hosts with a load above this value leave the update group if they are already a member of it.

Cutoff values are recomputed periodically to accommodate changes in the overall state of the system. This information is acquired by having all hosts in the system send update messages to the server at low frequency—much lower, in particular, than the frequency with which hosts in the candidate set send update messages. Using this information and subsequent feedback from the update group, it is possible to (perhaps iteratively) compute a cutoff value that, while limiting the size of the update group, ensures the group is large enough to handle reasonable variations in the host selection request frequency. Note that broadcasting an increased load cutoff value also allows quick replenishment of the server's candidate list should it become depleted by a burst of host selection requests.

However, this design still bothers busy hosts with cutoff messages and makes them generate periodic update messages. A refinement is to create another multicast group containing all hosts that have idle resources available for remote execution requests. Cutoff messages are sent only to this "idle group" since only hosts in this group need generate update messages. Membership in the group is determined in a completely decentralized fashion: Each host decides for itself when to join or leave the group, based on its own determination of whether or not it has idle resources.

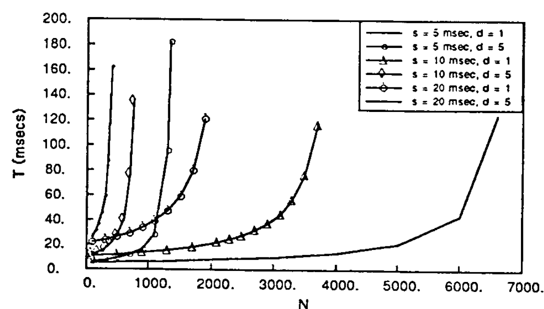

Figure 2: Central server performance with a restricted updater group. In all cases hosts generate one host selection request per minute and one status update message every 10 seconds. s in the figure stands for s_0.

Figure 2 shows the results equivalent to Figure 1 with our changes in update frequency taken into account, that is, with the second term of equation 4 set to 0. The most important thing to note is that the server is now able to handle in excess of 400 machines in all cases. For our "canonical" case of running a server with a 10 msec base service time and dealing with non-distributed applications, the server can deal with systems up to about 3500 machines in size. With respect to network load, it is now reduced to where over 3400 machines can be handled when $d = 1$, or about 680 hosts when $d = 5$.

At this level of size, the scheduling facilities will almost certainly no longer be any kind of bottleneck. In particular, the traffic generated between remotely executing programs and the user's workstation just for purposes of interacting with the user will typically dwarf that generated for scheduling.

4.3 An Implementation

In order to empirically evaluate our model a simple scheduling service was implemented which ignored selection issues such as memory requirements and specialized resource needs. A list of machines, ranked by load, is maintained by a scheduling server and the first element is always picked. The size of the list is bounded by multicasting a cutoff load value—namely, the normalized processor utilization—to the idle group.

Figure 3 shows response times for host selection as a function of the number of machines, N', updating the scheduler and the frequency, f, at which lightly loaded machines update the central server. In order to control the loads and parameters of the host managers, a parallel set of simulated host managers was created to generate the data for the graphs and data presented here. Al-

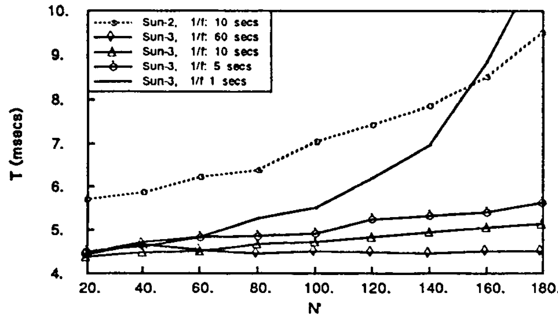

Figure 3: Central server host selection response times as a function of the number of updaters (in milliseconds).

though multiple host managers were run on each actual machine in order to obtain the results for larger system sizes, there was no contention between the host managers on any given machine because their update frequency to the scheduling server was at most once every second and the phase between host managers' update cycles was random. In order to force pessimistic processing times for update messages, the loads presented by the program managers were uniformly distributed and randomly changed on every update.

Figure 3 only shows results out to update group sizes of 180. The reason for this upper bound has to do with the number of workstations we were able to commandeer at any one time (at most 20) and with with the amount of kernel memory available for process descriptors and network buffers (which limited us to at most 10 simulated hosts on each workstation).

The scheduling server was run both on Sun-2 and Sun-3 machines, as labeled in Figure 3. The following conclusions can be drawn from the figure:

- The "base" service time for a host selection request, s_0, can be determined from the response times corresponding to small update groups. For a Sun-3 server, the service time is 4 to 5 msec. For a Sun-2 server, it is about 6 msec.

- The empirical results bear out our claims with respect to scalability. The curves for update frequencies of 5, 10, and 60 seconds indicate that we can easily handle several thousand machines, at least as far as scheduler response time goes. An update group size of 180 corresponds to a system size of 5400 if applications do not generate multiple subprograms.[5] If they generate, on average, 5 subprograms, then the system size corresponding to an update group of size 180 shrinks to 1000.

  Since our testbed is 80% idle most of the time, using the actual update rates obtained from host managers that only update on significant changes in state would yield almost no update traffic at all, corresponding best to the curve for updates once every 60 seconds. One can argue that this will also hold in the future for the set of machines that reside on the "least loaded host" list in the server.

[5] We only used one host selection client in our experiments, implying that the N' in the figure corresponds to $2N_g$.

4.4 Fault Tolerance

We have demonstrated that a centralized design is scalable to large systems. The next point to address is that of availability. A typical approach would be to provide $k+1$ replicas of the server if it is to survive k faults (machine failures, in our case). Numerous replication algorithms exist for updating replicated services such as this. We note only that, if all machines in the system have network interfaces that allow multicast filtering in hardware and we are willing to forego strong consistency by admitting the possibility of lost updates, then multicast provides a simple and cheap means of updating the multiple copies of the server.

However, if occasional delays in service of several seconds are acceptable, then one can take the simpler approach of *reinstantiation* instead. Rather than maintaining $k+1$ server replicas, we maintain a single server and ensure that there are at least k entities monitoring the server to detect its death. When death is detected, a new instance of the server is brought up, which reconstructs its state information by sending a multicast message out requesting immediate state update.[6] The time during which the scheduling service is unavailable will be the sum of the time to detect failure, the time to load the server program, the time to resolve the possibility of multiple concurrent instantiations, and the time to reconstruct the global state information. If requesting clients are used as the monitors, then the delay in response seen by the first client after failure will be exactly this sum. For our implementation this turns out to be approximately 18 seconds.

4.5 Server Placement

A final problem one faces with global system servers of any kind is where to run them. If separate machines are dedicated to running servers, then several such machines are required if fault tolerance is to be taken into account. Fortunately, schedulers are not intrinsically bound to any particular machine, so we can achieve the effect of having multiple server machines by simply running the server on idle workstations and migrating or reinstantiating it as needed. Indeed, since there is no point in considering remote execution in the first place if there are no idle resources, there is no need to dedicate machines for scheduling purposes.

5 Decentralized Scheduling

It has been demonstrated that it is possible to construct a centralized scheduling facility that scales well and is highly available. However, high availability comes at the expense of moderate complexity—for purposes of fault tolerance, in particular. As will be shown, decentralized scheduling facilities can reduce the level of complexity at the expense of less scalability.

[6] Actually, state update messages must be sent after a small random delay to avoid overloading the network interface at the server machine. This topic is covered in more detail in Section 5.

5.1 The Basic Model

A fully decentralized design requires that every machine perform its own host selection actions. This can be done either by having every machine keep track of the global system state continually or having it query the system for state information as needed. The former approach is an advertising approach in which machines generate update messages, as in a central design, and broadcast these to all machines. The major problem with such an approach is that every machine, including busy ones, must keep track of incoming updates as well as generate updates of its own. Moreover, if a machine crashes, then it must also somehow reconstruct the state information it has lost.

In contrast, in a query-based approach only those machines interested in host selection must worry about state information. If queries are multicast to a group containing only machines with idle resources, then busy machines never participate in the process at all, unless multicast is implemented in software. In that case, under our assumption of one selection request per machine per minute, a system of 1000 machines would incur a negligible overhead of about 0.7% on each (1 MIPS) machine for discarding multicast packets. For comparison, the overhead of providing round-robin scheduling in the V-System on the same machine is about 1.5% of the CPU. Additionally, there is no failure recovery procedure since state information is obtained only at the time it is needed.

The basic model, then, is one in which a client interested in obtaining a host selection sends a multicast query requesting current state information. The request may contain minimum resource requirement specifications to exclude some replies from being generated in the first place. The client receives replies back from all willing candidate machines and selects the best candidate from that set.

5.2 Dealing with a Large Number of Replies

There are two principal problems with the basic model: the enquirer may receive a very large number of replies almost simultaneously and the N^2 nature of the design quickly consumes network bandwidth. Receipt of many replies prolongs the time required to perform a host selection since the selector must process the reply messages before being able to make a selection. In contrast, a central server can process and order update messages more-or-less separately from selection requests. Moreover, receiving a large number of replies almost simultaneously can overload the hardware and local operating system, as described earlier, resulting in poor host selections if packets lost are from the most suitable hosts. Finally, a large number of replies may consume a significant fraction of a machine's packet buffering capacity, which could cause timeouts of unrelated network communication activities.

An obvious solution is to reduce the number of replies generated—beyond the reduction already obtained by virtue of the notion of an idle group. Doing so requires that global state information be maintained to determine which machines should be in the reply set, and this information must be propagated to those machines. This must be done on a continuing basis since the reply set changes over time. Unfortunately, we have no location, such as a central server, at which to gather the necessary state information and then rebroadcast it. Whereas one could come up with schemes that store state information in a distributed manner among the host machines of the idle group (remember that we don't want to require busy hosts to participate in scheduling), such schemes would add a non-trivial amount of complexity to the design. Since our primary motivation for considering a decentralized design instead of a centralized design is simplicity, we currently question the benefit of such an approach—in contrast to our initial beliefs, presented in [10].

An alternative, and far simpler, approach, is to relax the assumption of optimal scheduling employed thus far. In particular, if we are willing to accept host selections that are not "perfect" in the sense of being the "absolutely" least loaded host in the system, we can have an enquirer only examine the first n replies of a reply set of size r, discarding the rest. Response times and buffering requirements would thus be determined by the value of n, whereas reply set size, r, could be a much larger value.

The rationale for such an approach is that even if the "best" selection candidate isn't obtained, the requesting client will obtain a "good" selection candidate. Furthermore, over a period of several selections, the best candidate(s) will indeed be found by some client. Work by Eager, Lazowska, and Zahorjan [5] demonstrates that this type of approach works surprisingly well, even when only two or three replies are examined (see Section 6).

The error introduced by this approach can be reduced by having reply set members weight their random delays by a factor that reflects how good a candidate they think they are. However, these delays must be kept short in order to avoid excessively prolonging the response time to a host selection request—consider, for example, the situation where all machines are 90% loaded, in which case all replies will be needlessly delayed. The basic solution is to compute the delay as the product of the machine load (normalized to a value between 0 and 1), a standard delay interval, and a uniformly distributed random variable. The normalized machine load ensures that lightly loaded hosts will, on average, respond before more heavily loaded hosts, while the random variable allows for the possibility of a quick response even when all hosts are heavily loaded. Because this solution penalizes the common case of idle machines being available, a refinement is for any machine with a load below a given threshold to reply immediately.

Unfortunately, this design does not address the problem of network bandwidth consumption. If all N machines in a system were available for remote execution—that is, consider themselves members of the idle group, as defined above—and responded to host selection queries, then our limit of 1% of network bandwidth would be exceeded when $N = 85$ for nondistributed ($d = 1$) applications and when $N = 38$ for distributed applications with $d = 5$.

In practice, we expect the frequency of remote execution to be far smaller, so that it is not unreasonable to expect a system of up to a few hundred machines to be feasible. Indeed, if all N machines of a system really are available for remote execution, then they won't all be generating remote execution requests! Furthermore, in this situation there will be few applications generating network traffic, such that a greater fraction of the bandwidth can reasonably be allocated to scheduling activities. Therefore, in order to retain the simplicity of the

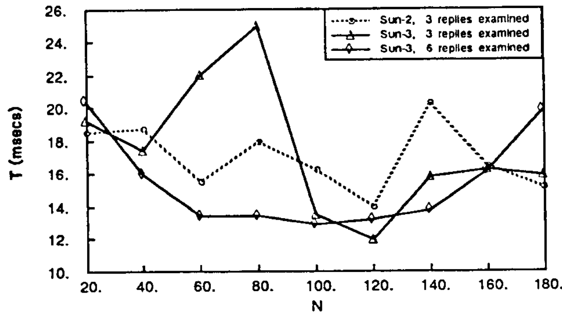

Figure 4: Decentralized scheduling host selection response times as a function of reply set size.

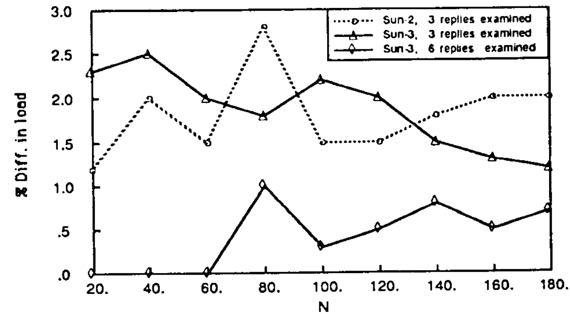

Figure 5: Comparison of host selection results obtained from concurrently running a central scheduler and decentralized host selection queries.

design, we choose not to introduce additional mechanisms to accommodate larger systems.

5.3 An Implementation

To verify our claims about the decentralized design, the centralized scheduling testbed described earlier was extended to have host managers respond to multicast load queries as well, each responding after a weighted delay interval that ranged from 0 to about 10 milliseconds. (Hosts with loads below 10% responded immediately.) Clients could have host selection performed by either the central scheduler, by performing a host selection query themselves, or by having one followed by the other. Thus, we could measure both the response times of the decentralized design as well as compare the quality of its host selections against those returned from the centralized design.

A problem we encountered with our experimental testbed was that V does not provide timing delay facilities with a granularity below 10 msec. In order to implement delays in the range from 0 to 10 msec we employed a suitable number of **GetTime()** calls to the kernel as the delaying mechanism. While calls introduce interference among multiple simulated hosts on a single real machine, resulting in delay intervals that are actually longer than they are supposed to be, this does not invalidate our load comparison test results because of the random nature in which it affects all simulated hosts. It only affects the average length of our decentralized query response times.

Figures 4 and 5 show the results obtained from our test runs. Load results are given relative to the interval over which loads in the simulated hosts were allowed to range. That is, in order to factor out the absolute load interval from which hosts reply to a host selection query and from which hosts update the central server, all load intervals were normalized to range from 0 to 100. The reader should be cautious in attaching any significance to the relative "volatility" of the graphs; the important result in each figure is the bound on the values—the range of response times in Figure 4 and the range in load differentials in Figure 5 (and subsequently, Figure 6). With these caveats in mind, the results and our conclusions can be summarized as follows:

- The response times for the decentralized design are on the order of 20 msec, which we feel are acceptably short, even though they are two to four times greater than those provided by our centralized design for comparable system sizes.
- The response time of the decentralized host selection procedure was independent of the size of the reply set. This is due to the fact that in our system machines allocate at most 20 to 30 packet buffers to their network device, implying that anything beyond the first 20 or 30 replies will never even enter the machine. Thus, our system actually violates our requirement that machines be able to accept every second network packet sent to them, in this case to our advantage.

  The fact that we saturate the network communications resources of a machine so quickly illustrates the point made earlier that multicast can easily interfere with other activities in the workstation by locking them out from the network. In order for this scheduling architecture to work without undue interference it must be possible to inform the operating system when to discard queued reply packets. Otherwise, a machine may be clogged with unneeded reply packets long after the scheduling client has examined the first n packets to a status query, made a scheduling decision, and gone on to other tasks.
- The load selection results in comparison to the "perfect" selections that are returned from the centralized scheduler show that the decentralized scheme performs within about 2% of the centralized scheme when 3 replies are examined. The average standard deviation is about 4%. When 6 replies are examined, the decentralized scheme performs within 1% of the centralized scheme, with the average standard deviation being about 2%. These results are so good because there are almost always machines with loads below the "immediate-reply threshold" (10%) and replies from those machines will be among the first to be received in response to a query.

Another set of experiments forced all simulated hosts to have loads that were in a range from l to 100 and had a single simulated host with a load of 0. These experiments demonstrate how well the decentralized scheme can pick out a single good host candidate from a background of

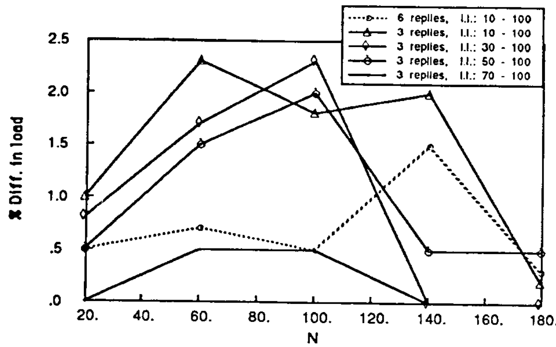

Figure 6: Comparison of host selection results for a reply set containing only one good candidate host.

less suitable host candidates that are nevertheless still in the reply set. Figure 6 displays the results for four values of l: 10, 30, 50, and 70. These correspond to increasingly bad reply sets containing a single good candidate host.

The conclusions we draw from these experiments are:

- The quality of host selections compared to the "perfect" centralized design depends on the odds of receiving the good candidate's reply message as well as the cost of missing it. The odds of missing go down as l increases because of the weighted reply delay times used. The weighting scheme we used made this factor dominate the increasing differences in load quality between the one good host and the rest of the reply set. Consequently the maximum difference in host selection values from those returned by the centralized design occurred for $l = 10\%$.
- In the worst case, however, the average difference in load selections still never exceeded 2.3% when the client examined only 3 host status replies. For this case we also ran the experiment with the client examining 6 replies; which reduced the maximum difference measured down to 1.5%.

5.4 Contention

Since host selections made are not evident to everyone until the time at which a remote execution request is actually sent to some host and accepted there, there is a time interval during which the global state information available to other host selection clients will be out-of-date. Any new host selections initiated during this time interval will see a global state that does not reflect the pending host selection of the first selection, and hence might pick the same host as their remote execution candidate.

Such transients can be detected by having a program execution request include the load it is expecting to see on the machine the request is being sent to. If the actual load is significantly worse, then rescheduling can be performed *before* the cost of program instantiation has been incurred. If the execution request also includes an alternative selection candidate and its expected load, then rescheduling can consist of simply forwarding the request.

One must now worry about cascading effects, where rescheduled execution requests conflict with the later selections made after the one being rescheduled. The forwarding trick as described only works if the probability of two additional selections occurring between a host selection and its associated program load is fairly small. If that is not the case, alternatives can be randomly selected from an appropriately large set of the best candidate hosts.

6 Related Work

Most of the work on multi-processor scheduling consists of theoretical models for load sharing algorithms. An excellent literature survey of this topic is provided in the appendix of Wang and Morris [13]. Our reservations with respect to these models were discussed in the introduction.

6.1 Random Probe Sets

The work most relevant to ours is that of Eager, Lazowska, and Zahorjan [5]. In their design, each server keeps track of a single-valued load metric for itself, such as the job queue length. When a new task is to be executed, it is executed locally if the load metric is below a given *threshold value*. Otherwise, a remote server is selected, which then performs the same threshold decision. Tasks are allowed to be "forwarded" at most some fixed number of times to prevent the system from thrashing (which one can prove that it will inevitably do). Remote servers are chosen by probing a small random set of servers in the system for their loads; the least loaded machine is selected from this *probe set*. Using queueing models and simulation, this simple scheme is shown to perform almost optimally—with remarkably low threshold values (job queue length of 1) and probe set sizes (2 to 5, depending on server utilization).

The advantages of this algorithm are clear: bounded, small communication costs, provable stability, and excellent performance under the environmental assumptions made. Unfortunately, the algorithm relies on homogeneity and statistical assumptions similar to those outlined earlier, which we have argued do not hold in our environment. Nevertheless, the approach provides the basis for our decentralized architecture. By restricting the set of hosts that are examined to only "reasonable" candidates, we achieve a degree of homogeneity similar to that required by their design. Since only reasonable candidates are examined, and replies are temporally ordered according to load, our design should out-perform their's—in terms of goodness of selection.

6.2 Stumm

In [9], Stumm advocates a decentralized scheduling design that is based on advertising rather than querying. He argues that advertising state information generates $O(N)$ update messages, whereas querying for state information will generate $O(N^2)$ query/query reply messages, and therefore a design based on advertising will be more scalable. In exchange for this added scalability, his decentralized design requires a more complicated implementation: Hosts must continually keep track of the global state of the system even when they have no intention of invoking any remotely executing programs, and new hosts must either wait for periodic status update messages to

initialize their "monitoring" records with or must query the system for that information. Stumm also dismisses centralized designs as being undesirable on general (aesthetic) principles.

We, however, consider a centralized design acceptable. Indeed, our centralized design is even more scalable than Stumm's decentralized design and is effectively as fault tolerant if clients are willing to occasionally wait a few seconds to allow reinstantiation of failed servers. We consider the principal advantage of a decentralized design to be its potential for greater *simplicity*, not *scalability*. Thus, when choosing a decentralized design, we chose to base it on querying rather than advertising.

6.3 Zhou and Ferrari

Zhou and Ferrari have performed a number of experiments with load balancing algorithms in the context of UNIX [14]. Several of their conclusions are consistent with ours—in particular with regard to the advantages of using a central server. On the other hand, their conclusion that decentralized algorithms are more scalable is based on their examination of the random probe set algorithm of Eager, Lazowska, and Zahorjan, whose workload assumptions we have already taken exception to above.

7 Concluding Remarks

We have described two radically different scheduling architectures that perform well (in terms of response time, quality of host selection, and overhead incurred by the system to support the facilities), and are highly available. Although the architectures are comparable in terms of availability and performance, tradeoffs exist with respect to complexity of implementation and scalability.

The centralized architecture we have presented is capable of handling a system of thousands of machines, at least as far as the scheduling service is concerned. This level of scalability is achieved by limiting status update traffic to only those machines that are likely host selection candidates. Unreliable multicast communication is used to efficiently control which machines should view themselves as such candidates, as well as to allow quick reinstantiation of the server after a crash.

Our decentralized architecture is based on similar notions: Host selection is performed by sending a query message to a multicast group containing only willing candidates and only processing the first n replies sent by the group's members. The rest of the replies are either discarded or lost in the flood of replies that each query message generates. Even if the reply from the "best" candidate is lost as a result, the requesting client will nevertheless obtain a "good" selection. Furthermore, over a period of several selections, the best candidates will indeed be found by some client. By relying on a statistical algorithm, we are able to avoid the problems of reliably receiving a large number of reply messages within a short period of time. This approach also limits the amount of time required to perform a host selection by limiting the number of replies that must actually be processed.

Unfortunately, network bandwidth considerations limit our decentralized architecture to handling systems of at most a few hundred machines, significantly less than the centralized architecture. In exchange for this limit on scalability we obtain a simpler design. There are no failure recovery procedures and no need to migrate a scheduling server between idle machines if a dedicated server machine isn't available. The last point becomes especially important if the system does not support migration, in which case the availability of a central server requires periodic reinstantiation (which is significantly slower than migration—see [11]).

Given only the tradeoff between scalability (assuming efficient, although unreliable, multicast) and simplicity, we prefer the latter and argue that for most systems the simple decentralized design will be more than adequate. The parameters we have picked for our scaling estimates are conservative—the entire premise of remote execution is based on the assumption that most machines are effectively idle, implying that much larger systems should be manageable in practice. Note that if we relax our network bandwidth limits from 1% to 4% then we should be able to handle systems twice as large as before—at least as far as network bandwidth constraints are concerned.

We argue that one should switch to a centralized design when scalability beyond a few hundred machines becomes a significant issue, or when other issues (such as global fairness or network management) come into play. Note that the implementation of the client code need not change for such a conversion—the centralized scheduling server can simply be the only member of the idle resources multicast group. We suspect that for system sizes beyond a few hundred machines issues other than simple load-based scheduling will become of dominant importance. A centralized architecture may be preferable when such considerations come into play. Principal of these is the question of how to extend our architectures to a wide-area or internetwork environment in which efficient multicast facilities may not be readily available. While efforts such as [3] and [6] may show how to provide for suitable multicast facilities, a centralized architecture can be extended into such an environment in a straightforward manner by providing a hierarchy of scheduling servers that cooperate with each other. Another alternative is a hybrid approach in which central servers would participate as "agents" of remote networks in a decentralized design (whose effective system size would still be small). Examination of some of these alternatives represents future work we intend to pursue.

Other areas that have not been dealt with in this paper are those of fairness and load balancing. As argued in [10], such issues should not be of concern in an environment in which there are many idle machines available. However, if long-running and/or massively parallel applications become more prevalent then this may no longer be true. In that case the tradeoffs between architectures may again be tipped toward a centralized one.

In conclusion, we view the scheduling architectures we have described as a demonstration that global scheduling facilities for finding idle machines in a workstation-based distributed system can be implemented that are highly scalable, fault tolerant, and straightforward to implement. The reliance on efficient multicast facilities has allowed us to improve both centralized and decentralized architectures in a significant manner. By investigating both approaches we have allowed ourselves room for various futures in which the nature of available multicast

facilities may be different and in which differing workloads may imply the need for additional global scheduling criteria such as fairness and load balancing.

8 Acknowledgments

The bulk of this research was conducted by the authors while they were with the Department of Computer Science, Stanford University. It was supported in part by the Defense Advanced Research Projects Agency under contracts MDA903-80-C-0102 and N00039-83-K-0431 and by the National Aeronautics and Space Administration under contract NAGW-419. Thanks also to the other members of the Stanford Distributed Systems Group, which created the V-System and provided a sounding board for the ideas presented here.

References

[1] R. Agrawal and A.K. Ezzat. Processor sharing in NEST: A network of computer workstations. In *Proc. 1st International Conference on Computer Workstations*, pages 198–208, IEEE, November 1985.

[2] E.J. Berglund. An introduction to the V-System. *IEEE Micro*, 35–52, August 1986.

[3] D.R. Cheriton and S. Deering. Host groups: a multicast extension for datagram internetworks. In *Proc. 9th Data Communications Symposium*, pages 172–184, IEEE, September 1985.

[4] D.R. Cheriton and W. Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, May 1985. Presented at the SIGCOMM '84 Symposium on Communications Architectures and Protocols, ACM, June 1984.

[5] D.L. Eager, E.D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, SE-12(5):662–675, May 1986.

[6] A.J. Frank, L.D. Wittie, and A.J. Bernstein. Multicast communications on network computers. *IEEE Software*, 2(3):49–61, May 1985.

[7] L. Kleinrock. *Queueing Systems Volume 1: Theory.* Wiley, 1975.

[8] R.F. Rashid. Threads of a new system. *UNIX Review*, 4(8):36–49, August 1986.

[9] M. Stumm. The design and implementation of a decentralized scheduling facility for a workstation cluster. In *Proc. 2nd IEEE Conference on Computer Workstations*, pages 12–22, IEEE, March 1988.

[10] M. M. Theimer. *Preemptable Remote Execution Facilities for Loosely-Coupled Distributed Systems.* PhD thesis, Stanford University, 1986. Technical Report STAN-CS-86-1128, Department of Computer Science.

[11] M.M. Theimer, K.A. Lantz, and D.R. Cheriton. Preemptable remote execution facilities for the V-System. In *Proc. 10th Symposium on Operating Systems Principles*, pages 2–12, ACM, December 1985. Proceedings published as *Operating System Review* 19(5).

[12] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS distributed operating system. In *Proc. 9th Symposium on Operating Systems Principles*, pages 49–70, ACM, October 1983. Published as *Operating Systems Review* 17(5).

[13] Y. Wang and R.J.T. Morris. Load sharing in distributed systems. *IEEE Transactions on Computers*, C-34(3):204–217, March 1985.

[14] S. Zhou and D. Ferrari. *An experimental study of load balancing performance.* Report UCB/CSD 87/336, Computer Science Division (EECS), University of California - Berkeley, January 1987.