PRACTICAL QUEUEING TECHNIQUES FOR EFFICIENT I/O

L Kleinrock

Professor of Engineering
Department of Computer Science
University of California at Los Angeles

L. KLEINROCK is Professor of Engineering in the Computer Science Department of the University of California, Los Angeles. He is also a consultant in the fields of communication systems, queueing theory, and computers for several national industrial firms as well as certain governmental agencies. He was formerly a staff member of the Systems Analysis group at Lincoln Laboratory at Massachusetts Institute of Technology. Professor Kleinrock's main areas of interest include: computer networks, modelling of computer systems, queueing theory, priority queueing and its applications to time-shared facilities, and communication theory. He has published over sixty papers in these fields and is the author of Communication networks: stochastic message flow and delay *(McGraw-Hill, 1964) and* Queueing systems: Vol I: Theory (1974) and Vol II: Computer Applications (1975) *(Wiley Interscience). As principal investigator for an Advanced Research Projects Agency contract on Computer Networks, he directs efforts involving mathematical modelling and analysis, measurement, and simulation. The computer facility under his direction is part of the nationwide ARPA experimental computer network. Professor Kleinrock was awarded a Guggenheim Fellowship for 1971-72 and was recently made a Fellow of the IEEE.*

PRACTICAL QUEUEING TECHNIQUES FOR EFFICIENT I/O

There are some fairly sophisticated means for connecting devices to computers and running these devices concurrently, in an attempt to share resources in computing machines and hopefully without too much conflict. When conflicts do occur, there are two ways of resolving them. First, data may get lost (but hopefully that does not happen). Secondly, buffers may be inserted to smoothe the flow. In this presentation I shall discuss some of the simple techniques that we have for describing the behaviour of these buffers. How full do they get? How long does data remain in the buffer? How can the models from queueing theory be effectively used to predict the performance of computer systems?

It is fair to say that one of the few methods we have for predicting and measuring the performance of data flow in computer systems is queueing theory. There are few other performance evaluation tools in computer science. Consequently, we should be familiar with some of the major results and some of the major applications that we have seen over the past six or seven years, both for predicting performance and for comparing these predictions to the measurement of real computer systems.

THE FUNDAMENTALS OF QUEUEING THEORY

Figure 1 illustrates a very simple model of a queueing system. In basic terms, a queueing system is one in which arrivals make their way into that system, remain there for a while, and finally leave, hopefully satisfied in some sense. The reason why they enter the system is to gain access to some resource. A computer system is a

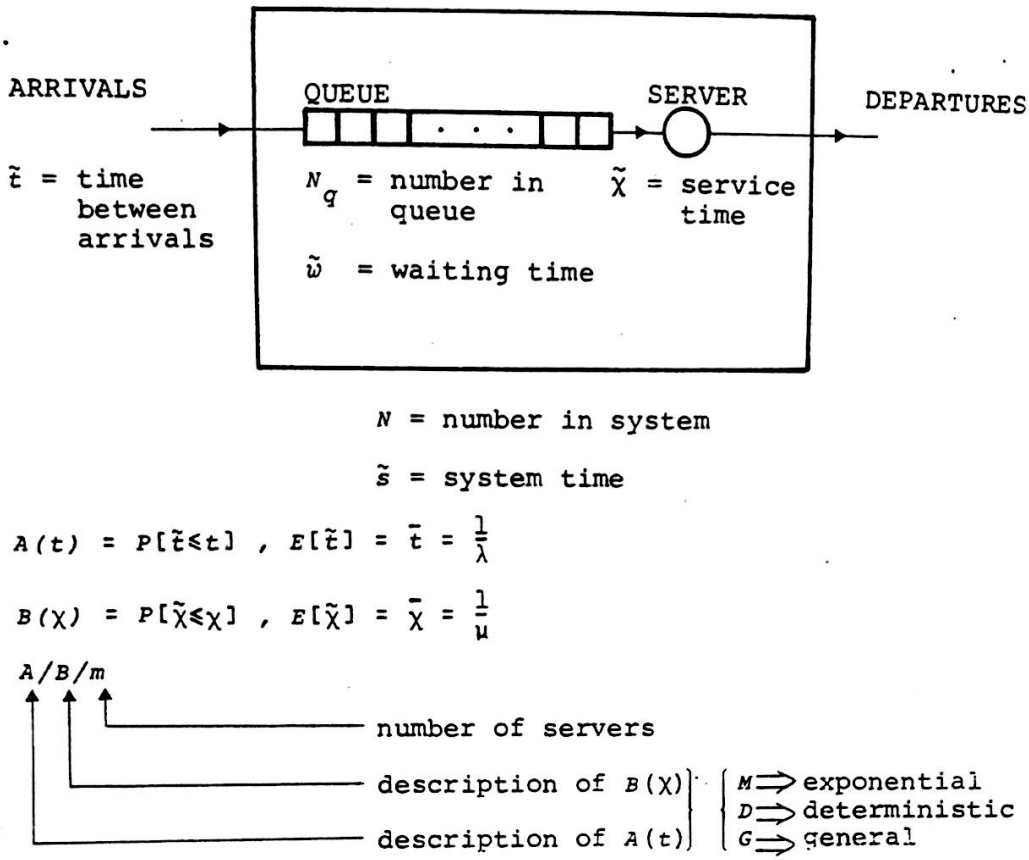


Figure 1: Model of a simple queueing system

collection of resources, each with finite capacity for doing work, for example, the CPU, a data channel, a terminal, a disk, and so on. Requests for these resources arrive at unpredictable times. One may not know when characters are coming from the teletype or when an interrupt is to occur on an I/O channel and, because they are unpredictable arrivals, they may arrive in bunches and so may temporarily overload the server, the CPU, for example. To resolve these overloads, jobs are placed in a queue.

The basic quantities

I should now like to define a few quantities. First, we must talk about the time between arrivals, which is a very basic quantity. For example, Dr Wiseman discusses the variability in that quantity for different devices. Not only must we talk about the time between

arrivals but we have also to talk about how much attention each of those arrivals demands of this resource; and that is called the service time. I shall denote those two quantities by $\tilde{t}$ and $\tilde{\chi}$ respectively. We shall also have to talk about how many items are in the queue (N_q) and how long each spends in the queue, that is, the waiting time ($\tilde{w}$). For the whole system, which includes the queue plus the server, there are similar quantities: the number in the system (N) and the time in the system ($\tilde{s}$). What we really want to know is how long is spent in the queue; how long is spent in the system; how large the queue gets; what fraction of the time the server is being used; what fraction of the customers who come in find no space in the queue (and are therefore perhaps lost). Those are the basic questions.

In order to describe the randomness in the time between arrivals, what is known as the probability distribution function is introduced, namely, the probability that the time between arrivals is less than some figure. The average of the time between arrivals is denoted either by $E[\tilde{t}]$ or $\bar{t}$ or $1/\lambda$. In particular, if $1/\lambda$ is the average time between arrivals, then λ is the average rate of arrivals (say, in characters per second,). $E[\tilde{\chi}]$ describes the average time that the customer will spend in the CPU and, in particular, the average is written as $\frac{1}{\mu}$ or as $\bar{\chi}$, and μ is therefore the service rate (the number of characters the processor can handle per second).

If there is one fundamental law of physics, it is that if a device can do work at a certain rate, it should not be asked to do work at a higher rate than that, since the device cannot handle it. In this context, that means that the arrival rate of jobs should be smaller than the service rate of jobs, that is, $\lambda < \mu$; otherwise the queue will grow to infinity and waiting times will become arbitrarily large. This does not mean that simultaneous arrivals cannot overload the system. Such arrivals will occur but, on the average, more work should not be demanded than the system can provide. Everybody knows that law but very few people have used it. Whenever someone introduces a new configuration of a computer system, I ask where the bottlenecks are, that is, which device is most overloaded. Usually, they have no idea and they wonder what I mean.

Note that λ is not permitted to equal μ. Until a few years ago, in most computer systems, people were quite happy to provide an input rate equal to the rate at which the system could handle that flow. They worked on the assumption that if they could handle 100 characters

per second, they would allow an input rate of 100 characters per second on average. Many time-sharing systems fell into this trap and they soon found that waiting times were becoming arbitrarily large and the queues were becoming arbitrarily large, because they forgot that there was variation in the time between arrivals, and variation in the time required in service. It was those fluctuations, those random quantities, that caused very large queues. Large queues will also occur when λ is close to μ.

To continue my brief description of the standard notation in queueing theory, three other quantities are described in relation to a queue. First, there is the number of servers, denoted by m. Then there is a description of the nature of the time between arrivals, denoted by A, and a description of the nature of the service time requirement, denoted by B. The two descriptions, A and B, can each take on one of many values (in fact, characters). The letter M, for example indicates that the quantity can be described by the expontial distribution. The letter D means it is deterministic; that is, the time between arrivals, for example, is constant at, say, ½ a second between every character. If there is nothing specific about the quantity, then G, for general, is used.

Let me discuss some of these systems. I shall be talking mainly about the exponential case, since it happens to be nice to handle theoretically and it also turns out to be a fairly accurate description based on some of the measured statistics that have been made.

That is the name of the game: people come in, join the queue, get served, and leave. We want to know how long are the queues and how long are the waiting times.

$$p_k = P[N=k] \quad , \quad E[N] = \bar{N}$$

$$W(y) = P[\tilde{w} \leq y] \quad , \quad E[\tilde{w}] = W$$

$$S(y) = P[\tilde{s} \leq y] \quad , \quad E[\tilde{s}] = T$$

$$T = W + \bar{\chi}$$

$$\text{Utilization factor} = \rho = \begin{cases} \bar{\chi}/\bar{t} = \lambda\bar{\chi} & G/G/1 \\ \bar{\chi}/m\bar{t} = \lambda\bar{\chi}/m & G/G/m \end{cases}$$

$$\lambda T = \bar{N}$$

Figure 2: Some measures of performance

Figure 2 shows some further notation for the quantities we wish to find. I want to find the probability that there are k items in the system, and typically that is denoted by p_k, but, more important, I want to know the average number in the system $\bar{N}$. For example, on the average, how many characters are there in the buffer? I want to know the distribution of waiting time and, more importantly, for a simple measure, what is W, the average time one spends waiting in the queue? Similarly, I want to know the distribution of time spent in the system or average time spent in the system. The time in system includes the service time; $\tilde{w}$ does not. So, in general, the average time in system equals the average time in queue plus the average time in service. That is generally true for any queueing system.

Another important definition is the utilization factor, ρ, which indicates the fraction of time that the resource is busy. For example, a utilization factor of 0.9 for a CPU would indicate that the CPU was occupied for 90% of the time and idle for 10%. The value of this factor is easily determined by multiplying the arrival rate by the average service time in an arbitrary single-server queueing system or, in a multiple server system with m servers, it is that same quantity divided by the number of servers. What I am saying is that a trivial calculation yields the fraction of time the server is busy.

Given any two of T, $\bar{N}$, and λ, I can solve for the third by the very important formula at the bottom of Figure 2: the average arrival rate of characters in a system × the average time that those characters spend in the queueing system = the average number of jobs in the system. This is a linear relationship between how long a job spends in the system and how many there are in the system. This was a folk theorem in queueing theory for many years, until it was finally proved in 1961. It is a very general result that applies to any queue one is likely to encounter.

The *M/M/1* queueing system

Figure 3 shows the behaviour of the simplest possible queueing system, one with a single server but where the inter-arrival time is exponential and the time required in the service facility is exponential. that is, an *M/M/1* system.

The equation at the top of Figure 3 gives the value for ρ and the

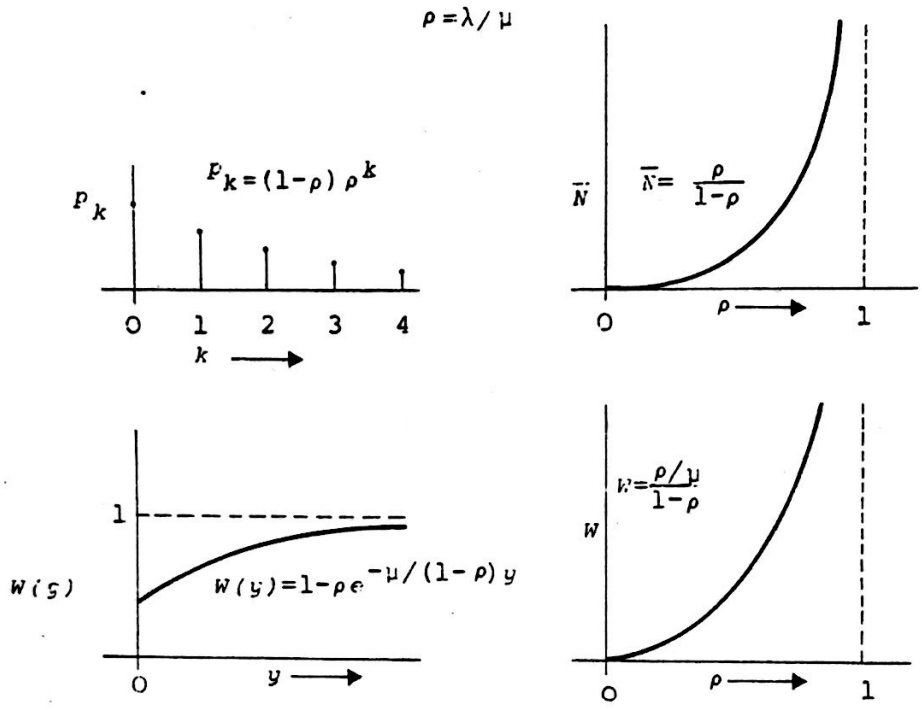


Figure 3: Theoretical values of an M/M/1 system

various curves give the performance of the system. Before discussing this performance, let me point out that the techniques used to give these results are very special, because the mathematics are very simple but the results themselves are rather general. Most queueing systems exhibit behaviour of this type, as I shall describe, so this is more than just the study of a very special system. This behaviour holds for all types of system including G/G/1 systems.

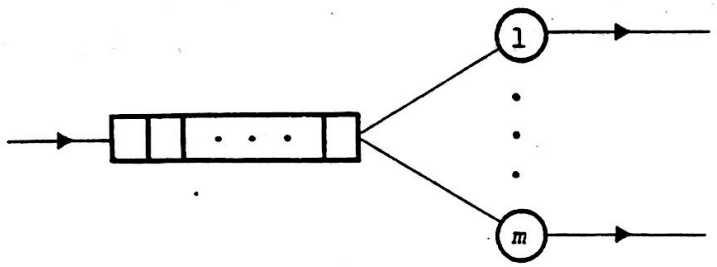
The curve at the top left of Figure 3 shows that the probability of finding k tasks in the system is a geometrically decreasing function; that is, it becomes less and less likely to find more and more tasks in the system. That formula is specific to this system, but the expression for $\bar{N}$ is not. The curve at the top right shows that the average number of jobs one is likely to find in a system as a function of the fraction of time that the system is being used, grows in a very nasty way. It is given by $\rho/(1-\rho)$.

So, for example, with a utilization factor of 0.9, you get a value of 9.0; that is, if the server is busy for 90% of the time, then on average there will be nine customers within the system. If the utilization factor is increased to 0.99, then there will be 99 customers. The utilization factor cannot be increased to the value 1.0 without creating a queue of infinite length on the average, which is why I said earlier that λ must not equal μ, but must be less than μ. The reason for the large growth in the queue length is the fluctuations in arrival times and service times. Every so often, the CPU will be sitting idle for longer than it should on the average and, since on the average it has to be busy for, say, 99% of the time, there will be a huge burst of characters to compensate for that relative idleness, which is when the queues get very long. It is the fluctuations that cause those effects and we shall see later what happens when we exceed saturation. The queues that form in the stable case ($\rho<1$) are due to the fluctuations in the arrival times and service times. Even though, on the average, they can be handled, large queues and large waiting times are encountered. The curve at the bottom right of Figure 3 shows the behaviour of waiting times. Note that $W = \bar{N}/\mu$. The average time spent in the queue gets out of hand very quickly if one tries to drive the system close to saturation. There is a rule of thumb that suggests that 80% is a reasonable figure for utilization; with higher rates, the system begins to suffer very seriously.

The final curve in Figure 3 shows the distribution of the time spent in a queue. The only interesting point to observe is that it is so simple. One can actually calculate the form for this waiting time distribution. For example, we may calculate the probability that a character spends more than 13 seconds in the queue, rather than merely calculating the average time that the character spends in the queue.

The *M/M/m* queueing system

Figure 3 showed the simplest kind of result for queueing theory. Figure 4 gives the formulae for a slightly more complex but much more interesting system: one with a single queue but with multiple servers (multiple channels from the device to the computer, for example). The expression for ρ gives the average number of servers that are busy doing work. The figure also gives the expression for the probability of finding k people in the system.



$$\rho = \frac{\lambda}{m\mu}$$

$$p_k = \begin{cases} p_o \dfrac{(m\rho)^k}{k!} & k \leq m \\ p_o \dfrac{(m\rho)^k m^{m-k}}{m!} & k \geq m \end{cases}$$

$$p_o = \frac{1-\rho}{\dfrac{(m\rho)^m}{m!} + (1-\rho)\displaystyle\sum_{k=0}^{m-1} \frac{(m\rho)^k}{k!}}$$

$$P[\text{must queue}] = C(m,\lambda/\mu) = \frac{p_m}{1-\lambda/\mu}$$

$$W(y) = 1 - C(m,\lambda/\mu)e^{-\mu m(1-\rho)y}$$

$$W = \frac{C(m,\lambda/\mu)}{\mu m(1-\rho)}$$

Figure 4: Theoretical values of an M/M/m system

More important is the next item in the figure, which is the probability that a task cannot find an unoccupied server on arrival and so must join the queue.

Again we see the same form for the distribution of waiting time in the next equation, which is exponential, and we see a simple form in the average waiting time; the curve for T is shown in Figure 6. The important point in this case, once again, is the nasty behaviour evident in the denominator when an attempt is made to drive the utilization close to 1; then the average waiting time blows up. That always happens.

Let us see what the curves look like. First, let us take a look at the probability of joining the queue. What is the probability that, when a character comes in, it does not find that there are any channels empty? I shall plot that probability versus load on the system in Figure 5. ρ is the expected number of busy servers. There is an interesting effect here, namely, the way in which the system degrades, that is, the probability of finding no empty server degrades uniformly with the load on the system in the single-server case, a little bit

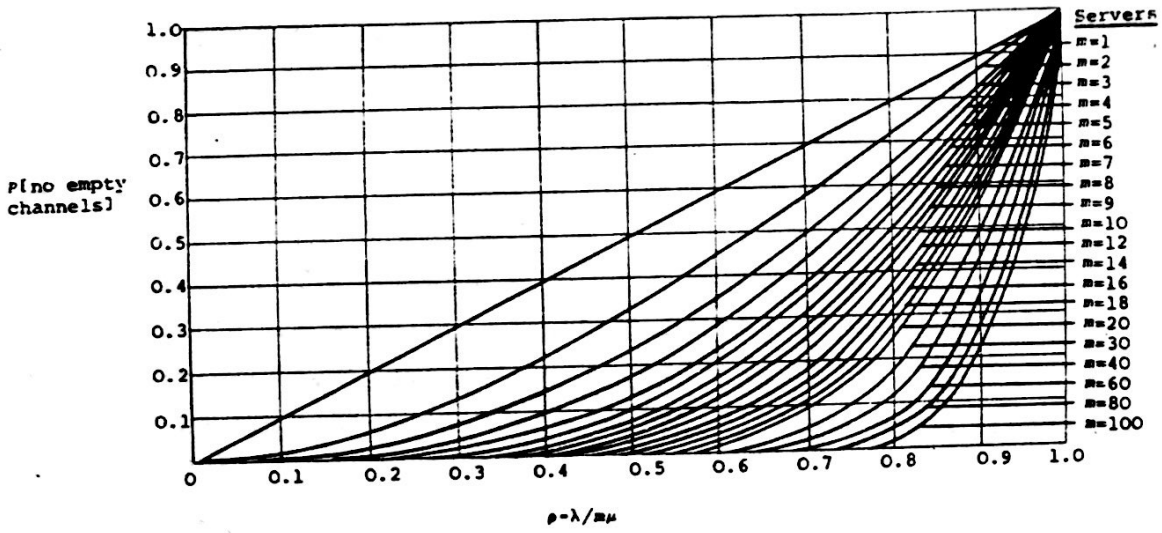


Figure 5: Probability of queueing

slower with two servers, and, as the number of servers increases, the system can be seen to get bad very slowly, and then to get worse very quickly. It hangs on in a fairly good way, and then has a threshold effect to saturation.

This effect is evident in Figure 6, which plots not the probability of finding no empty servers, but rather the average waiting time for any task versus load on the system. For one server, things grow pretty much as in Figure 5 but look at the behaviour as m increases. The

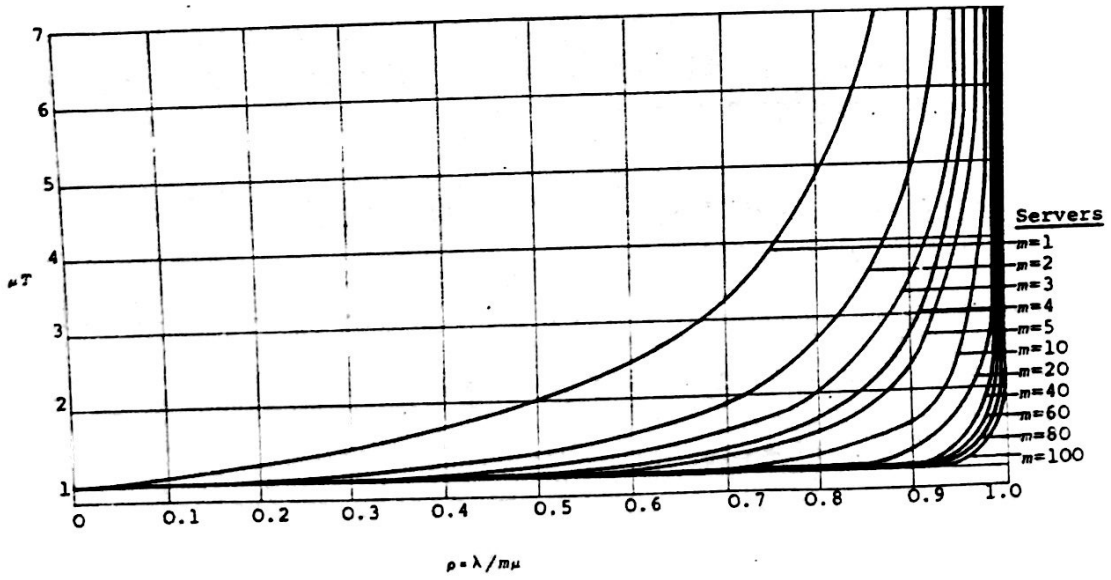


Figure 6: Average time in system (normalized)

system performs beautifully, until it is very close to saturation: then, bang!, it saturates in a nasty way. That means that the utilization can be run way up when there are many servers without suffering very serious degradation. This effect, as we shall see later, occurs also in networks; that is, the system behaves well up to a certain point, then the queues get disastrously long. This is therefore a situation that can be accurately modelled by queueing techniques.

So the queueing effect in a multiple-server system can be described in terms of two parameters: the performance value when it is good, and the place where it gets terrible. It is a very simple model, as a first approximation.

The *M/G/1* queueing system

By generalizing a little, instead of insisting on exponential service times, permitting a general service time, we can obtain more results, in particular for the average number of items waiting for service in the system and the average waiting time. Once again the same effect can be seen in Figure 7: as ρ gets close to 1, we get into trouble.

$$\rho = \lambda\bar{\chi} \quad , \quad Var(\tilde{\chi}) = \sigma_b^2$$

$$C^2 = \frac{\sigma_b^2}{(\bar{\chi})^2}$$

$$\bar{N} = \rho + \frac{\rho^2(1+C^2)}{2(1-\rho)}$$

$$W = \frac{\rho\bar{\chi}(1+C^2)}{2(1-\rho)}$$

Figure 7: Theoretical values of an M/G/1 system (Pollaczek-Khinchin formulae)

More importantly, note what is happening in the numerator of W. There is the quantity C^2 which happens to be a function of the variance of the service time. How much variability is there in the processing time for each character or each task? The larger that variability is, the the larger is the waiting time and therefore the larger the number of tasks waiting in the system. When there is no

variation in the service time, as for example, when servicing characters, then $C = 0$. There is a significant savings if in fact the variance of service times can be reduced. As I have already said, it is these fluctuations that cause queues to build up even when a system is not saturated.

There are two sources of fluctuations: first, the processing time, (which is the C^2 term) and, secondly, the time between arrivals. If there is a general distribution of inter-arrival time and the variation is driven down to 0, then there is a saving.

You will find that the average waiting time when the service time is constant is only half of the average waiting time when the service time is exponentially distributed. Most people are unaware of this difference but, clearly, it is of great importance.

It might be thought that waiting times can be reduced by changing the order in which tasks are processed, by processing the last arrival first, or in some other arbitrary order. In fact, if tasks are selected to be processed in some arbitrary order, the mean waiting time cannot be reduced unless the selection depends on the processing time of the task itself. If, on the other hand, I have a set of tasks to process and I know how long each one of them will take, then I can do something to reduce the mean waiting time. The best thing is to take the shortest task first. But if I do not know how long they will take to process, I cannot improve the mean waiting time by changing the order in which they are processed. What I might do is to introduce an increase in the standard deviation of the waiting time.

So, one might suppose that changing the order of service is beneficial but there is no benefit in terms of the mean waiting time and there may very well be some penalty in terms of the variation in the waiting time.

The *G/G/1* queueing system

Figure 8 shows the properties of a queueing system with an arbitrary time between arrivals, an arbitrary service time, and a single server, that is, a *G/G/1* system. The first equation defines the variance of the time between arrivals. In fact, there is almost nothing we can solve for in this case. We cannot even say what the average

$$Var(\tilde{t}) = \sigma_a^2$$

$$W \leq \frac{\lambda(\sigma_a^2+\sigma_b^2)}{2(1-\rho)}$$

Heavy traffic approximation:

$$\text{As } \rho \longrightarrow 1,\ W(y) \cong 1-exp\left(-\frac{2(1-\rho)}{\lambda(\sigma_a^2+\sigma_b^2)}\,y\right)$$

Figure 8: Theoretical values of a G/G/1 system

waiting time is, only what its upper bound is, the second item in Figure 8. The same effect was shown in Figure 2, namely that the behaviour will be like $1/(1-\rho)$. In fact, as ρ gets close to 1, near the critical point, the upper bound becomes a very excellent approximation. And, more interestingly, the distribution of the time spent in queue is again exponentially distributed, just like in the *M/M/1* case, the very simplest queue. The behaviour is easily described in terms of an exponential distribution. This is called a heavy traffic approximation, because the traffic is close to saturation point, and we can find the probability that the task will have to wait more than so long in the queue.

Now, that is the effect of queues when the system is not saturated but what happens when the system is overloaded? Obviously, the queues will grow but how fast will they grow? Supposing I put in two jobs per second when the system can handle only one job per second. Obviously, the queue will grow at the rate of one job per second. If I put in ten jobs per second when the system can handle one job per second, then it will grow at the rate of nine jobs per second. In other words, it will grow linearly in time.

Let us make an approximation. Let us forget the fluctuations in time between arrivals and the service time, and let us consider a highway that can carry traffic at a given rate: so many cars a minute. When the arrival rate exceeds the rate at which traffic can exit, a backlog will form.

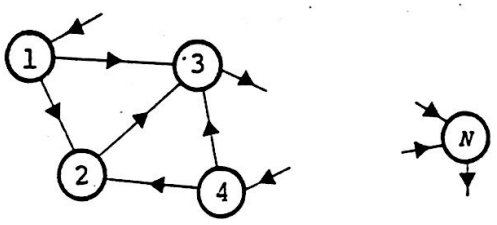
That is a very simple model of a queueing system but it contains basically all of the effects that occur when the system is overloaded. If the system is overloaded, it is a fluid approximation that makes sense. When the system is underloaded, the fluctuations must be

taken care of. One must think about them to predict how long the queues get. When the system is overloaded, it is very easy to calculate what happens. It gets bad very quickly, and in particular notice that the peak backlog occurs when the rush hour ends. In the United States the rush hour is from about 4 to 6 pm but the freeways are still busy until 7.30 pm. It is really slow, and people wonder why. They say that nobody is coming on, but they are all there, on the freeway, slowing each other down. The same thing happens with computer systems, if they get overloaded. We have to look at things that way, in a very simple fashion.

APPLICATIONS OF QUEUEING THEORY

Having described briefly what queueing theory is all about, I should now like to apply it to some computer problems. The difficulty with the models that we have been discussing is that they have been concerned with a single resource. There was typically a single server, or many servers doing the same thing. In a computer facility, there are multiple resources: CPUs, disks, I/O channels, terminals, and so on. What we have to do is to try to model the flow of jobs through this multiple-resource system and ask how long they spend at each position, how long before they pass through the entire system, how many jobs are queued up, and where queues are forming. In order to do that, a network of queues will have to be considered. Figure 9 shows such a network of queues. Each circle represents a queueing facility, for example, a terminal. Jobs arrive at that terminal requesting service and, after they finish typing in whatever task they have, we think of the task as moving, for example, to the CPU, then to a disk, then to memory, then finally back to the terminal. Tasks move around among these various resources, joining queues if the resource is busy in some fashion. The question is whether we can create a model of this, a mathematical model that will predict performance in a network. The answer is that we can.

Figure 9 also gives some notation to show what some of the results are. Let us assume N nodes or N resources in the system. Let us assume that the net traffic into the ith one of these resources is a number, λ_i. Now I have to talk about how tasks move around from one centre to another centre, and r_{ij} will be a model of that, because r_{ij} is the probability that a task will go next to resource j, given that it is currently in resource i. We also need to consider



N nodes

λ_i = traffic rate into node i

r_{ij} = $P[j \text{ next} \mid i \text{ now}]$

γ_i = external traffic rate into node i (Poisson)

m_i = number of servers in node i

$E[\tilde{X}_i] = 1/\mu_i$

Figure 9: A network of queues

tasks coming into the system from outside. In each station there is a way in which jobs can enter from outside this network. Let γ_i be the rate at which jobs enter the ith node from external sources. For example, if there is an idle terminal, what is the rate at which people approach that terminal? Let us assume that arrivals are distributed according to a Poisson process. Further, let m_i parallel servers exist in each of the facilities; m_i might be 1 if the resource is a CPU, 16 if it is a set of data channels, and so forth. Then, for each device, we shall assume that the average service time is unique to that device, and we shall call it $1/\mu_i$. In this context, there are two kinds of systems. One kind of system is the kind I have described, where jobs come in from outside the system, get processed, and finally leave. Such systems are called open systems since there is a way in and a way out. There are also closed systems, in which there is no way in and no way out and jobs in such systems stay there forever, just moving around among the stations. We shall look at both kinds of system.

Figure 10 gives the major results for open networks. First, the

probability that a job leaves the network after it finishes at station i is just 1 minus the probability that it goes somewhere else within the network. The next equation one has to solve, and it is a trivial equation, is to determine what the traffic is into each of the nodes of the network, which is the sum of traffic coming into that node from outside sources and the traffic coming from other stations within the network.

$$P[\text{leave network after node } i] = 1 - \sum_{j=1}^{N} r_{ij}$$

$$\lambda_i = \gamma_i + \sum_{j=i}^{N} \lambda_j r_{ji}$$

$$P[k_1, k_2, \ldots, k_N] = P_1[k_1]\, P_2[k_2] \ldots P_N[k_N]$$

where

$$P_i[k_i] \Longrightarrow M/M/m_i \text{ node}$$

with λ_i , μ_i

Figure 10: Theoretical values for an open network

The final equation means that the probability of finding k_1 jobs queueing up for resource 1, k_2 for resource 2, and k_N for resource N, equals the probability of finding k_1 at node 1 times the probability of k_2 at node 2, and so forth. Therefore, each of these nodes can be extracted from the network and studied by itself. In fact, each one will look like an $M/M/m_i$ system and we have already seen the solution for such systems. The only thing that one needs to know is the arrival rate: what is the parameter associated with the arrival time? That is given by λ_i. The network part of the problem is trivial. You just have to solve for the traffic.

That is a very important result. It means that one can decompose networks if one will accept the statistical assumptions without worrying about interplay among devices, just to know what the average flow is between devices.

Figure 11 gives the corresponding results for closed networks. Here we have K tasks in the system, and there will be no external arrivals.

K = constant number of customers in network

$$\gamma_i = 0$$

$$K = \sum_{i=1}^{N} k_i$$

$$\mu_i \chi_i = \sum_{j=1}^{N} \mu_j \chi_j r_{ji}$$

$$p(k_1, k_2, \ldots, k_N) = \frac{1}{G(K)} \prod_{i=1}^{N} \frac{\chi_i^{k_i}}{\beta_i(k_i)}$$

where

$$\beta_i(k_i) = \begin{cases} k_i! & k_i \leq m_i \\ m_i!\, m_i^{k_i - m_i} & k_i \geq m_i \end{cases}$$

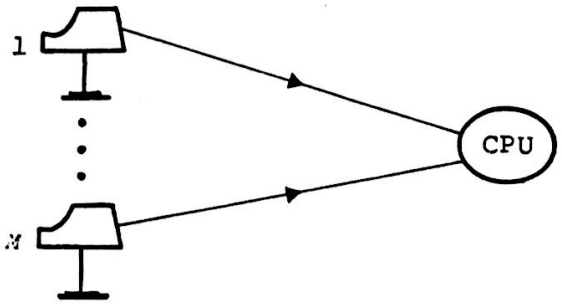
Figure 11: Theoretical values of a closed network

The set of equations we now have to solve is the same as for open networks without external arrivals. In fact we can give an explicit form for the distribution of number at various stations. One can see that it is roughly of the same type as in Figure 10, namely, that it seems to be a product of a set of terms, each of which depends only upon what is happening in the ith station. Again, it almost decomposes.

Terminal polling

We shall study some applications using these network tools. Figure 12 shows the first application, terminal polling. If there is a collection of M terminals from which data has to be transferred to the CPU, we should like to know how long it takes from when a char-

acter is typed in until it is accepted by the CPU.



CASE 1: DIFFUSION APPROXIMATE

$$\bar{N} = \frac{\sigma^2}{2\bar{m}}$$

$$\sigma^2 = \frac{\sigma_a^2}{(\bar{t})^3} + \frac{\sigma_x^2}{(\bar{x})^3}$$

$$\bar{m} = 2\left(\frac{1}{\bar{t}} - \frac{1}{\bar{x}}\right)$$

$$x = \tilde{X}_1 + \tilde{X}_2 + \ldots + \tilde{X}_M$$

CASE 2: EXACT ANALYSIS

$$\bar{N} = \frac{\alpha^2 \bar{t}}{2(\bar{t}-M)} + \frac{M(\bar{t}-1)}{2\bar{t}(\bar{t}-M)}$$

α^2 = *Var* (number of arrivals/unit time)

Figure 12: Terminal polling emptying each terminal in cyclic poll

There are two cases, depending on how much we know about the statistics of character generation. The first case applies to situations where we can make almost no assumptions about the nature of the statistics and the expression given in Figure 12 is an approximation to the behaviour of the system. It tells the average number of characters buffered in a terminal at any point in time. This is an important quantity because from it we can find, by the folk theorem I discussed earlier, how long each character spends in the buffer. The second case is different because it is possible to obtain an exact analysis rather than an approximation. Once again, it is critical that the input rate should be less than the resource rate. We can see from the denominator of the two expressions that as $\bar{t}$, the average time between characters, approaches M, the number of terminals, the average number of buffered characters grows without

bound.

It is important to see that these very simple queueing models give some rather elegant results to terminal polling.

Time-sharing systems

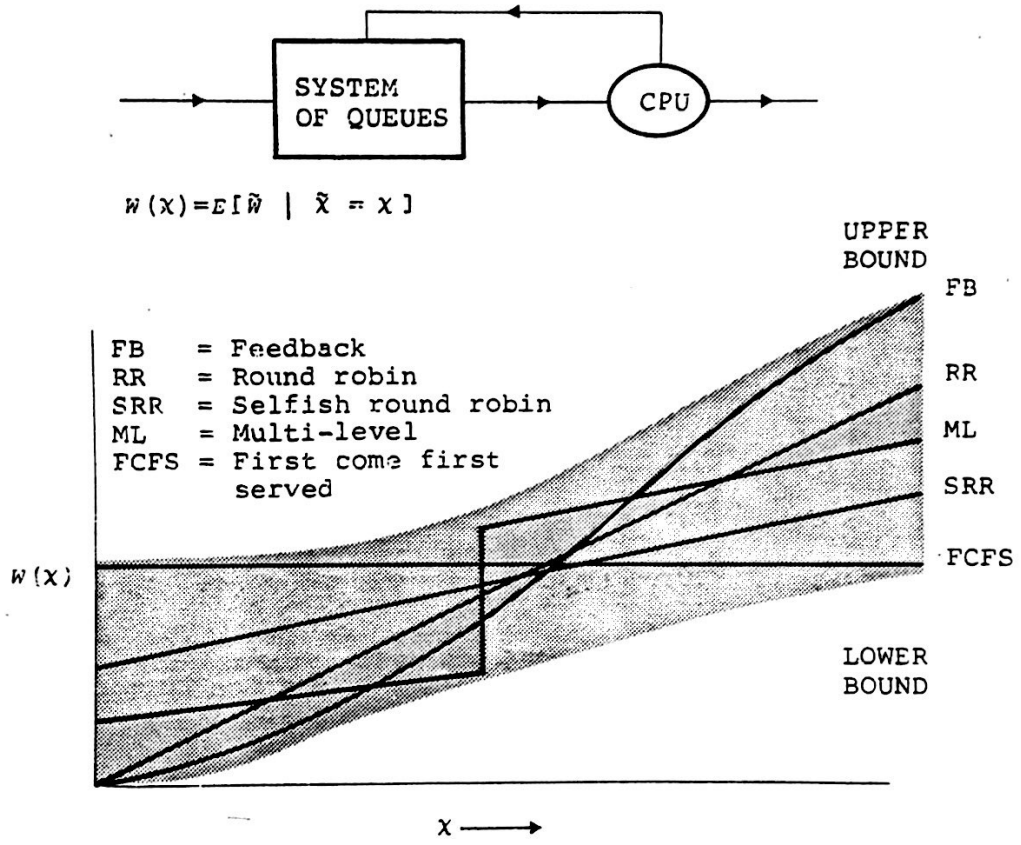


Figure 13: Time-sharing system algorithms

Figure 13 shows another application for a network of queues, a simple model of a time-sharing system. We are once again assuming that there is a single resource available in this case, a CPU. We think of jobs coming in and gaining access to the CPU but, in the time-sharing system, a job does not retain the resource. Instead, it is shared between various jobs by time slicing or some other means. A job comes in and joins a system of queues and eventually gains access to the CPU. If it is given enough, the job will leave. If it is not enough, the time slice may end, and the job gets thrown back into the queues to await another time slice. The jobs will run

around that loop until they finally leave. What we want to know is the effect of the different scheduling algorithms.

First of all, we should like to solve for the average time that a job spends in this system, or in the system of queues. We are talking about the waiting time, so it is a system of queues. We know how long the job will spend in service so we should like to find the average waiting time depending on how many seconds, or milliseconds, the job will spend in the CPU. One of the objectives of time sharing, apart from resource sharing, is to give rapid response to short jobs. If I want to see how that rapid response is given to short jobs, I need to solve the quantity above as a function of job length.

The curves at the bottom of Figure 13 show the performances of a variety of algorithms. For example, in a batch processing system, where jobs are taken from the queue to the CPU and left till they finish, in first-come-first-served fashion, the performance is represented by the horizontal line. In other words, the waiting time, as a function of the job length, is a constant; there is no discrimination in favour of short jobs. It is therefore a lousy time-sharing system. A round-robin algorithm gives a linear discrimination; if the job is twice as long as another, it will spend twice as long, on the average, in the system. The FB algorithm gives the CPU next to that job that has so far had it least, which gives very short jobs the opportunity to get in and get out very quickly. It is thus most discriminatory in favour of the shortest jobs.

I do not want to go into detail about these various algorithms except to say that many schemes have been developed and we know exactly how they behave. For example, there exists a lower bound and an upper bound to $W(\chi)$.

It is an interesting fact that the batch processing algorithm is the worst for short jobs but it is the best for long jobs. One would expect that; since no discrimination has been made in favour of short jobs, the long jobs have not been hurt. Conversely, the FB scheme, the one which gives access next to the job which so far has had least access, is the best for the short jobs and, as might be expected, it is the worst for the long jobs. In between these two is the round-robin, a very common scheme.

Figure 14 shows what is happening outside the time-sharing system. It contains part of the last diagram, the CPU and the system of queues,

but also shows how jobs arise as entries to the system. If there are M consoles with people sitting at them typing requests for access to the time-sharing system, how does this system behave? If we define $1/\gamma$ as the average thinking time, the average time that it takes to compose a character or line at the terminal, and p_0, a quantity which I can solve for, as the probability that there is nobody in the queue or CPU (shown in the box), then the average time, T, from when one enters until one leaves this box, under some statistical assumptions, is given simply by the formula in the Figure.

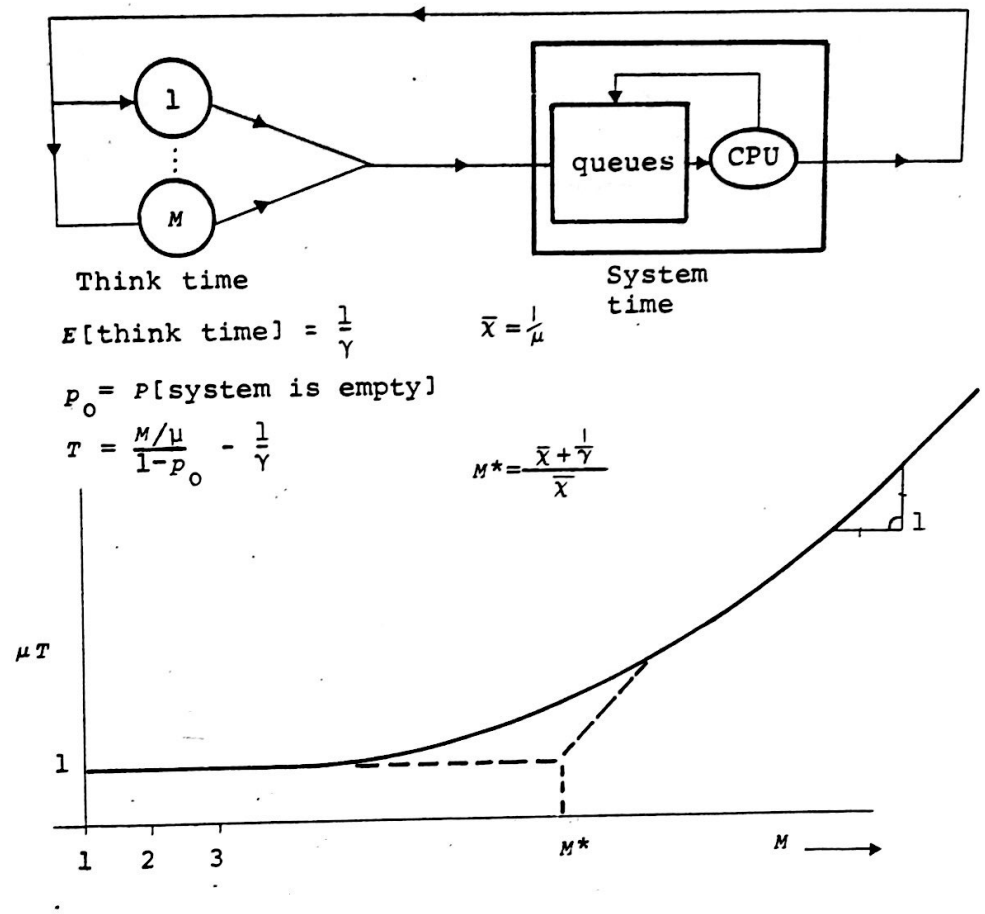


Figure 14: Average waiting versus number of users

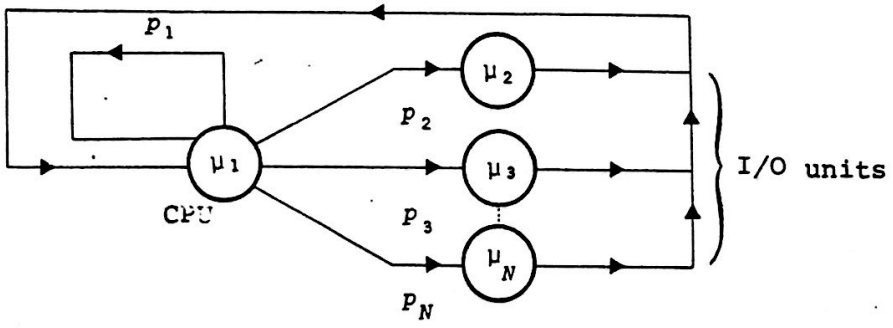
Before discussing the rest of Figure 14, let me point out that this equation was compared favourably to some measurements many years ago on an MIT system, the CTSS compatible time-sharing system.

It might be thought that it is possible to produce an infinite backlog in that system but it is impossible. There can be at the most M people in the system since this is an example of a closed network. People cannot enter and they cannot leave. If I consider this to

be a network, there are $M + 1$ stations essentially, and people keep circulating around, so that the earlier network equations describe the behaviour. There can be at most M people waiting so it is impossible to saturate the CPU if by saturation one means that the delays and queues are driven to infinity.

On the other hand, we know that a time-sharing system is supposed to share resources efficiently. Now, a system is efficiently sharing resources if, when a new user is brought into the system, the other users of the system do not feel his presence in a very strong way. If that is true, then the system absorbed him gracefully. If, on the other hand, when he comes in, everybody gets delayed by an amount equal to how much time he needs on the CPU, then he was not absorbed gracefully; this is what I should like to define as saturation. Figure 14 gives an equation for M^*, the saturation number of the system, which is equal to the average time that the users need the CPU plus the average time spent in thinking, divided by the average time that the users need the CPU. For example, if users spend, on average, 18 seconds thinking and two seconds processing, the system can handle ten users, nine people thinking and one working. This is a very important measure of this kind of system. In Figure 14, we plot T, the time that a job spends waiting and gaining service, and the critical number, M^*, is just about where that curve begins to rise in a nasty way. After this critical point, the curve becomes linear and every job will be delayed by an amount equal to any new job's processing time and the system will not be absorbing them gracefully any more. Below that point, we see the effect of time-sharing; there is no increase in delay, as more and more users come in, until the number approaches that saturation point and the curve begins to climb, eventually becoming linear.

Figure 15 is a model of a multiprogramming system with a fixed number, K, of partitions. The model says that a job is either in the CPU or in some external I/O device, typically accessing memory, a card reader, and so on. After it gets its data, it goes back, and requires service on the CPU again, and so there are K jobs circulating around that loop. It is a closed network of the kind that I described earlier and the solution here is a special case of that more general network. This is a very special network and we understand exactly how it works.



$$p(k_1, k_2, \ldots, k_N) = \frac{1}{G(K)} \prod_{i=2}^{N} \left(\frac{\mu_1 p_i}{\mu_i} \right)^{k_i}$$

$$A_i = P[\text{node } i \text{ busy}] = \begin{cases} G(K-1)/G(K) & i=1 \\ \dfrac{\mu_1 p_i}{\mu_i} A_1 & i=2, \ldots, N \end{cases}$$

Balanced system:

$$\frac{\partial}{\partial \mu_i} A_1 \mu_1 p_1 = \frac{\partial}{\partial \mu_j} A_1 \mu_1 p_1$$

Figure 15: A multiprogramming model

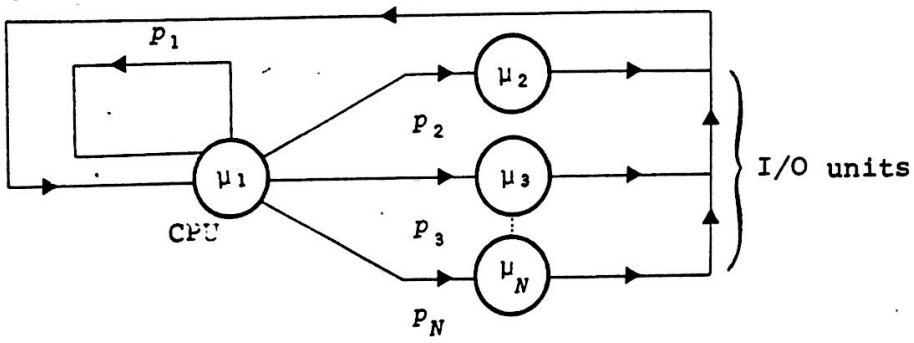
Computer networks

Finally, I should like to make a brief mention of computer networks. Perhaps the ultimate in input/output is to separate not only the two terminals from the computer but to separate computers from each other and to let them communicate with each other, with terminals connected to the computers themselves. In such a case, we would create something called a message service, whereby computers throw messages into the communication network, and the network delivers these messages to other computers. The question is how long does it take the messages to propagate through that network. This is no different than throwing messages into a single facility between two devices, except that it is on a geographically distributed scale.

be a network, there are $M + 1$ stations essentially, and people keep circulating around, so that the earlier network equations describe the behaviour. There can be at most M people waiting so it is impossible to saturate the CPU if by saturation one means that the delays and queues are driven to infinity.

On the other hand, we know that a time-sharing system is supposed to share resources efficiently. Now, a system is efficiently sharing resources if, when a new user is brought into the system, the other users of the system do not feel his presence in a very strong way. If that is true, then the system absorbed him gracefully. If, on the other hand, when he comes in, everybody gets delayed by an amount equal to how much time he needs on the CPU, then he was not absorbed gracefully; this is what I should like to define as saturation. Figure 14 gives an equation for M^*, the saturation number of the system, which is equal to the average time that the users need the CPU plus the average time spent in thinking, divided by the average time that the users need the CPU. For example, if users spend, on average, 18 seconds thinking and two seconds processing, the system can handle ten users, nine people thinking and one working. This is a very important measure of this kind of system. In Figure 14, we plot T, the time that a job spends waiting and gaining service, and the critical number, M^*, is just about where that curve begins to rise in a nasty way. After this critical point, the curve becomes linear and every job will be delayed by an amount equal to any new job's processing time and the system will not be absorbing them gracefully any more. Below that point, we see the effect of time-sharing; there is no increase in delay, as more and more users come in, until the number approaches that saturation point and the curve begins to climb, eventually becoming linear.

Figure 15 is a model of a multiprogramming system with a fixed number, K, of partitions. The model says that a job is either in the CPU or in some external I/O device, typically accessing memory, a card reader, and so on. After it gets its data, it goes back, and requires service on the CPU again, and so there are K jobs circulating around that loop. It is a closed network of the kind that I described earlier and the solution here is a special case of that more general network. This is a very special network and we understand exactly how it works.



$$p(k_1, k_2, \ldots, k_N) = \frac{1}{G(K)} \prod_{i=2}^{N} \left(\frac{\mu_1 p_i}{\mu_i} \right)^{k_i}$$

$$A_i = P[\text{node } i \text{ busy}] = \begin{cases} G(K-1)/G(K) & i=1 \\ \dfrac{\mu_1 p_i}{\mu_i} A_1 & i=2, \ldots, N \end{cases}$$

Balanced system:

$$\frac{\partial}{\partial \mu_i} A_1 \mu_1 p_1 = \frac{\partial}{\partial \mu_j} A_1 \mu_1 p_1$$

Figure 15: A multiprogramming model

Computer networks

Finally, I should like to make a brief mention of computer networks. Perhaps the ultimate in input/output is to separate not only the two terminals from the computer but to separate computers from each other and to let them communicate with each other, with terminals connected to the computers themselves. In such a case, we would create something called a message service, whereby computers throw messages into the communication network, and the network delivers these messages to other computers. The question is how long does it take the messages to propagate through that network. This is no different than throwing messages into a single facility between two devices, except that it is on a geographically distributed scale.

If one attempts to create a model of that, a queueing model, one can find one that works pretty well, a model that describes what the delays are to message traffic in computer networks. If we compare theory with simulation, we see that the delay is quite acceptable up until a critical throughput, at which point the delay blows up. It is not the slow degradation that one might expect. It is a very critical threshold which occurs in networks. Just like in the multiple server case we saw before, when there were 50 or 100 servers, things were fine, and we had that model which I said behaved this way, with a critical load. The same with networks. A very simple model seems to suffice in predicting and estimating the queuing delays in networks. Again it comes right out of the queueing theory models.

CONCLUSION

In conclusion, I should like to say that, although queueing theory is tough and hard to work with, the results that we have give a fairly good description and provide a very good model of systems. For many terminal-oriented input/output computer systems, the flow of data between devices and finite-capacity resources in the computer system can be accurately modelled. A lot of work is going on and the models that people have come up with seem to work fairly well. The queueing theory part is hard but, if one is willing to accept approximations, either in the modelling or in the calculations, one can find some results that predict quite accurately how the jobs are going to queue up, how long they spend waiting, where the bottlenecks are, how to tune up systems, and where one should put additional capacity.