A LOAD DISTRIBUTION POLICY FOR A NEW TRANSCATION SERVICE CONSIDERING THE PRE-LOADED SERVICES

by

SUKANYA SURANAUWARAT

Department of Computer Science, School of Applied Statistics,
National Institute of Development Administration,
Bangkok 10240, Thailand

October 16, 2003

A LOAD DISTRIBUTION POLICY FOR A NEW TRANSCATION SERVICE CONSIDERING THE PRE-LOADED SERVICES

SUKANYA SURANAUWARAT
Department of Computer Science, School of Applied Statistics,
National Institute of Development Administration,
Bangkok 10240, Thailand

ABSTRACT

In this paper, we describe and evaluate our load distribution policy. This policy makes placement decisions for processes of each service in such a way that the estimated response time is optimized while the estimated throughput is in the range that the user desires. Our target service is a transaction service consisting of multiple processes that communicate with each other. Our target system is a distributed system consisting of workstations connected by a local area network. The outstanding feature of our policy is that it takes account of pre-loaded services, ones for which process allocation has already been determined before the newly arrived service. In other words, our policy tries to keep the performance of the pre-loaded services at a desirable level while the newly arrived service processes are loaded and run. We measured the response time and the throughput when the process allocation is determined using our policy, and we compared them with those our policy estimates. The results show that our policy provides an appropriate process allocation, and that calculated results agree well with the measured ones.

A LOAD DISTRIBUTION POLICY FOR A NEW TRANSCATION SERVICE CONSIDERING THE PRE-LOADED SERVICES

SUKANYA SURANAUWARAT
Department of Computer Science, School of Applied Statistics,
National Institute of Development Administration,
Bangkok 10240, Thailand

1 Introduction

The advent of inexpensive microcomputers, powerful workstations, and wide-bandwidth networks caused distributed systems to become increasingly popular. One of the most important features of such systems is the possibility of achieving high performance by means of load distribution. Load distribution attempts to improve the performance of application programs by distributing their processes (i.e., the load) among the computers in a network.

The effect of load distribution depends on the load distribution policy. So far, many load distribution policies (e.g., [1]-[9], [11]) have been proposed, most of which focused on the load on a single resource in a computer especially a CPU resource, even though processes usually require various types of resources for their execution. In this case, it may not be meaningful to assign an incoming I/O-bound process to a computer that the CPU is lightly loaded while the I/O resources it uses are not. In other words, an incoming I/O-bound process will not necessarily run faster on a more lightly loaded CPU, and it might even run slower if the I/O resources it uses are congested. Moreover, the previous studies [1]-[8] have demonstrated the performance benefit of their policies using simple workload and system models. For example, the workload consists of only CPU-bound processes and all the computers in a network are identical, which makes the comparison of the load among the computers easier since the information about the specifications of the computers and the network is not a concern, and thus the processes are simply assigned to the computers that have the least number of processes. Another example, only application programs that generate a single process are considered, thus the relation and interaction between processes of the same program is not a concern.

Therefore, the policies mentioned above are not suitable for the

programs that are widely used in business and require have high performance like transaction processing programs, because the execution of each transaction processing program usually causes the creation and the execution of multiple cooperative processes which require various types of resources and interprocess communication (IPC) in order to carry out each transaction. Hence, we propose a load distribution policy for transaction processing programs on distributed systems consisting of heterogeneous workstations connected by a local area network (LAN). Unlike previous policies, our policy takes into account the information about the specifications of the computers and the network and the information about the execution behavior of the programs, i.e., the resource consumption behavior of each process and its interaction with other processes (i.e., IPC), in making placement decisions for each process. To be more specific, our policy estimates the response time and the throughput of any transaction service using the information about the specifications of the computers and the network and the information about the execution behavior of the programs, and distributes each process of the service in such a way that the estimated response time is optimized while the estimated throughput is in the range that users desire (i.e., our policy allows users to set the lower and the upper limits of the throughput).

However, when some services have already been executed on the distributed system (pre-loaded services), it becomes harder to estimate the response time, because the correlation between the newly arrived service (new service) and the pre-loaded services has to be considered. For example, consider a situation in which a process of a new service is assigned to a computer. Beginning an execution of a new process on a computer requires reallocation of the computing resource (such as the CPU or memory) for each process running on the computer. This will affect the time it takes to complete one transaction in each process, and thus affecting the processes of the same services running on the other computers, because the processes use IPC. Consequently, just one new process affects all the other processes in the system. The effect spreads through the system until all the processes on the system are stable. This makes it difficult to take account of the pre-loaded services in the calculation. We name this problem the *process chain problem*. As a solution, we introduce the *bottleneck coefficient* into our load distribution policy.

The rest of this paper is organized as follows. Section 2 briefly discusses the related work. Section 3 describes the model of our system. Section 4 describes the method of estimating response time of a transaction service. Section 5 describes our load distribution policy. Section 6 describes our experiments and explains the results we obtained. Section 7 offers our conclusions and future work.

2 Related Work

Load distribution methods that have been proposed in previous studies [1]-[8] can be classified roughly into two types: (1) those based on the centralized dispatcher model [1]-[3], and (2) those based on the autonomous dispatcher model [4]-[8]. In the centralized dispatcher model, just one computer called dispatch server makes the placement decisions for all processes. This model is simple but not scalable, because a dispatch server might end up being a bottleneck of an entire system. In the autonomous dispatcher model, each computer in the system can make placement decisions for the processes submitted to it, which leads to a robust system from the viewpoint of fault tolerance. However, each computer has to collect load information from other computers, which may cause the network to get clogged up. Besides, in these models, the target services are the services that consist of a single process, which has no IPC [1]-[8]. Therefore, they are not suitable for handling transaction services with multiple processes.

Load distribution methods that support multiple processes on a multiprocessor system have also been reported [9]. However, their system models are very different from those of distributed systems.

For the discussion of load balancing, simplified models of distributed systems are usually adopted [1]-[9]. For example, no network latency [4][5][9], or no I/O is considered [1]-[9]. Another example, all the computers in a network are assumed to be homogeneous [4][7], since it is easier to make the comparison of the load among the computers and thus the processes are simply assigned to the computers that have the least number of processes.

Therefore, the methods mentioned above are not suitable for transaction processing programs because the execution of them causes the creation and the execution of multiple processes which require various types of resources including I/O and need to communicate with one another. Also, practically and naturally, computers in the distributed systems tend to be heterogeneous.

Previous methods proposed in [4][6][8][10] make the placement decisions for processes in a similar way to ours, using the following steps: (1) select a process to reallocate, (2) select a workstation for the selected process, and (3) decide whether or not to reallocate. The details of each step vary from method to the method and can be found in the corresponding papers. Our method differs from the previous ones in that it takes account of pre-loaded services to estimate response times and throughputs more precisely. Exact estimation is very important, because an inaccurate estimation may cause unpredictable performance degradation [10][12]. In Section 6, we will show that taking account of pre-loaded services allows more precise estimation.

Selecting a process to reallocate can be done either before beginning its execution or after its execution has begun. Our method and those mentioned above are the former type, while the methods based on the latter type can be found in [13]-[15], for example.

3 System Model

Our target service is a transaction service in which the behavior of each process has been understood before executing our load distribution policy. Our target system is a distributed system consisting of heterogeneous workstations connected by a LAN. Each workstation has its own CPU and I/O devices. The process scheduling at each workstation is based on the roundrobin algorithm used in UNIX operating systems. Each process can be executed at any workstation in the network. We consider only local file access.

The model of our system can be divided into *two* parts: the *process structure*, which represents the behavior of each process, and the *system structure*, which represents the specifications of the workstations and the network. Tables 1 and 2 show the elements and the parameters of the process structure and the system structure respectively.

Table 1: Elements of the process structure.

	element	parameter	notes
1	CPU	number of dynamic	effect on IPC and I/O
2	IPC	frequency and timing,	with or without a
3	IPC	frequency and timing,	with or without a
4	I/O	frequency and timing,	–
5	I/O	frequency and timing,	–
6	service scale	number of processes	–
7	process scale	process size (bits)	–

Table 2: Elements of the system structure.

	classificati	element	parameter
1	hardware	CPU	performance (MIPS)
2		I/O device	rate (bits/s)
3		network interface	rate (bits/s)
4		system scale	number of CPUs
5	OS	I/O processing	number of dynamic steps
6		protocol processing	number of dynamic steps
7		IPC processing	number of dynamic steps

4 Method of Estimating the Response Time of a Transaction Service

The response time of each program is dependent on the behavior of the processes it creates. Some programs may create a single process while some may create several processes. For the programs consisting of a single process, their processes generally involve two kinds of processing — CPU processing and I/O. For the programs consisting of multiple cooperative processes such as transaction processing programs, their processes involve not only CPU processing and I/O but also IPC. We will refer to the time spent on CPU processing and I/O as the *process time*.

In this section, we first describe our method to estimate the process time of a single process, and we then give an extended version of this method to estimate the process time of a cooperative process. After that, we will describe how to estimate the response time of a transaction service using the process times of cooperative processes that we estimated.

4.1 Estimating the Process Time of a Single Process

Any single process will generally change among run, wait and ready states in order to carry out its job, as shown in Figure 1. A process is said to be running in the run state if it is currently using the CPU to do its CPU processing work. When the process in the run state makes an I/O request, it will change from run state to wait state. If at that time there is another process using the desired I/O resource (e.g., a disk drive), then its I/O request will be put into the I/O queue until the desired I/O resource becomes available. On the other hand, if the desired I/O resource is available at that time, then that process will have access to it; thus its I/O request will be serviced immediately. When its I/O request completes, an I/O interrupt is caused and then that process will change from wait state to ready state. In this case also, if there is another process using the CPU, then that process needs to wait in the ready queue until the CPU becomes available. In this paper, we will refer to the time a process uses the CPU until it makes an I/O request as the *CPU processing time (CpuTime)*, the time a process uses to wait for an I/O device to become available in the I/O queue as the *I/O queue time (IoQueueTime)*, the time a process uses to access an I/O device as the *I/O accessing time (AccessTime)*, and the time a process waits for the CPU to become available in the ready queue as the *ready queue time (ReadyQueueTime)*. Therefore, the process time of any single process is the sum of its total CPU processing time, its total I/O queue time, its total I/O accessing time, and its total ready queue time.

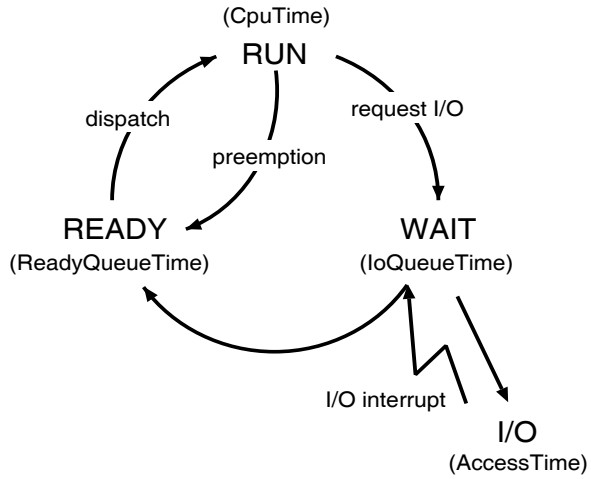

Figure 1: Diagram of process state.

4.1.1 CPU Processing Time

In this section, we describe how we estimate the CPU processing time of each process.

Suppose that $steps_i$ is the total amount of CPU processing process *i* does, and *IO* is the total number of times process *i* makes I/O requests, then the average amount of CPU processing between I/O requests (μ) of process *i* can be calculated as follows.

$$\mu_i = \frac{steps_i}{IO_i} \tag{1}$$

Therefore, the CPU processing time (*CpuTime*) of process *i* when it runs on a workstation *m* where the performance of the CPU is $mips_m$, can be estimated as follows.

$$CpuTime(m, \mu_i) = \frac{\mu_i}{mips_m} \tag{2}$$

We note that, $steps_i$ is given through the first element of the process structure, IO_i is given through the fourth and the fifth elements of the process structure, and $mips_m$ is given through the first element of the system structure.

4.1.2 I/O Queue Time

In this section, we first define the following two terms: a *CPU-intensive process* and an I/O-*intensive process*, and we then describe how we estimate the I/O queue time of each process by using an example shown in Figure 2.

A CPU-intensive process is a process that has the CPU processing time longer than the I/O accessing time ($CpuTime > AccessTime$), while an I/O-intensive process is a process that has the CPU processing time shorter than the I/O accessing time ($CpuTime < AccessTime$).

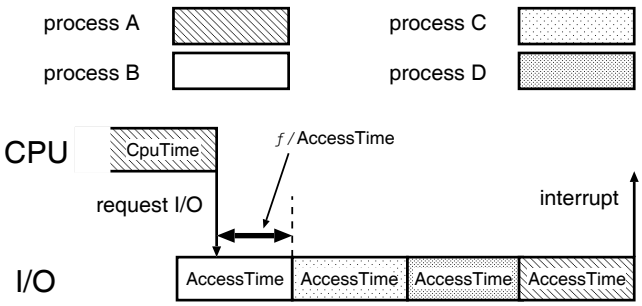

Figure 2: I/O queue time.

Figure 2 shows a situation in which both CPU-intensive and I/O-intensive processes running on the same workstation make I/O requests. In this figure, process A is a CPU-intensive process while processes B, C, and D are I/O-intensive processes. In order to simplify the figure, we show an I/O request and an I/O interrupt pertaining to only process A. If process A makes an I/O request to the I/O device that process B is still accessing to at time *t*, then the rest of the I/O accessing time of process B ($\Delta AccessTime$) can be expected to be $AccessTime/2$. Suppose that ζ is the time used to operate the I/O queue for an I/O request, η is the time used to handle an I/O interrupt caused by the completion of an I/O request, and n_{io} is the total number of I/O-intensive processes on workstation *m*, then the I/O queue time (*IoQueueTime*) of process *i* on workstation *m* could be estimated as follows.

$$IoQueueTim\ e(m, n_{io}) = (\Delta AccessTime\ + \zeta) + \sum_{j}^{n_{io}-1} \{AccessTime\ (m) + \eta\} \qquad (3)$$

4.1.3 I/O Accessing Time

In this section, we describe how we estimate the I/O accessing time of each process.

The I/O accessing time of each process depends on the size of the data it requests to read from or to write to an I/O device for each I/O request. Therefore, we used the results obtained from the following measurement to estimate the I/O accessing time. In our measurement, we ran a process that loops an I/O request of writing 1 KB of data to a disk drive, and measured the time used for each I/O request. Since our process is an I/O-intensive process that does not do any CPU processing (i.e., $\mu = 0$) and does not coexist with any other process while running, its CPU

processing time, its I/O queue time and its ready queue time are all zero. Therefore, the measured time can be thought as the summation of the I/O accessing time and the time used to handle the I/O interrupt (η). Since η is very small compared to the I/O accessing time, it can be negligible. Therefore, the I/O accessing time (*AccessTime*) of process *i* for writing 1 KB of data on workstation *m* can be estimated to the measured time on workstation *m*.

4.1.4 Ready Queue Time

In the following, we describe how we estimate the ready queue time of each process by using an example shown in Figure 3.

Figure 3 shows the CPU consumption behavior of both CPU-intensive processes (i.e., processes A, B, and C) and I/O-intensive processes (i.e., processes D and E) that are running on the same workstation. In this figure, blocks A, B, C, D, and E represent the CPU usage of processes A, B, C, D, and E, respectively. In addition, we assume that the process scheduling algorithm is that used in UNIX operating systems, in which processes that move to the ready state after finishing an I/O in the wait state are given high priority. Under this assumption, the ready queue time of an I/O-intensive process can be thought of as zero.

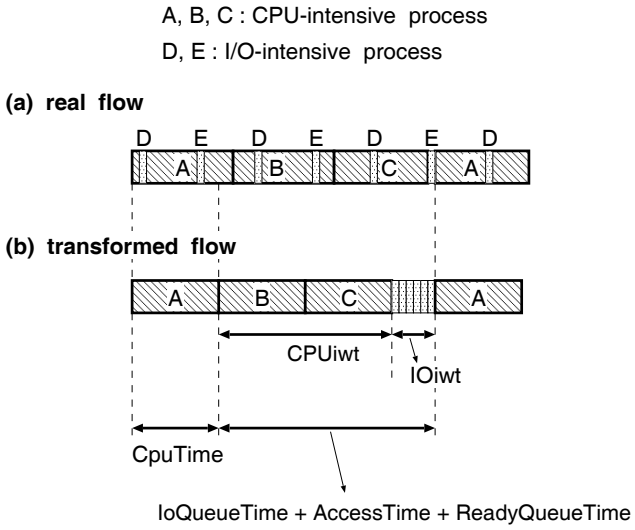

Figure 3: Ready queue time.

Figure 3(a) shows the real behavior of processes while Figure 3(b) shows the simplified version of Figure 3(a). As shown in Figure 3(b), the sum of the I/O queue time, the I/O accessing time, and the ready queue

time of CPU-intensive process A, equals the time process A needs to wait in the ready queue until it will be dispatched to the run state again, which is the sum of the time (*CPUiwt*) it needs to wait for other CPU-intensive processes to use the CPU and the time (*IOiwt*) to wait for all the I/O-intensive processes to use the CPU.

Therefore, the ready queue time (*ReadyQueueTime*) of process *i* on workstation *m* can be estimated as follows, where $i = I/O$ represents the case that process *i* is an I/O-intensive process whereas $i = CPU$ represents the case that process *i* is a CPU-intensive process.

$$ReadyQueueTime(m, \mu_i, n_{cpu}, n_{io}) = \begin{cases} 0 & , i = I/O \\ CPUiwt + IOiwt & , i = CPU \\ -(IoQueueTime + AccessTime) & \end{cases} \quad (4)$$

How to estimate *CPUiwt*. When thinking about a sufficiently long period of time, the amount of time the CPU is given to each CPU-intensive process is about the same , as shown in Figure 3, without regard to the average amount of CPU processing between I/O requests (μ) of each process. Therefore, given the CPU processing time (*CpuTime*) of process *i* on workstation *m* and the total number of CPU-intensive processes (n_{cpu}) on workstation *m*, the time (*CPUiwt*) process *i* needs to wait for other CPU-intensive processes to use the CPU can be estimated as follows.

$$CPUiwt(m, \mu_i, n_{cpu}) = CpuTime(m, \mu_i) \times (n_{cpu} - 1) \quad (5)$$

How to estimate *IOiwt*. *IOiwt* is dependent on how often each I/O-intensive process is dispatched to the run state and on how much CPU time each I/O-intensive process consumes at each dispatch. Since the number of times each I/O-intensive process is dispatched is in turn the number of I/O requests it makes, we decided to measure the number of I/O requests each I/O-intensive process makes per unit of time as the frequency of each I/O-intensive process being dispatched. The measurement result is shown in Figure 4. In this figure in which we varied the number of I/O-intensive processes from 1 to 15, we measured the number of I/O requests (each of which is to write 1 KB of data) each process makes per second. Figure 4 shows that, as the number of processes increases, the number of I/O requests each process makes decreases without regard to its average amount of CPU processing between I/O requests (μ). We also measured the CPU time each I/O-intensive process consumes, and the measurement result is shown in Figure 5. Figure 5 shows that, as the average amount of CPU processing between I/O requests (μ) is increased, the CPU time each

process consumes is increased. This means that an increase in the CPU time consumed by each I/O-intensive process is due to its CPU processing time (*CpuTime*). Hence, the CPU time (Δ*IOiwt*) I/O-intensive process *j* on workstation *m* consumes at each dispatch can be estimated as follows, where θ is the CPU time it consumes in the case that its CPU processing time is zero (i.e., $\mu = 0$), which is in turn the time used to handle an I/O interrupt.

$$\Delta IOiwt(m, \mu_j) = \theta + CpuTime(m, \mu_j) \tag{6}$$

Therefore, the time (*IOiwt*) process *i* on workstation *m* needs to wait for all the I/O-intensive processes to use the CPU can be estimated as follows, where n_{io} is the total number of I/O-intensive processes on workstation *m* and $\xi(n_{io})$ is the number of times each I/O-intensive process is dispatched.

$$\begin{aligned} IOiwt(m, \mu_i) = \sum_{j}^{n_{io}} \Delta IOiwt(m, \mu_j) \times \xi(n_{io}) \\ \times \left\{ CpuTime(m, \mu_i) + IOiwt(m, \mu_i) + CPUiwt(m, \mu_i, n_{cpu}) \right\} \end{aligned} \tag{7}$$

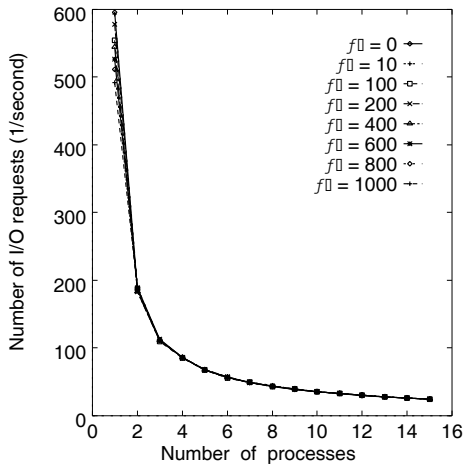

Figure 4: Number of I/O requests each I/O-intensive process makes per second.

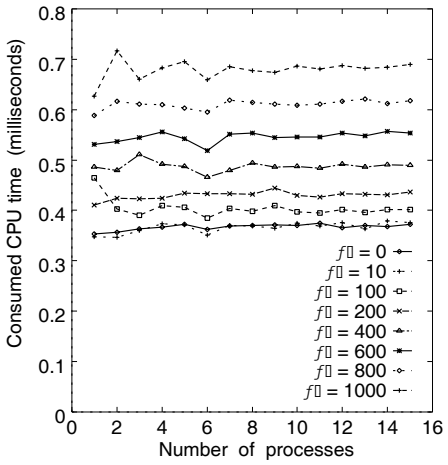

Figure 5: The CPU time each I/O-intensive process consumes.

4.1.5 Process Time

Suppose that v_i is the number of times process *i* on workstation *m* makes I/O requests, n_{cpu} is the total number of CPU-intensive processes on workstation *m*, n_{io} is the total number of I/O-intensive processes on workstation *m*, then the process time (*ProcessTime*) of process *i* can be estimated as follows.

$$\begin{aligned} ProcessTime(m, \mu_i, n_{cpu}, n_{io}, v_i) = \{ & CpuTime(m, \mu_i) \\ & + IoQueueTime(m, n_{io}) \\ & + AccessTime(m) \\ & + ReadyQueueTime(m, \mu_i, n_{cpu}, n_{io}) \} \\ & \times v_i \end{aligned} \quad (8)$$

4.2 Estimating the Process Time of a Cooperative Process

The process chain problem makes it difficult to take account of pre-loaded services in estimating the process time. We describe our idea, which we call the bottleneck coefficient, for solving this problem. Our approach is to estimate continuously, taking account of changes in all the processes. The following describes our approach in detail using an example shown in Figure 6.

In Figure 6, A, B, and C are cooperative processes of the same transaction service. If the processes in Figure 6(a) do not need to cooperate with one another, in other words, if each of them acts as a single

process, then their process time can be estimated using equation (8). In such a case, the reciprocal of the process time of each process, which will be referred to as *process throughput*, will be 8, 4, and 2. Since cooperative processes need to communicate with one another, the process throughput of the process that has the smallest process throughput among all the processes of the same service could be the throughput of the service. We will refer to such a process as a *bottleneck process (BP)*. As shown in Figure 6 (a), process C is the BP and its process throughput is the throughput of the service, which is 2. Figure 6(b) shows how processes A, B, and C are being executed. In this figure, a dotted line and a solid line indicate respectively a period of time when a process is waiting for the arrival of the data from another cooperative process and when it is not, in which case a process may use the CPU in the run state, or may wait for an I/O device to become available in the wait state, or may access to an I/O device in the wait state, or may wait for the CPU to become available in the ready state.

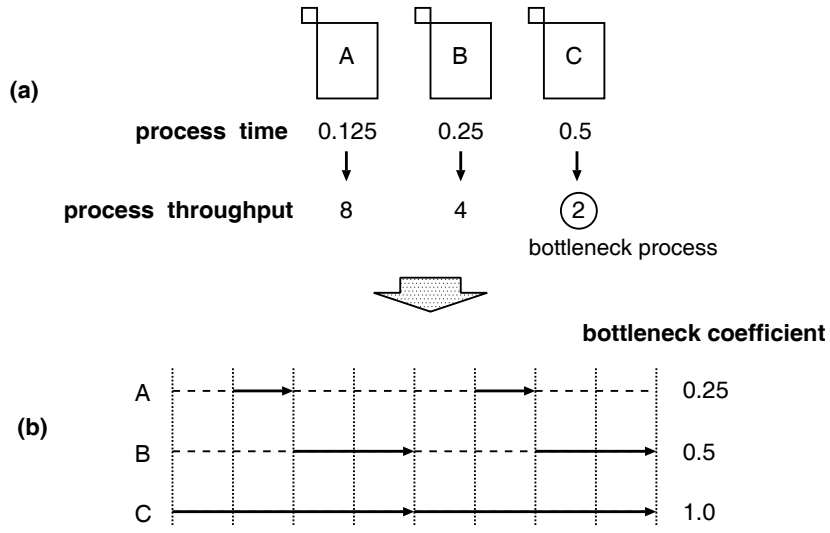

Figure 6: Bottleneck coefficients of cooperative processes of the same service.

Suppose that PTH_i is the process throughput of process *i*, we define the bottleneck coefficient (*BC*) of process *i* as follows.

$$BC_i = \frac{PTH_{BP}}{PTH_i} = \frac{ProcessTime_i}{ProcessTime_{BP}} \quad (9)$$

BC_i indicates the probability that process *i* is running (i.e., process *i* is not waiting for the arrival of the data from another cooperative process) at an arbitrary time *t*, as shown in Figure 6(b). Thus, the expected number of CPU-intensive processes (n'_{cpu}) that are running simultaneously on workstation *m* at an arbitrary time *t* can be estimated from $\sum_j^{n_{cpu}} BC_j \times 1$, and the expected number of I/O intensive processes (n'_{io}) that are running simultaneously on workstation *m* at an arbitrary time *t* can be estimated from $\sum_j^{n_{io}} BC_j \times 1$. Therefore, the process time of a cooperative process *i* on workstation *m* can be estimated by substituting n'_{cpu} for n_{cpu} and n'_{io} for n_{io} in equation (8). In this way, we can get a new process time and a new process throughput for each process of the same service.

Since the change in the new and the old process times affects the rest of the services, and therefore we have to take them into account in order to estimate the response time or throughput more precisely. Our idea is that it is possible to generate even the new *BC* from the new process time. This implies that we can repeatedly estimate the response time of each service on the basis of a new process time, until the difference between the new and old process times becomes small.

We will describe how this idea can be applied to our load distribution policy in Section 5.

4.3 Estimating the Response Time

The response time of a transaction service is determined by (1) the process time of cooperative processes, (2) the IPC time between processes, and (3) the processing flow of the program. The process time of cooperative processes can be estimated using our proposed method. The IPC time between processes is dependent on network protocols and the utilization of communication channels. Therefore, we decided not to discuss the method of estimating it here, since it will make our study too complicated to start. Finally, the processing flow of a program is known through the second element in the process structure (i.e., IPC (synchronous)) of each process of the same service.

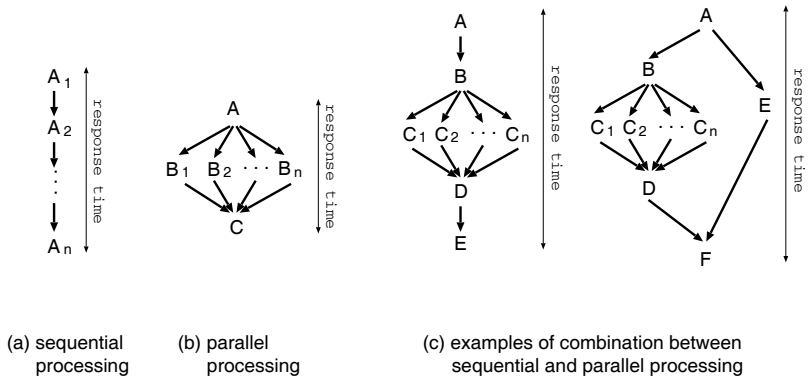

Figure 7: Examples of the processing flow of a program.

Examples of the processing flow of a program are shown in Figure 7. In this figure, A, B, C, D, E, and F represent processes A, B, C, D, E, and F respectively, and an arrow represents IPC. For a transaction processing program, its processing is usually a combination between sequential and parallel processing as shown in Figure 7(c).

Also, the throughput of a transaction service can be regarded as the reciprocal of the response time.

5 Load Distribution Policy

This section describes our load distribution policy, which makes placement decisions for each process before beginning its execution. The goal of our policy is to minimize the response time of a new coming service for a range of throughputs which is set by users in advance.

Figure 8 shows the processing flowchart of our policy and it is described in detail as the following steps.

(1) Select a process to reallocate (or allocate if it has not been allocated yet) from all the processes of a new service, based on the *process selecting rule*. If all the processes have already been *LOCKED*, then exit the load distribution process since there are no more eligible processes for reallocating. We note that when a process is LOCKED, it is not allowed to be reallocated.

(2) Select a workstation for the selected process based on the *workstation selecting rule*. If there are no more eligible workstations, then the selected process will be LOCKED since reallocating the selected process is not likely to make any improvement in the response time.

(3) Estimate the response times and the throughputs of all the services on the assumption that the selected process has been reallocated on the selected workstation.

(4) If the estimated response times and the estimated throughputs satisfy the *reallocating conditions*, reallocate the selected process to the selected workstation and go back to step 1 to select a new process. Otherwise, go back to step 2 to select a new workstation.

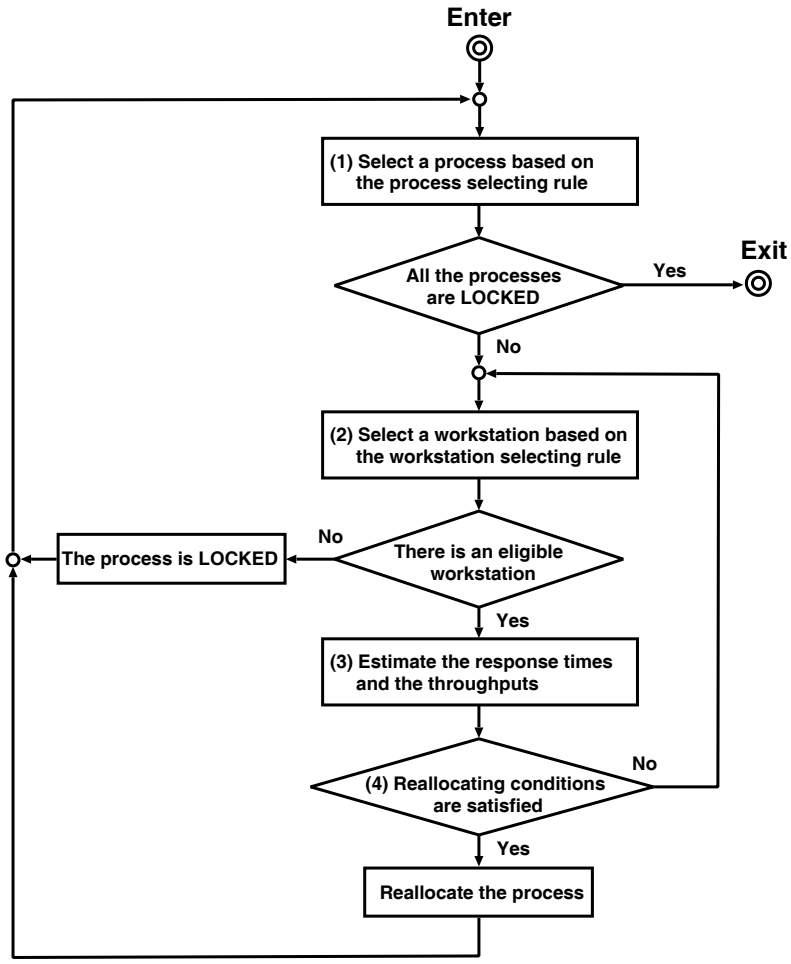

Figure 8: Processing flowchart of our load distribution policy.

The process selecting rule (Section 5.1), the workstation selecting rule (Section 5.2), a description of how to estimate the response time and throughputs under the assumption that the selected process is reallocated to the selected workstation (Section 5.3), and reallocating conditions (Section 5.4), are described in detail below.

5.1 Process Selecting Rule

The following describes how to select a process to reallocate.

(1) If there are some processes that have not been allocated yet, then select the process that has the largest amount of CPU processing among them.

(2) If all the processes have been allocated, then select the process that has the longest process time among all the processes that are not LOCKED.

5.2 Workstation Selecting Rule

We define the *CPU priority* and the *I/O priority* for each workstation. The CPU priority expresses the CPU utilization and the degree of contention at the CPU. A workstation with high CPU priority is expected to provide high performance for the CPU processing. In the same way, the I/O priority is used. Since a process can be either CPU-intensive or I/O-intensive according to the computer it is running on, we cannot determine which workstation — a workstation with high CPU priority or that with high I/O priority — should be assigned to the selected process. We solve this problem by making two lists: the CPU priority list and the I/O priority list. The CPU and I/O priority lists are respectively composed of all the workstations sorted by their CPU and I/O priorities. Based on these two lists, a workstation will be selected as follows.

(1) Select the workstation at top of the CPU priority list. In this case, if the reallocating conditions are not satisfied, then

(2) select the workstation at the top of the I/O priority list. In this case, if the reallocating conditions are not satisfied, then

(3) select the next workstation from the CPU priority list or from the I/O priority list alternately until the reallocating conditions are satisfied or until there are no more workstations left in both lists, in which case the selected process will be LOCKED.

When the assignment of the selected process to the selected workstation is determined, both CPU and I/O priority lists will be updated and recomposed if necessary.

5.3 Estimating Response Time and Throughput Under Reallocating Assumption

The following steps describes how to estimate the response times and the throughputs of all the services on the assumption that the selected process *i* has been reallocated from workstation *s* to the selected workstation *d*.

(1) Regard the bottleneck coefficients (*BC*s) of all the processes on workstations *s* and *d* as 1. (This means regard each of them as a single process.)

(2) Apply equation (8) to estimate the process times for all the processes on workstations *s* and *d*.

(3) If there is no change in any of the process times that exceeds the threshold ρ, then go to step 4. Otherwise, calculate n'_{cpu} and n'_{io} to repeat the estimation, and then go back to step 3.

(4) Calculate each process throughput as the reciprocal of each process time, and then find the bottleneck process (*BP*) of each service.

(5) Calculate *BC*s of all the processes on all the workstations.

(6) Calculate n'_{cpu} and n'_{io} on each of the workstations except workstations *s* and *d*. Apply equation (8) with n'_{cpu} and n'_{io} to estimate the process times of all the process on all the workstations except workstations *s* and *d*.

(7) If there is no change in any of the process times that exceeds the threshold ρ, then calculate the response times and throughputs of all the services and finish the estimation. Otherwise, go back to step 5.

5.4 Reallocating Conditions

Our load distribution policy allows users to set an upper limit for the response time, and both lower and upper limits for the throughput of each service. We will respectively refer to the upper limit of response time, the lower limit of throughput, and the upper limit of throughput of any service as the maximum response time, the minimum throughput, and the maximum throughput. The assignment of the selected process to the selected workstation will be determined according to the following reallocating conditions:

(1) the maximum response times, and the minimum and the maximum throughputs of all services are satisfied, and

(2) the response time of a new service is improved.

6 Evaluation

In this section, we measure the response time and the throughput of each service when the placement decisions for their processes are determined using our load distribution policy, and we verify the usefulness of our policy by comparing the measured response time and the measured throughput with those our policy estimates. This section will start with a description of our service model, followed by a description of our system environment, and then proceed to present results and discussion of our measurements.

6.1 Service Model

We wrote a transaction processing program based on the TPC Benchmark^{TM} A specification [16][17], which simulates a typical transaction service such as a banking service. The diagram of how our program works is shown in Figure 9 and is described in detail below. We note that, in Figure 9, A, B, C, and D represent cooperative processes that are created by the program, and each arrow represents either IPC or I/O.

Process A (branch of a bank) generates a transaction by sending 200 bytes of data to process B. After receiving 200 bytes of data back from process B, it sends 200 bytes of data to processes B, C, and D. While waiting for each 200 bytes of data back from processes B, C, and D, it performs a lot of amount of CPU processing (i.e., repeats a simple calculation for 1,000,000 times) and then writes 1 KB of data to its own hard disk.

Process B (database manager) receives 200 bytes of data from process A. Next, it performs some amount of CPU processing (i.e., repeats the simple calculation for 50,000 times), and then writes 1 KB of data to its own hard disk twice. Following this, it sends 200 bytes of data to process A. After receiving 200 bytes of data back from process A, it performs some amount of CPU processing (i.e., repeats the simple calculation for 50,000 times), then writes 1 KB of data to its own hard disk twice, and finally sends 200 bytes of data to process A.

Process C (recovery log manager) receives 200 bytes of data from process A. Next, process C performs a little amount of CPU processing (i.e., repeats the simple calculation for 1,000 times), and then writes 1 KB of data to its own hard disk four times. Finally, it sends 200 bytes of data to process A.

Process D (history manager) behaves in the same way as process C.

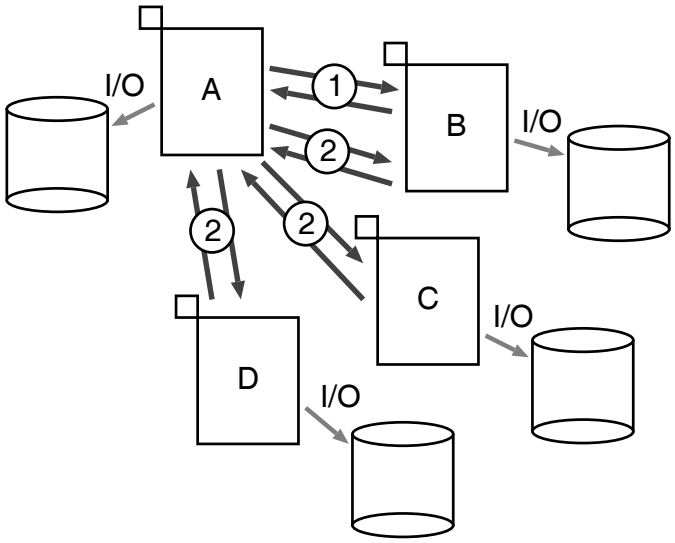

Figure 9: Model of a service based on the TPC-A specification.

6.2 System Environment

We use three workstations (WS1, WS2, and WS3) connected by a 10 Mb/s Ethernet LAN in our measurements. WS1 is equipped with a Pentium Pro 180 MHz processor and 64 MB of memory. WS2 is equipped with a Pentium 133 MHz processor and 32 MB of memory. WS3 is equipped with a Pentium 90 MHz processor and 24 MB of memory. All workstations are running on BSD/OS version 2.1. In the process of estimating the response times of services, we assume that the IPC time between processes on the same workstation is 0.2 milliseconds while that on different workstations is 1 millisecond. These IPC times are set based on actual measurements. Also, we set the threshold ρ (see Section 5.3) to 0.01.

6.3 Results and Discussion

We used our policy to find the process allocation in three cases, as shown in Table 3. Our policy in case 1 and previous policies [4][6][8][10] share similarities in that they do not consider pre-loaded services, and take three steps to decide process allocation as follows: (1) select a process to reallocate, (2) select a workstation for the selected process, and (3) decide whether or not to reallocate. In case 2, pre-loaded services are taken into account by using a bottleneck coefficient. In case 3, the conditions are the same as case 2 except that the reallocating conditions are set.

Table 3: Cases for evaluation.

	bottleneck	reallocating
case 1	not used	not set
case 2	used	not set
case 3	used	set

In each case, we varied the number of services from one to three, and we measured the response time and the throughput of each service and compared them with the estimated ones. Also, all the services are the same and each of them consists of four cooperative processes as we described in Section 6.1. The results are shown in Figures 10 to 17. In each figure, the x-axis represents the service identification (service ID), and "a-*n*" (*n*=1,2,3) represents service "a" of *n* services. For example, service "b-2" is distributed after "a-2" when there are two services.

We first discuss the effectiveness of the bottleneck coefficient by comparing the results between cases 1 and 2, and proceed to discuss the effect of setting reallocating conditions on quality of estimation by comparing the results between cases 2 and 3.

Figure 10 shows the estimated response times in milliseconds (msec) in cases 1 and 2. This figure shows that the estimated response times in case 2 are better than those in case 1. Also, in case 2, no matter how many services are in the system, all the services are likely to produce equivalent response times. On the other hand, in case 1, the last distributed services (i.e., b-2 and c-3) tend to yield better response times than those that were there before their arrival (i.e., the pre-loaded services). This is because adopting a bottleneck coefficient causes equal treatment between the new service and the pre-loaded services. Figure 11 shows the estimated throughputs in transactions per second (tps) in cases 1 and 2; the trend of the results is the same as those in Figure 10.

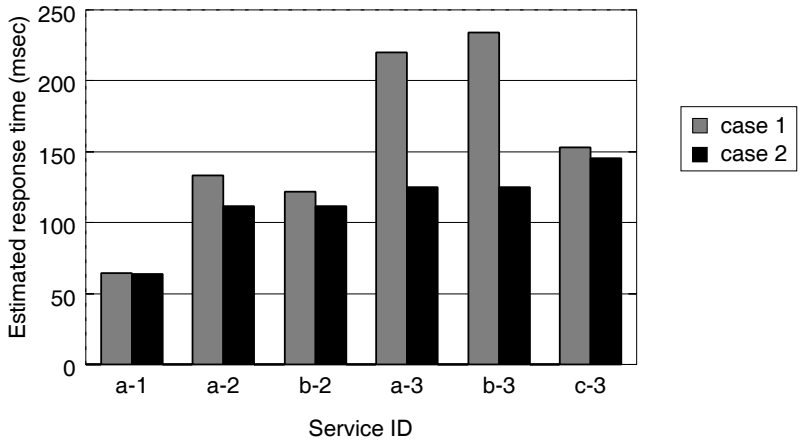

Figure 10: Estimated response times in cases 1 and 2.

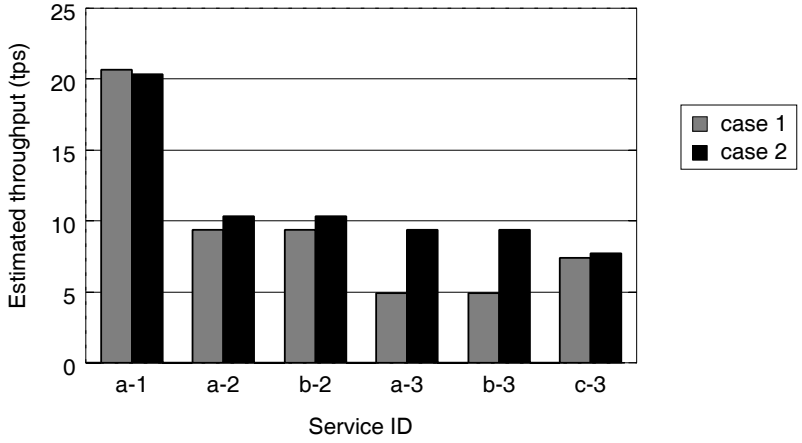

Figure 11: Estimated throughputs in cases 1 and 2.

We can see the effectiveness of the bottleneck coefficient in the estimated response times and the estimated throughputs in Figures 10 and 11. Next, we have to compare the estimated response times and the estimated throughputs with the measured ones. In Figure 12, the comparisons of the estimated and the measured response times are shown in terms of M/E *ratio* in which we defined as $\textit{Measured response time}/\textit{Estimated response time}$. In this figure, there is a very wide range of M/E ratios in case 1. On the other hand, all the M/E ratios in case 2 are close to one; this shows that we can get reliable estimation by using the bottleneck coefficient in our policy. Figure 13 shows comparisons of the estimated and the measured throughputs. It can be noticed that the results in both cases 1 and 2 are close to one. In case 1, they are a little bit closer to one than those in case 2. However, in case 1, the increase in the number of services is likely to make the M/E ratio farther from one. This

is because our policy in case 1 does not take sufficient account of the effects of the pre-loaded services, and thus cannot give a high-quality estimate when the number of services is large.

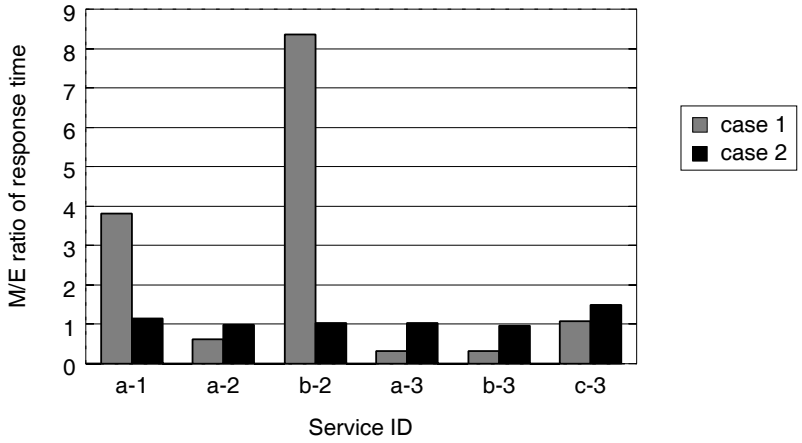

Figure 12: M/E ratio of the response time in cases 1 and 2.

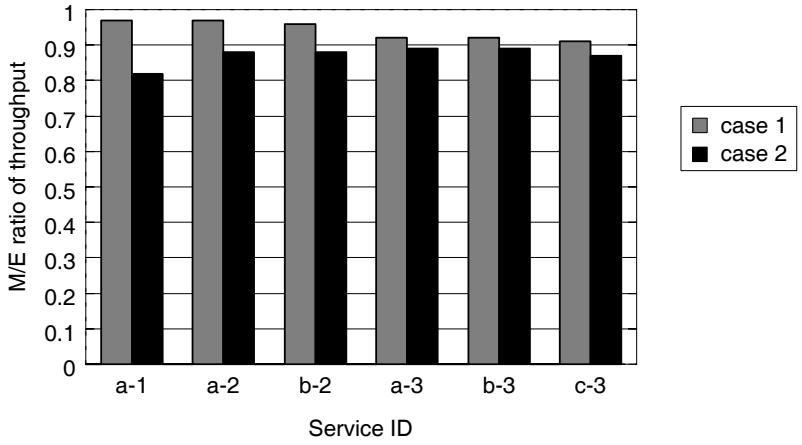

Figure 13: M/E ratio of the throughput in cases 1 and 2.

Table 4 Three patterns of reallocating conditions in case 3.

cases	reallocating conditions
case 3-1	maximum response time =
case 3-2	minimum throughput = 15 maximum throughput = 16
case 3-3	maximum response time = minimum throughput = 15 maximum throughput = 16

In case 3, we set three patterns of reallocating conditions, as shown in Table 4. All the conditions are set to only service a-*n* (*n*=1,2,3). Figure 14 shows the comparisons of the response times our policy estimates when it takes into account the pre-loaded services by using a bottleneck coefficient in case 2 and when it considers both bottleneck coefficient and reallocating conditions in cases 3-1 and 3-3. In this figure, the response times of a-*n* in cases 3-1 and 3-3 are kept below eighty milliseconds, whereas those in case 2 exceed eighty, and those of b-*n* and c-*n* are longer than those in case 2. Figure 15 shows the comparisons of the throughputs our policy estimates when it uses a bottleneck coefficient in case 2 and when it considers both bottleneck coefficient and reallocating conditions in cases 3-2 and 3-3. In this figure, the throughputs of a-*n* are kept between fifteen and sixteen transactions per second, whereas those in case 2 do not reach fifteen. As a result, the throughputs of b-*n* and c-*n* are less than those in case 2. Thus, we see that reallocating conditions set in advance worked well at the estimation level. Next, we compare the estimated response times and the estimated throughputs of a-*n* with the measured ones. As shown in Figures 16 and 17, all the M/E ratios of the response times and the throughputs are close to one. Therefore, setting the reallocating conditions in advance does not affect the quality of the estimation.

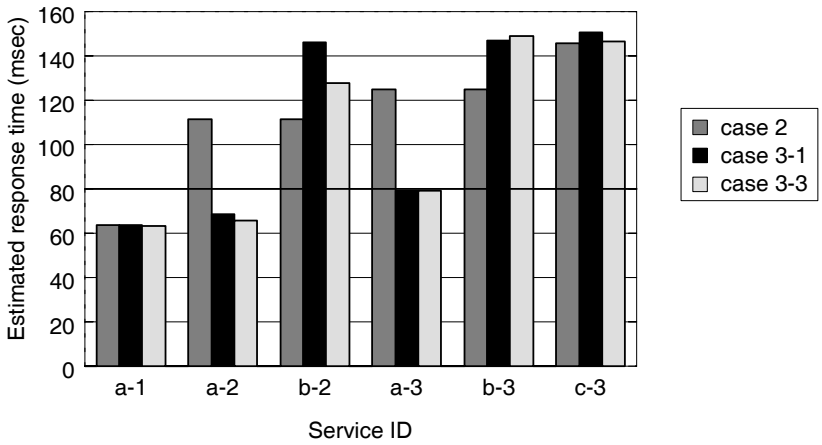

Figure 14: Estimated response times in cases 2, 3-1, and 3-3.

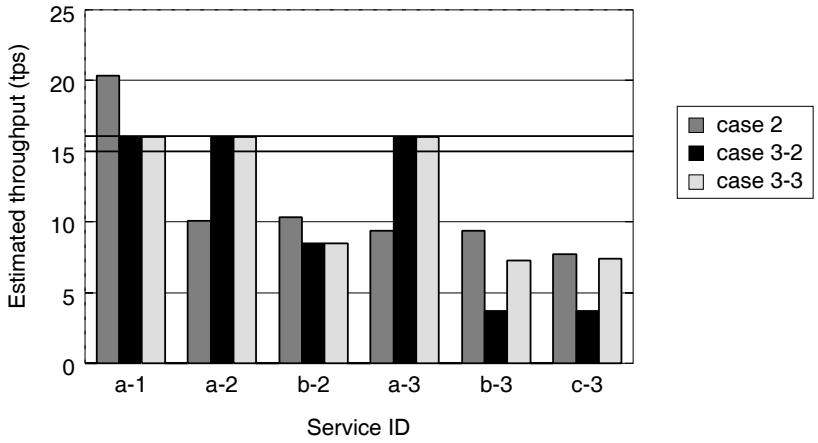

Figure 15: Estimated throughputs in cases 2, 3-2, and 3-3.

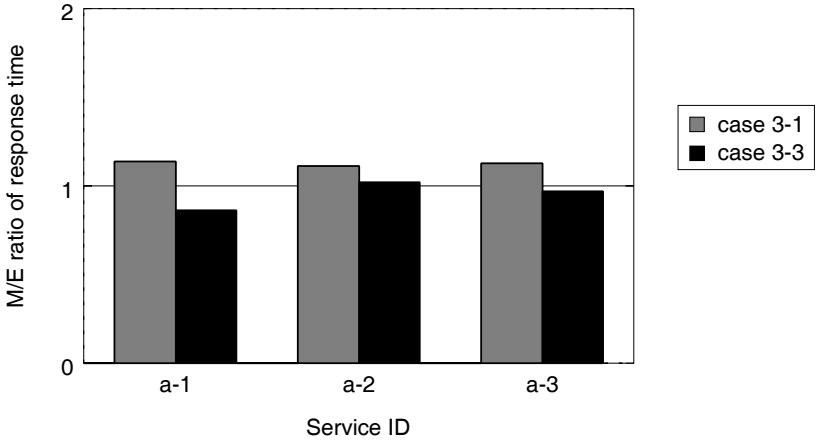

Figure 16: M/E ratio of the response time.

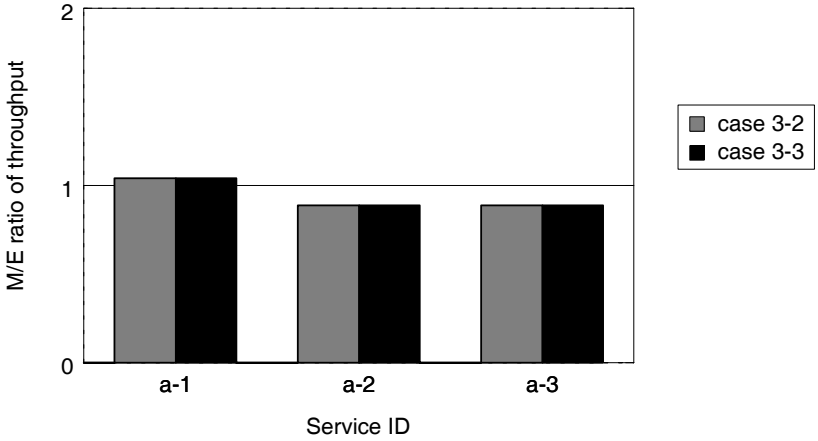

Figure 17: M/E ratio of the throughput.

7 Conclusions

In this paper, we described a model of a distributed system based on a process structure and a system structure. The process structure has elements that represent the behavior of each process, and the system structure has elements that represent the specification of each workstation and the network. We also explained the method of estimating response times and throughputs for transaction services that consist of processes requiring interprocess communication in order to carry out each transaction. Our estimation is based on the process structure and the system structure, and we adopted the original idea of a bottleneck coefficient to take account of the effect of pre-loaded services.

Besides this, we described our load distribution policy, which determines the process allocation on the basis of estimation. Our policy allows us to set an upper limit of response time and a range of throughput for each service. We evaluated our policy by actual measurement, and the results show that the measured response times and the measured throughputs are close to the estimated ones when the bottleneck coefficient is taken into account. The results also show that the conditions set by users in advance do not affect the quality of estimation.

Our future work will include enhancement of our policy to estimate the network latency.

Acknowledgement

I would like to thank my mentors, Prof. Hideo Taniguchi and Dr. Yoshinori Aoki for all of their advices and contributions to this work.

References

[1] Y. C. Chow and W. H. Kohler, “Models for dynamic load balancing in a heterogeneous multiple processor system,” *IEEE Trans. Comput.*, vol.c-28, no.5, pp.354-361, May 1979.

[2] S. Shenker and A. Weinrib, “The optimal control of heterogeneous queueing systems: A paradigm for load-sharing and routing,” *IEEE Trans. Comput.*, vol.38, no.12, pp.1724-1735, Dec. 1989.

[3] F. Bonomi and A. Kumar, “Adaptive optimal load balancing in a nonhomogeneous multiserver system with a central job scheduler,” *IEEE Trans. Comput.*, vol.39, no.10, pp.1232-1250, Oct. 1990.

[4] D. L. Eager, E. D. Lazowska, and J. Zahorjan, “Adaptive load sharing in homogeneous distributed systems,” *IEEE Trans. Software Eng.*, vol.SE-12, no.5, pp.662-675, May 1986.

[5] K. M. Baumgartner and B. W. Wah, “GAMMON: A load balancing strategy for local computer systems with multiaccess networks,” *IEEE Trans. Comput.*, vol.38, no.8, pp.1098-1109, Aug. 1989.

[6] F. C. H. Lin and R. M. Keller, “The gradient model load balancing method,” *IEEE Trans. Software Eng.*, vol.SE-13, no.1, pp.32-38, Jan. 1987.

[7] S. Dandamudi, “Performance impact of scheduling discipline on adaptive load sharing in homogeneous distributed systems,” In *Proc. of the 15th IEEE International Conference on Distributed Computing Systems*, pp.484-492, May 1995.

[8] P. K. Niranjan, G. Shivaratri, and M. Singhal, “Load distributing for locally distributed systems,” *IEEE Computer*, pp.33-44, Dec. 1992.

[9] M. Furuichi, K. Taki, and N. Ichiyoshi, “A multi-level load balancing scheme for or-parallel exhaustive search programs on the multi-psi,” In *Proc. of the 2nd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pp.50-59, Mar. 1990.

[10] B. A. Shirazi, A. R. Hurson, and K. M. Kavi, *Scheduling and Load Balancing in Parallel and Distributed Systems*, IEEE Computer Society Press, 1995.

[11] R. Mirchandaney, D. Towsley, and J. A. Stankovic, “Analysis of the effects of delays on load sharing,” *IEEE Trans. Comput.*, vol.38, no.11, pp.1513-1525, Nov. 1989.

[12] B. Lee, A. R. Hurson, and T. Y. Feng, “A vertically layered allocation scheme for data flow systems,” *J. Parallel and Distributed Computing*, vol.11, no.3, pp.175-187, Mar. 1991

[13] D. L. Eager, E. D. Lazowska, and J. Zhaorjan, “The limited performance benefits of migrating active processes for load sharing,” In *Proc of the 1988 ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, pp.74-85, May 1988.

[14] S. Petri and H. Langendörfer, “Load balancing and fault tolerance in workstation clusters migrating groups of communicating processes,” *ACM SIGOPS Operating Systems Review*, vol.29, no.4, pp.25-36, Oct. 1995.

[15] M. Harchol-balter and A. B. Downey, “Exploiting process lifetime distributions for dynamic load balancing,” *ACM Trans. Computer Syst.*, vol.15, no.3, pp.253-285, Aug. 1997.

[16] http://www.tpc.org

[17] J. Gray, The Benchmark Handbook: For Database and Transaction Processing Systems, Morgan Kaufmann, 1991.